Firebird 3.0 Sprachreferenz

Dmitry Filippov, Alexander Karpeykin, Alexey Kovyazin, Dmitry Kuzmenko, Denis Simonov, Paul Vinkenoog, Dmitry Yemanov, Mark Rotteveel, Martin Köditz

Version 1.13.1-de, 4. April 2024

Die Quelle des meist kopierten Referenzmaterials: Paul Vinkenoog
Weitere Quelle kopierten Referenzmaterials: Thomas Woinke
Übersetzung ins Deutsche: Martin Köditz

Inhaltsverzeichnis

Kapitel 1. Über die Firebird 3.0-Sprachreferenz

Diese Sprachreferenz beschreibt die von Firebird 3.0 unterstützte SQL-Sprache.

Diese Firebird SQL Sprachreferenz ist das zweite zusammenhängende Dokument, das alle Aspekte der Abfragesprache betrachtet, die von Entwicklern für die Kommunikation ihrer Anwendungen mit dem Firebird Relation Database Management System benötigt werden.

1.1. Thema

Das Kernthema dieses Dokuments ist die vollständige Implementierung der SQL-Abfragesprache. Firebird entspricht weitestgehend den internationalen Standards für SQL in Bezug auf Datentypen, Datenspeicherstrukturen, Mechanismen zur referentiellen Integrität sowie Fähigkeiten zur Datenmanipulation und Zugriffsrechte. Firebird beinhaltet außerdem eine robuste Prozedursprache — procedural SQL (PSQL) — für gespeicherte Prozeduren, Trigger und dynamisch ausführbare Code Blöcke. Dies sind die Themen, die in diesem Dokument behandelt werden.

1.2. Urheberschaft

Für die Firebird 3.0-Version wurde die *Firebird 2.5-Sprachreferenz* als Basis verwendet, und Firebird 3.0-Informationen wurden basierend auf den Firebird 3.0-Versionshinweisen, der Funktionsdokumentation und der russischen Firebird 3.0-Sprachreferenz hinzugefügt. Dieses Dokument ist jedoch keine direkte Übersetzung der russischen Firebird 3.0-Sprachreferenz.

1.2.1. Mitwirkende

Direkter Inhalt

- Dmitry Filippov (Autor)
- Alexander Karpeykin (Autor)
- Alexey Kovyazin (Autor, Editor)
- Dmitry Kuzmenko (Autor, Editor)
- Denis Simonov (Autor, Editor)
- Paul Vinkenoog (Autor, Designer)
- Dmitry Yemanov (Autor)
- Mark Rotteveel (Autor)

Ressourcen Inhalt

- Adriano dos Santos Fernandes
- Alexander Peshkov
- Vladyslav Khorsun

- Claudio Valderrama
- Helen Borrie
- und andere

Übersetzungen

- Martin Köditz, it & synergy GmbH

1.3. Anmerkungen

Sponsoren und andere Spender

Sponsoren der russischen Sprachreferenz

- Moscow Exchange (Russland)

  Moscow Exchange ist die größte Börse in Russland sowie Osteuropa, gegründet am 19. Dezember 2011, durch den Zusammenschluss der Börsengruppen MICEX (gegründet 1992) und RTS (gegründet 1995). Moscow Exchange zählt zu den 20 weltweit führenden Börsen im Handel von Anleihen und Aktien, als zu den 10 größten Börsenplattformen für Handelsderivate.

- IBSurgeon (ibase.ru) (Russland)

  Technischer Support und Entwickler administrativer Tools für das Firebird DBMS.

1.4. Beitragen

Es gibt verschiedene Möglichkeiten, wie Sie zur Dokumentation von Firebird oder Firebird im Allgemeinen beitragen können:

- Nehmen Sie an den Mailinglisten teil (siehe https://www.firebirdsql.org/en/mailing-lists/)
- Melden Sie Fehler oder senden Sie Pull-Anfragen auf GitHub (https://github.com/FirebirdSQL/)
- Werden Sie Entwickler (für Dokumentationen kontaktieren Sie uns unter firebird-docs, für Firebird im Allgemeinen verwenden Sie die Firebird-Entwickler-Mailingliste)
- Spenden Sie an die Firebird Foundation (siehe https://www.firebirdsql.org/en/donate/)
- Werden Sie zahlendes Mitglied oder Sponsor der Firebird Foundation (siehe https://www.firebirdsql.org/en/firebird-foundation/)

Kapitel 2. SQL Sprachstruktur

Diese Referenz beschreibt die von Firebird unterstützte SQL-Sprache.

2.1. Hintergrund zu Firebirds SQL-Sprache

Zu Beginn, ein paar Punkte über die Eigenschaften die im Hintergrund von Firebirds Sprache eine Rolle spielen.

2.1.1. SQL Bestandteile

Verschiedene *Teilmengen von SQL* gehören wiederum in verschiedene Aktivitätsbereiche. Die Teilmengen in Firebirds Sprachimplementation sind:

- Dynamic SQL (DSQL)
- Procedural SQL (PSQL)
- Embedded SQL (ESQL)
- Interactive SQL (ISQL)

Dynamic SQL macht den Hauptteil der Sprache aus, der in Abschnitt (SQL/Foundation) 2 der SQL-Spezifikation beschrieben wird. DSQL repräsentiert Statements, die von Anwendungen über die Firebird API durch die Datenbank verarbeitet werden.

Procedural SQL erweitert Dynamic SQL, um zusammengesetzte Anweisungen zu ermöglichen, die lokale Variablen, Zuweisungen, Bedingungen, Schleifen und andere prozedurale Konstrukte enthalten. PSQL entspricht dem Teil 4 (SQL/PSM) Teil der SQL-Spezifikationen. Ursprünglich waren PSQL-Erweiterungen nur in persistent gespeicherten Modulen (Prozeduren und Trigger) verfügbar, aber in neueren Versionen wurden sie auch in Dynamic SQL aufgetaucht (siehe `EXECUTE BLOCK`).

Embedded SQL definiert die DSQL-Untermenge, die von Firebird *gpre* unterstützt wird, der Anwendung, mit der Sie SQL-Konstrukte in Ihre Host-Programmiersprache (C, C++, Pascal, Cobol usw.) einbetten und diese eingebetteten Konstrukte in die richtigen Firebird-API-Aufrufe vorverarbeiten können.

Nur ein Teil der in DSQL implementierten Anweisungen und Ausdrücke wird in ESQL unterstützt.

Interactive ISQL bezieht sich auf die Sprache, die mit Firebird *isql* ausgeführt werden kann, der Befehlszeilenanwendung für den interaktiven Zugriff auf Datenbanken. Als normale Client-Anwendung ist ihre Muttersprache DSQL. Es bietet auch einige zusätzliche Befehle, die außerhalb seiner spezifischen Umgebung nicht verfügbar sind.

Sowohl DSQL- als auch PSQL-Teilmengen werden in dieser Referenz vollständig vorgestellt. Weder ESQL- noch ISQL-Varianten werden hier beschrieben, sofern nicht explizit erwähnt.

2.1.2. SQL-Dialekte

SQL-Dialekt ist ein Begriff, der die spezifischen Funktionen der SQL-Sprache definiert, die beim Zugriff auf eine Datenbank verfügbar sind. SQL-Dialekte können auf Datenbankebene definiert und auf Verbindungsebene angegeben werden. Drei Dialekte stehen zur Verfügung:

- *Dialekt 1* dient ausschließlich dazu, die Abwärtskompatibilität mit Legacy-Datenbanken aus sehr alten InterBase-Versionen, v.5 und darunter, zu ermöglichen. Dialekt 1-Datenbanken behalten bestimmte Sprachfunktionen bei, die sich von Dialekt 3 unterscheiden, dem Standard für Firebird-Datenbanken.
  - Datums- und Uhrzeitinformationen werden im Datentyp `DATE` gespeichert. Ein Datentyp `TIMESTAMP` ist ebenfalls verfügbar, der mit dieser `DATE`-Implementierung identisch ist.
  - Anführungszeichen können als Alternative zu Apostrophen zum Trennen von Zeichenfolgendaten verwendet werden. Dies steht im Gegensatz zum SQL-Standard – doppelte Anführungszeichen sind sowohl in Standard-SQL als auch in Dialekt 3 für einen bestimmten syntaktischen Zweck reserviert. Strings in doppelten Anführungszeichen sind daher streng zu vermeiden.
  - Die Genauigkeit für die Datentypen `NUMERIC` und `DECIMAL` ist kleiner als in Dialekt 3 und wenn die Genauigkeit einer festen Dezimalzahl größer als 9 ist, speichert Firebird sie intern als langen Gleitkommawert.
  - Der Datentyp `BIGINT` (64-Bit-Ganzzahl) wird nicht unterstützt.
  - Bei Bezeichnern wird die Groß-/Kleinschreibung nicht beachtet und müssen immer den Regeln für normale Bezeichner entsprechen — siehe Abschnitt Bezeichner weiter unten.
  - Obwohl Generatorwerte als 64-Bit-Ganzzahlen gespeichert werden, gibt eine Dialekt-1-Client-Anfrage, beispielsweise `SELECT GEN_ID (MyGen, 1)` den Generatorwert auf 32 Bit gekürzt zurück.
- *Dialekt 2* ist nur über die Firebird-Client-Verbindung verfügbar und kann nicht in der Datenbank eingestellt werden. Es soll das Debuggen möglicher Probleme mit Altdaten bei der Migration einer Datenbank von Dialekt 1 auf 3 unterstützen.
- In *Dialekt 3*-Datenbanken,
  - Zahlen (Datentypen `DECIMAL` und `NUMERIC`) werden intern als lange Festkommawerte (skalierte Ganzzahlen) gespeichert, wenn die Genauigkeit größer als 9 ist.
  - Der Datentyp `TIME` ist nur zum Speichern von Uhrzeitdaten verfügbar.
  - Der Datentyp `DATE` speichert nur Datumsinformationen.
  - Der 64-Bit-Integer-Datentyp `BIGINT` ist verfügbar.
  - Doppelte Anführungszeichen sind für die Abgrenzung nicht regulärer Bezeichner reserviert, um Objektnamen zu ermöglichen, bei denen die Groß-/Kleinschreibung beachtet wird oder die auf andere Weise nicht die Anforderungen für reguläre Bezeichner erfüllen.
  - Alle Strings müssen durch einfache Anführungszeichen (Apostrophe) getrennt werden.
  - Generatorwerte werden als 64-Bit-Ganzzahlen gespeichert.

Für neu entwickelte Datenbanken und Anwendungen wird die Verwendung von

> Dialect 3 dringend empfohlen. Sowohl Datenbank- als auch Verbindungsdialekte sollten übereinstimmen, außer unter Migrationsbedingungen mit Dialekt 2.
>
> Diese Referenz beschreibt die Semantik von SQL Dialect 3, sofern nicht anders angegeben.

2.1.3. Fehlerbedingungen

Die Verarbeitung jeder SQL-Anweisung wird entweder erfolgreich abgeschlossen oder schlägt aufgrund einer bestimmten Fehlerbedingung fehl. Die Fehlerbehandlung kann sowohl auf der Clientseite der Anwendung als auch auf der Serverseite mit PSQL erfolgen.

2.2. Grundelemente: Aussagen, Klauseln, Schlüsselwörter

Das primäre Konstrukt in SQL ist die *Anweisung*. Eine Anweisung definiert, was das Datenbankverwaltungssystem mit einem bestimmten Daten- oder Metadatenobjekt tun soll. Komplexere Anweisungen enthalten einfachere Konstrukte — *Klauseln* und *Optionen*.

Klauseln

Eine Klausel definiert eine bestimmte Art von Direktive in einer Anweisung. Zum Beispiel spezifiziert die Klausel `WHERE` in einer `SELECT`-Anweisung und in einigen anderen Datenmanipulationsanweisungen (`UPDATE`, `DELETE`) Kriterien zum Durchsuchen einer oder mehrerer Tabellen nach den Zeilen, die ausgewählt, aktualisiert oder gelöscht werden sollen . Die `ORDER BY`-Klausel gibt an, wie die Ausgabedaten — die Ergebnismenge — sortiert werden sollen.

Optionen

Optionen sind die einfachsten Konstrukte und werden in Verbindung mit bestimmten Schlüsselwörtern angegeben, um eine Qualifizierung für Klauselelemente bereitzustellen. Wenn alternative Optionen verfügbar sind, ist es üblich, dass eine von ihnen die Standardeinstellung ist, die verwendet wird, wenn für diese Option nichts angegeben ist. Zum Beispiel gibt die `SELECT` -Anweisung alle Zeilen zurück, die den Suchkriterien entsprechen, es sei denn, die `DISTINCT` -Option beschränkt die Ausgabe auf nicht duplizierte Zeilen.

Schlüsselwörter

Alle Wörter, die im SQL-Lexikon enthalten sind, sind Schlüsselwörter. Einige Schlüsselwörter sind *reserviert*, was bedeutet, dass ihre Verwendung als Bezeichner für Datenbankobjekte, Parameternamen oder Variablen in einigen oder allen Kontexten verboten ist. Nicht reservierte Schlüsselwörter können als Bezeichner verwendet werden, obwohl dies nicht empfohlen wird. Von Zeit zu Zeit können nicht reservierte Schlüsselwörter reserviert werden, wenn eine neue Sprachfunktion eingeführt wird.

Die folgende Anweisung wird beispielsweise ohne Fehler ausgeführt, da `ABS` zwar ein Schlüsselwort, aber kein reserviertes Wort ist.

```
CREATE TABLE T (ABS INT NOT NULL);
```

Im Gegenteil, die folgende Anweisung gibt einen Fehler zurück, da `ADD` sowohl ein Schlüsselwort als auch ein reserviertes Wort ist.

```
CREATE TABLE T (ADD INT NOT NULL);
```

Siehe die Liste der reservierten Wörter und Schlüsselwörter im Kapitel Reservierte Wörter und Schlüsselwörter.

2.3. Bezeichner

Alle Datenbankobjekte haben Namen, die oft als *Identifier* oder *Bezeichner* angegeben werden. Die maximale Identifier-Länge beträgt 31 Byte. Als Bezeichner sind zwei Arten von Namen gültig: *reguläre* Namen, ähnlich den Variablennamen in regulären Programmiersprachen, und *getrennte* Namen, die für SQL spezifisch sind. Um gültig zu sein, muss jeder Bezeichnertyp einer Reihe von Regeln entsprechen, wie folgt:

2.3.1. Regeln für reguläre Objektbezeichner

- Länge darf 31 Zeichen nicht überschreiten
- Der Name muss mit einem alphabetischen 7-Bit-ASCII-Zeichen ohne Akzent beginnen. Es können weitere 7-Bit-ASCII-Buchstaben, Ziffern, Unterstriche oder Dollarzeichen folgen. Andere Zeichen, einschließlich Leerzeichen, sind nicht gültig. Bei dem Namen wird die Groß-/Kleinschreibung nicht beachtet, dh er kann in Groß- oder Kleinschreibung deklariert und verwendet werden. Somit sind aus Sicht des Systems die folgenden Namen gleich:

```
fullname
FULLNAME
FuLlNaMe
FullName
```

Reguläre Namenssyntax

```
<name> ::=
  <letter> | <name><letter> | <name><digit> | <name>_ | <name>$

<letter> ::= <upper letter> | <lower letter>

<upper letter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M |
                   N | O | P | Q | R | S | T | U | V | W | X | Y | Z

<lower letter> ::= a | b | c | d | e | f | g | h | i | j | k | l | m |
                   n | o | p | q | r | s | t | u | v | w | x | y | z
```

```
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

2.3.2. Regeln für getrennte Objektbezeichner

- Länge darf 31 Byte nicht überschreiten. Bezeichner werden im Zeichensatz `UNICODE_FSS` gespeichert, dh Zeichen außerhalb des ASCII-Bereichs werden mit 2 oder 3 Byte gespeichert.
- Der gesamte String muss in doppelte Anführungszeichen eingeschlossen werden, z.B. `"anIdentifier"`
- Es kann jedes Zeichen aus dem Zeichensatz `UNICODE_FSS` enthalten, einschließlich Akzentzeichen, Leerzeichen und Sonderzeichen
- Ein Bezeichner kann ein reserviertes Wort sein
- Bei Bezeichnern mit Trennzeichen muss die Groß-/Kleinschreibung in allen Kontexten beachtet werden
- Nachgestellte Leerzeichen in durch Trennzeichen getrennten Namen werden wie bei jeder Stringkonstanten entfernt
- Begrenzte Bezeichner sind nur in Dialekt 3 verfügbar. Weitere Informationen zu Dialekten finden Sie unter SQL-Dialekte

Getrennte Namenssyntax

```
<delimited name> ::= "<permitted_character>[<permitted_character> ...]"
```

Ein durch Trennzeichen getrennter Bezeichner wie `"FULLNAME"` entspricht den regulären Bezeichnern `FULLNAME`, `fullname`, `FullName` und so weiter. Der Grund dafür ist, dass Firebird reguläre Bezeichner in Großbuchstaben speichert, unabhängig davon, wie sie definiert oder deklariert wurden. Begrenzte Bezeichner werden immer nach der genauen Schreibweise ihrer Definition oder Deklaration gespeichert. Somit unterscheidet sich `"FullName"` (quoted) von `FullName` (unquoted, d.h. regulär), das als `FULLNAME` in den Metadaten gespeichert wird.

2.4. Literals

Literale werden verwendet, um Daten direkt darzustellen. Beispiele für Standardtypen von Literalen sind:

```
integer        - 0, -34, 45, 0X080000000;
fixed-point    - 0.0, -3.14
floating-point - 3.23e-23;
string         - 'text', 'don''t!';
binary string  - x'48656C6C6F20776F726C64'
date           - DATE '2018-01-19';
time           - TIME '15:12:56';
timestamp      - TIMESTAMP '2018-01-19 13:32:02';
boolean        - true, false, unknown
```

```
null state     - null
```

Details zum Umgang mit den Literalen für jeden Datentyp werden im nächsten Kapitel Datentypen und Untertypen besprochen.

2.5. Operatoren und Sonderzeichen

Eine Reihe von Sonderzeichen ist für die Verwendung als Operatoren oder Trennzeichen reserviert.

```
<special char> ::=
    <space> | " | % | & | ' | ( | ) | * | + | , | -
  | . | / | : | ; | < | = | > | ? | [ | ] | ^ | { | }
```

Einige dieser Zeichen können einzeln oder in Kombination als Operatoren (arithmetisch, string, logisch), als Trennzeichen für SQL-Befehle, als Anführungszeichen für Bezeichner und als Begrenzung von String-Literalen oder Kommentaren verwendet werden.

Operatorsyntax

```
<operator> ::=
    <string concatenation operator>
  | <arithmetic operator>
  | <comparison operator>
  | <logical operator>

<string concatentation operator> ::= "||"

<arithmetic operator> ::= * | / | + | - |

<comparison operator> ::=
    =  | <> | != | ~= | ^= | > | < | >= | <=
  | !> | ~> | ^> | !< | ~< | ^<

<logical operator> ::= NOT | AND | OR
```

Weitere Informationen zu Operatoren finden Sie unter Ausdrücke.

2.6. Bemerkungen

Kommentare können in SQL-Skripten, SQL-Anweisungen und PSQL-Modulen vorhanden sein. Ein Kommentar kann ein beliebiger Text sein, der vom Code-Autor angegeben wird und normalerweise verwendet wird, um zu dokumentieren, wie bestimmte Teile des Codes funktionieren. Der Parser ignoriert den Text von Kommentaren.

Firebird unterstützt zwei Arten von Kommentaren: *block* und *in-line*.

Syntax

```
<comment> ::= <block comment> | <single-line comment>

<block comment> ::=
  /* <character>[<character> ⋯] */

<single-line comment> ::=
  -- <character>[<character> ⋯]<end line>
```

Blockkommentare beginnen mit dem Zeichenpaar `/*` und enden mit dem Zeichenpaar `*/`. Text in Blockkommentaren kann beliebig lang sein und mehrere Zeilen belegen.

Inline-Kommentare beginnen mit einem Bindestrich-Paar `--` und werden bis zum Ende der aktuellen Zeile fortgesetzt.

Beispiele

```
CREATE PROCEDURE P(APARAM INT)
  RETURNS (B INT)
AS
BEGIN
  /* This text will be ignored during the execution of the statement
     since it is a comment
  */
  B = A + 1; -- In-line comment
  SUSPEND;
END
```

Kapitel 3. Datentypen und Untertypen

Daten verschiedener Art werden verwendet, um:

- Spalten in einer Datenbanktabelle in der `CREATE TABLE`-Anweisung definieren oder Spalten mit `ALTER TABLE` ändern
- deklarieren oder ändern Sie eine *Domäne* mit den Anweisungen `CREATE DOMAIN` oder `ALTER DOMAIN`
- lokale Variablen in Stored Procedures, PSQL-Blöcken und Triggern deklarieren und Parameter in Stored Procedures angeben
- Argumente und Rückgabewerte indirekt angeben, wenn externe Funktionen deklariert werden (UDFs — benutzerdefinierte Funktionen)
- Argumente für die Funktion `CAST()` bereitstellen, wenn Daten explizit von einem Typ in einen anderen konvertiert werden

Tabelle 1. Übersicht der Datentypen

Name	Größe	Präzision & Grenzen	Beschreibung
`BIGINT`	64 Bits	Von -2^{63} bis (2^{63} - 1)	Nur in Dialekt 3 verfügbar
`BLOB`	unterschiedlich	Die Größe eines BLOB-Segments ist auf 64K begrenzt. Die maximale Größe eines BLOB-Feldes sind 4GB.	Ein Datentyp mit dynamisch unterschiedlicher Größe für die Ablage von großen Datenmengen, wie z.B. Bilder, Texte, Audiodaten. Die strukturelle Basiseinheit is das Segment. Der BLOB-Untertyp definiert dessen Inhalt.
`BOOLEAN`	8 Bits	false, true, unknown	Boolean-Datentyp
`CHAR(n)`, `CHARACTER(n)`	*n* Zeichen. Größe in Bytes abhängig von der Encodierung, der Anzahl Bytes pro Zeichen	von 1 bis 32,767 Bytes	Ein Datentyp mit fester Länge. Bei Anzeige der Daten werden Leerzeichen an das Ende der Zeichenkette bis zur angegebenen Länge angefügt. Die Leerzeichen werden nicht in der Datenbank gespeichert, jedoch wiederhergestellt, um die definierte Länge bei Anzeige am Client zu erreichen. Die Leerzeichen werden nicht über das Netzwerk versendet, was den Datenverkehr reduziert. Wurde kein Zeichenlänge angegeben, wird 1 als Standardwert verwendet.

Name	Größe	Präzision & Grenzen	Beschreibung
`DATE`	32 Bits	von 01.01.0001 AD bis 31.12.9999 AD	`ISC_DATE`. Nur Datum, kein Zeitelement
`DECIMAL ( precision, scale)`	Varying (16, 32 or 64 bits)	*precision* = von 1 bis 18, legt die mindestmögliche Anzahl zu speichernder Ziffern fest; _scale) = von 0 bis 18, gibt die Anzahl der Nachkommastellen an.	Eine Kommazahl mit *scale* Nachkommastellen. *scale* muss kleiner oder gleich -precision_ sein. Beispiel: `NUMERIC(10,3)` ist eine Zahl im Format: `PPPPPPP.sss`
`DOUBLE PRECISION`	64 Bits	$2.225 * 10^{-308}$ bis $1.797 * 10^{308}$	Doppelte Präzision nach IEEE, ~15 Stellen, zuverlässige Größe hängt von der Plattform ab.
`FLOAT`	32 bits	$1.175 * 10^{-38}$ bis $3.402 * 10^{38}$	Einfache Präzision nach IEEE, ~7 Stellen
`INTEGER, INT`	32 Bits	-2.147.483.648 up to 2.147.483.647	Ganzzahlen mit Vorzeichen
`NUMERIC ( precision, scale)`	Unterschiedlich (16, 32 oder 64 Bits)	*precision* = von 1 bis 18, legt die genaue Anzahl zu speichernder Stellen fest; *scale* = von 0 bis 18, legt die Anzahl der Nachkommastellen fest.	Eine Kommazahl mit *scale* Nachkommastellen. *scale* muss kleiner oder gleich -precision_ sein. Beispiel: `NUMERIC(10,3)` ist eine Zahl im Format: `PPPPPPP.sss`
`SMALLINT`	16 Bits	-32.768 bis 32.767	Ganzzahlen mit Vorzeichen (word)
`TIME`	32 Bits	0:00 to 23:59:59.9999	`ISC_TIME`. Tageszeit. Kann nicht zum Spiechern von Zeitintervallen verwendet werden.
`TIMESTAMP`	64 Bits (2 X 32 Bits)	Von Anfang des Tages 01.01.0001 AD bis Ende des Tages 31.12.9999 AD	Datum und Uhrzeit eines Tages

Name	Größe	Präzision & Grenzen	Beschreibung
`VARCHAR(n)`, `CHAR VARYING`, `CHARACTER VARYING`	*n* Zeichen. Größe in Bytes, abhängig von der Enkodierung, der Anzahl von Bytes für ein Zeichen	von 1 bis 32,765 Bytes	Zeichenkette mit variabler Länge. Die Gesamtgröße der Zeichen darf (32KB-3) nicht übersteigen. Dies berücksichtigt auch die hinterlegte Enkodierung. Die beiden hinteren Bytes speichern die deklarierte Länge. Es gibt keine Standardgröße. Das Argument *n* ist erforderlich. Führende und abschließende Leerzeichen werden gespeichert und nicht abgeschnitten, außer den Leerzeichen, die hinter der definierten Länge liegen.

Hinweis zu Daten

Beachten Sie, dass eine Zeitreihe, bestehend aus Daten der letzten Jahrhunderte, verarbeitet wird, ohne auf historische Gegebenheiten Rücksicht zu nehmen. Dennoch ist der Gregorianische Kalender komplett anwendbar.

3.1. Ganzzahlen-Datentypen

Die Datentypen `SMALLINT`, `INTEGER` und `BIGINT` werden für Ganzzahlen verschiedener Präzisionen in Dialekt 3 verwendet. Firebird unterstützt keine vorzeichenlosen (unsigned) Integer.

3.1.1. `SMALLINT`

Der 16-Bit-Datentyp "SMALLINT" dient der kompakten Datenspeicherung von Integer-Daten, für die nur ein enger Bereich möglicher Werte benötigt wird. Zahlen vom Typ `SMALLINT` liegen im Bereich von -2^{16} bis $2^{16} - 1$, also von -32.768 bis 32.767.

`SMALLINT`-*Beispiele*

```
CREATE DOMAIN DFLAG AS SMALLINT DEFAULT 0 NOT NULL
  CHECK (VALUE=-1 OR VALUE=0 OR VALUE=1);

CREATE DOMAIN RGB_VALUE AS SMALLINT;
```

3.1.2. `INTEGER`

Der Datentyp `INTEGER` ist eine 32-Bit-Ganzzahl. Die Kurzbezeichnung des Datentyps lautet 'INT'. Zahlen vom Typ `INTEGER` liegen im Bereich von -2^{32} bis $2^{32} - 1$, also von -2.147.483.648 bis 2.147.483.647.

INTEGER-*Beispiele*

```
CREATE TABLE CUSTOMER (
  CUST_NO INTEGER NOT NULL,
  CUSTOMER VARCHAR(25) NOT NULL,
  CONTACT_FIRST VARCHAR(15),
  CONTACT_LAST VARCHAR(20),
  ...
    PRIMARY KEY (CUST_NO) )
```

3.1.3. BIGINT

BIGINT ist ein SQL:99-kompatibler 64-Bit-Integer-Datentyp, der nur in Dialect 3 verfügbar ist. Wenn ein Client Dialekt 1 verwendet, wird der vom Server gesendete Generatorwert auf eine 32-Bit-Ganzzahl (INTEGER) reduziert. Wenn Dialekt 3 für die Verbindung verwendet wird, ist der Generatorwert vom Typ BIGINT.

Zahlen des Typs 'BIGINT' liegen im Bereich von -2^{63} bis $2^{63} - 1$, oder von -9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807.

3.1.4. Hexadezimales Format für Integer-Zahlen

Ab Firebird 2.5 können Konstanten der drei Integer-Typen im hexadezimalen Format mit 9 bis 16 hexadezimalen Stellen für BIGINT bzw. 1 bis 8 Stellen für INTEGER angegeben werden. Die Hex-Darstellung zum Schreiben in SMALLINT wird nicht explizit unterstützt, aber Firebird wandelt eine Hex-Zahl bei Bedarf transparent in SMALLINT um, sofern sie in den negativen und positiven SMALLINT -Bereich fällt.

Die Verwendung und die numerischen Wertebereiche der hexadezimalen Notation werden in der Diskussion zu Zahlenkonstanten im Kapitel *Allgemeine Sprachelemente* genauer beschrieben.

Beispiele mit Integer-Typen

```
CREATE TABLE WHOLELOTTARECORDS (
  ID BIGINT NOT NULL PRIMARY KEY,
  DESCRIPTION VARCHAR(32)
);

INSERT INTO MYBIGINTS VALUES (
  -236453287458723,
  328832607832,
  22,
  -56786237632476,
  0X6F55A09D42,       -- 478177959234
  0X7FFFFFFFFFFFFFFF, -- 9223372036854775807
  0XFFFFFFFFFFFFFFFF, -- -1
  0X80000000,         -- -2147483648, ein INTEGER
  0X080000000,        -- 2147483648, ein BIGINT
  0XFFFFFFFF,         -- -1, ein INTEGER
  0X0FFFFFFFF         -- 4294967295, ein BIGINT
```

```
);
```

Die hexadezimalen INTEGERs im obigen Beispiel werden automatisch in BIGINT umgewandelt, bevor sie in die Tabelle eingefügt werden. Dies geschieht jedoch *nach* der Zahlenwert ermittelt wurde, also werden 0x80000000 (8 Stellen) und 0x080000000 (9 Stellen) als unterschiedliche BIGINT Werte gespeichert.

3.2. Gleitkomma-Datentypen

Gleitkomma-Datentypen werden in einem binären IEEE 754-Format gespeichert, das Vorzeichen, Exponent und Mantisse umfasst. Die Genauigkeit ist dynamisch und entspricht dem physischen Speicherformat des Werts, das genau 4 Byte für den Typ "FLOAT" und 8 Byte für "DOUBLE PRECISION" beträgt.

Angesichts der Besonderheiten beim Speichern von Gleitkommazahlen in einer Datenbank werden diese Datentypen nicht zum Speichern von Gelddaten empfohlen. Aus den gleichen Gründen werden Spalten mit Gleitkommadaten nicht für die Verwendung als Schlüssel oder für die Anwendung von Eindeutigkeitsbeschränkungen empfohlen.

Zum Testen von Daten in Spalten mit Gleitkomma-Datentypen sollten Ausdrücke anhand eines Bereichs, beispielsweise BETWEEN, prüfen, anstatt nach genauen Übereinstimmungen zu suchen.

Bei der Verwendung dieser Datentypen in Ausdrücken ist bei der Rundung der Auswertungsergebnisse äußerste Vorsicht geboten.

3.2.1. FLOAT

Der Datentyp FLOAT hat eine ungefähre Genauigkeit von 7 Stellen nach dem Komma. Um die Sicherheit der Lagerung zu gewährleisten, verlassen Sie sich auf 6 Ziffern.

3.2.2. DOUBLE PRECISION

Der Datentyp "DOUBLE PRECISION" wird mit einer ungefähren Genauigkeit von 15 Stellen gespeichert.

3.3. Festkomma-Datentypen

Festkomma-Datentypen stellen die Vorhersagbarkeit von Multiplikations- und Divisionsoperationen sicher und machen sie zur ersten Wahl zum Speichern von Geldwerten. Firebird implementiert zwei Festkomma-Datentypen: NUMERIC und DECIMAL. Beide Typen begrenzen laut Norm die gespeicherte Zahl auf die angegebene Skala (die Anzahl der Nachkommastellen).

Unterschiedliche Behandlungen begrenzen die Genauigkeit für jeden Typ: Die Genauigkeit für NUMERIC-Spalten ist genau „wie deklariert“, während DECIMAL-Spalten Zahlen akzeptieren, deren Genauigkeit mindestens der Deklaration entspricht.

HINWEIS: Das Verhalten von NUMERIC und DECIMAL in Firebird entspricht dem SQL-Standard DECIMAL; die Genauigkeit entspricht mindestens der Deklaration.

Beispielsweise definiert `NUMERIC(4, 2)` eine Zahl, die insgesamt aus vier Ziffern besteht, einschließlich zweier Nachkommastellen; das heißt, er kann bis zu zwei Ziffern vor dem Punkt und nicht mehr als zwei Ziffern nach dem Punkt haben. Wird in eine Spalte mit dieser Datentypdefinition die Zahl 3.1415 geschrieben, wird der Wert 3.14 in der Spalte `NUMERIC(4, 2)` gespeichert.

Die Deklarationsform für Festkommadaten, zB `NUMERIC(p, s)`, ist beiden Typen gemeinsam. Es ist wichtig zu wissen, dass das Argument `s` in dieser Vorlage *scale* ist und nicht "eine Anzahl von Stellen nach dem Komma". Das Verständnis des Mechanismus zum Speichern und Abrufen von Festkommadaten sollte dabei helfen, zu veranschaulichen, warum: Zum Speichern wird die Zahl mit 10^s (10 hoch `s`) multipliziert und in eine ganze Zahl umgewandelt; beim Lesen wird die ganze Zahl zurückkonvertiert.

Die Methode zum Speichern von Festkommadaten im DBMS hängt von mehreren Faktoren ab: Deklarierte Genauigkeit, Datenbankdialekt, Deklarationstyp.

Tabelle 2. Methode der physischen Speicherung für reelle Zahlen

Skalierung	Datentyp	Dialekt 1	Dialekt 3
1 - 4	`NUMERIC`	`SMALLINT`	`SMALLINT`
1 - 4	`DECIMAL`	`INTEGER`	`INTEGER`
5 - 9	`NUMERIC` oder `DECIMAL`	`INTEGER`	`INTEGER`
10 - 18	`NUMERIC` oder `DECIMAL`	`DOUBLE PRECISION`	`BIGINT`

3.3.1. `NUMERIC`

Datenformat für die Deklaration

```
  NUMERIC
| NUMERIC(precision)
| NUMERIC(precision, scale)
```

Tabelle 3. `NUMERIC`-Typparameter

Parameter	Beschreibung
precision	Präzision zwischen 1 und 18. Standardmäßig auf 9.
scale	Skala, zwischen 0 und *scale*. Standardmäßig auf 0.

Speicherbeispiele

Zusätzlich zu der obigen Erläuterung speichert das DBMS "NUMERISCHE" Daten gemäß der deklarierten *Präzision* und *Skala*. Einige weitere Beispiele sind:

```
NUMERIC(4) gespeichert als SMALLINT (genaue Daten)
NUMERIC(4,2)               SMALLINT (Daten * 10^2)
NUMERIC(10,4) (Dialekt 1)  DOUBLE PRECISION
              (Dialekt 3)  BIGINT (Daten * 10^4)
```

Denken Sie immer daran, dass das Speicherformat von der Genauigkeit abhängt. Zum Beispiel definieren Sie den Spaltentyp als `NUMERIC(2,2)` unter der Annahme, dass sein Wertebereich -0,99…0,99 beträgt. Der tatsächliche Wertebereich für die Spalte beträgt jedoch -327,68..327,67, was darauf zurückzuführen ist, dass der Datentyp `NUMERIC(2,2)` im `SMALLINT`-Format gespeichert ist. Im Speicher sind die Datentypen `NUMERIC(4,2)`, `NUMERIC(3,2)` und `NUMERIC(2,2)` tatsächlich gleich. Das heißt, wenn Sie wirklich Daten in einer Spalte mit dem Datentyp `NUMERIC(2,2)` speichern und den Bereich auf -0,99…0,99 begrenzen möchten, müssen Sie dafür eine Einschränkung erstellen.

3.3.2. DECIMAL

Datendeklarationsformat

```
  DECIMAL
| DECIMAL(precision)
| DECIMAL(precision, scale)
```

Tabelle 4. `DECIMAL`*-Typparameter*

Parameter	Beschreibung
precision	Präzision zwischen 1 und 18. Standardmäßig auf 9.
scale	Skala, zwischen 0 und *scale*. Standardmäßig auf 0.

Speicherbeispiele

Das Speicherformat in der Datenbank für `DECIMAL` ist `NUMERIC` sehr ähnlich, mit einigen Unterschieden, die anhand einiger weiterer Beispiele leichter zu erkennen sind:

```
DECIMAL(4) gespeichert als INTEGER (exact data)
DECIMAL(4,2)               INTEGER (data * 10²)
DECIMAL(10,4) (Dialekt 1)  DOUBLE PRECISION
              (Dialekt 3)  BIGINT (data * 10⁴)
```

3.4. Datentypen für Datum und Uhrzeit

Die Datentypen `DATE`, `TIME` und `TIMESTAMP` werden verwendet, um mit Daten zu arbeiten, die Datums- und Uhrzeitangaben enthalten. Dialekt 3 unterstützt alle drei Typen, während Dialekt 1 nur `DATUM` hat. Der `DATE`-Typ in Dialekt 3 ist “nur Datum”, während der `DATE`-Typ von Dialekt 1 sowohl Datum als auch Uhrzeit speichert, was `TIMESTAMP` in Dialekt 3 entspricht. Dialekt 1 hat keinen Typ “nur Datum”.

Dialekt 1 `DATE` Daten können alternativ als `TIMESTAMP` definiert werden und dies wird für neue Definitionen in Dialekt 1 Datenbanken empfohlen.

1. Sekundenbruchteile Wenn Sekundenbruchteile in Datums- und Zeitdatentypen gespeichert

sind, speichert Firebird sie auf Zehntausendstelsekunden. Wenn eine niedrigere Granularität bevorzugt wird, kann der Bruchteil in Dialekt-3-Datenbanken von ODS 11 oder höher explizit als Tausendstel, Hundertstel oder Zehntelsekunden angegeben werden.

Einige nützliche Informationen über die Genauigkeit im Sekundenbereich:

Der Zeitteil von `TIME` oder `TIMESTAMP` ist ein 4-Byte-WORD, mit Platz für Dezimillisekunden-Genauigkeit und Zeitwerte werden als die Anzahl der seit Mitternacht verstrichenen Dezimillisekunden gespeichert. Die tatsächliche Genauigkeit von Werten, die in Zeit(stempel)-Funktionen und -Variablen gespeichert oder daraus gelesen werden, beträgt:

- `CURRENT_TIME` ist standardmäßig auf Sekunden genau und kann mit `CURRENT_TIME (0|1|2|3)` bis auf Millisekunden genau angegeben werden
- `CURRENT_TIMESTAMP` Millisekunden-Genauigkeit. Die Genauigkeit von Sekunden bis Millisekunden kann mit `CURRENT_TIMESTAMP (0|1|2|3)` specified angegeben werden
- Literal `'NOW'`: Millisekunden-Präzision
- Die Funktionen `DATEADD()` und `DATEDIFF()` unterstützen eine Genauigkeit von bis zu Millisekunden. Dezi-Millisekunden können angegeben werden, sie werden jedoch auf die nächste ganze Zahl gerundet, bevor eine Operation ausgeführt wird
- Die Funktion `EXTRACT()` gibt mit den Argumenten `SECOND` und `MILLISECOND` eine Genauigkeit von Dezi-Millisekunden zurück
- Für *TIME- und TIMESTAMP-Literale* akzeptiert Firebird gerne eine Genauigkeit von Dezi-Millisekunden, kürzt (nicht rundet) den Zeitteil jedoch auf die nächste kleinere oder gleiche Millisekunde. Versuchen Sie zum Beispiel `SELECT TIME '14:37:54.1249' FROM rdb$database`
- die Operatoren ‘+’ und ‘-’ arbeiten mit Dezimillisekunden-Genauigkeit, aber nur *innerhalb* des Ausdrucks. Sobald etwas gespeichert oder auch nur aus `RDB$DATABASE` ausgewählt wird, geht es auf Millisekunden genau zurück

Die Genauigkeit von Dezi-Millisekunden ist selten und wird derzeit nicht in Spalten oder Variablen gespeichert. Die beste Annahme aus all dem ist, dass, obwohl Firebird `TIME` und die `TIMESTAMP`-Zeitteilwerte als die Anzahl der seit Mitternacht verstrichenen Dezi-Millisekunden (10^{-4} Sekunden) speichert, die tatsächliche Genauigkeit variieren kann von Sekunden bis Millisekunden.

3.4.1. DATE

Der Datentyp `DATE` in Dialect 3 speichert nur das Datum ohne Uhrzeit. Der verfügbare Bereich zum Speichern von Daten reicht vom 01. Januar bis zum 31. Dezember 9999.

Dialekt 1 hat keinen Typ “Nur Datum”.

In Dialekt 1 erhalten Datumsliterale ohne Zeitteil sowie `'TODAY'`, `'YESTERDAY'` und `'TOMORROW'` automatisch einen Null-Zeitteil.

> Wenn es Ihnen aus irgendeinem Grund wichtig ist, ein Dialekt-1-Zeitstempelliteral mit einem expliziten Zeitteil Null zu speichern, akzeptiert die Engine ein Literal wie `'2016-12-25 00:00:00.0000'`. Allerdings hätte `'2016-12-25'` genau den gleichen Effekt, mit weniger Tastenanschlägen!

3.4.2. TIME

Der Datentyp `TIME` ist nur in Dialekt 3 verfügbar. Es speichert die Tageszeit im Bereich von 00:00:00.0000 bis 23:59:59,9999.

Wenn Sie den Zeitteil von `DATE` in Dialekt 1 benötigen, können Sie die `EXTRACT`-Funktion verwenden.

Beispiele für die Verwendung von `EXTRACT()`

```
EXTRACT (HOUR FROM DATE_FIELD)
EXTRACT (MINUTE FROM DATE_FIELD)
EXTRACT (SECOND FROM DATE_FIELD)
```

Siehe auch `EXTRACT()`-Funktion im Kapitel *Eingebaute Funktionen*.

3.4.3. TIMESTAMP

Der Datentyp `TIMESTAMP` ist in Dialekt 3 und Dialekt 1 verfügbar. Es besteht aus zwei 32-Bit-Wörtern – einem Datumsteil und einem Zeitteil – um eine Struktur zu bilden, die sowohl Datum als auch Uhrzeit speichert. Es ist das gleiche wie der Typ `DATE` in Dialekt 1.

Die Funktion `EXTRACT` funktioniert mit `TIMESTAMP` genauso gut wie mit dem Dialekt 1 `DATE`-Typ.

3.4.4. Operationen mit Datums- und Uhrzeitwerten

Die Methode der Speicherung von Datums- und Uhrzeitwerten ermöglicht es, diese als Operanden in einige arithmetische Operationen einzubeziehen. Im Speicher wird ein Datumswert oder ein Datumsteil eines Zeitstempels als die Anzahl von Tagen dargestellt, die seit “Datum Null” - 17. November 1898 - verstrichen sind, während ein Zeitwert oder der Zeitteil eines Zeitstempels dargestellt wird als Anzahl der Sekunden (mit Berücksichtigung von Sekundenbruchteilen) seit Mitternacht.

Ein Beispiel ist das Subtrahieren eines früheren Datums, einer früheren Zeit oder eines Zeitstempels von einem späteren, was zu einem Zeitintervall in Tagen und Bruchteilen von Tagen führt.

Tabelle 5. Arithmetische Operationen für Datums- und Uhrzeitdatentypen

Operand 1	Operation	Operand 2	Ergebnis
`DATE`	+	`TIME`	`TIMESTAMP`
`DATE`	+	Numerischer Wert `n`	`DATE` um *n* ganze Tage erhöht. Gebrochene Werte werden auf die nächste Ganzzahl gerundet (nicht abgeschnitten).

Operand 1	Operation	Operand 2	Ergebnis
TIME	+	DATE	TIMESTAMP
TIME	+	Numerischer Wert n	TIME um *n* Sekunden erhöht. Bruchteile werden berücksichtigt.
TIMESTAMP	+	Numerischer Wert n	TIMESTAMP, wobei das Datum um die Anzahl der Tage und der Teil eines Tages durch die Zahl *n* repräsentiert wird — somit wird "+ 2.75" das Datum um 2 Tage und 18 Stunden weiterstellen wird
DATE	-	DATE	Anzahl der vergangenen Tage innerhalb des Bereichs DECIMAL(9, 0)
DATE	-	Numerischer Wert n	DATE um *n* ganze Tage reduziert. Gebrochene Werte werden auf die nächste Ganzzahl gerundet (nicht abgeschnitten).
TIME	-	TIME	Anzahl der vergangenen Sekunden, innerhalb des Bereichs DECIMAL(9, 4)
TIME	-	Numerischer Wert n	TIME um *n* Sekunden reduziert. Bruchteile werden berücksichtigt.
TIMESTAMP	-	TIMESTAMP	Anzahl der Tage und der Tageszeit, innerhalb des Bereichs DECIMAL(18, 9)
TIMESTAMP	-	Numerischer Wert n	TIMESTAMP wobei das Datum sich auf der Anzahl der Tage und der Tageszeit beruht, die durch die Zahl *n* repräsentiert wird — somit wird "-2.25" das Datum um 2 Tage und 6 Stunden reduzieren.

Hinweise

Der Typ DATE wird in Dialekt 1 als TIMESTAMP betrachtet.

Siehe auch

DATEADD, DATEDIFF

3.5. Zeichendatentypen

Für die Arbeit mit Zeichendaten hat Firebird die Datentypen CHAR mit fester Länge und VARCHAR mit variabler Länge. Die maximale Größe der in diesen Datentypen gespeicherten Textdaten beträgt 32.767 Byte für 'CHAR' und 32.765 Byte für 'VARCHAR'. Die maximale Anzahl von *Zeichen*, die in diese Grenzen passt, hängt davon ab, welches CHARACTER SET für die betrachteten Daten verwendet wird. Die Sortierreihenfolge hat keinen Einfluss auf dieses Maximum, kann sich jedoch auf die maximale Größe eines Index auswirken, der die Spalte umfasst.

Wenn beim Definieren eines Zeichenobjekts kein Zeichensatz explizit angegeben wird, wird der beim Erstellen der Datenbank angegebene Standardzeichensatz verwendet. Wenn in der Datenbank kein Standardzeichensatz definiert ist, erhält das Feld den Zeichensatz NONE.

3.5.1. Unicode

Die meisten aktuellen Entwicklungstools unterstützen Unicode, implementiert in Firebird mit den Zeichensätzen UTF8 und UNICODE_FSS. UTF8 enthält Kollationen für viele Sprachen. UNICODE_FSS ist eingeschränkter und wird hauptsächlich von Firebird intern zum Speichern von Metadaten verwendet. Beachten Sie, dass ein UTF8-Zeichen bis zu 4 Byte belegt, wodurch die Größe von CHAR -Feldern auf 8.191 Zeichen (32.767/4) begrenzt ist.

Der tatsächliche Wert von “Bytes pro Zeichen” hängt vom Bereich ab, zu dem das Zeichen gehört. Lateinische Buchstaben ohne Akzent belegen 1 Byte, kyrillische Buchstaben der Codierung WIN1251 belegen 2 Byte in UTF8, Zeichen anderer Codierungen können bis zu 4 Byte belegen.

Der in Firebird implementierte UTF8-Zeichensatz unterstützt die neueste Version des Unicode-Standards und empfiehlt daher seine Verwendung für internationale Datenbanken.

3.5.2. Client-Zeichensatz

Bei der Arbeit mit Strings ist es wichtig, den Zeichensatz der Client-Verbindung im Auge zu behalten. Wenn die Zeichensätze der gespeicherten Daten nicht mit denen der Client-Verbindung übereinstimmen, werden die Ausgabeergebnisse für String-Spalten automatisch neu codiert, sowohl beim Senden der Daten vom Client an den Server als auch beim Zurücksenden von der Server an den Client. Wenn die Datenbank beispielsweise in der Codierung WIN1251 erstellt wurde, aber KOI8R oder UTF8 in den Verbindungsparametern des Clients angegeben ist, ist die Abweichung transparent.

3.5.3. Sonderzeichensätze

Zeichensatz NONE

Der Zeichensatz NONE ist ein *Sonderzeichensatz* in Firebird. Es kann so charakterisiert werden, dass jedes Byte Teil einer Zeichenkette ist, die Zeichenkette jedoch im System ohne Hinweise darauf gespeichert wird, was ein Zeichen darstellt: Zeichencodierung, Sortierung, Groß-/Kleinschreibung usw. sind einfach unbekannt. Es liegt in der Verantwortung der Clientanwendung, mit den Daten umzugehen und die Mittel bereitzustellen, um die Bytefolge auf eine für die Anwendung und den menschlichen Benutzer sinnvolle Weise zu interpretieren.

Zeichensatz OCTETS

Daten in der OCTETS-Kodierung werden als Bytes behandelt, die möglicherweise nicht wirklich als Zeichen interpretiert werden. OCTETS bietet eine Möglichkeit, Binärdaten zu speichern, die das Ergebnis einiger Firebird-Funktionen sein können. Die Datenbank-Engine hat keine Vorstellung davon, was sie mit einer Bitfolge in OCTETS tun soll, außer sie nur zu speichern und abzurufen. Auch hier ist die Clientseite dafür verantwortlich, die Daten zu validieren, sie in für die Anwendung und ihre Benutzer sinnvollen Formaten darzustellen und alle Ausnahmen zu behandeln, die sich aus der Decodierung und Codierung ergeben.

3.5.4. Sortierreihenfolge

Jeder Zeichensatz hat eine Standardkollatierungssequenz (COLLATE), die die Sortierreihenfolge angibt. Normalerweise ist dies nichts anderes als eine Sortierung basierend auf dem numerischen Code der Zeichen und eine grundlegende Zuordnung von Groß- und Kleinbuchstaben. Wenn für Strings ein Verhalten erforderlich ist, das nicht von der Standardsortierreihenfolge bereitgestellt wird, und eine geeignete alternative Kallation für diesen Zeichensatz unterstützt wird, kann eine COLLATE collation-Klausel in der Spaltendefinition angegeben werden.

Eine COLLATE collation-Klausel kann neben der Spaltendefinition auch in anderen Kontexten angewendet werden. Für Größer-als/Kleiner-Vergleichsoperationen kann es in der WHERE-Klausel einer SELECT-Anweisung hinzugefügt werden. Wenn die Ausgabe in einer speziellen alphabetischen Reihenfolge oder ohne Beachtung der Groß-/Kleinschreibung sortiert werden muss und die entsprechende Sortierung vorhanden ist, kann eine COLLATE-Klausel in die ORDER BY-Klausel eingefügt werden, wenn Zeilen nach einem Zeichenfeld sortiert werden und mit die GROUP BY -Klausel bei Gruppierungsoperationen.

Suche ohne Berücksichtigung der Groß-/Kleinschreibung

Für eine Suche ohne Beachtung der Groß-/Kleinschreibung könnte die Funktion UPPER verwendet werden, um sowohl das Suchargument als auch die gesuchten Zeichenfolgen in Großbuchstaben umzuwandeln, bevor eine Übereinstimmung versucht wird:

```
...
where upper(name) = upper(:flt_name)
```

Bei Zeichenfolgen in einem Zeichensatz, der eine Sortierung ohne Beachtung der Groß-/Kleinschreibung zur Verfügung hat, können Sie einfach die Sortierung anwenden, um das Suchargument und die gesuchten Zeichenfolgen direkt zu vergleichen. Wenn Sie beispielsweise den Zeichensatz WIN1251 verwenden, ist die Sortierung PXW_CYRL zu diesem Zweck unabhängig von der Groß-/Kleinschreibung:

```
...
WHERE FIRST_NAME COLLATE PXW_CYRL >= :FLT_NAME
...
ORDER BY NAME COLLATE PXW_CYRL
```

Siehe auch
CONTAINING

UTF8-Sortierreihenfolgen

Die folgende Tabelle zeigt die möglichen Sortierfolgen für den Zeichensatz UTF8.

Tabelle 6. Sortierfolgen für Zeichensatz UTF8

Kollation	Eigenschaften
UCS_BASIC	Die Sortierung funktioniert nach der Position des Zeichens in der Tabelle (binär). In Firebird 2.0 hinzugefügt
UNICODE	Die Sortierung funktioniert nach dem UCA-Algorithmus (Unicode Collation Algorithm) (alphabetisch). In Firebird 2.0 hinzugefügt
UTF8	Die standardmäßige, binäre Sortierung, identisch mit UCS_BASIC, die aus Gründen der SQL-Kompatibilität hinzugefügt wurde
UNICODE_CI	Sortierung ohne Berücksichtigung der Groß-/Kleinschreibung, funktioniert ohne Berücksichtigung der Groß-/Kleinschreibung. Hinzugefügt in Firebird 2.1
UNICODE_CI_AI	Groß-/Kleinschreibung, akzentunabhängige Sortierung, arbeitet alphabetisch ohne Berücksichtigung von Groß-/Kleinschreibung oder Akzenten. Hinzugefügt in Firebird 2.5

Beispiel

Ein Beispiel für die Sortierung für den UTF8-Zeichensatz ohne Berücksichtigung der Groß-/Kleinschreibung oder der Akzentuierung von Zeichen (ähnlich wie COLLATE PXW_CYRL).

```
...
ORDER BY NAME COLLATE UNICODE_CI_AI
```

3.5.5. Zeichenindizes

In Firebird vor Version 2.0 kann ein Problem beim Erstellen eines Indexes für Zeichenspalten auftreten, die eine nicht standardmäßige Kollatierungssequenz verwenden: Die Länge eines indizierten Felds ist auf 252 Byte begrenzt, wenn COLLATE nicht angegeben ist, oder 84 Byte, wenn `COLLATE ` ist angegeben. Multi-Byte-Zeichensätze und zusammengesetzte Indizes begrenzen die Größe noch weiter.

Ab Firebird 2.0 beträgt die maximale Länge für einen Index ein Viertel der Seitengröße, d.h. von 1.024 — für Seitengröße 4.096 — bis 4.096 Bytes — für Seitengröße 16.384. Die maximale Länge einer indizierten Zeichenfolge beträgt 9 Byte weniger als diese Viertelseitenbegrenzung.

Berechnen der maximalen Länge eines indizierten Zeichenfolgenfelds

Die folgende Formel berechnet die maximale Länge einer indizierten Zeichenfolge (in Zeichen):

```
max_char_length = FLOOR((page_size / 4 - 9) / N)
```

wobei N die Anzahl der Bytes pro Zeichen im Zeichensatz ist.

Die folgende Tabelle zeigt die maximale Länge einer indizierten Zeichenfolge (in Zeichen), je nach Seitengröße und Zeichensatz, berechnet mit dieser Formel.

Tabelle 7. Maximale Indexlängen nach Seitengröße und Zeichengröße

Seitengröße	Bytes je Zeichen				
	1	2	3	4	6
4.096	1.015	507	338	253	169
8.192	2.039	1.019	679	509	339
16.384	4.087	2.043	1.362	1.021	682

Bei Sortierungen, bei denen die Groß-/Kleinschreibung nicht beachtet wird ("_CI"), belegt ein Zeichen im *index* nicht 4, sondern 6 (sechs) Bytes, sodass die maximale Schlüssellänge für eine Seite von z 169 Zeichen.

Siehe auch

`CREATE DATABASE`, Sortierreihenfolge, `SELECT`, `WHERE`, `GROUP BY`, `ORDER BY`

3.5.6. Zeichenarten im Detail

CHAR

`CHAR` ist ein Datentyp mit fester Länge. Wenn die eingegebene Anzahl von Zeichen kleiner als die angegebene Länge ist, werden dem Feld abschließende Leerzeichen hinzugefügt. Im Allgemeinen muss das Auffüllzeichen kein Leerzeichen sein: Es hängt vom Zeichensatz ab. Das Füllzeichen für den Zeichensatz `OCTETS` ist beispielsweise null.

Der vollständige Name dieses Datentyps ist `CHARACTER`, aber es ist nicht erforderlich, vollständige Namen zu verwenden, und die Leute tun dies selten.

Zeichendaten mit fester Länge können verwendet werden, um Codes zu speichern, deren Länge Standard ist und eine bestimmte "Breite" in Verzeichnissen hat. Ein Beispiel für einen solchen Code ist ein EAN13-Barcode – 13 Zeichen, alle ausgefüllt.

Deklarationssyntax

```
{CHAR | CHARACTER} [(length)]
  [CHARACTER SET <set>] [COLLATE <name>]
```

Wenn keine Länge *length* angegeben ist, wird sie mit 1 angenommen.

Eine gültige Länge *length* reicht von 1 bis zur maximalen Anzahl von Zeichen, die innerhalb von 32.767 Bytes untergebracht werden können.

Formal ist die `COLLATE`-Klausel nicht Teil der Datentyp-Deklaration und ihre Position hängt von der Syntax der Anweisung ab.

VARCHAR

`VARCHAR` ist der grundlegende Stringtyp zum Speichern von Texten variabler Länge, bis maximal 32.765 Byte. Die gespeicherte Struktur entspricht der tatsächlichen Größe der Daten plus 2 Byte, wobei die Länge der Daten aufgezeichnet wird.

Alle Zeichen, die von der Clientanwendung an die Datenbank gesendet werden, werden als aussagekräftig angesehen, einschließlich der führenden und abschließenden Leerzeichen. Nachgestellte Leerzeichen werden jedoch nicht gespeichert: Sie werden beim Abrufen bis zur aufgezeichneten Länge der Zeichenfolge wiederhergestellt.

Der vollständige Name dieses Typs ist `CHARACTER VARYING`. Eine andere Variante des Namens wird als `CHAR VARYING` geschrieben.

Syntax

```
{VARCHAR | {CHAR | CHARACTER} VARYING} (length)
  [CHARACTER SET <set>] [COLLATE <name>]
```

Formal ist die `COLLATE`-Klausel nicht Teil der Datentyp-Deklaration und ihre Position hängt von der Syntax der Anweisung ab.

NCHAR

`NCHAR` ist ein Zeichendatentyp fester Länge mit dem vordefinierten Zeichensatz `ISO8859_1`. Ansonsten ist es dasselbe wie `CHAR`.

Syntax

```
{NCHAR | NATIONAL {CHAR | CHARACTER}} [(length)]
```

Wenn keine Länge *length* angegeben ist, wird sie mit 1 angenommen.

Ein ähnlicher Datentyp ist für den String-Typ variabler Länge verfügbar: `NATIONAL {CHAR | CHARAKTER} VERSCHIEDLICH`.

3.6. Boolean-Datentyp

Firebird 3.0 führte einen vollwertigen booleschen Datentyp ein.

3.6.1. BOOLEAN

Der SQL:2008-konforme Datentyp `BOOLEAN` (8 Bit) umfasst die unterschiedlichen Wahrheitswerte `TRUE` und `FALSE`. Sofern nicht durch eine `NOT NULL`-Beschränkung verboten, unterstützt der `BOOLEAN`-Datentyp auch den Wahrheitswert `UNKNOWN` als Nullwert. Die Spezifikation macht keinen Unterschied zwischen dem `NULL`-Wert dieses Datentyps und dem Wahrheitswert `UNKNOWN`, der das Ergebnis eines SQL-Prädikats, einer Suchbedingung oder eines booleschen Wertausdrucks ist: Sie sind austauschbar und bedeuten das gleiche.

Wie bei vielen Programmiersprachen können die `BOOLEAN`-Werte von SQL mit impliziten Wahrheitswerten getestet werden. Beispielsweise sind `field1 OR field2` und `NOT field1` gültige Ausdrücke.

Der IS-Operator

Prädikate können den Operator Boolean `IS [NOT]` zum Abgleich verwenden. Zum Beispiel `field1 IS FALSE` oder `field1 IS NOT TRUE`.

- Äquivalenzoperatoren (“=”, “!=”, “<>” und so weiter) sind in allen Vergleichen gültig.

`BOOLEAN`-Beispiele

1. Einfügen und abfragen

```
CREATE TABLE TBOOL (ID INT, BVAL BOOLEAN);
COMMIT;

INSERT INTO TBOOL VALUES (1, TRUE);
INSERT INTO TBOOL VALUES (2, 2 = 4);
INSERT INTO TBOOL VALUES (3, NULL = 1);
COMMIT;

SELECT * FROM TBOOL;
          ID    BVAL
============ =======
           1 <true>
           2 <false>
           3 <null>
```

2. Test auf Wert `TRUE`

```
SELECT * FROM TBOOL WHERE BVAL;
          ID    BVAL
============ =======
           1 <true>
```

3. Test auf Wert `FALSE`

```
SELECT * FROM TBOOL WHERE BVAL IS FALSE;
          ID    BVAL
============ =======
           2 <false>
```

4. Test auf Wert `UNKNOWN`

```
SELECT * FROM TBOOL WHERE BVAL IS UNKNOWN;
          ID    BVAL
============ =======
```

```
           3 <null>
```

5. Boolean-Werte in SELECT-Anweisung

```
SELECT ID, BVAL, BVAL AND ID < 2
  FROM TBOOL;
          ID    BVAL
============ ======= =======
           1 <true>  <true>
           2 <false> <false>
           3 <null>  <false>
```

6. PSQL-Deklaration mit Startwert

```
DECLARE VARIABLE VAR1 BOOLEAN = TRUE;
```

7. Gültige Syntax, aber wie bei einem Vergleich mit NULL, wird nie ein Datensatz zurückgegeben

```
SELECT * FROM TBOOL WHERE BVAL = UNKNOWN;
SELECT * FROM TBOOL WHERE BVAL <> UNKNOWN;
```

Verwendung von Boolean gegen andere Datentypen

Obwohl BOOLEAN von Natur aus in keinen anderen Datentyp konvertierbar ist, werden ab Version 3.0.1 die Strings 'true' und 'false' (Groß-/Kleinschreibung nicht beachtet) in Wertausdrücken implizit in BOOLEAN umgewandelt, z.B.

```
if (true > 'false') then ...
```

'false' wird in BOOLEAN umgewandelt. Jeder Versuch, die booleschen Operatoren AND, NOT, OR und IS zu verwenden, schlägt fehl. NOT 'False' ist beispielsweise ungültig.

Ein BOOLEAN kann mit CAST explizit in und aus einem String umgewandelt werden. UNKNOWN ist für keine Form des Castings verfügbar.

Weitere Hinweise

- Der Typ wird in der API mit dem Typ FB_BOOLEAN und den Konstanten FB_TRUE und FB_FALSE dargestellt.
- Der Wert TRUE ist größer als der Wert FALSE.

3.7. Binärdatentypen

BLOBs (Binary Large Objects) sind komplexe Strukturen, die verwendet werden, um Text und binäre Daten undefinierter Länge, oft sehr groß, zu speichern.

Syntax

```
BLOB [SUB_TYPE <subtype>]
  [SEGMENT SIZE <segment size>]
  [CHARACTER SET <character set>]
  [COLLATE <collation name>]
```

Verkürzte Syntax

```
BLOB (<segment size>)
BLOB (<segment size>, <subtype>)
BLOB (, <subtype>)
```

Formal ist die `COLLATE`-Klausel nicht Teil der Datentyp-Deklaration und ihre Position hängt von der Syntax der Anweisung ab.

Segmentgröße

Die Angabe der BLOB-Segmentgröße ist ein Rückfall in vergangene Zeiten, als Anwendungen zum Arbeiten mit BLOB-Daten in C (Embedded SQL) mit Hilfe des Pre-Compilers *gpre* geschrieben wurden. Heutzutage ist es praktisch irrelevant. Die Segmentgröße für BLOB-Daten wird von der Clientseite bestimmt und ist in der Regel auf jeden Fall größer als die Datenseitengröße.

3.7.1. `BLOB`-Untertypen

Der optionale Parameter `SUB_TYPE` gibt die Art der in die Spalte geschriebenen Daten an. Firebird bietet zwei vordefinierte Untertypen zum Speichern von Benutzerdaten:

Subtyp 0: `BINARY`

Wenn kein Subtyp angegeben wird, wird angenommen, dass die Spezifikation für nicht typisierte Daten gilt, und der Standardwert `SUB_TYPE 0` wird angewendet. Der Alias für den Subtyp null ist `BINARY`. Dies ist der Untertyp, um anzugeben, ob es sich bei den Daten um eine Binärdatei oder einen Stream handelt: Bilder, Audio, Textverarbeitungsdateien, PDFs usw.

Untertyp 1: `TEXT`

Subtyp 1 hat einen Alias, `TEXT`, der in Deklarationen und Definitionen verwendet werden kann. Zum Beispiel `BLOB SUB_TYPE TEXT`. Es ist ein spezialisierter Untertyp, der verwendet wird, um Nur-Text-Daten zu speichern, die zu groß sind, um in einen String-Typ zu passen. Ein `CHARACTER SET` kann angegeben werden, wenn das Feld Text mit einer anderen Kodierung als der für die Datenbank angegebenen speichern soll. Ab Firebird 2.0 wird auch eine `COLLATE`-Klausel unterstützt.

Die Angabe eines `CHARACTER SET` ohne `SUB_TYPE` impliziert `SUB_TYPE TEXT`.

Benutzerdefinierte Untertypen

Es ist auch möglich, benutzerdefinierte Datenuntertypen hinzuzufügen, für die der Aufzählungsbereich von -1 bis -32.768 reserviert ist. Benutzerdefinierte Subtypen, die mit positiven Zahlen aufgezählt werden, sind nicht zulässig, da die Firebird-Engine die Zahlen ab 2 aufwärts für einige interne Subtypen in Metadaten verwendet.

3.7.2. BLOB-Besonderheiten

Größe

Die maximale Größe eines 'BLOB'-Feldes ist auf 4 GB begrenzt, unabhängig davon, ob der Server 32-Bit oder 64-Bit ist. (Die internen Strukturen, die sich auf BLOBs beziehen, unterhalten ihre eigenen 4-Byte-Zähler.) Bei einer Seitengröße von 4 KB (4096 Byte) ist die maximale Größe geringer – etwas weniger als 2 GB.

Operationen und Ausdrücke

Text-BLOBs beliebiger Länge und beliebiger Zeichensätze – auch Multibyte – können Operanden für praktisch jede Anweisung oder interne Funktion sein. Die folgenden Operatoren werden vollständig unterstützt:

=	(Zuordnung)
=, <>, <, ⇐, >, >=	(Vergeleich)
||	(Verkettung)
BETWEEN,	IS [NOT] DISTINCT FROM,
IN,	ANY | SOME,
ALL	

Teilunterstützung:

- Bei diesen tritt ein Fehler auf, wenn das Suchargument größer oder gleich 32 KB ist:

  STARTING [WITH], LIKE,
  CONTAINING

- Aggregationsklauseln wirken sich nicht auf den Inhalt des Feldes selbst aus, sondern auf die BLOB-ID. Abgesehen davon gibt es einige Macken:

SELECT DISTINCT	gibt fälschlicherweise mehrere NULL-Werte zurück, wenn sie vorhanden sind
ORDER BY	—
GROUP BY	verkettet dieselben Zeichenfolgen, wenn sie nebeneinander liegen, tut dies jedoch nicht, wenn sie voneinander entfernt sind

BLOB-*Speicher*

- Standardmäßig wird für jedes BLOB ein regulärer Datensatz erstellt und auf einer ihm zugeordneten Datenseite gespeichert. Passt das gesamte BLOB auf diese Seite, wird es als *level 0 BLOB* bezeichnet. Die Nummer dieses Sondersatzes wird im Tabellensatz gespeichert und belegt

8 Byte.

- Wenn ein BLOB nicht auf eine Datenseite passt, wird sein Inhalt auf separate, ihm exklusiv zugeordnete Seiten (Blob-Seiten) gelegt, während die Nummern dieser Seiten im BLOB-Record gespeichert werden. Dies ist ein *Level 1 BLOB*.
- Wenn das Array von Seitennummern, das die BLOB-Daten enthält, nicht auf eine Datenseite passt, wird das Array auf separate Blob-Seiten gelegt, während die Nummern dieser Seiten in den BLOB-Datensatz geschrieben werden. Dies ist ein *Level-2-BLOB*.
- Level höher als 2 werden nicht unterstützt.

Siehe auch

FILTER, DECLARE FILTER

3.7.3. ARRAY-Datentyp

Die Unterstützung von Arrays im Firebird DBMS ist eine Abkehr vom traditionellen relationalen Modell. Die Unterstützung von Arrays im DBMS könnte die Lösung einiger Datenverarbeitungsaufgaben mit großen Mengen ähnlicher Daten erleichtern.

Arrays in Firebird werden in BLOB eines spezialisierten Typs gespeichert. Arrays können eindimensional und mehrdimensional sein und jeden Datentyp außer BLOB und ARRAY haben.

Beispiel

```
CREATE TABLE SAMPLE_ARR (
  ID INTEGER NOT NULL PRIMARY KEY,
  ARR_INT INTEGER [4]
);
```

In diesem Beispiel wird eine Tabelle mit einem Feld vom Typ Array erstellt, das aus vier ganzen Zahlen besteht. Die Indizes dieses Arrays sind von 1 bis 4.

Angeben von expliziten Grenzen für Bemaßungen

Standardmäßig sind Dimensionen 1-basiert – tiefgestellte Indizes werden ab 1 nummeriert. Verwenden Sie die folgende Syntax, um explizite Ober- und Untergrenzen der tiefgestellten Werte anzugeben:

```
'[' <lower>:<upper> ']'
```

Hinzufügen weiterer Dimensionen

Eine neue Dimension wird mit einem Komma in der Syntax hinzugefügt. In diesem Beispiel erstellen wir eine Tabelle mit einem zweidimensionalen Array, wobei die Untergrenze der Indizes in beiden Dimensionen bei Null beginnt:

```
CREATE TABLE SAMPLE_ARR2 (
```

```
  ID INTEGER NOT NULL PRIMARY KEY,
  ARR_INT INTEGER [0:3, 0:3]
);
```

Das DBMS bietet nicht viel an Sprache oder Werkzeugen, um mit dem Inhalt von Arrays zu arbeiten. Die Datenbank `employee.fdb`, die sich im Verzeichnis `../examples/empbuild` eines Firebird-Distributionspakets befindet, enthält eine gespeicherte Beispielprozedur, die einige einfache Arbeiten mit Arrays zeigt:

PSQL-Quelle für `SHOW_LANGS`, eine Prozedur mit einem Array

```
CREATE OR ALTER PROCEDURE SHOW_LANGS (
  CODE VARCHAR(5),
  GRADE SMALLINT,
  CTY VARCHAR(15))
RETURNS (LANGUAGES VARCHAR(15))
AS
  DECLARE VARIABLE I INTEGER;
BEGIN
  I = 1;
  WHILE (I <= 5) DO
  BEGIN
    SELECT LANGUAGE_REQ[:I]
    FROM JOB
    WHERE (JOB_CODE = :CODE)
      AND (JOB_GRADE = :GRADE)
      AND (JOB_COUNTRY = :CTY)
      AND (LANGUAGE_REQ IS NOT NULL))
    INTO :LANGUAGES;

    IF (LANGUAGES = '') THEN
    /* 'NULL' ANSTELLE VON LEERZEICHEN AUSGEBEN */
      LANGUAGES = 'NULL';
    I = I +1;
    SUSPEND;
  END
END
```

Wenn die beschriebenen Funktionen für Ihre Aufgaben ausreichen, können Sie in Ihren Projekten Arrays verwenden. Derzeit sind keine Verbesserungen geplant, um die Unterstützung für Arrays in Firebird zu verbessern.

3.8. Spezielle Datentypen

“Spezielle” Datentypen …

3.8.1. SQL_NULL-Datentypen

Der Typ SQL_NULL enthält keine Daten, sondern nur einen Zustand: NULL oder NOT NULL. Als Datentyp zum Deklarieren von Tabellenfeldern, PSQL-Variablen oder Parameterbeschreibungen steht er nicht zur Verfügung. Es wurde hinzugefügt, um die Verwendung nicht typisierter Parameter in Ausdrücken zu unterstützen, die das Prädikat IS NULL beinhalten.

Ein Auswertungsproblem tritt auf, wenn optionale Filter verwendet werden, um Abfragen des folgenden Typs zu schreiben:

```
WHERE col1 = :param1 OR :param1 IS NULL
```

Nach der Verarbeitung auf API-Ebene sieht die Abfrage wie folgt aus:

```
WHERE col1 = ? OR ? IS NULL
```

Dies ist ein Fall, in dem der Entwickler eine SQL-Abfrage schreibt und :param1 als eine *Variable* betrachtet, auf die er zweimal verweisen kann. Auf API-Ebene enthält die Abfrage jedoch zwei separate und unabhängige *_Parameter*. Der Server kann den Typ des zweiten Parameters nicht bestimmen, da er mit IS NULL verknüpft ist.

Der Datentyp SQL_NULL löst dieses Problem. Immer wenn die Engine in einer Abfrage auf ein Prädikat "? IS NULL" stößt, weist sie dem Parameter den Typ SQL_NULL zu, was anzeigt, dass es sich bei dem Parameter nur um "Nulligkeit" und den Datentyp handelt oder der Wert muss nicht angesprochen werden.

Das folgende Beispiel zeigt die Anwendung in der Praxis. Es nimmt zwei benannte Parameter an — sagen wir :size und :colour — die zum Beispiel Werte aus Bildschirmtextfeldern oder Dropdown-Listen erhalten können. Jeder benannte Parameter entspricht zwei Positionsparametern in der Abfrage.

```
SELECT
  SH.SIZE, SH.COLOUR, SH.PRICE
FROM SHIRTS SH
WHERE (SH.SIZE = ? OR ? IS NULL)
  AND (SH.COLOUR = ? OR ? IS NULL)
```

Um zu erklären, was hier passiert, wird davon ausgegangen, dass der Leser mit der Firebird-API und der Übergabe von Parametern in XSQLVAR-Strukturen vertraut ist — was unter der Oberfläche passiert, ist für diejenigen nicht von Interesse, die keine Treiber oder Anwendungen schreiben, die mit der "nakten" API kommunizieren.

Die Anwendung übergibt die parametrisierte Anfrage an den Server in der üblichen positionellen ? -Form. Paare von "identischen" Parametern können nicht zu einem zusammengeführt werden, daher werden beispielsweise für zwei optionale Filter vier Positionsparameter benötigt: einer für jedes ? in unserem Beispiel.

Nach dem Aufruf von `isc_dsql_describe_bind()` wird der SQLTYPE des zweiten und vierten Parameters auf `SQL_NULL` gesetzt. Firebird hat keine Kenntnis von ihrer speziellen Beziehung zum ersten und dritten Parameter: Diese Verantwortung liegt vollständig auf der Anwendungsseite.

Nachdem die Werte für Größe und Farbe vom Benutzer festgelegt (oder nicht festgelegt) wurden und die Abfrage ausgeführt werden soll, muss jedes Paar von `XSQLVAR`s wie folgt gefüllt werden:

Der Benutzer hat einen Wert angegeben

Erster Parameter (Wertvergleich): setze `*sqldata` auf den angegebenen Wert und `*sqlind` auf `0` (für `NOT NULL`)

Zweiter Parameter (`NULL` Test): setze `sqldata` auf `null` (Nullzeiger, nicht SQL `NULL`) und `*sqlind` auf `0` (für `NOT NULL`)

Der Benutzer hat das Feld leer gelassen

Beide Parameter: setze `sqldata` auf `null` (Nullzeiger, nicht SQL `NULL`) und `*sqlind` auf `-1` (zeigt `NULL`)

Mit anderen Worten: Der Parameter Wertvergleich wird immer wie gewohnt gesetzt. Der Parameter `SQL_NULL` wird gleich gesetzt, außer dass `sqldata` immer `null` bleibt.

3.9. Konvertierung von Datentypen

Beim Verfassen eines Ausdrucks oder der Angabe einer Operation sollte das Ziel sein, kompatible Datentypen für die Operanden zu verwenden. Wenn eine Mischung von Datentypen verwendet werden muss, sollten Sie nach einer Möglichkeit suchen, inkompatible Operanden zu konvertieren, bevor Sie sie der Operation unterziehen. Die Möglichkeit, Daten zu konvertieren, kann durchaus ein Problem darstellen, wenn Sie mit Dialekt-1-Daten arbeiten.

3.9.1. Explizite Datentypkonvertierung

Die `CAST`-Funktion ermöglicht die explizite Konvertierung zwischen vielen Paaren von Datentypen.

Syntax

```
CAST (<expression> AS <target_type>)

<target_type> ::= <domain_or_non_array_type> | <array_datatype>

<domain_or_non_array_type> ::=
  !! Vgl. Syntax für Scalardatentypen !!

<array_datatype> ::=
  !! Vgl. Syntax für Array-Datentypen !!
```

Siehe auch `CAST()` im Abschnitt *Eingebaute Skalarfunktionen*.

Casting auf eine Domain

Beim Casting in eine Domäne werden alle dafür deklarierten Constraints berücksichtigt, d. h. `NOT NULL`- oder `CHECK`-Constraints. Wenn der *Wert* die Prüfung nicht besteht, schlägt die Umwandlung fehl.

Wenn zusätzlich `TYPE OF` angegeben wird — Umwandlung in seinen Basistyp — werden alle Domäneneinschränkungen während der Umwandlung ignoriert. Wird `TYPE OF` mit einem Zeichentyp (`CHAR`/`VARCHAR`) verwendet, bleiben Zeichensatz und Kollatierung erhalten.

Casting in `Spaltentyp`

Wenn Operanden in den Typ einer Spalte umgewandelt werden, kann die angegebene Spalte aus einer Tabelle oder einer Sicht stammen.

Es wird nur der Typ der Spalte selbst verwendet. Bei Zeichentypen enthält die Besetzung den Zeichensatz, aber nicht die Sortierung. Die Einschränkungen und Standardwerte der Quellspalte werden nicht angewendet.

Beispiel

```
CREATE TABLE TTT (
  S VARCHAR (40)
  CHARACTER SET UTF8 COLLATE UNICODE_CI_AI
);
COMMIT;

SELECT
  CAST ('I have many friends' AS TYPE OF COLUMN TTT.S)
FROM RDB$DATABASE;
```

Konvertierungen für die `CAST`-Funktion möglich

Tabelle 8. Umwandlungen mit CAST

Von Datentyp	Zu Datentyp
Numerische Typen	Numerische Typen, `[VAR]CHAR`, `BLOB`
`[VAR]CHAR`	`[VAR]CHAR`, `BLOB`, Numerische Typen, `DATE`, `TIME`, `TIMESTAMP`, `BOOLEAN`
`BLOB`	`[VAR]CHAR`, `BLOB`, Numerische Typen, `DATE`, `TIME`, `TIMESTAMP`, `BOOLEAN`
`DATE`, `TIME`	`[VAR]CHAR`, `BLOB`, `TIMESTAMP`
`TIMESTAMP`	`[VAR]CHAR`, `BLOB`, `DATE`, `TIME`
`BOOLEAN`	`BOOLEAN`, `[VAR]CHAR`, `BLOB`

Um String-Datentypen in den Typ `BOOLEAN` zu konvertieren, muss der Wert (ohne Berücksichtigung der Groß-/Kleinschreibung) `'true'` oder `'false'` oder `NULL` sein.

Beachten Sie, dass ein teilweiser Informationsverlust möglich ist. Wenn Sie beispielsweise den Datentyp `TIMESTAMP` in den Datentyp `DATE` umwandeln, geht der

> Zeitteil verloren.

Literale Formate

Um String-Datentypen in die Datentypen `DATE`, `TIME` oder `TIMESTAMP` umzuwandeln, muss das String-Argument eines der vordefinierten Datums- und Uhrzeitliterale sein (siehe Tabelle 9) oder eine Darstellung des Datums in einem der zulässigen *Datum-Uhrzeit-Literal*-Formate:

```
<timestamp_format> ::=
    { [YYYY<p>]MM<p>DD[<p>HH[<p>mm[<p>SS[<p>NNNN]]]]
    | MM<p>DD[<p>YYYY[<p>HH[<p>mm[<p>SS[<p>NNNN]]]]]
    | DD<p>MM[<p>YYYY[<p>HH[<p>mm[<p>SS[<p>NNNN]]]]]
    | MM<p>DD[<p>YY[<p>HH[<p>mm[<p>SS[<p>NNNN]]]]]
    | DD<p>MM[<p>YY[<p>HH[<p>mm[<p>SS[<p>NNNN]]]]]
    | NOW
    | TODAY
    | TOMORROW
    | YESTERDAY }

<date_format> ::=
    { [YYYY<p>]MM<p>DD
    | MM<p>DD[<p>YYYY]
    | DD<p>MM[<p>YYYY]
    | MM<p>DD[<p>YY]
    | DD<p>MM[<p>YY]
    | TODAY
    | TOMORROW
    | YESTERDAY }

<time_format> :=
    { HH[<p>mm[<p>SS[<p>NNNN]]]
    | NOW }

<p> ::= whitespace | . | : | , | - | /
```

Tabelle 9. Datums- und Uhrzeit-Literalformatargumente

Argument	Beschreibung
timestamp_format	Format des Zeitstempelliterals
date_literal	Format des Datumsliterals
time_literal	Format des Zeitliterals
YYYY	Vierstelliges Jahr
YY	Zweistelliges Jahr

Argument	Beschreibung
MM	Monat Kann 1 oder 2 Stellen enthalten (1-12 oder 01-12). Sie können auch den aus drei Buchstaben bestehenden Kurznamen oder den vollständigen Namen eines Monats in Englisch angeben. Groß-/Kleinschreibung nicht beachten
DD	Tag. Es kann 1 oder 2 Stellen enthalten (1-31 oder 01-31)
HH	Stunde. Es kann 1 oder 2 Stellen enthalten (0-23 oder 00-23)
mm	Minuten. Es kann 1 oder 2 Stellen enthalten (0-59 oder 00-59)
SS	Sekunden. Es kann 1 oder 2 Stellen enthalten (0-59 oder 00-59)
NNNN	Zehntausendstelsekunden. Es kann 1 bis 4 Stellen (0-9999) enthalten.
p	Ein Trennzeichen, eines der zulässigen Zeichen. Führende und nachgestellte Leerzeichen werden ignoriert

Tabelle 10. Literale mit vordefinierten Werten für Datum und Uhrzeit

Literal	Beschreibung	Datentyp	
		Dialekt 1	Dialekt 3
`'NOW'`	Aktuelles Datum und Zeit	`DATE`	`TIMESTAMP`
`'TODAY'`	Aktuelles Datum	`DATE` mit Nullzeit-Teil	`DATE`
`'TOMORROW'`	Aktualles Datum + 1 (Tag)	`DATE` mit Nullzeit-Teil	`DATE`
`'YESTERDAY'`	Aktualles Datum - 1 (Tag)	`DATE` mit Nullzeit-Teil	`DATE`

Die Verwendung der vollständigen Jahresangabe in vierstelliger Form — `YYYY` — wird dringend empfohlen, um Verwirrung bei Datumsberechnungen und Aggregationen zu vermeiden.

Beispielhafte Interpretationen der Datumsliterale

```
select
  cast('04.12.2014' as date) as d1, -- DD.MM.YYYY
  cast('04 12 2014' as date) as d2, -- MM DD YYYY
  cast('4-12-2014' as date) as d3,  -- MM-DD-YYYY
  cast('04/12/2014' as date) as d4, -- MM/DD/YYYY
  cast('04,12,2014' as date) as d5, -- MM,DD,YYYY
  cast('04.12.14' as date) as d6,   -- DD.MM.YY
  -- DD.MM mit aktuellem Jahr
  cast('04.12' as date) as d7,
  -- MM/DD mit aktuellem Jahr
  cast('04/12' as date) as d8,
```

```
  cast('2014/12/04' as date) as d9, -- YYYY/MM/DD
  cast('2014 12 04' as date) as d10, -- YYYY MM DD
  cast('2014.12.04' as date) as d11, -- YYYY.MM.DD
  cast('2014-12-04' as date) as d12, -- YYYY-MM-DD
  cast('4 Jan 2014' as date) as d13, -- DD MM YYYY
  cast('2014 Jan 4' as date) as dt14, -- YYYY MM DD
  cast('Jan 4, 2014' as date) as dt15, -- MM DD, YYYY
  cast('11:37' as time) as t1, -- HH:mm
  cast('11:37:12' as time) as t2, -- HH:mm:ss
  cast('11:31:12.1234' as time) as t3, -- HH:mm:ss.nnnn
  cast('11.37.12' as time) as t4, -- HH.mm.ss
  -- DD.MM.YYYY HH:mm
  cast('04.12.2014 11:37' as timestamp) as dt1,
  -- MM/DD/YYYY HH:mm:ss
  cast('04/12/2014 11:37:12' as timestamp) as dt2,
  -- DD.MM.YYYY HH:mm:ss.nnnn
  cast('04.12.2014 11:31:12.1234' as timestamp) as dt3,
  -- MM/DD/YYYY HH.mm.ss
  cast('04/12/2014 11.37.12' as timestamp) as dt4
from rdb$database
```

Kurzformumwandlungen für Datums- und Uhrzeitdatentypen

Firebird erlaubt die Verwendung einer abgekürzten Typsyntax im "C-Stil" für Umwandlungen von Strings in die Typen "DATE", "TIME" und "TIMESTAMP". Der SQL-Standard ruft diese Datetime-Literale auf.

Syntax

```
<data_type> 'date_literal_string'
```

Beispiel

```
-- 1
  UPDATE PEOPLE
  SET AGECAT = 'SENIOR'
  WHERE BIRTHDATE < DATE '1-Jan-1943';
-- 2
  INSERT INTO APPOINTMENTS
  (EMPLOYEE_ID, CLIENT_ID, APP_DATE, APP_TIME)
  VALUES (973, 8804, DATE 'today' + 2, TIME '16:00');
-- 3
  NEW.LASTMOD = TIMESTAMP 'now';
```

Diese Kurzausdrücke werden direkt beim Parsen ausgewertet, als ob die Anweisung bereits für die Ausführung vorbereitet wäre. Selbst wenn die Abfrage mehrmals ausgeführt wird, bleibt der Wert von beispielsweise `timestamp 'now'` gleich, egal wie viel Zeit vergeht.

Wenn die Zeit bei jeder Ausführung ausgewertet werden soll, verwenden Sie die vollständige CAST-Syntax. Ein Beispiel für die Verwendung eines solchen Ausdrucks in einem Trigger:

```
NEW.CHANGE_DATE = CAST('now' AS TIMESTAMP);
```

Firebird 4 lässt diese impliziten Datetime-Werte wie 'now'', ''today'' usw. in diesen Kurzformumsetzungen nicht mehr zu. Es ist ratsam, für implizite Werte auf die Verwendung des vollständigen `CAST-Ausdrucks umzusteigen.

3.9.2. Implizite Datentypkonvertierung

Eine implizite Datenkonvertierung ist in Dialekt 3 nicht möglich — die CAST-Funktion wird fast immer benötigt, um Datentypkonflikte zu vermeiden.

In Dialekt 1 wird in vielen Ausdrücken ein Typ implizit in einen anderen umgewandelt, ohne dass die CAST-Funktion verwendet werden muss. Zum Beispiel gilt die folgende Aussage in Dialekt 1:

```
UPDATE ATABLE
  SET ADATE = '25.12.2016' + 1
```

und das Datumsliteral wird implizit in den Datumstyp umgewandelt.

In Dialekt 3 wird diese Anweisung den Fehler 35544569 ausgeben, "`Dynamic SQL Error: expression evaluation not supported, Strings cannot be added or subtracted in dialect 3" — eine Umwandlung ist erforderlich:

```
UPDATE ATABLE
  SET ADATE = CAST ('25.12.2016' AS DATE) + 1
```

oder mit der kurzen Umwandlung:

```
UPDATE ATABLE
  SET ADATE = DATE '25.12.2016' + 1
```

In Dialekt 1 ist es normalerweise möglich, ganzzahlige Daten und numerische Zeichenfolgen zu mischen, da der Parser versucht, die Zeichenfolge implizit umzuwandeln. Beispielsweise,

```
2 + '1'
```

wird korrekt ausgeführt.

In Dialekt 3 führt ein solcher Ausdruck zu einem Fehler, daher müssen Sie ihn als CAST-Ausdruck schreiben:

```
2 + CAST('1' AS SMALLINT)
```

Die Ausnahme von der Regel ist während der *String-Verkettung*.

Implizite Konvertierung während der String-Verkettung

Wenn mehrere Datenelemente verkettet werden, werden alle Nicht-String-Daten nach Möglichkeit implizit in Strings umgewandelt.

Beispiel

```
SELECT 30||' days hath September, April, June and November' CONCAT$
  FROM RDB$DATABASE;

CONCAT$
==============================================
30 days hath September, April, June and November
```

3.10. Benutzerdefinierte Datentypen – Domains

In Firebird ist das Konzept eines “benutzerdefinierten Datentyps” in Form der *Domain* implementiert. Das Erstellen einer Domain erzeugt natürlich nicht wirklich einen neuen Datentyp. Eine Domain bietet die Möglichkeit, einen vorhandenen Datentyp mit einem Satz von Attributen zu kapseln und diese “Kapsel” für die mehrfache Verwendung in der gesamten Datenbank verfügbar zu machen. Wenn mehrere Tabellen Spalten mit identischen oder nahezu identischen Attributen benötigen, ist eine Domäne sinnvoll.

Die Domänenverwendung ist nicht auf Spaltendefinitionen für Tabellen und Ansichten beschränkt. Domänen können verwendet werden, um Eingabe- und Ausgabeparameter und Variablen in PSQL-Code zu deklarieren.

3.10.1. Domaineigenschaften

Eine Domaindefinition enthält erforderliche und optionale Attribute. Der *Datentyp* ist ein erforderliches Attribut. Zu den optionalen Attributen gehören:

- ein Standardwert
- um `NULL` zu erlauben oder zu verbieten
- `CHECK`-Einschränkungen
- Zeichensatz (für Zeichendatentypen und Text-BLOB-Felder)
- Sortierung (für Zeichendatentypen)

Beispieldomaindefinition

```
CREATE DOMAIN BOOL3 AS SMALLINT
```

```
  CHECK (VALUE IS NULL OR VALUE IN (0, 1));
```

Siehe auch

Explizite Datentypumwandlung zur Beschreibung von Unterschieden im Datenkonvertierungsmechanismus, wenn Domänen für die Modifikatoren `TYPE OF` und `TYPE OF COLUMN` angegeben werden.

3.10.2. Domain-Überschreibung

Beim Definieren einer Spalte mithilfe einer Domäne ist es möglich, einige der von der Domäne geerbten Attribute zu überschreiben. Tabelle 3.9 fasst die Regeln für die Domänenüberschreibung zusammen.

Tabelle 11. Regeln zum Überschreiben von Domänenattributen in der Spaltendefinition

Attribute	Überschreiben?	Hinweise
Datentyp	Nein	
Standardwert	Ja	
Textzeichensatz	Ja	Es kann auch verwendet werden, um die Standarddatenbankwerte für die Spalte wiederherzustellen
Reihenfolge der Textsortierung	Ja	
`CHECK`-Constraints	Ja	Um der Prüfung neue Bedingungen hinzuzufügen, können Sie die entsprechenden `CHECK`-Klauseln in den Anweisungen CREATE und ALTER auf Tabellenebene verwenden.
`NOT NULL`	Nein	Oft ist es besser, die Domain in ihrer Definition nullbar zu lassen und zu entscheiden, ob sie auf `NOT NULL` gesetzt werden soll, wenn die Domain zum Definieren von Spalten verwendet wird.

3.10.3. Erstellen und Verwalten von Domains

Eine Domain wird mit der DDL-Anweisung `CREATE DOMAIN` erstellt.

Kurzschreibweise

```
CREATE DOMAIN name [AS] <type>
  [DEFAULT {<const> | <literal> | NULL | <context_var>}]
  [NOT NULL] [CHECK (<condition>)]
  [COLLATE <collation>]
```

Siehe auch

`CREATE DOMAIN` im Abschnitt Datendefinitionssprache (DDL).

Domain ändern

Um die Attribute einer Domain zu ändern, verwenden Sie die DDL-Anweisung `ALTER DOMAIN`. Mit dieser Aussage können Sie:

- die Domain umbenennen
- den Datentyp ändern
- den aktuellen Standardwert löschen
- einen neuen Standardwert setzen
- lösche die `NOT NULL`-Beschränkung
- setze die `NOT NULL`-Beschränkung
- eine bestehende `CHECK`-Einschränkung löschen
- füge eine neue `CHECK`-Einschränkung hinzu

Kurzsyntax

```
ALTER DOMAIN name
  [{TO new_name}]
  [{SET DEFAULT { <literal> | NULL | <context_var> } |
    DROP DEFAULT}]
  [{SET | DROP} NOT NULL ]
  [{ADD [CONSTRAINT] CHECK (<dom_condition>) |
    DROP CONSTRAINT}]
  [{TYPE <datatype>}]
```

Beispiel

```
ALTER DOMAIN STORE_GRP SET DEFAULT -1;
```

Beim Wechsel einer Domain müssen deren Abhängigkeiten berücksichtigt werden: ob Tabellenspalten, beliebige Variablen, Ein- und/oder Ausgabeparameter mit dem im PSQL-Code deklarierten Typ dieser Domain vorhanden sind. Wenn Sie Domains in Eile ändern, ohne sie sorgfältig zu überprüfen, funktioniert Ihr Code möglicherweise nicht mehr!

Wenn Sie Datentypen in einer Domain konvertieren, dürfen Sie keine Konvertierungen durchführen, die zu Datenverlusten führen können. Wenn Sie beispielsweise `VARCHAR` in `INTEGER` konvertieren, prüfen Sie sorgfältig, ob alle Daten, die diese Domain verwenden, erfolgreich konvertiert werden können.

Siehe auch

`ALTER DOMAIN` im Abschnitt Datendefinitionssprache (DDL).

Löschen (Dropping) einer Domain

Die DDL-Anweisung `DROP DOMAIN` löscht eine Domain aus der Datenbank, sofern sie nicht von anderen Datenbankobjekten verwendet wird.

Syntax

```
DROP DOMAIN name
```

Jeder mit der Datenbank verbundene Benutzer kann eine Domäne löschen.

Beispiel

```
DROP DOMAIN Test_Domain
```

Siehe auch

`DROP DOMAIN` im Abschnitt Datendefinitionssprache (DDL).

3.11. Syntax der Datentyp-Deklaration

In diesem Abschnitt wird die Syntax der Deklaration von Datentypen dokumentiert. Die Datentypdeklaration erfolgt am häufigsten in DDL-Anweisungen, aber auch in `CAST` und <<fblangref30-dml-execblock-de,`EXECUTE BLOCK`> >.

Auf die unten dokumentierte Syntax wird von anderen Teilen dieser Sprachreferenz verwiesen.

3.11.1. Syntax für Skalardatentypen

Die skalaren Datentypen sind einfache Datentypen, die einen einzelnen Wert enthalten. Aus organisatorischen Gründen wird die Syntax der BLOB-Typen separat in Syntax der BLOB-Datentypen definiert.

Syntax für skalare Datentypen

```
<domain_or_non_array_type> ::=
    <scalar_datatype>
  | <blob_datatype>
  | [TYPE OF] domain
  | TYPE OF COLUMN rel.col

<scalar_datatype> ::=
    SMALLINT | INT[EGER] | BIGINT
  | FLOAT | DOUBLE PRECISION
  | BOOLEAN
  | DATE | TIME | TIMESTAMP
  | {DECIMAL | NUMERIC} [(precision [, scale])]
  | {VARCHAR | {CHAR | CHARACTER} VARYING} (length)
    [CHARACTER SET charset]
  | {CHAR | CHARACTER} [(length)] [CHARACTER SET charset]
  | {NCHAR | NATIONAL {CHARACTER | CHAR}} VARYING (length)
  | {NCHAR | NATIONAL {CHARACTER | CHAR}} [(length)]
```

Tabelle 12. Argumente für die Syntax der skalaren Datentypen

Argument	Beschreibung
domain	Domain (nur Nicht-Array-Domains)
rel	Name einer Tabelle oder Ansicht (View)
col	Name einer Spalte in einer Tabelle oder Ansicht (nur Spalten eines Nicht-Array-Typs)
precision	Numerische Genauigkeit in Dezimalstellen. Von 1 bis 18
scale	Skalierung oder Anzahl der Dezimalstellen. Von 0 bis 18. Sie muss kleiner oder gleich *precision* sein.
length	Die maximale Länge einer Zeichenfolge in Zeichen
charset	Zeichensatz
domain_or_non_array_type	Nicht-Array-Typen, die in PSQL-Code und -Casts verwendet werden können

Verwendung von Domains in Deklarationen

Ein Domainname kann als Typ eines PSQL-Parameters oder einer lokalen Variablen angegeben werden. Der Parameter oder die Variable erbt alle Domänenattribute. Wenn für den Parameter oder die Variable ein Standardwert angegeben wird, überschreibt er den in der Domaindefinition angegebenen Standardwert.

Wenn die `TYPE OF`-Klausel vor dem Domainnamen hinzugefügt wird, wird nur der Datentyp der Domain verwendet: alle anderen Attribute der Domain — `NOT NULL`-Einschränkung, `CHECK`-Einschränkungen, Standardwert — sind weder geprüft noch benutzt. Handelt es sich bei der Domain jedoch um einen Texttyp, werden immer deren Zeichensatz und Kollatierungsreihenfolge verwendet.

Verwendung des Spaltentyps in Deklarationen

Ein- und Ausgabeparameter oder lokale Variablen können auch über den Datentyp von Spalten in bestehenden Tabellen und Views deklariert werden. Dafür wird die `TYPE OF COLUMN`-Klausel verwendet, die *relationname.columnname* als Argument angibt.

Wenn `TYPE OF COLUMN` verwendet wird, erbt der Parameter oder die Variable nur den Datentyp und – bei String-Typen – den Zeichensatz und die Kollatierungssequenz. Die Einschränkungen und der Standardwert der Spalte werden ignoriert.

3.11.2. Syntax der BLOB-Datentypen

Die BLOB-Datentypen enthalten Binär-, Zeichen- oder benutzerdefinierte Formatdaten unbestimmter Größe. Weitere Informationen finden Sie unter Binärdatentypen.

Syntax der BLOB-Datentypen

```
<blob_datatype> ::=
    BLOB [SUB_TYPE {subtype_num | subtype_name}]
    [SEGMENT SIZE seglen] [CHARACTER SET charset]
```

```
  | BLOB [(seglen [, subtype_num])]
```

Tabelle 13. Argumente für die Syntax der Blob-Datentypen

Argument	Beschreibung
charset	Zeichensatz (wird für andere Untertypen als TEXT/1 ignoriert)
subtype_num	BLOB-Untertypnummer
subtype_name	mnemonischer Name des 'BLOB'-Untertyps; dies kann TEXT, BINARY oder einer der (anderen) Standard- oder benutzerdefinierten Namen sein, die in RDB$TYPES für RDB$FIELD_NAME = 'RDB$FIELD_SUB_TYPE' definiert sind.
seglen	Segmentgröße, darf nicht größer als 65.535 sein, Standardwert 80, wenn nicht angegeben. Siehe auch Segmentgröße

3.11.3. Syntax der Array-Datentypen

Die Array-Datentypen enthalten mehrere Skalarwerte in einem ein- oder mehrdimensionalen Array. Weitere Informationen finden Sie unter ARRAY-Datentyp

Syntax der Array-Datentypen

```
<array_datatype> ::=
    {SMALLINT | INT[EGER] | BIGINT} <array_dim>
  | {FLOAT | DOUBLE PRECISION} <array_dim>
  | BOOLEAN <array_dim>
  | {DATE | TIME | TIMESTAMP} <array_dim>
  | {DECIMAL | NUMERIC} [(precision [, scale])] <array_dim>
  | {VARCHAR | {CHAR | CHARACTER} VARYING} (length)
    <array_dim> [CHARACTER SET charset]
  | {CHAR | CHARACTER} [(length)] <array_dim>
    [CHARACTER SET charset]
  | {NCHAR | NATIONAL {CHARACTER | CHAR}} VARYING (length)
    <array_dim>
  | {NCHAR | NATIONAL {CHARACTER | CHAR}}
    [(length)] <array_dim>

<array_dim> ::= '[' [m:]n [,[m:]n ...] ']'
```

Tabelle 14. Argumente für die Syntax der Array-Datentypen

Argument	Beschreibung
array_dim	Array-Dimensionen
precision	Numerische Genauigkeit in Dezimalstellen. Von 1 bis 18
scale	Skala oder Anzahl der Dezimalstellen. Von 0 bis 18. Sie muss kleiner oder gleich *precision* sein.
length	Die maximale Länge einer Zeichenfolge in Zeichen; optional für Zeichentypen mit fester Breite, standardmäßig 1

Argument	Beschreibung
charset	Zeichensatz
m, n	Ganzzahlen, die den Indexbereich einer Array-Dimension definieren

Kapitel 4. Allgemeine Sprachelemente

Dieser Abschnitt behandelt die Elemente, die in der SQL-Sprache als allgemeingültig betrachtet werden können — die *Ausdrücke*, die verwendet werden um Fakten aus Daten zu extrahieren, diese zu verarbeiten und die *Prädikate*, die den Wahrheitswert dieser Fakten prüfen.

4.1. Ausdrücke

SQL-Ausdrücke bieten formelle Methoden zum Auswerten, Transformieren und Vergleichen von Werten. SQL-Ausdrücke können Tabellenspalten, Variablen, Konstanten, Literale, andere Statements und Prädikate sowie andere Ausdrücke enthalten. Folgend die vollständige Liste möglicher Elemente.

Beschreibung der Ausdruck-Elemente

Spaltenname

Kennung einer Spalte aus einer angegebenen Tabelle, die in Auswertungen oder als Suchbedingung verwendet wird. Eine Spalte des Array-Typs kann kein Element innerhalb eines Ausdrucks sein, es sei denn sie wird mit dem `IS [NOT] NULL`-Prädikat verwendet.

Array-Element

Ein Ausdruck kann einen Verweis auf ein Array-Element enthalten.

Arithmetische Operatoren

Die Zeichen +, -, *, / werden verwendet um Berechnungen durchzuführen.

Verkettungsoperator

Der Operator || (“Doppel-Pipe”) wird verwendet um Strings zu verketten.

Logische Operatoren

Die reservierten Wörter `NOT`, `AND` sowie `OR` werden verwendet um einfache Suchbedingungen oder komplexere Behauptungen zu erstellen.

Vergleichsoperatoren

Die Zeichen =, <>, !=, ~=, ^=, <, <=, >, >=, !<, ~<, ^<, !>, ~> und ^>

Vergleichsprädikate

`LIKE`, `STARTING WITH`, `CONTAINING`, `SIMILAR TO`, `BETWEEN`, `IS [NOT] NULL` und `IS [NOT] DISTINCT FROM`

Existenzprädikate

Prädikate, die für die Existenzprüfung von Werten Verwendung finden. Das Prädikat `IN` kann sowohl innerhalb von Listen kommagetrennter Konstanten als auch mit Unterabfragen, die nur eine Spalte zurückgeben, verwendet werden. Die Prädikate `EXISTS`, `SINGULAR`, `ALL`, `ANY` und `SOME` können nur mit Unterabfragen verwendet werden.

Konstante oder Litaral

In Apostrophen eingeschlossene Zahlen oder String-Literale, Boolesche Werte `TRUE`, `FALSE` und `UNKOWN`, `NULL

Datum-/Zeitliterale

Ein Ausdruck, ähnlich zu Zeichenketten, eingeschlossen in Apostrophs, der als Datum, Zeit oder Zeitstempel interpretiert wird. Datumsliterale können vordefinierte Literale ('TODAY', 'NOW', etc.) oder Zeichenketten aus Buchstaben oder Zahlen sein, wie zum Beispiel '30.12.2016 15:30:35', die zu einem Datum und/oder einer Zeit aufgelöst werden können.

Kontextvariablen

Ein intern definierte Kontextvariable

Lokale Variablen

Deklarierte lokale Variablen, Über- und Rückgabeparameter eines PSQL-Moduls (Stored Procedure, Trigger, unbenannter PSQL-Block in DSQL)

Positionale Parameter

Ein Mitglied innerhalb einer geordneten Gruppe von einem oder mehreren unbenannten Parametern, die an eine gespeicherte Prozedur oder eine vorbereitete Abfrage übergeben wurden.

Unterabfrage

Eine `SELECT`-Anweisung, die in Klammern eingeschlossen ist, die einen einzelnen (skalaren) Wert zurückgibt oder, wenn er in existenziellen Prädikaten verwendet wird, einen Satz von Werten.

Funktionskennung

Die Kennung einer internen oder externen Funktion in einem Funktionsausdruck

Type-Cast

Ein Ausdruck, der explizit Daten von einem in einen anderen Datentyp unter Verwendung der `CAST`-Funktion (`CAST (<value> AS <datatype>)`) konvertiert. Nur für Datum-/Zeit-Literale ist die Kurzschreibweise <datatype> <value> (`DATE '30.12.2016'`) möglich.

Bedingter Ausdruck

Ausdrücke mit `CASE` und verwandten internen Funktionen

Klammern

Klammernpaare (⋯) werden verwendet, um Ausdrücke zu gruppieren. Operationen innerhalb der Klammern werden vor Operationen außerhalb von ihnen durchgeführt. Wenn eingebettete Klammern verwendet werden, werden die tiefsten eingebetteten Ausdrücke zuerst ausgewertet und dann bewegen sich die Auswertungen von innen nach außen durch die Einbettungsstufen.

COLLATE-Klausel

Klausel, die für CHAR- und VARCHAR-Datentypen angewendet werden kann, um die Collation für String-Vergleiche festzulegen.

`NEXT VALUE FOR sequence`

Ausdruck zum Ermitteln des nächsten Wertes eines bestimmten Generators (Sequenz). Die interne Funktion `GEN_ID()` tut das Gleiche.

4.1.1. Konstanten

Eine Konstante ist ein Wert der direkt in einem SQL-Statement verwendet wird und weder von einem Ausdruck, einem Parameter, einem Spaltenverweis noch einer Variablen abgeleitet wird. Dies kann eine Zeichenkette oder eine Zahl sein.

Zeichenkonstanten (Literale)

Eine String-Konstante ist eine Aneinanderreihung von Zeichen, die zwischen einem Paar von Apostrophen (“einfache Anführungszeichen”) eingeschlossen werden. Die größtmögliche Länge dieser Zeichenketten ist 32.767 Bytes; die maximale Anzahl der Zeichen wird durch die verwendete Zeichenkodierung bestimmt.

- Doppelte Anführungszeichen sind *NICHT GÜLTIG* für das Anführungszeichen von Zeichenfolgen. Der SQL-Standard reserviert doppelte Anführungszeichen für einen anderen Zweck: das Anführen von Bezeichnern.
- Wenn ein literaler Apostroph innerhalb einer String-Konstante erforderlich ist, wird er “escaped”, indem ihm ein anderer Apostroph vorangestellt wird. Zum Beispiel ’Mutter O''Reillys hausgemachter Hooch''.
- Bei der Stringlänge ist Vorsicht geboten, wenn der Wert in eine `CHAR`- oder `VARCHAR`-Spalte geschrieben werden soll. Die maximale Länge für ein `CHAR`- oder VARCHAR`-Literal beträgt 32.765 Byte.

Es wird angenommen, dass der Zeichensatz einer Zeichenkonstanten der gleiche ist wie der Zeichensatz seines Bestimmungsspeichers.

Stringkonstanten in Hexadezimalnotation

Ab Firebird 2.5 können String-Literale in hexadezimaler Notation eingegeben werden, sogenannte “binary strings”. Jedes Paar von Hex-Ziffern definiert ein Byte in der Zeichenfolge. Auf diese Weise eingegebene Zeichenfolgen haben standardmäßig den Zeichensatz `OCTETS`, aber die *Einführer-Syntax* kann verwendet werden, um zu erzwingen, dass ein String als ein anderer Zeichensatz interpretiert wird.

Syntax

```
{x|X}'<hexstring>'

<hexstring> ::= eine gerade Anzahl von <hexdigit>
<hexdigit>  ::= eines aus 0..9, A..F, a..f
```

Beispiele

```
select x'4E657276656E' from rdb$database
-- liefert 4E657276656E, ein 6-Byte 'Binärstring'

select _ascii x'4E657276656E' from rdb$database
-- liefert 'Nerven' (gleiche Zeichenfolge, jetzt als ASCII-Text interpretiert)
```

```
select _iso8859_1 x'53E46765' from rdb$database
-- liefert 'Säge' (4 Zeichen, 4 Bytes)

select _utf8 x'53C3A46765' from rdb$database
-- liefert 'Säge' (4 Zeichen, 5 Bytes)
```

Hinweise

Die Client-Schnittstelle legt fest, wie Binärzeichenfolgen dem Benutzer angezeigt werden. Das *isql*-Werkzeug beispielsweise, nutzt großgeschriebene Buchstaben A-F, während FlameRobin Kleinschreibung verwendet. Andere Client-Applikationen könnten andere Konventionen bevorzugen, zum Beispiel Leerzeichen zwischen den Bytepaaren: '4E 65 72 76 65 6E'.

Mit der hexadezimalen Notation kann jeder Bytewert (einschließlich 00) an beliebiger Stelle im String eingefügt werden. Allerdings, wenn Sie diesen auf etwas anderes als OCTETS erzwingen wollen, liegt es in Ihrer Verantwortung, die Bytes in einer Sequenz zu liefern, die für den Zielzeichensatz gültig ist.

Alternative String-Literale

Seit Firebird 3.0 ist es möglich, ein anderes Zeichen oder Zeichenpaar als das doppelte (escaped) Apostroph zu verwenden, um einen String in Anführungszeichen in einen anderen String einzubetten. Das Schlüsselwort `q` oder `Q` vor einem String in Anführungszeichen informiert den Parser darüber, dass bestimmte Links-Rechts-Paare oder Paare identischer Zeichen innerhalb des Strings die Begrenzer des eingebetteten String-Literals sind.

Syntax

```
<alternative string literal> ::=
    { q | Q } <quote> <start char> [<char> ...] <end char> <quote>
```

Regeln

Wenn `<start char>` `(', '{', '[' oder '<' ist, `< end char>` wird mit seinem jeweiligen "partner" gepaart, nämlich ')', '}', ']' und '` >'. In anderen Fällen ist `<end char>` dasselbe wie `<start char>`.

Innerhalb des Strings, d. h. `<char>`-Elemente, können einfache (nicht maskierte) Anführungszeichen verwendet werden. Jedes Anführungszeichen ist Teil der Ergebniszeichenfolge.

Beispiel

```
select q'{abc{def}ghi}' from rdb$database;        -- Ergebnis: abc{def}ghi
select q'!That's a string!' from rdb$database;    -- Ergebnis: That's a string
```

Introducer-Syntax für String-Literale

Gegebenenfalls kann einem Zeichenfolgenliteral ein Zeichensatzname vorangestellt werden, dem ein Unterstrich "_" vorangestellt ist. Dies ist als *Introducer-Syntax* bekannt. Sein Zweck besteht darin, die Engine darüber zu informieren, wie die eingehende Zeichenfolge zu interpretieren und zu speichern ist.

Beispiel

```
INSERT INTO People
VALUES (_ISO8859_1 'Hans-Jörg Schäfer')
```

Zahlenkonstanten

Eine Zahlkonstante ist eine gültige Zahl in einer unterstützten Notation:

- In SQL wird der Dezimalpunkt, für Zahlen in der Standard-Dezimal-Notation, immer durch das Punkt-Zeichen dargestellt. Tausender werden nicht getrennt. Einbeziehung von Komma, Leerzeichen usw. führt zu Fehlern.
- Exponentielle Notation wird unterstützt. Zum Beispiel kann 0.0000234 auch als `2.34e-5` geschrieben werden.
- Hexadezimal-Notation wird von Firebird 2.5 und höheren Versionen unterstützt — siehe unten.

Das Format des Literals bestimmt den Typ (`<d>` für eine Dezimalziffer, `<h>` für eine Hexadezimalziffer):

Format	Typ
`<d>[<d> ⋯]`	`INTEGER` oder `BIGINT` (hängt davon ab, ob der Wert in den Typ passt)
`0{x|X} <h><h>[<h><h> ⋯]`	`INTEGER` für 1-8 `<h><h>` Paare oder `BIGINT` für 9-16 Paare
`<d>[<d> ⋯] "." [<d> ⋯]`	`NUMERIC(18, n)` wobei *n* von der Anzahl der Nachkommastellen abhängt
`<d>[<d> ⋯]["." [<d> ⋯]] E <d>[<d> ⋯]`	`DOUBLE PRECISION`

Hexadezimale Notation für Ziffern

Von Firebird 2.5 aufwärts können ganzzahlige Werte in hexadezimaler Notation eingegeben werden. Zahlen mit 1-8 Hex-Ziffern werden als Typ `INTEGER` interpretiert; Zahlen mit 9-16 Hex-Ziffern als Typ `BIGINT`.

Syntax

```
0{x|X}<hexdigits>

<hexdigits>  ::=  1-16 of <hexdigit>
```

```
<hexdigit>   ::=  one of 0..9, A..F, a..f
```

Beispiele

```
select 0x6FAA0D3 from rdb$database          -- liefert 117088467
select 0x4F9 from rdb$database              -- liefert 1273
select 0x6E44F9A8 from rdb$database         -- liefert 1850014120
select 0x9E44F9A8 from rdb$database         -- liefert -1639646808 (an INTEGER)
select 0x09E44F9A8 from rdb$database        -- liefert 2655320488 (a BIGINT)
select 0x28ED678A4C987 from rdb$database    -- liefert 720001751632263
select 0xFFFFFFFFFFFFFFFF from rdb$database -- liefert -1
```

Hexadezimale Wertebereiche

- Hex-Nummern im Bereich 0 .. 7FFF FFFF sind positive `INTEGER` mit Dezimalwerten zwischen 0 .. 2147483647. Um eine Zahl als `BIGINT` zu erzwingen, müssen Sie genügend Nullen voranstellen, um die Gesamtzahl der Hex-Ziffern auf neun oder mehr zu bringen. Das ändert den Typ, aber nicht den Wert.
- Hex-Nummern zwischen 8000 0000 .. FFFF FFFF erfordern etwas Aufmerksamkeit:
  - Bei der Eingabe mit acht Hex-Ziffern, wie in 0x9E44F9A8, wird ein Wert als 32-Bit-`INTEGER` interpretiert. Da das erste Bit (Vorzeichenbit) gesetzt ist, wird es dem negativen Dezimalbereich -2147483648 .. -1 zugeordnet.
  - Bei einer oder mehreren Nullen, die wie in 0x09E44F9A8 vorangestellt werden, wird ein Wert als 64-Bit-`BIGINT` im Bereich 0000 0000 8000 0000 .. 0000 0000 FFFF FFFF interpretiert. Das Zeichen-Bit ist jetzt nicht gesetzt, also wird der Dezimalwert dem positiven Bereich 2147483648 .. 4294967295 zugewiesen.

  So ergibt sich in diesem Bereich — und nur in diesem Bereich — anhand einer mathematisch unbedeutenden 0 ein gänzlich anderer Wert. Dies ist zu beachten.
- Hex-Zahlen zwischen 1 0000 0000 .. 7FFF FFFF FFFF FFFF sind alle positiv `BIGINT`.
- Hex-Zahlen zwischen 8000 0000 0000 0000 .. FFFF FFFF FFFF FFFF sind alle negativ `BIGINT`.
- Ein `SMALLINT` kann nicht in Hex geschrieben werden, streng genommen zumindest, da sogar 0x1 als `INTEGER` ausgewertet wird. Wenn Sie jedoch eine positive Ganzzahl innerhalb des 16-Bit-Bereichs 0x0000 (Dezimal-Null) bis 0x7FFF (Dezimalzahl 32767) schreiben, wird sie transparent in `SMALLINT` umgewandelt.

  Es ist möglich einen negativen `SMALLINT` in Hex zu schreiben, wobei eine 4-Byte-Hexadezimalzahl im Bereich 0xFFFF8000 (Dezimal -32768) bis 0xFFFFFFFF (Dezimal -1) verwendet wird.

Boolesche Literale

Ein boolesches Literal ist eines von `TRUE`, `FALSE` oder `UNKNOWN`.

4.1.2. SQL-Operatoren

SQL-Operatoren umfassen Operatoren zum Vergleichen, Berechnen, Auswerten und Verketten von Werten.

Vorrang der Operatoren

SQL Operatoren sind in vier Typen unterteilt. Jeder Operator-Typ hat eine *Priorität*, eine Rangfolge, die die Reihenfolge bestimmt, in der die Operatoren und die mit ihrer Hilfe erhaltenen Werte in einem Ausdruck ausgewertet werden. Je höher der Vorrang des Operator-Typs ist, desto früher wird er ausgewertet. Jeder Operator hat seine eigene Priorität innerhalb seines Typs, der die Reihenfolge bestimmt, in der sie in einem Ausdruck ausgewertet werden.

Operatoren der gleichen Rangfolge werden von links nach rechts ausgewertet. Um dieses Verhalten zu beeinflussen, können Gruppen mittels Klammern erstellt werden.

Tabelle 15. Vorrang der Operatortypen

Operatortyp	Vorrang	Erläuterung
Verkettung	1	Strings werden verkettet, bevor andere Operationen stattfinden
Arithmetik	2	Arithmetische Operationen werden durchgeführt, nachdem Strings verkettet sind, aber vor Vergleichs- und logischen Operationen
Vergleiche	3	Vergleichsoperationen erfolgen nach String-Verkettung und arithmetischen Operationen, aber vor logischen Operationen
Logical	4	Logische Operatoren werden nach allen anderen Operatortypen ausgeführt

Verkettungsoperator

Der Verkettungsoperator, zwei Pipe-Zeichen, auch “Doppel-Pipe” — ‘||’ — verkettet (verbindet) zwei Zeichenketten zu einer einzigen Zeichenkette. Zeichenketten können dabei Konstante Werte oder abgeleitet von einer Spalte oder einem Ausdruck sein.

Beispiel

```
SELECT LAST_NAME || ', ' || FIRST_NAME AS FULL_NAME
FROM EMPLOYEE
```

Arithmetische Operatoren

Tabelle 16. Vorrang arithmetischer Operatoren

Operator	Zweck	Vorrang
`+Zahl mit Vorzeichen`	unäres Plus	1
`-Zahl mit Vorzeichen`	unäres Minus	1

Operator	Zweck	Vorrang
*	Multiplikation	2
/	Division	2
+	Addition	3
-	Subtraktion	3

Beispiel

```
UPDATE T
    SET A = 4 + 1/(B-C)*D
```

Wenn Operatoren den gleichen Vorrang besitzen, werden diese von links nach rechts ausgewertet.

Vergleichsoperatoren

Tabelle 17. Prioritäten der Vergleichsoperatoren

Operator	Zweck	Priorität
IS	Überprüft, ob der Ausdruck auf der linken Seite (nicht) NULL oder der boolesche Wert auf der rechten Seite ist	1
=	Ist gleich, ist identisch mit	2
<>, !=, ~=, ^=	Ist ungleich zu	2
>	Ist größer als	2
<	Ist kleiner als	2
>=	Ist größer gleich als	2
<=	Ist kleiner gleich als	2
!>, ~>, ^>	Ist nicht größer als	2
!<, ~<, ^<	Ist nicht kleiner als	2

Diese Gruppe umfasst auch Vergleichsprädikate BETWEEN, LIKE, CONTAINING, SIMILAR TO und andere.

Beispiel

```
IF (SALARY > 1400) THEN
...
```

Siehe auch

Andere Vergleichsprädikate.

Logische Operatoren

Tabelle 18. Prioritäten logischer Operatoren

Operator	Zweck	Priorität
`NOT`	Negierung eines Suchkriteriums	1
`AND`	Kombiniert zwei oder mehr Prädikate, wobei jedes als wahr angesehen werden muss, damit der Gesamtausdruck ebenfalls als wahr aufgelöst wird	2
`OR`	Kombiniert zwei oder mehr Prädikate, wobei mindestens eines als wahr angesehen werden muss, damit der Gesamtausdruck ebenfalls als wahr aufgelöst wird	3

Beispiel

```
IF (A < B OR (A > C AND A > D) AND NOT (C = D)) THEN ⋯
```

NEXT VALUE FOR

Verfügbar in

DSQL, PSQL

Syntax

```
NEXT VALUE FOR Sequenzname
```

`NEXT VALUE FOR` gibt den nächsten Wert einer Sequenz zurück. `SEQUENCE` ist ein SQL-konformer Begriff für Generatoren in Firebird und dessen Vorgänger, InterBase. Der Operator `NEXT VALUE FOR` ist equivalent zur ursprünglichen Funktion `GEN_ID (⋯, 1)` und ist die empfohlene Syntax zum Holen des nächsten Wertes.

Anders als `GEN_ID (⋯, 1)` verwendet `NEXT VALUE FOR` keine Parameter, wodurch es nicht möglich ist den *aktuellen Wert* einer Sequenz zu ermitteln sowie eine andere Schrittweite als 1 zu nutzen. `GEN_ID (⋯, <step value>)` wird noch immer für diesen Zweck verwendet. Eine *Schrittweite* von 0 gibt den aktuellen Sequenzwert zurück.

Beispiel

```
NEW.CUST_ID = NEXT VALUE FOR CUSTSEQ;
```

Siehe auch

SEQUENCE (GENERATOR), `GEN_ID()`

4.1.3. Bedingte Ausdrücke

Ein bedingter Ausdruck ist einer der verschiedene Werte zurückgibt, je nach verwendeter Bedingung. Es besteht aus einem bedingten Funktionskonstrukt, wovon Firebird mehrere unterstützt. Dieser Abschnitt beschreibt nur ein bedingtes Ausdruckskonstrukt: `CASE`. Alle anderen bedingten Ausdrücke sind interne Funktionen und leiten sich von `CASE` ab und werden in Bedingte Funktionen beschrieben.

`CASE`

Verfügbar in

DSQL, PSQL

Das `CASE`-Konstrukt gibt einen einzigen Wert aus einer Reihe von Werten zurück. Zwei syntaktische Varianten werden unterstützt:

- Das *einfache* `CASE`, vergleichbar zu einem *CASE-Konstrukt* in Pascal oder einem *Switch* in C
- Das *gesuchte* `CASE`, welches wie eine Reihe aus "`if ⋯ else if ⋯ else if`"-Klauseln funktioniert.

Einfaches `CASE`

Syntax

```
...
CASE <test-expr>
  WHEN <expr> THEN <result>
  [WHEN <expr> THEN <result> ...]
  [ELSE <defaultresult>]
END
...
```

Bei dieser Variante wird *test-expr* mit dem ersten *expr*, dem zweiten *expr* usw. verglichen, bis eine Übereinstimmung gefunden wird und das entsprechende Ergebnis zurückgegeben wird. Wenn keine Übereinstimmung gefunden wird, wird *defaultresult* aus der optionalen `ELSE`-Klausel zurückgegeben. Wenn es keine Übereinstimmungen und keine `ELSE`-Klausel gibt, wird `NULL` zurückgegeben.

Das Matching funktioniert genauso wie der Operator "=". Das heißt, wenn *test-expr* `NULL` ist, stimmt es mit keinem *expr* überein, nicht einmal mit einem Ausdruck, der in `NULL` aufgelöst wird.

Das zurückgegebene Ergebnis muss kein Literalwert sein: Es kann ein Feld- oder Variablenname, ein zusammengesetzter Ausdruck oder ein `NULL`-Literal sein.

Beispiel

```
SELECT
  NAME,
  AGE,
  CASE UPPER(SEX)
    WHEN 'M' THEN 'Male'
```

```
    WHEN 'F' THEN 'Female'
    ELSE 'Unknown'
  END GENDER,
RELIGION
    FROM PEOPLE
```

Eine Kurzform des einfachen CASE-Konstrukts wird auch in der DECODE -Funktion verwendet.

Gesuchtes CASE

Syntax

```
CASE
  WHEN <bool_expr> THEN <result>
  [WHEN <bool_expr> THEN <result> ⋯]
  [ELSE <defaultresult>]
END
```

Der *bool_expr*-Ausdruck gibt ein ternäres logisches Ergebnis zurück: TRUE, FALSE oder NULL. Der erste Ausdruck, der TRUE ermittelt, wird als Ergebnis verwendet. Gibt kein Ausdruck TRUE zurück, kommt *defaultresult* aus der optionalen ELSE-Klausel zum Einsatz. Gibt kein Ausdruck TRUE zurück und gibt es keine ELSE-Klausel, ist der Rückgabewert NULL.

So wie im einfachen CASE-Konstrukt, muss das Ergebnis nicht zwangsläufig ein Literal sein: es kann ein Feld- oder Variablenname, ein zusammengesetzter Ausdruck oder NULL sein.

Beispiel

```
CANVOTE = CASE
  WHEN AGE >= 18 THEN 'Yes'
  WHEN AGE < 18 THEN 'No'
  ELSE 'Unsure'
END
```

4.1.4. NULL in Ausdrücken

NULL ist in SQL kein Wert, sondern ein *state*, der anzeigt, dass der Wert des Elements entweder *unbekannt* ist oder nicht existiert. Es ist weder eine Null, noch ein Leerzeichen, noch ein “leerer String”, und es verhält sich nicht wie ein Wert.

Wenn Sie NULL in numerischen, String- oder Datums-/Uhrzeit-Ausdrücken verwenden, ist das Ergebnis immer NULL. Wenn Sie NULL in logischen (booleschen) Ausdrücken verwenden, hängt das Ergebnis vom Typ der Operation und von anderen beteiligten Werten ab. Wenn Sie einen Wert mit NULL vergleichen, ist das Ergebnis *unbekannt*.

NULL heißt NULL, jedoch gilt in Firebird, dass das logische Ergebnis *unknown* ebenfalls durch NULL *repräsentiert* wird.

Ausdrücke die NULL zurückgeben

Ausdrücke in dieser Liste werden immer NULL zurückgeben:

```
1 + 2 + 3 + NULL
'Home ' || 'sweet ' || NULL
MyField = NULL
MyField <> NULL
NULL = NULL
not (NULL)
```

Wenn es Ihnen schwerfällt dies zu verstehen, beachten Sie, dass NULL ein Status ist, der für "unknown" (unbekannt) steht.

NULL in logischen Ausdrücken

Es wurde bereits gezeigt, dass not (NULL) in NULL aufgeht. Dieser Effekt ist etwas komplizierter für logische AND- sowie logische OR-Operatoren:

```
NULL or false  → NULL
NULL or true   → true
NULL or NULL   → NULL
NULL and false → false
NULL and true  → NULL
NULL and NULL  → NULL
```

Als grundlegende Faustregel gilt: Wenn die Anwendung von TRUE anstelle von NULL zu einem anderen Ergebnis führt als die Anwendung von FALSE, dann ist das Ergebnis des ursprünglichen Ausdrucks *unknown* oder NULL.

Beispiele

```
(1 = NULL) or (1 <> 1)    -- Ergebnis NULL
(1 = NULL) or FALSE       -- Ergebnis NULL
(1 = NULL) or (1 = 1)     -- Ergebnis TRUE
(1 = NULL) or TRUE        -- Ergebnis TRUE
(1 = NULL) or (1 = NULL)  -- Ergebnis NULL
(1 = NULL) or UNKNOWN     -- Ergebnis NULL
(1 = NULL) and (1 <> 1)   -- Ergebnis FALSE
(1 = NULL) and FALSE      -- Ergebnis FALSE
(1 = NULL) and (1 = 1)    -- Ergebnis NULL
(1 = NULL) and TRUE       -- Ergebnis NULL
(1 = NULL) and (1 = NULL) -- Ergebnis NULL
(1 = NULL) and UNKNOWN    -- Ergebnis NULL
```

4.1.5. Unterabfragen

Eine Unterabfrage ist eine spezielle Form eines Ausdrucks, die innerhalb einer anderen Abfrage eingebettet wird. Unterabfragen werden in der gleichen Weise geschrieben wie reguläre `SELECT` -Abfragen, werden jedoch von Klammern umschlossen. Unterabfrage-Ausdrücke können in folgender Art und Weise verwendet werden:

- Um eine Ausgabespalte in der SELECT-Liste anzugeben
- Um Werte zu holen oder als Kriterium für Suchprädikate (die `WHERE`- und `HAVING`-Klauseln)
- Um ein Set zu erstellen, das die Eltern-Abfrage verwenden kann, so als wäre dies eine reguläre Tabelle oder View. Unterabfragen wie diese erscheinen in der FROM-Klausel (Derived Tables) oder in einer Common Table Expression (CTE)

Korrelierte Unterabfragen

Eine Unterabfrage kann *korreliert* sein. Eine Abfrage ist korreliert, wenn die Unterabfrage und die Hauptabfrage voneinander abhängig sind. Um jeden Datensatz in der Unterabfrage zu verarbeiten, muss ein Datensatz in der Hauptabfrage abgerufen werden; d.h. die Unterabfrage hängt vollständig von der Hauptabfrage ab.

Beispiel einer korrelierten Unterabfrage

```
SELECT *
FROM Customers C
WHERE EXISTS
  (SELECT *
   FROM Orders O
   WHERE C.cnum = O.cnum
     AND O.adate = DATE '10.03.1990');
```

Werden Unterabfragen verwendet um Werte einer Ausgabespalte aus einer SELECT-Liste zu holen, muss die Unterabfrage ein *skalares* Ergebnis zurückliefern.

Skalare Ergebnisse

Unterabfragen, die in Suchprädikaten verwendet werden, mit Ausnahme von existenziellen und quantifizierten Prädikaten, müssen ein *skalares* Ergebnis zurückgeben; Das heißt, nicht mehr als eine Spalte von nicht mehr als einer passenden Zeile oder Aggregation. Sollte mehr zurückgegeben werden, wird es zu einem Laufzeitfehler kommen (“Multiple rows in a singleton select…”).

Obwohl es einen echten Fehler berichtet, kann die Nachricht etwas irreführend sein. Ein “singleton SELECT” ist eine Abfrage, die nicht mehr als eine Zeile zurückgeben kann. Jedoch sind “singleton” und “skalar” nicht gleichzusetzen: nicht alle singleton SELECTs müssen zwangsläufig skalar sein; und Einspalten-SELECTs können mehrere Zeilen für existenzielle und quantifizierte Prädikate zurückgeben.

Unterabfrage-Beispiele

1. Eine Unterabfrage als Ausgabespalte in einer SELECT-Liste:

```
SELECT
  e.first_name,
  e.last_name,
  (SELECT
       sh.new_salary
   FROM
       salary_history sh
   WHERE
       sh.emp_no = e.emp_no
   ORDER BY sh.change_date DESC ROWS 1) AS last_salary
FROM
  employee e
```

2. Eine Unterabfrage in der WHERE-Klausel, um das höchste Gehalt eines Mitarbeiters zu ermitteln und hierauf zu filtern:

```
SELECT
  e.first_name,
  e.last_name,
  e.salary
FROM employee e
WHERE
  e.salary = (
    SELECT MAX(ie.salary)
    FROM employee ie
  )
```

4.2. Prädikate

Ein Prädikat ist ein einfacher Ausdruck, der eine Behauptung aufstellt, wir nennen sie P. Wenn P zu TRUE (wahr) aufgelöst wird, ist die Behauptung erfolgreich. Wird sie zu FALSE (unwahr, falsch) oder NULL (UNKNOWN) aufgelöst, ist die Behauptung falsch. Hier gibt es einen Fallstrick: Nehmen wir an, das Prädikat P gibt FALSE zurück. In diesem Falle gilt, dass NOT(P) TRUE zurückgeben wird. Andererseits gilt, falls P NULL (unknown) zurückgibt, dann gibt NOT(P) ebenfalls NULL zurück.

In SQL können Prädikate in CHECK-Constraints auftreten, WHERE- und HAVING-Klauseln, CASE -Ausdrücken, der IIF()-Funktion und in der ON-Bedingung der JOIN-Klausel.

4.2.1. Bedingungen

Eine Behauptung ist ein Statement über Daten, die, wie ein Prädikat, zu TRUE, FALSE oder NULL aufgelöst werden können. Behauptungen bestehen aus einem oder mehr Prädikaten, möglicherweise mittels NOT negiert und verbunden durch AND- sowie OR-Operatoren. Klammern können verwendet werden um Prädikate zu gruppieren und die Ausführungsreihenfolge festzulegen.

Ein Prädikat kann andere Prädikate einbetten. Die Ausführung ist nach außen gerichtet, das heißt, das innenliegendste Prädikat wird zuerst ausgeführt. Jede “Ebene” wird in ihrer Rangfolge ausgewertet bis der Wahrheitsgehalt der endgültigen Behauptung aufgelöst wird.

4.2.2. Vergleichs-Prädikate

Ein Vergleichsprädikat besteht aus zwei Ausdrücken, die mit einem Vergelichsoperator verbunden sind. Es existieren traditionel sechs Vergleichsoperatoren:

```
=, >, <, >=, <=, <>
```

Für die vollständige Liste der Vergleichsoperatoren mit ihren Variantenformen siehe Vergleichsoperatoren.

Wenn eine der Seiten (links oder rechts) eines Vergleichsprädikats `NULL` darin hat, wird der Wert des Prädikats UNKNOWN.

Beispiele

1. Abrufen von Informationen über Computer mit der CPU-Frequenz nicht weniger als 500 MHz und der Preis niedriger als $800:

   ```
   SELECT *
   FROM Pc
   WHERE speed >= 500 AND price < 800;
   ```

2. Abrufen von Informationen über alle Punktmatrixdrucker, die weniger als $300 kosten:

   ```
   SELECT *
   FROM Printer
   WHERE ptrtype = 'matrix' AND price < 300;
   ```

3. Die folgende Abfrage gibt keine Daten zurück, auch nicht wenn es Drucker ohne zugewiesenen Typ gibt, da ein Prädikat, das `NULL` mit `NULL` vergleicht, `NULL` zurückgibt:

   ```
   SELECT *
   FROM Printer
   WHERE ptrtype = NULL AND price < 300;
   ```

   Andererseits kann `ptrtype` auf `NULL` getestet werden und ein Ergebnis zurückgeben: es ist nur kein _Vergleichstest:

   ```
   SELECT *
   FROM Printer
   WHERE ptrtype IS NULL AND price < 300;
   ```

— Siehe auch `IS [NOT] NULL`.

Hinweis zu String-Vergleichen

Wenn die Felder `CHAR` und `VARCHAR` auf Gleichheit verglichen werden, werden abschließende Leerzeichen in allen Fällen ignoriert.

Andere Vergleichsprädikate

Andere Vergleichsprädikate werden durch Schlüsselwörter gekennzeichnet.

`BETWEEN`

Verfügbar in

DSQL, PSQL, ESQL

Syntax

```
<value> [NOT] BETWEEN <value_1> AND <value_2>
```

Das Prädikat `BETWEEN` testet, ob ein Wert in einen angegebenen Bereich von zwei Werten fällt. (`NOT BETWEEN` testet, ob der Wert nicht in diesen Bereich fällt.)

Die Operanden für das Prädikat `BETWEEN` sind zwei Argumente kompatibler Datentypen. Im Gegensatz zu einigen anderen DBMS ist das Prädikat `BETWEEN` in Firebird nicht symmetrisch — wenn der niedrigere Wert nicht das erste Argument ist, gibt das Prädikat `BETWEEN` immer FALSE zurück. Die Suche ist inklusiv (die von beiden Argumenten repräsentierten Werte werden in die Suche eingeschlossen). Mit anderen Worten, das Prädikat `BETWEEN` könnte umgeschrieben werden:

```
<value> >= <value_1> AND <value> <= <value_2>
```

Wenn `BETWEEN` in den Suchbedingungen von DML-Abfragen verwendet wird, kann der Firebird-Optimierer einen Index für die durchsuchte Spalte verwenden, falls dieser verfügbar ist.

Beispiel

```
SELECT *
FROM EMPLOYEE
WHERE HIRE_DATE BETWEEN date '1992-01-01' AND CURRENT_DATE
```

`LIKE`

Verfügbar in

DSQL, PSQL, ESQL

Syntax

```
<match_value> [NOT] LIKE <pattern>
   [ESCAPE <escape character>]
```

```
<match_value>      ::= character-type expression
<pattern>          ::= search pattern
<escape character> ::= escape character
```

Das Prädikat LIKE vergleicht den zeichenartigen Ausdruck mit dem im zweiten Ausdruck definierten Muster. Die Groß-/Kleinschreibung oder Akzent-Sensitivität für den Vergleich wird durch die verwendete Kollatierung bestimmt. Bei Bedarf kann für jeden Operanden eine Kollatierung angegeben werden.

Wildcards

Zwei Wildcard-Zeichen sind für die Suche verfügbar:

- Das Prozentzeichen (%) berücksichtigt alle Sequenzen von null oder mehr Zeichen im getesteten Wert
- Das Unterstrichzeichen (_) berücksichtigt jedes beliebige Einzelzeichen im getesteten Wert

Wenn der getestete Wert dem Muster entspricht, unter Berücksichtigung von Wildcard-Zeichen ist das Prädikat TRUE.

Verwendung der ESCAPE-Zeichen-Option

Wenn der Such-String eines der Wildcard-Zeichen beinhaltet, kann die ESCAPE-Klausel verwendet werden, um ein Escape-Zeichen zu definieren. Das Escape-Zeichen muss dem ‘%’ oder ‘_’ Symbol im Suchstring vorangestellt werden, um anzuzeigen, dass das Symbol als wörtliches Zeichen interpretiert werden soll.

Beispiele für LIKE

1. Finde die Nummern der Abteilung, deren Namen mit dem Wort “Software” starten:

   ```
   SELECT DEPT_NO
   FROM DEPT
   WHERE DEPT_NAME LIKE 'Software%';
   ```

   Es ist möglich einen Index für das Feld DEPT_NAME zu verwenden, sofern dieser existiert.

   

   Über LIKE und den Optimizer

   Eigentlich verwendet das LIKE-Prädikat keinen Index. Wird das Prädikat jedoch in Form von `LIKE 'string%'` verwendet, wird dieses zum Prädikat STARTING WITH konvertiert, welches einen Index verwendet.

   Somit gilt — wenn Sie nach einem Wortanfang suchen, sollten Sie das Prädikat STARTING WITH anstelle von LIKE verwenden.

2. Suchen Sie nach Mitarbeitern, deren Namen aus 5 Buchstaben bestehen, mit den Buchstaben “Sm” beginnen und mit “th” enden. Das Prädikat gilt für Namen wie “Smith” und “Smyth”.

```
SELECT
  first_name
FROM
  employee
WHERE first_name LIKE 'Sm_th'
```

3. Suche nach allen Mandanten, deren Adresse den String “Rostov” enthält:

```
SELECT *
FROM CUSTOMER
WHERE ADDRESS LIKE '%Rostov%'
```

Benötigen Sie eine Suche, die Groß- und Kleinschreibung *innerhalb* einer Zeichenkette ignoriert (`LIKE '%Abc%'`), sollten Sie das `CONTAINING`-Prädikat, anstelle des `LIKE`-Prädikates, verwenden.

4. Suchen Sie nach Tabellen, die den Unterstrich im Namen enthalten. Als Escape-Zeichen wird das Zeichen ‘#’ verwendet:

```
SELECT
  RDB$RELATION_NAME
FROM RDB$RELATIONS
WHERE RDB$RELATION_NAME LIKE '%#_%' ESCAPE '#'
```

Siehe auch

`STARTING WITH`, `CONTAINING`, `SIMILAR TO`

`STARTING WITH`

Verfügbar in

DSQL, PSQL, ESQL

Syntax

```
<value> [NOT] STARTING WITH <value>
```

Das Prädikat `STARTING WITH` sucht nach einer Zeichenkette oder einem zeichenkettenähnlichen Datentyp, die mit den Zeichen des Argumentes *value* beginnt. Die Suche unterscheidet zwischen Groß- und Kleinschreibung.

Wenn `STARTING WITH` als Suchkriterium in DML-Abfragen verwendet wird, nutzt der Firebird-Optimizer einen Index auf der Suchspalte, sofern vorhanden.

Beispiel

Suche nach Mitarbeitern deren Namen mit “Jo” beginnen:

```
SELECT LAST_NAME, FIRST_NAME
FROM EMPLOYEE
WHERE LAST_NAME STARTING WITH 'Jo'
```

Siehe auch

LIKE

CONTAINING

Verfügbar in

DSQL, PSQL, ESQL

Syntax

```
<value> [NOT] CONTAINING <value>
```

Das Prädikat CONTAINING sucht nach einem String oder einem stringähnlichen Typ und sucht nach der Zeichenfolge, die seinem Argument entspricht. Es kann für eine alphanumerische (stringartige) Suche nach Zahlen und Datumsangaben verwendet werden. Bei einer CONTAINING-Suche wird die Groß-/Kleinschreibung nicht beachtet. Wenn jedoch eine akzentsensitive Sortierung verwendet wird, erfolgt die Suche akzentsensitiver.

Beispiele

1. Suche nach Projekten, deren Namen die Teilzeichenfolge “Map” enthalten:

   ```
   SELECT *
   FROM PROJECT
   WHERE PROJ_NAME CONTAINING 'Map';
   ```

   Zwei Zeilen mit den Namen “AutoMap” und “MapBrowser port” werden zurückgegeben.

2. Suche nach Änderungen in den Gehältern, die die Zahl 84 im Datum enthalten (in diesem Falle heißt dies, Änderungen im Jahr 1984):

   ```
   SELECT *
   FROM SALARY_HISTORY
   WHERE CHANGE_DATE CONTAINING 84;
   ```

Siehe auch

LIKE

SIMILAR TO

Verfügbar in

DSQL, PSQL

Syntax

```
string-expression [NOT] SIMILAR TO <pattern> [ESCAPE <escape-char>]

<pattern>     ::= an SQL regular expression
<escape-char> ::= a single character
```

`SIMILAR TO` findet eine Zeichenkette anhand eines Regulären Ausdruck-Musters in SQL (engl. SQL Regular Expression Pattern). Anders als in einigen anderen Sprachen muss das Muster mit der gesamten Zeichenkette übereinstimmen, um erfolgreich zu sein — die Übereinstimmung eines Teilstrings reicht nicht aus. Ist ein Operand `NULL`, ist auch das Ergebnis `NULL`. Andernfalls ist das Ergebnis `TRUE` oder `FALSE`.

Syntax: SQL Reguläre Ausdrücke

Die folgende Syntax definiert das SQL-Standardausdruckformat. Es ist eine komplette und korrekte Top-down-Definition. Es ist auch sehr formell, ziemlich lang und vermutlich perfekt geeignet, jeden zu entmutigen, der nicht schon Erfahrungen mit Regulären Ausdrücken (oder mit sehr formalen, eher langen Top-down-Definitionen) gesammelt hat. Fühlen Sie sich frei, dies zu überspringen und den nächsten Abschnitt, Aufbau Regulärer Ausdrücke, zu lesen, der einen Bottom-up-Ansatz verfolgt und sich an den Rest von uns richtet.

```
<regular expression> ::= <regular term> ['|' <regular term> ...]

<regular term> ::= <regular factor> ...

<regular factor> ::= <regular primary> [<quantifier>]

<quantifier> ::= ? | * | + | '{' <m> [,[<n>]] '}'

<m>, <n> ::= unsigned int, mit <m> <= <n> wenn beide vorhanden

<regular primary> ::=
    <character> | <character class> | %
  | (<regular expression>)

<character> ::= <escaped character> | <non-escaped character>

<escaped character> ::=
  <escape-char> <special character> | <escape-char> <escape-char>

<special character> ::= eines der Zeichen []()|^-+*%\?{}

<non-escaped character> ::=
  ein Zeichen, das nicht ein <special character> ist
  und nicht gleich <escape-char> (wenn definiert)_

<character class> ::=
    '' | '[' <member> ... ']' | '[^' <non-member> ... ']'
```

```
  | '[' <member> ... '^' <non-member> ... ']'

<member>, <non-member> ::= <character> | <range> | <predefined class>

<range> ::= <character>-<character>

<predefined class> ::= '[:' <predefined class name> ':]'

<predefined class name> ::=
  ALPHA | UPPER | LOWER | DIGIT | ALNUM | SPACE | WHITESPACE
```

Aufbau Regulärer Ausdrücke

Dieser Abschnitt behandelt die Elemente und Regeln zum Aufbau Regulärer Ausdrücke in SQL.

Zeichen

Innerhalb Regulärer Ausdrücke repräsentieren die meisten Zeichen sich selbst. Die einzige Ausnahme bilden die folgenden Zeichen:

```
[ ] ( ) | ^ - + * % _ ? { }
```

... und das Escape-Zeichen, sofern definiert.

Ein Regulärer Ausdruck, der keine Sonderzeichen oder Escape-Zeichen beinhaltet, findet nur Strings, die identisch zu sich selbst sind (abhängig von der verwendeten Collation). Das heißt, es agiert wie der '='-Operator:

```
'Apple' similar to 'Apple'  -- true
'Apples' similar to 'Apple' -- false
'Apple' similar to 'Apples' -- false
'APPLE' similar to 'Apple'  -- abhängig von der Collation
```

Wildcards

Die bekannten SQL-Wildcards '_' und '%' finden beliebige Einzelzeichen und Strings beliebiger Länge:

```
'Birne' similar to 'B_rne'   -- true
'Birne' similar to 'B_ne'    -- false
'Birne' similar to 'B%ne'    -- true
'Birne' similar to 'Bir%ne%' -- true
'Birne' similar to 'Birr%ne' -- false
```

Beachten Sie, wie '%' auch den leeren String berücksichtigt.

Zeichenklassen

Ein Bündel von Zeichen, die in Klammern eingeschlossen sind, definiert eine Zeichenklasse. Ein Zeichen in der Zeichenfolge entspricht einer Klasse im Muster, wenn das Zeichen Mitglied der Klasse ist:

```
'Citroen' similar to 'Cit[arju]oen'     -- true
'Citroen' similar to 'Ci[tr]oen'        -- false
'Citroen' similar to 'Ci[tr][tr]oen'    -- true
```

Wie aus der zweiten Zeile ersichtlich ist, entspricht die Klasse nur einem einzigen Zeichen, nicht einer Sequenz.

Innerhalb einer Klassendefinition definieren zwei Zeichen, die durch einen Bindestrich verbunden sind, einen Bereich. Ein Bereich umfasst die beiden Endpunkte und alle Zeichen, die zwischen ihnen in der aktiven Sortierung liegen. Bereiche können überall in der Klassendefinition ohne spezielle Begrenzer platziert werden, um sie von den anderen Elementen zu trennen.

```
'Datte' similar to 'Dat[q-u]e'          -- true
'Datte' similar to 'Dat[abq-uy]e'       -- true
'Datte' similar to 'Dat[bcg-km-pwz]e'   -- false
```

Vordefinierte Zeichenklassen

Die folgenden vordefinierten Zeichenklassen können auch in einer Klassendefinition verwendet werden:

`[:ALPHA:]`

Lateinische Buchstaben a..z und A..Z. Mit einer akzentunempfindlichen Sortierung stimmt diese Klasse auch mit akzentuierten Formen dieser Zeichen überein.

`[:DIGIT:]`

Dezimalziffern 0..9.

`[:ALNUM:]`

Gesamtheit aus `[:ALPHA:]` und `[:DIGIT:]`.

`[:UPPER:]`

Großgeschriebene Form der lateinischen Buchstaben A..Z. Findet auch kleingeschriebene Strings mit groß- und kleinschreibunempfindlicher Collation sowie akzentunempfindlicher Collation.

`[:LOWER:]`

Kleingeschriebene Form der lateinischen Buchstaben A..Z. Findet auch großgeschriebene Strings mit groß- und kleinschreibunempfindlicher Collation sowie akzentunempfindlicher Collation.

`[:SPACE:]`

Findet das Leerzeichen (ASCII 32).

`[:WHITESPACE:]`

Findet horizontalen Tabulator (ASCII 9), Zeilenvorschub (ASCII 10), vertikalen Tabulator (ASCII 11), Seitenvorschub (ASCII 12), Wagenrücklauf (ASCII 13) und Leerzeichen (ASCII 32).

Das Einbinden einer vordefinierten Klasse hat den gleichen Effekt wie das Einbinden all seiner Mitglieder. Vordefinierte Klassen sind nur in Klassendefinitionen erlaubt. Wenn Sie gegen eine vordefinierte Klasse prüfen und gegen nichts sonst, platzieren Sie ein zusätzliches Paar von Klammern um sie herum.

```
'Erdbeere' similar to 'Erd[[:ALNUM:]]eere'     -- true
'Erdbeere' similar to 'Erd[[:DIGIT:]]eere'     -- false
'Erdbeere' similar to 'Erd[a[:SPACE:]b]eere'   -- true
'Erdbeere' similar to [[:ALPHA:]]              -- false
'E'        similar to [[:ALPHA:]]              -- true
```

Wenn eine Klassendefinition mit einem Caret-Zeichen beginnt, wird alles, was folgt, aus der Klasse ausgeschlossen. Alle anderen Zeichen stimmen überein:

```
'Framboise' similar to 'Fra[^ck-p]boise'       -- false
'Framboise' similar to 'Fr[^a][^a]boise'       -- false
'Framboise' similar to 'Fra[^[:DIGIT:]]boise'  -- true
```

If the caret is not placed at the start of the sequence, the class contains everything before the caret, except for the elements that also occur after the caret:

```
'Grapefruit' similar to 'Grap[a-m^f-i]fruit'   -- true
'Grapefruit' similar to 'Grap[abc^xyz]fruit'   -- false
'Grapefruit' similar to 'Grap[abc^de]fruit'    -- false
'Grapefruit' similar to 'Grap[abe^de]fruit'    -- false

'3' similar to '[[:DIGIT:]^4-8]'               -- true
'6' similar to '[[:DIGIT:]^4-8]'               -- false
```

Zuletzt sei noch erwähnt, dass die Wildcard-Zeichen ‘_’ eine eigene Zeichenklasse sind, die einem beliebigen einzelnen Zeichen entspricht.

Bezeichner

Ein Fragezeichen, direkt von einem weiteren Zeichen oder Klasse gefolgt, gibt an, dass das folgende Element gar nicht oder einmalig vorkommen darf:

```
'Hallon' similar to 'Hal?on'                   -- false
'Hallon' similar to 'Hal?lon'                  -- true
'Hallon' similar to 'Halll?on'                 -- true
'Hallon' similar to 'Hallll?on'                -- false
'Hallon' similar to 'Halx?lon'                 -- true
```

```
'Hallon' similar to 'H[a-c]?llon[x-z]?'        -- true
```

Ein Sternchen (‘*’) unmittelbar nach einem Zeichen oder einer Klasse zeigt an, dass das vorangehende Element 0-mal oder öfter vorkommen kann, damit es übereinstimmt:

```
'Icaque' similar to 'Ica*que'                  -- true
'Icaque' similar to 'Icar*que'                 -- true
'Icaque' similar to 'I[a-c]*que'               -- true
'Icaque' similar to '_*'                       -- true
'Icaque' similar to '[[:ALPHA:]]*'             -- true
'Icaque' similar to 'Ica[xyz]*e'               -- false
```

Ein Pluszeichen (‘+’) unmittelbar nach einem Zeichen oder einer Klasse gibt an, dass das vorangehende Element mindestens einmal vorkommen muss, damit es übereinstimmt:

```
'Jujube' similar to 'Ju_+'                     -- true
'Jujube' similar to 'Ju+jube'                  -- true
'Jujube' similar to 'Jujuber+'                 -- false
'Jujube' similar to 'J[jux]+be'                -- true
'Jujube' sililar to 'J[[:DIGIT:]]+ujube'       -- false
```

Wenn auf ein Zeichen oder eine Klasse eine Zahl in geschweiften Klammern folgt (‘{’ und ‘}’), muss sie genau so oft wiederholt werden, damit sie übereinstimmt:

```
'Kiwi' similar to 'Ki{2}wi'                    -- false
'Kiwi' similar to 'K[ipw]{2}i'                 -- true
'Kiwi' similar to 'K[ipw]{2}'                  -- false
'Kiwi' similar to 'K[ipw]{3}'                  -- true
```

Wenn der Zahl ein Komma folgt (‘,’), muss das Element mindestens so oft wiederholt werden, damit es übereinstimmt:

```
'Limone' similar to 'Li{2,}mone'               -- false
'Limone' similar to 'Li{1,}mone'               -- true
'Limone' similar to 'Li[nezom]{2,}'            -- true
```

Wenn die geschweiften Klammern zwei durch ein Komma getrennte Zahlen enthalten, wobei die zweite Zahl nicht kleiner als die erste ist, muss das Element mindestens die erste Zahl und höchstens die zweite Zahl wiederholt werden, um zu entsprechen:

```
'Mandarijn' similar to 'M[a-p]{2,5}rijn'       -- true
'Mandarijn' similar to 'M[a-p]{2,3}rijn'       -- false
'Mandarijn' similar to 'M[a-p]{2,3}arijn'      -- true
```

Die Bezeichner ‘?’, ‘*’ und ‘+’ sind Kurzschreibweisen für `{0,1}`, `{0,}` und `{1,}`.

Oder-verknüpfte Terme

Reguläre Ausdrücke können Oder-verknüpft werden mittels ‘|’-Operator. Eine Gesamtübereinstimmung tritt auf, wenn die Argumentzeichenkette mit mindestens einem Term übereinstimmt.

```
'Nektarin' similar to 'Nek|tarin'              -- false
'Nektarin' similar to 'Nektarin|Persika'       -- true
'Nektarin' similar to 'M_+|N_+|P_+'            -- true
```

Unterausdrücke

Ein oder mehrere Teile der regulären Ausdrücke können in Unterausdrücke gruppiert werden (auch Untermuster genannt), indem diese in runde Klammern eingeschlossen werden. Ein Unterausdruck ist ein eigener regulärer Ausdruck. Dieser kann alle erlaubten Elemente eines regulären Ausdrucks enthalten, und auch eigene Bezeichner.

```
'Orange' similar to 'O(ra|ri|ro)nge'           -- true
'Orange' similar to 'O(r[a-e])+nge'            -- true
'Orange' similar to 'O(ra){2,4}nge'            -- false
'Orange' similar to 'O(r(an|in)g|rong)?e'      -- true
```

Sonderzeichen escapen

Um mit einem Sonderzeichen in regulären Ausdrücken abzugleichen, muss dieses Zeichen mit Escapezeichen versehen werden. Es gibt kein Standard-Escape-Zeichen; Stattdessen gibt der Benutzer bei Bedarf eine an:

```
'Peer (Poire)' similar to 'P[^ ]+ \(P[^ ]+\)' escape '\'    -- true
'Pera [Pear]'  similar to 'P[^ ]+ #[P[^ ]+#]' escape '#'    -- true
'Päron-äppledryck' similar to 'P%$-ä%' escape '$'           -- true
'Pärondryck' similar to 'P%--ä%' escape '-'                 -- false
```

Die letzte Zeile demonstriert, dass das Escape-Zeichen auch sich selbst escapen kann, wenn notwendig.

`IS [NOT] DISTINCT FROM`

Verfügbar in

DSQL, PSQL

Syntax

```
<operand1> IS [NOT] DISTINCT FROM <operand2>
```

Zwei Operanden werden als *DISTINCT* angesehen, wenn sie unterschiedliche Werte besitzen oder wenn einer `NULL` ist und der andere nicht-`NULL`. Sie werden als *NOT DISTINCT* angesehen, wenn sie den gleichen Wert besitzen oder beide Operanden `NULL` sind.

`IS [NOT] DISTINCT FROM` liefert immer TRUE oder FALSE und niemals UNKNOWN (NULL) (unbekannter Wert). Die Operatoren '=' und '<>' geben umgekehrt UNKNOWN (NULL) zurück, wenn einer oder beide Operanden NULL sind.

Tabelle 19. Ergebnisse verschiedener Vergleichsprädikate

Operandenwerte	**Ergebnis verschiedener Prädikate**			
	`=`	`IS NOT DISTINCT FROM`	`<>`	`IS DISTINCT FROM`
Gleiche Werte	`TRUE`	`TRUE`	`FALSE`	`FALSE`
Verschiedene Werte	`FALSE`	`FALSE`	`TRUE`	`TRUE`
Beide `NULL`	`UNKNOWN`	`TRUE`	`UNKNOWN`	`FALSE`
Einer `NULL`, einer nicht-`NULL`	`UNKNOWN`	`FALSE`	`UNKNOWN`	`TRUE`

Beispiele

```
SELECT ID, NAME, TEACHER
FROM COURSES
WHERE START_DAY IS NOT DISTINCT FROM END_DAY;

-- PSQL-Fragment
IF (NEW.JOB IS DISTINCT FROM OLD.JOB)
THEN POST_EVENT 'JOB_CHANGED';
```

Siehe auch

`IS [NOT] NULL`, Boolesches `IS [NOT]`

Boolesches `IS [NOT]`

Verfügbar in

DSQL, PSQL

Syntax

```
<value> IS [NOT] { TRUE | FALSE | UNKNOWN }
```

Das IS-Prädikat mit booleschen Literalwerten prüft, ob der Ausdruck auf der linken Seite mit dem booleschen Wert auf der rechten Seite übereinstimmt. Der Ausdruck auf der linken Seite muss vom Typ `BOOLEAN` sein, sonst kommt es zu einer Ausnahme.

Das `IS [NOT] UNKNOWN` entspricht `IS [NOT] NULL`.

Die rechte Seite des Prädikats akzeptiert nur die Literale `TRUE`, `FALSE` und `UNKNOWN`

(und NULL). Es akzeptiert keine Ausdrücke.

Verwenden des IS-Prädikats mit einem booleschen Datentyp

```
-- FALSE-Wert prüfen
SELECT * FROM TBOOL WHERE BVAL IS FALSE;

ID            BVAL
============= =======
2             <false>

-- UNKNOWN-Wert prüfen
SELECT * FROM TBOOL WHERE BVAL IS UNKNOWN;

ID            BVAL
============= =======
3             <null>
```

Siehe auch

IS [NOT] NULL

IS [NOT] NULL

Verfügbar in

DSQL, PSQL, ESQL

Syntax

```
<value> IS [NOT] NULL
```

Da NULL kein Wert ist, sind diese Operatoren keine Vergleichsoperatoren. Das Prädikat IS [NOT] NULL prüft die Behauptung, dass der Ausdruck auf der linken Seite einen Wert (*IS NOT NULL*) oder keinen Wert hat (*IS NULL*).

Beispiel

Suche nach Verkäufen, die kein Versanddatum besitzen:

```
SELECT * FROM SALES
WHERE SHIP_DATE IS NULL;
```

Hinweis bezüglich des IS-Prädikates

Bis einschließlich Firebird 2.5, hat das Prädikat IS, wie andere Vergleichsprädikate, keinen Vorrang gegenüber anderer. Ab Firebird 3.0 hat dieses Prädikat Vorrang gegenüber den anderen.

4.2.3. Existenzprädikate

Diese Gruppe von Prädikaten umfasst diejenigen, die Unterabfragen verwenden, um Werte für alle Arten von Zusicherungen in Suchbedingungen zu übermitteln. Existenzielle Prädikate werden so genannt, weil sie verschiedene Methoden verwenden, um auf *existence* oder *non-existence* einer Bedingung zu testen, und `TRUE` zurückgeben, wenn die Existenz oder Nichtexistenz bestätigt wird oder `FALSE` andernfalls.

`EXISTS`

Verfügbar in

DSQL, PSQL, ESQL

Syntax

```
[NOT] EXISTS (<select_stmt>)
```

Das Prädikat `EXISTS` verwendet als Argument einen Unterabfrageausdruck. Es gibt `TRUE` zurück, wenn das Ergebnis der Unterabfrage mindestens eine Zeile enthalten würde; andernfalls gibt es `FALSE` zurück.

`NOT EXISTS` gibt `FALSE` zurück, wenn das Ergebnis der Unterabfrage mindestens eine Zeile enthalten würde; andernfalls gibt es `TRUE` zurück.

Die Unterabfrage kann mehrere Spalten enthalten, oder `SELECT *`, da die Prüfung anhand der zurückgegebenen Datenzeilen vorgenommen wird, die die Bedingungen erfüllen.

Beispiele

1. Finde die Mitarbeiter, die Projekte haben.

```
SELECT *
FROM employee
WHERE EXISTS(SELECT *
             FROM  employee_project ep
             WHERE ep.emp_no = employee.emp_no)
```

2. Finde die Mitarbeiter, die keine Projekte haben.

```
SELECT *
FROM employee
WHERE NOT EXISTS(SELECT *
                 FROM employee_project ep
                 WHERE ep.emp_no = employee.emp_no)
```

IN

Verfügbar in

DSQL, PSQL, ESQL

Syntax

```
<value> [NOT] IN (<select_stmt> | <value_list>)

<value_list> ::= <value_1> [, <value_2> ⋯]
```

Das Prädikat `IN` prüft, ob der Wert des Ausdrucks auf der linken Seite im Wertesatz der rechten Seite vorkommt. Der Wertesatz darf nicht mehr als 1500 Elemente enthalten. Das `IN`-Prädikat kann mit folgender äquivalenter Form ersetzt werden:

```
(<value> = <value_1> [OR <value> = <value_2> ⋯])

<value> = { ANY | SOME } (<select_stmt>)
```

Wenn das Prädikat `IN` als Suchbedingung in DML-Abfragen verwendet wird, kann der Firebird-Optimizer einen Index auf die Suchspalte nutzen, sofern einer vorhanden ist.

In seiner zweiten Form prüft das Prädikat `IN`, ob der linke Ausdruckswert im Ergebnis der Unterabfrage vorhanden ist (oder nicht vorhanden, wenn `NOT IN` verwendet wird).

Die Unterabfrage darf nur eine Spalte abfragen, andernfalls wird es zum Fehler “count of column list and variable list do not match” kommen.

Abfragen, die das Prädikat `IN` mit einer Unterabfrage verwenden, können durch eine ähnliche Abfrage mittels des `EXISTS`-Prädikates ersetzt werden. Zum Beispiel folgende Abfrage:

```
SELECT
  model, speed, hd
FROM PC
WHERE
model IN (SELECT model
          FROM product
          WHERE maker = 'A');
```

kann ersetzt werden mittels EXISTS-Prädikat:

```
SELECT
  model, speed, hd
FROM PC
WHERE
 EXISTS (SELECT *
         FROM product
         WHERE maker = 'A'
```

```
            AND product.model = PC.model);
```

Jedoch gilt zu beachten, dass eine Abfrage mittels `NOT IN` und einer Unterabfrage nicht immer das gleiche Ergebnis zurückliefert wie sein Gegenpart mit `NOT EXISTS`. Dies liegt daran, dass `EXISTS` immer TRUE oder FALSE zurückgibt, wohingegen `IN` `NULL` in diesen beiden Fällen zurückliefert:

a. wenn der geprüfte Wert `NULL` ist und die `IN ()`-Liste nicht leer ist
b. wenn der geprüfte Wert keinen Treffer in der `IN ()`-Liste enthält und mindestens ein Element `NULL` ist.

Nur in diesen beiden Fällen wird `IN () NULL` zurückgeben, während das `EXISTS`-Prädikat `FALSE` zurückgibt ('keine passende Zeile gefunden', engl. 'no matching row found'). In einer Suche oder, zum Beispiel in einem `IF (…)`-Statement, bedeuten beide Ergebnisse einen “Fehler” und es macht damit keinen Unterschied.

Aber für die gleichen Daten gibt `NOT IN () NULL` zurück, während `NOT EXISTS TRUE` zurückgibt, was das Gegenteilige Ergebnis ist.

Schauen wir uns das folgendes Beispiel an:

```
-- Suche nach Bürgern die nicht am gleichen Tag wie eine
-- berühmte New Yorker Persönlichkeit geboren wurden
SELECT P1.name AS NAME
FROM Personnel P1
WHERE P1.birthday NOT IN (SELECT C1.birthday
                          FROM Celebrities C1
                          WHERE C1.birthcity = 'New York');
```

Nehmen wir nun an, dass die Liste der New Yorker Berühmtheiten nicht leer ist und mindestens einen NULL-Geburtstag aufweist. Dann gilt für alle Bürger, die nicht am gleichen Tag mit einer Berühmtheit Geburtstag haben, dass `NOT IN NULL` zurückgibt, da dies genau das ist was `IN` tut. Die Suchbedingung wurde nicht erfüllt und die Bürger werden nicht im Ergebnis des `SELECT` berücksichtigt, da die Aussage falsch ist.

Bürger, die am gleichen Tag wie eine Berühmtheit Geburtstag feiern, wird `NOT IN` korrekterweise `FALSE` zurückgeben, womit diese ebenfalls aussortiert werden, und damit keine Zeile zurückgegeben wird.

Wird die Form `NOT EXISTS` verwendet:

```
-- Suche nach Bürgern, die nicht am gleichen Tag wie eine
-- berühmte New Yorker Persönlichkeit geboren wurden
SELECT P1.name AS NAME
FROM Personnel P1
WHERE NOT EXISTS (SELECT *
                  FROM Celebrities C1
                  WHERE C1.birthcity = 'New York'
```

```
                    AND C1.birthday = P1.birthday);
```

nicht-Übereinstimmungen werden im `NOT EXISTS`-Ergebnis `TRUE` erhalten und ihre Datensätze landen im Rückgabesatz.

Wenn bei der Suche nach einer Nichtübereinstimmung die Möglichkeit besteht, dass `NULL` gefunden wird, sollten Sie `NOT EXISTS` verwenden.

Beispiele für die Verwendung

1. Finde Mitarbeiter mit den Namen “Pete”, “Ann” und “Roger”:

```
SELECT *
FROM EMPLOYEE
WHERE FIRST_NAME IN ('Pete', 'Ann', 'Roger');
```

2. Finde alle Computer, die deren Hersteller mit dem Buchstaben “A” beginnt:

```
SELECT
  model, speed, hd
FROM PC
WHERE
  model IN (SELECT model
            FROM product
            WHERE maker STARTING WITH 'A');
```

Siehe auch

EXISTS

SINGULAR

Verfügbar in

DSQL, PSQL, ESQL

Syntax

```
[NOT] SINGULAR (<select_stmt>)
```

Das Prädikat `SINGULAR` nimmt eine Unterabfrage als Argument und wertet sie als TRUE, wenn die Unterabfrage genau eine Ergebniszeile zurückgibt; andernfalls wird das Prädikat als FALSE ausgewertet. Die Unterabfrage kann mehrere Ausgabespalten auflisten, da die Zeilen sowieso nicht zurückgegeben werden. Sie werden nur auf (singuläre) Existenz geprüft. Der Kürze halber wird normalerweise ‘`SELECT *`’ angegeben. Das Prädikat `SINGULAR` kann nur zwei Werte zurückgeben: `TRUE` oder `FALSE`.

Beispiel

Finden Sie die Mitarbeiter, die nur ein Projekt haben.

```
SELECT *
FROM employee
WHERE SINGULAR(SELECT *
               FROM employee_project ep
               WHERE ep.emp_no = employee.emp_no)
```

4.2.4. Quantifizierte Unterabfrage-Prädikate

Ein Quantifizierer ist ein logischer Operator, der die Anzahl der Objekte festlegt, für die diese Behauptung wahr ist. Es ist keine numerische Größe, sondern eine logische, die die Behauptung mit dem vollen Satz möglicher Objekte verbindet. Solche Prädikate basieren auf logischen universellen und existentiellen Quantifizierern, die in der formalen Logik erkannt werden.

In Unterabfrageausdrücken ermöglichen quantifizierte Prädikate den Vergleich einzelner Werte mit den Ergebnissen von Unterabfragen; sie haben die folgende gemeinsame Form:

```
<value expression> <comparison operator> <quantifier> <subquery>
```

ALL

Verfügbar in

DSQL, PSQL, ESQL

Syntax

```
<value> <op> ALL (<select_stmt>)
```

Wenn der ALL-Quantifizierer verwendet wird, ist das Prädikat TRUE, wenn jeder Wert, der von der Unterabfrage zurückgegeben wird, die Bedingung des Prädikates in der Hauptabfrage erfüllt ist.

Beispiel

Zeige nur jene Kunden an, deren Bewertungen höher sind als die Bewertung jedes Kunden in Paris.

```
SELECT c1.*
FROM Customers c1
WHERE c1.rating > ALL
      (SELECT c2.rating
       FROM Customers c2
       WHERE c2.city = 'Paris')
```

Wenn die Unterabfrage einen leeren Satz zurückgibt, ist das Prädikat TRUE für jeden linken Wert, unabhängig vom Operator. Dies mag widersprüchlich erscheinen, denn jeder linke Wert wird gegenüber dem rechten betrachtet als: kleiner als, größer als, gleich sowie ungleich.

Dennoch passt dies perfekt in die formale Logik: Wenn der Satz leer ist, ist das

> Prädikat 0 mal wahr, d.h. für jede Zeile im Satz.

ANY and SOME

Verfügbar in

DSQL, PSQL, ESQL

Syntax

```
<value> <op> {ANY | SOME} (<select_stmt>)
```

Die Quantifizierer ANY und SOME sind in ihrem Verhalten identisch. Offensichtlich sind beide im SQL-Standard vorhanden, so dass sie austauschbar verwendet werden können, um die Lesbarkeit der Operatoren zu verbessern. Wird der ANY- oder SOME-Quantifizierer verwendet, ist das Prädikat TRUE, wenn einer der zurückgegebenen Werte der Unterabfrage die Suchbedingung der Hauptabfrage erfüllt. Gibt die Unterabfrage keine Zeile zurück, wird das Prädikat automtisch als FALSE angesehen.

Beispiel

Zeige nur die Kunden, deren Bewertungen höher sind als die eines oder mehrerer Kunden in Rom.

```
SELECT *
FROM Customers
WHERE rating > ANY
      (SELECT rating
       FROM Customers
       WHERE city = 'Rome')
```

Kapitel 5. Anweisungen der Datendefinitionssprache (DDL)

DDL ist die Teilmenge der Datendefinitionssprache der SQL-Sprache von Firebird. DDL-Anweisungen werden verwendet, um von Benutzern erstellte Datenbankobjekte zu erstellen, zu ändern und zu löschen. Wenn eine DDL-Anweisung festgeschrieben wird, werden die Metadaten für das Objekt erstellt, geändert oder gelöscht.

5.1. DATABASE

In diesem Abschnitt wird beschrieben, wie Sie eine Datenbank erstellen, eine Verbindung zu einer vorhandenen Datenbank herstellen, die Dateistruktur einer Datenbank ändern und eine Datenbank löschen. Es erklärt auch, wie Sie eine Datenbank auf zwei ganz unterschiedliche Weisen sichern und wie Sie die Datenbank in den "kopiersicheren" Modus schalten, um eine externe Sicherung sicher durchzuführen.

5.1.1. CREATE DATABASE

Verwendet für

Erstellen einer neuen Datenbank

Verfügbar in

DSQL, ESQL

Syntax

```
CREATE {DATABASE | SCHEMA} <filespec>
  [<db_initial_option> [<db_initial_option> ...]]
  [<db_config_option> [<db_config_option> ...]]

<db_initial_option> ::=
    USER username
  | PASSWORD 'password'
  | ROLE rolename
  | PAGE_SIZE [=] size
  | LENGTH [=] num [PAGE[S]]
  | SET NAMES 'charset'

<db_config_option> ::=
    DEFAULT CHARACTER SET default_charset
      [COLLATION collation] -- not supported in ESQL
  | <sec_file>
  | DIFFERENCE FILE 'diff_file' -- not supported in ESQL

<filespec> ::= "'" [server_spec]{filepath | db_alias} "'"

<server_spec> ::=
    host[/{port | service}]:
```

```
  | \\host\
  | <protocol>://[host[:{port | service}]/]

<protocol> ::= inet | inet4 | inet6 | wnet | xnet

<sec_file> ::=
  FILE 'filepath'
  [LENGTH [=] num [PAGE[S]]
  [STARTING [AT [PAGE]] pagenum]
```

Jede *db_initial_option* und *db_config_option* kann höchstens einmal vorkommen, außer *sec_file*, die null oder mehrmals vorkommen kann.

Tabelle 20. `CREATE DATABASE`-*Anweisungsparameter*

Parameter	Beschreibung
filespec	Dateispezifikation für primäre Datenbankdatei
server_spec	Spezifikation des Remote-Servers. Einige Protokolle erfordern die Angabe eines Hostnamens. Enthält optional eine Portnummer oder einen Dienstnamen. Erforderlich, wenn die Datenbank auf einem Remoteserver erstellt wird.
filepath	Vollständiger Pfad und Dateiname einschließlich seiner Erweiterung. Der Dateiname muss gemäß den Regeln des verwendeten Plattform-Dateisystems angegeben werden.
db_alias	Datenbankalias, der zuvor in der Datei `databases.conf` erstellt wurde
host	Hostname oder IP-Adresse des Servers, auf dem die Datenbank erstellt werden soll
port	Die Portnummer, auf der der Remote-Server lauscht (Parameter *RemoteServicePort* in der Datei `firebird.conf`)
service	Dienstname. Muss mit dem Parameterwert von *RemoteServiceName* in der Datei `firebird.conf` übereinstimmen
username	Benutzername des Besitzers der neuen Datenbank. Er kann aus bis zu 31 Zeichen bestehen. Der Benutzername kann wahlweise in einfache oder doppelte Anführungszeichen eingeschlossen werden. Wenn ein Benutzername in doppelte Anführungszeichen eingeschlossen ist, muss die Groß-/Kleinschreibung entsprechend den Regeln für Bezeichner in Anführungszeichen beachtet werden. Wenn es in einfache Anführungszeichen eingeschlossen ist, verhält es sich so, als ob der Wert ohne Anführungszeichen angegeben wurde. Der Benutzer muss ein Administrator sein oder die Berechtigung `CREATE DATABASE` besitzen.
password	Passwort des Benutzers als Datenbankbesitzer. Bei Verwendung des Authentifizierungs-Plugins `Legacy_Auth` werden nur die ersten 8 Zeichen verwendet. Groß-/Kleinschreibung beachten

Parameter	Beschreibung
rolename	Der Name der Rolle, deren Rechte beim Anlegen einer Datenbank berücksichtigt werden sollen. Der Rollenname kann in einfache oder doppelte Anführungszeichen eingeschlossen werden. Wenn der Rollenname in doppelte Anführungszeichen eingeschlossen ist, muss die Groß-/Kleinschreibung entsprechend den Regeln für Bezeichner in Anführungszeichen beachtet werden. Wenn es in einfache Anführungszeichen eingeschlossen ist, verhält es sich so, als ob der Wert ohne Anführungszeichen angegeben wurde.
size	Seitengröße für die Datenbank in Byte. Mögliche Werte sind 4096, 8192 und 16384. Die Standardseitengröße ist 8192.
num	Maximale Größe der primären Datenbankdatei oder einer sekundären Datei in Seiten
charset	Gibt den Zeichensatz der Verbindung an, die einem Client zur Verfügung steht, nachdem die Datenbank erfolgreich erstellt wurde. Einzelne Anführungszeichen sind erforderlich.
default_charset	Gibt den Standardzeichensatz für Zeichenfolgendatentypen an
collation	Standardsortierung für den Standardzeichensatz
sec_file	Dateispezifikation für eine Sekundärdatei
pagenum	Anfangsseitennummer für eine sekundäre Datenbankdatei
diff_file	Dateipfad und Name für Differenzdateien (.delta-Dateien) für den Sicherungsmodus

Die Anweisung `CREATE DATABASE` erstellt eine neue Datenbank. Sie können `CREATE DATABASE` oder `CREATE SCHEMA` verwenden. Sie sind synonym, aber wir empfehlen immer `CREATE DATABASE` zu verwenden, da sich dies in einer zukünftigen Version von Firebird ändern kann.

Eine Datenbank kann aus einer oder mehreren Dateien bestehen. Die erste (Haupt-)Datei wird als *Primärdatei* bezeichnet, nachfolgende Dateien werden als *Sekundärdatei(en)* bezeichnet.

Mehrdatei-Datenbanken

Heutzutage gelten Multi-File-Datenbanken als Anachronismus. Es war sinnvoll, Multi-File-Datenbanken auf alten Dateisystemen zu verwenden, bei denen die Größe einer Datei begrenzt ist. Sie können beispielsweise auf FAT32 keine Datei erstellen, die größer als 4 GB ist.

Die primäre Dateispezifikation ist der Name der Datenbankdatei und ihre Erweiterung mit dem vollständigen Pfad zu ihr gemäß den Regeln des verwendeten Dateisystems der OS-Plattform. Die Datenbankdatei darf zum Zeitpunkt der Datenbankerstellung noch nicht vorhanden sein. Wenn sie vorhanden ist, erhalten Sie eine Fehlermeldung und die Datenbank wird nicht erstellt.

Wenn kein vollständiger Pfad zur Datenbank angegeben ist, wird die Datenbank in einem der Systemverzeichnisse erstellt. Das jeweilige Verzeichnis hängt vom Betriebssystem ab. Geben Sie daher beim Erstellen einer Datenbank immer entweder den absoluten Pfad oder einen *Alias* an, es

sei denn, Sie haben einen triftigen Grund, diese Situation zu bevorzugen.

Einen Datenbankalias verwenden

Sie können Aliasse anstelle des vollständigen Pfads zur primären Datenbankdatei verwenden. Aliase werden in der Datei `databases.conf` im folgenden Format definiert:

```
alias = filepath
```

Die Ausführung einer `CREATE DATABASE`-Anweisung erfordert besondere Überlegungen in der Client-Anwendung oder dem Datenbanktreiber. Daher ist es nicht immer möglich, eine `CREATE DATABASE`-Anweisung auszuführen. Einige Treiber bieten andere Möglichkeiten zum Erstellen von Datenbanken. Jaybird stellt zum Beispiel die Klasse `org.firebirdsql.management.FBManager` bereit, um programmgesteuert eine Datenbank zu erstellen.

Bei Bedarf können Sie jederzeit auf *isql* zurückgreifen, um eine Datenbank zu erstellen.

Erstellen einer Datenbank auf einem Remote-Server

Wenn Sie eine Datenbank auf einem Remote-Server erstellen, müssen Sie die Remote-Server-Spezifikation angeben. Die Spezifikation des Remote-Servers hängt vom verwendeten Protokoll ab. Wenn Sie das TCP/IP-Protokoll verwenden, um eine Datenbank zu erstellen, sollte die primäre Dateispezifikation wie folgt aussehen:

```
host[/{port|service}]:{filepath | db_alias}
```

Wenn Sie das Named Pipes-Protokoll verwenden, um eine Datenbank auf einem Windows-Server zu erstellen, sollte die primäre Dateispezifikation so aussehen:

```
\\host\{filepath | db_alias}
```

Seit Firebird 3.0 gibt es auch eine einheitliche URL-ähnliche Syntax für die Remote-Server-Spezifikation. In dieser Syntax gibt der erste Teil den Namen des Protokolls an, dann einen Hostnamen oder eine IP-Adresse, eine Portnummer und einen Pfad der primären Datenbankdatei oder einen Alias.

Als Protokoll können folgende Werte angegeben werden:

inet

TCP/IP (versucht zuerst eine Verbindung über das IPv6-Protokoll herzustellen, wenn dies fehlschlägt, dann IPv4)

inet4

TCP/IP v4 (seit Firebird 3.0.1)

inet6

TCP/IP v6 (seit Firebird 3.0.1)

wnet

NetBEUI oder Named Pipes Protocol

xnet

lokales Protokoll (enthält keinen Host-, Port- und Servicenamen)

```
<protocol>://[host[:{port | service}]/]{filepath | db_alias}
```

Optionale Parameter für `CREATE DATABASE`

`USER` und `PASSWORD`

Klauseln zur Angabe des Benutzernamens bzw. des Passworts eines bestehenden Benutzers in der Sicherheitsdatenbank (`security3.fdb` oder was auch immer in der *SecurityDatabase* Konfiguration konfiguriert ist). Sie müssen den Benutzernamen und das Kennwort nicht angeben, wenn die Umgebungsvariablen `ISC_USER` und `ISC_PASSWORD` gesetzt sind. Der beim Erstellen der Datenbank angegebene Benutzer ist der Eigentümer. Dies ist wichtig, wenn Sie Datenbank- und Objektberechtigungen berücksichtigen.

`ROLE`

Die `ROLE`-Klausel gibt den Namen der Rolle an (normalerweise `RDB$ADMIN`), die beim Erstellen der Datenbank berücksichtigt wird. Die Rolle muss dem Benutzer in der entsprechenden Sicherheitsdatenbank zugewiesen werden.

`PAGE_SIZE`

Klausel zum Angeben der Datenbankseitengröße. Diese Größe wird für die Primärdatei und alle Sekundärdateien der Datenbank festgelegt. Wenn Sie eine Datenbankseitengröße von weniger als 4.096 angeben, wird sie automatisch auf 4.096 aufgerundet. Andere Werte, die nicht gleich 4.096, 8.192 oder 16.384 sind, werden auf den nächst kleineren unterstützten Wert geändert. Wenn die Datenbankseitengröße nicht angegeben wird, wird sie auf den Standardwert 8.192 gesetzt.

`LENGTH`

Klausel, die die maximale Größe der primären oder sekundären Datenbankdatei in Seiten angibt. Wenn eine Datenbank erstellt wird, belegen ihre Primär- und Sekundärdateien die minimale Anzahl von Seiten, die zum Speichern der Systemdaten erforderlich sind, unabhängig von dem in der `LENGTH`-Klausel angegebenen Wert. Der Wert `LENGTH` hat keinen Einfluss auf die Größe der einzigen (oder letzten, in einer Datenbank mit mehreren Dateien) Datei. Die Datei wird bei Bedarf automatisch weiter vergrößert.

`SET NAMES`

Klausel, die den Zeichensatz der verfügbaren Verbindung angibt, nachdem die Datenbank erfolgreich erstellt wurde. Standardmäßig wird der Zeichensatz `NONE` verwendet. Beachten Sie, dass der Zeichensatz in ein Paar Apostrophe (einfache Anführungszeichen) eingeschlossen werden sollte.

`DEFAULT CHARACTER SET`

Klausel, die den Standardzeichensatz zum Erstellen von Datenstrukturen von Zeichenfolgendatentypen angibt. Zeichensätze werden für die Datentypen `CHAR`, `VARCHAR` und `BLOB SUB_TYPE TEXT` verwendet. Standardmäßig wird der Zeichensatz `NONE` verwendet. Es ist auch möglich, die Standard-"COLLATION" für den Standardzeichensatz anzugeben, wodurch diese Kollatierungssequenz zum Standard für den Standardzeichensatz wird. Der Standardwert wird für die gesamte Datenbank verwendet, es sei denn, ein alternativer Zeichensatz mit oder ohne festgelegter Sortierung wird explizit für ein Feld, eine Domäne, eine Variable, einen Umwandlungsausdruck usw. verwendet.

`STARTING AT`

Klausel, die die Seitennummer der Datenbank angibt, bei der die nächste sekundäre Datenbankdatei beginnen soll. Wenn die vorherige Datei gemäß der angegebenen Seitenzahl vollständig mit Daten gefüllt ist, beginnt das System, der nächsten Datenbankdatei neue Daten hinzuzufügen.

`DIFFERENCE FILE`

Klausel, die den Pfad und den Namen für das Datei-Delta angibt, das alle Mutationen in der Datenbankdatei speichert, nachdem sie durch die Anweisung `ALTER DATABASE BEGIN BACKUP` in den "kopiersicheren" Modus geschaltet wurde. Eine detaillierte Beschreibung dieser Klausel finden Sie unter `ALTER DATABASE`.

Angabe des Datenbankdialekts

Datenbanken werden standardmäßig in Dialekt 3 erstellt. Damit die Datenbank im SQL-Dialekt 1 erstellt wird, müssen Sie die Anweisung `SET SQL DIALECT 1` aus dem Skript oder der Client-Anwendung ausführen, z.B. in *isql* vor der `CREATE DATABASE`-Anweisung.

Wer kann eine Datenbank erstellen?

Die `CREATE DATABASE`-Anweisung kann ausgeführt werden durch:

- Administratoren
- Benutzer mit dem Privileg `CREATE DATABASE`BASE

Beispiele für die Verwendung von `CREATE DATABASE`

1. Erstellen einer Datenbank in Windows, die sich auf Datenträger D mit einer Seitengröße von 4.096 befindet. Der Eigentümer der Datenbank ist der Benutzer *wizard*. Die Datenbank ist in Dialekt und verwendet `WIN1251` als Standardzeichensatz.

```
SET SQL DIALECT 1;
CREATE DATABASE 'D:\test.fdb'
USER 'wizard' PASSWORD 'player'
PAGE_SIZE = 4096 DEFAULT CHARACTER SET WIN1251;
```

2. Erstellen einer Datenbank im Linux-Betriebssystem mit einer Seitengröße von 8.192 (Standard). Der Eigentümer der Datenbank ist der Benutzer *wizard*. Die Datenbank wird in Dialekt 3 sein

und verwendet UTF8 als Standardzeichensatz mit UNICODE_CI_AI als Standardsortierung.

```
CREATE DATABASE '/home/firebird/test.fdb'
USER 'wizard' PASSWORD 'player'
DEFAULT CHARACTER SET UTF8 COLLATION UNICODE_CI_AI;
```

3. Erstellen einer Datenbank auf dem entfernten Server “baseserver” mit dem im Alias “test” angegebenen Pfad, der zuvor in der Datei databases.conf definiert wurde. Es wird das TCP/IP-Protokoll verwendet. Der Eigentümer der Datenbank ist der Benutzer *wizard*. Die Datenbank wird in Dialekt 3 sein und verwendet UTF8 als Standardzeichensatz.

```
CREATE DATABASE 'baseserver:test'
USER 'wizard' PASSWORD 'player'
DEFAULT CHARACTER SET UTF8;
```

4. Erstellen einer Datenbank in Dialekt 3 mit UTF8 als Standardzeichensatz. Die Primärdatei enthält bis zu 10.000 Seiten mit einer Seitengröße von 8.192. Sobald die Primärdatei die maximale Seitenzahl erreicht hat, beginnt Firebird damit, Seiten der Sekundärdatei test.fdb2 zuzuordnen. Wenn auch diese Datei maximal gefüllt ist, wird test.fdb3 der Empfänger aller neuen Seitenzuweisungen. Als letzte Datei hat Firebird keine Seitenbegrenzung. Neue Zuweisungen werden so lange fortgesetzt, wie das Dateisystem dies zulässt oder bis auf dem Speichergerät kein freier Speicherplatz mehr vorhanden ist. Wenn für diese letzte Datei ein LENGTH-Parameter angegeben würde, würde er ignoriert.

```
SET SQL DIALECT 3;
CREATE DATABASE 'baseserver:D:\test.fdb'
USER 'wizard' PASSWORD 'player'
PAGE_SIZE = 8192
DEFAULT CHARACTER SET UTF8
FILE 'D:\test.fdb2'
STARTING AT PAGE 10001
FILE 'D:\test.fdb3'
STARTING AT PAGE 20001;
```

5. Erstellen einer Datenbank in Dialekt 3 mit UTF8 als Standardzeichensatz. Die Primärdatei enthält bis zu 10.000 Seiten mit einer Seitengröße von 8.192. In Bezug auf die Dateigröße und die Verwendung von Sekundärdateien verhält sich diese Datenbank genau wie im vorherigen Beispiel.

```
SET SQL DIALECT 3;
CREATE DATABASE 'baseserver:D:\test.fdb'
USER 'wizard' PASSWORD 'player'
PAGE_SIZE = 8192
LENGTH 10000 PAGES
DEFAULT CHARACTER SET UTF8
FILE 'D:\test.fdb2'
```

```
FILE 'D:\test.fdb3'
STARTING AT PAGE 20001;
```

Siehe auch

ALTER DATABASE, DROP DATABASE

5.1.2. ALTER DATABASE

Verwendet für

Ändern der Dateiorganisation einer Datenbank, Umschalten ihres "kopiersicheren" Zustands, Verwalten der Verschlüsselung und anderer datenbankweiter Konfigurationen

Verfügbar in

DSQL, ESQL — eingeschränkter Funktionsumfang

Syntax

```
ALTER {DATABASE | SCHEMA} <alter_db_option> [<alter_db_option> ...]

<alter_db_option> :==
    <add_sec_clause>
  | {ADD DIFFERENCE FILE 'diff_file' | DROP DIFFERENCE FILE}
  | {BEGIN | END} BACKUP
  | SET DEFAULT CHARACTER SET charset
  | SET LINGER TO linger_duration
  | DROP LINGER
  | {ENCRYPT WITH plugin_name [KEY key_name] | DECRYPT}

<add_sec_clause> ::= ADD <sec_file> [<sec_file> ...]

<sec_file> ::=
  FILE 'filepath'
  [STARTING [AT [PAGE]] pagenum]
  [LENGTH [=] num [PAGE[S]]
```

Mehrere Dateien können in einer ADD-Klausel hinzugefügt werden:

```
ALTER DATABASE
  ADD FILE x LENGTH 8000
    FILE y LENGTH 8000
    FILE z
```

Mehrfaches Vorkommen von *add_sec_clause* (ADD FILE-Klauseln) ist erlaubt; eine ADD FILE-Klausel, die mehrere Dateien hinzufügt (wie im obigen Beispiel), kann mit anderen gemischt werden, die nur eine Datei hinzufügen. Die Anweisung wurde in der alten *InterBase 6 Language Reference* falsch dokumentiert.

Tabelle 21. `ALTER DATABASE`-*Anweisungsparameter*

Parameter	Beschreibung
add_sec_clause	Hinzufügen einer sekundären Datenbankdatei
sec_file	Dateispezifikation für Sekundärdatei
filepath	Vollständiger Pfad und Dateiname der Deltadatei oder sekundären Datenbankdatei
pagenum	Seitennummer, ab der die sekundäre Datenbankdatei beginnen soll
num	Maximale Größe der Sekundärdatei in Seiten
diff_file	Dateipfad und Name der .delta-Datei (Differenzdatei)
charset	Neuer Standardzeichensatz der Datenbank
linger_duration	Dauer der *linger* Verzögerung in Sekunden; muss größer oder gleich 0 (null) sein
plugin_name	Der Name des Verschlüsselungs-Plugins
key_name	Der Name des Verschlüsselungsschlüssels

Die `ALTER DATABASE`-Anweisung kann:

- Sekundärdateien zu einer Datenbank hinzufügen
- Umschalten einer Einzeldatei-Datenbank in den “copy-safe”-Modus (nur DSQL)
- Pfad und Name der Delta-Datei für physische Backups setzen oder aufheben (nur DSQL)

`SCHEMA` ist derzeit ein Synonym für `DATABASE`; dies kann sich in einer zukünftigen Version ändern, daher empfehlen wir immer `DATABASE` zu verwenden

Wer kann die Datenbank ändern?

Die `ALTER DATABASE`-Anweisung kann ausgeführt werden durch:

- Administratoren
- Benutzer mit dem `ALTER DATABASE`-Privileg

Paramter für `ALTER DATABASE`

`ADD (FILE)`

Fügt der Datenbank sekundäre Dateien hinzu. Es ist notwendig, den vollständigen Pfad zur Datei und den Namen der Sekundärdatei anzugeben. Die Beschreibung für die Sekundärdatei ähnelt der für die Anweisung `CREATE DATABASE`.

`ADD DIFFERENCE FILE`

Gibt den Pfad und den Namen der Delta-Datei an, die alle Mutationen in der Datenbank speichert, wenn sie in den "kopiersicheren" Modus geschaltet wird. Diese Klausel fügt tatsächlich keine Datei hinzu. Es überschreibt nur den Standardnamen und -pfad der .delta-Datei. Um die bestehenden Einstellungen zu ändern, sollten Sie die zuvor angegebene Beschreibung der .delta-

Datei mit der DROP DIFFERENCE FILE-Klausel löschen, bevor Sie die neue Beschreibung der Delta-Datei angeben. Wenn Pfad und Name der .delta-Datei nicht überschrieben werden, hat die Datei denselben Pfad und Namen wie die Datenbank, jedoch mit der Dateierweiterung .delta.

Wird nur ein Dateiname angegeben, wird die .delta-Datei im aktuellen Verzeichnis des Servers erstellt. Unter Windows ist dies das Systemverzeichnis – ein sehr unkluger Ort, um flüchtige Benutzerdateien zu speichern und im Gegensatz zu den Windows-Dateisystemregeln.

DROP DIFFERENCE FILE

Löscht die Beschreibung (Pfad und Name) der .delta-Datei, die zuvor in der Klausel ADD DIFFERENCE FILE angegeben wurde. Die Datei wird nicht wirklich gelöscht. DROP DIFFERENCE FILE löscht den Pfad und den Namen der .delta-Datei aus dem Datenbank-Header. Wenn die Datenbank das nächste Mal in den "copy-safe"-Modus geschaltet wird, werden die Standardwerte verwendet (d. h. der gleiche Pfad und Name wie die der Datenbank, aber mit der Erweiterung .delta).

BEGIN BACKUP

Schaltet die Datenbank in den "kopiersicher" Modus. ALTER DATABASE mit dieser Klausel friert die Hauptdatenbankdatei ein, sodass sie mit Dateisystemtools sicher gesichert werden kann, selbst wenn Benutzer verbunden sind und Operationen mit Daten ausführen. Bis der Sicherungsstatus der Datenbank auf *NORMAL* zurückgesetzt wird, werden alle an der Datenbank vorgenommenen Änderungen in die .delta (Differenz)-Datei geschrieben.

Trotz ihrer Syntax startet eine Anweisung mit der BEGIN BACKUP-Klausel keinen Backup-Prozess, sondern schafft lediglich die Bedingungen für die Ausführung einer Aufgabe, die erfordert, dass die Datenbankdatei temporär schreibgeschützt ist.

END BACKUP

Schaltet die Datenbank vom "kopiersicheren" Modus in den normalen Modus um. Eine Anweisung mit dieser Klausel führt die .delta-Datei mit der Hauptdatenbankdatei zusammen und stellt den normalen Betrieb der Datenbank wieder her. Nach dem Start des Prozesses END BACKUP sind die Voraussetzungen für die Erstellung sicherer Backups mittels Dateisystemtools nicht mehr gegeben.

Die Verwendung von BEGIN BACKUP und END BACKUP und das Kopieren der Datenbankdateien mit Dateisystemtools ist bei Mehrdateidatenbanken *nicht sicher*! Verwenden Sie diese Methode nur für Datenbanken mit einer einzigen Datei.

Eine sichere Sicherung mit dem Dienstprogramm *gbak* ist jederzeit möglich, es wird jedoch nicht empfohlen, *gbak* auszuführen, während sich die Datenbank im Status *LOCKED* oder *MERGE* befindet.

SET DEFAULT CHARACTER SET

Ändert den Standardzeichensatz der Datenbank. Diese Änderung wirkt sich nicht auf

vorhandene Daten oder Spalten aus. Der neue Standardzeichensatz wird nur in nachfolgenden DDL-Befehlen verwendet.

`SET LINGER TO`

Setzt die *linger*-Verzögerung. Die *linger*-Verzögerung gilt nur für Firebird SuperServer und gibt an, wie viele Sekunden der Server eine Datenbankdatei (und ihre Caches) geöffnet hält, nachdem die letzte Verbindung zu dieser Datenbank geschlossen wurde. Dies kann dazu beitragen, die Leistung kostengünstig zu verbessern, wenn die Datenbank häufig geöffnet und geschlossen wird, indem Ressourcen für die nächste Verbindung "warm" gehalten werden.

Dieser Modus kann für Webanwendungen - ohne Verbindungspool - nützlich sein, bei denen die Verbindung zur Datenbank normalerweise nur für sehr kurze Zeit "lebt".

Die Klauseln `SET LINGER TO` und `DROP LINGER` können in einer einzigen Anweisung kombiniert werden, aber die letzte Klausel "gewinnt". Zum Beispiel setzt `ALTER DATABASE SET LINGER TO 5 DROP LINGER` die *linger*-Verzögerung auf 0 (kein linger), während `ALTER DATABASE DROP LINGER SET LINGER to 5` die *linger* -Verzögerung auf 5 Sekunden setzt.

`DROP LINGER`

Löscht die *linger*-Verzögerung (setzt sie auf Null). Die Verwendung von `DROP LINGER` entspricht der Verwendung von `SET LINGER TO 0`.

Das Löschen von 'LINGER' ist keine ideale Lösung für die gelegentliche Notwendigkeit, es für einen einmaligen Zustand auszuschalten, in dem der Server erzwungenes Herunterfahren benötigt. Das Dienstprogramm *gfix* hat jetzt den Schalter `-NoLinger`, der die angegebene Datenbank sofort schließt, nachdem der letzte Anhang verschwunden ist, unabhängig von der `LINGER` -Einstellung in der Datenbank. Die Einstellung 'LINGER' wird beibehalten und funktioniert beim nächsten Mal normal.

Dieselbe einmalige Überschreibung ist auch über die Dienste-API unter Verwendung des Tags `isc_spb_prp_nolinger` verfügbar, z. (in einer Zeile):

```
fbsvcmgr host:service_mgr user sysdba password xxx
       action_properties dbname employee prp_nolinger
```

Die Klauseln `DROP LINGER` und `SET LINGER TO` können in einer einzigen Anweisung kombiniert werden, aber die letzte Klausel "gewinnt".

`ENCRYPT WITH`

Vgl. Eine Datenbank verschlüsseln im Abschnitt Sicherheit.

`DECRYPT`

Vgl. Eine Datenbank entschlüsseln im Abschnitt Sicherheit.

Beispiele zur Verwendung von ALTER DATABASE

1. Hinzufügen einer sekundären Datei zur Datenbank. Sobald 30000 Seiten in der vorherigen Primär- oder Sekundärdatei gefüllt sind, beginnt die Firebird-Engine, Daten in die Sekundärdatei `test4.fdb` hinzuzufügen.

```
ALTER DATABASE
  ADD FILE 'D:\test4.fdb'
    STARTING AT PAGE 30001;
```

2. Angabe von Pfad und Name der Delta-Datei:

```
ALTER DATABASE
  ADD DIFFERENCE FILE 'D:\test.diff';
```

3. Löschen der Beschreibung der Delta-Datei:

```
ALTER DATABASE
  DROP DIFFERENCE FILE;
```

4. Umschalten der Datenbank in den “kopiersicher” Modus:

```
ALTER DATABASE
  BEGIN BACKUP;
```

5. Zurückschalten der Datenbank vom “copy-safe”-Modus in den normalen Betriebsmodus:

```
ALTER DATABASE
  END BACKUP;
```

6. Ändern des Standardzeichensatzes für eine Datenbank in `WIN1251`

```
ALTER DATABASE
  SET DEFAULT CHARACTER SET WIN1252;
```

7. Einstellen einer *linger*-verzögerung von 30 Sekunden

```
ALTER DATABASE
  SET LINGER TO 30;
```

8. Verschlüsseln der Datenbank mit einem Plugin namens `DbCrypt`

```
ALTER DATABASE
  ENCRYPT WITH DbCrypt;
```

9. Entschlüsseln der Datenbank

```
ALTER DATABASE
  DECRYPT;
```

Siehe auch

`CREATE DATABASE`, `DROP DATABASE`

5.1.3. DROP DATABASE

Verwendet für

Löschen der Datenbank, mit der Sie gerade verbunden sind

Verfügbar in

DSQL, ESQL

Syntax

```
DROP DATABASE
```

Die Anweisung `DROP DATABASE` löscht die aktuelle Datenbank. Bevor Sie eine Datenbank löschen, müssen Sie sich mit ihr verbinden. Die Anweisung löscht die Primärdatei, alle Sekundärdateien und alle Shadow-Dateien.

Im Gegensatz zu `CREATE DATABASE` und `ALTER DATABASE` ist `DROP SCHEMA` kein gültiger Alias für `DROP DATABASE`. Dies ist beabsichtigt.

Wer kann eine Datenbank löschen?

Die `DROP DATABASE`-Anweisung kann ausgeführt werden durch:

- Administratoren
- Benutzer mit dem Privileg `DROP DATABASE`BASE

Beispiel zur Verwendung von `DROP DATABASE`

Löschen der aktuellen Datenbank

```
DROP DATABASE;
```

Siehe auch

`CREATE DATABASE`, `ALTER DATABASE`

5.2. SHADOW

Ein *shadow* ist eine exakte, seitenweise Kopie einer Datenbank. Sobald ein Schatten erstellt wurde, werden alle in der Datenbank vorgenommenen Änderungen sofort im Schatten widergespiegelt. Wenn die primäre Datenbankdatei aus irgendeinem Grund nicht verfügbar ist, wechselt das DBMS zum Shadow.

In diesem Abschnitt wird beschrieben, wie Sie Schattendateien erstellen und löschen.

5.2.1. CREATE SHADOW

Verwendet für

Schatten für die aktuelle Datenbank erstellen

Verfügbar in

DSQL, ESQL

Syntax

```
CREATE SHADOW <sh_num> [{AUTO | MANUAL}] [CONDITIONAL]
  'filepath' [LENGTH [=] num [PAGE[S]]]
  [<secondary_file> ...]

<secondary_file> ::=
  FILE 'filepath'
  [STARTING [AT [PAGE]] pagenum]
  [LENGTH [=] num [PAGE[S]]]
```

Tabelle 22. CREATE SHADOW-Anweisungsparameter

Parameter	Beschreibung
sh_num	Schattennummer – eine positive Zahl, die den Schattensatz identifiziert
filepath	Der Name der Schattendatei und der Pfad zu ihr gemäß den Regeln des Betriebssystems
num	Maximale Schattengröße in Seiten
secondary_file	Sekundärdateispezifikation
page_num	Die Nummer der Seite, bei der die sekundäre Schattendatei beginnen soll

Die Anweisung CREATE SHADOW erstellt einen neuen Schatten. Der Shadow beginnt sofort mit dem Duplizieren der Datenbank. Es ist einem Benutzer nicht möglich, sich mit einem Shadow zu verbinden.

Wie eine Datenbank kann ein Schatten aus mehreren Dateien bestehen. Die Anzahl und Größe der Dateien eines Shadows hängt nicht von der Anzahl und Größe der Dateien der Datenbank ab, die er beschattet.

Die Seitengröße für Schattendateien ist gleich der Seitengröße der Datenbank und kann nicht

geändert werden.

Wenn eine Katastrophe mit der Originaldatenbank auftritt, konvertiert das System den Schatten in eine Kopie der Datenbank und wechselt zu dieser. Der Schatten ist dann *unverfügbar*. Was als nächstes passiert, hängt von der Option `MODE` ab.

`AUTO | MANUAL`-Modus

Wenn ein Schatten in eine Datenbank konvertiert wird, ist er nicht verfügbar. Alternativ kann ein Schatten nicht mehr verfügbar sein, weil jemand seine Datei versehentlich löscht oder der Speicherplatz, auf dem die Schattendateien gespeichert sind, erschöpft oder selbst beschädigt ist.

- Wenn der AUTO-Modus ausgewählt ist (Standardwert), wird das Spiegeln automatisch beendet, alle Verweise darauf werden aus dem Datenbankheader gelöscht und die Datenbank funktioniert normal weiter.

  Wenn die Option `CONDITIONAL` gesetzt wurde, versucht das System, einen neuen Schatten zu erstellen, um den verlorenen zu ersetzen. Es gelingt jedoch nicht immer, und möglicherweise muss manuell ein neues erstellt werden.

- Wenn das MANUAL-Modusattribut gesetzt ist, wenn der Shadow nicht verfügbar ist, werden alle Versuche, eine Verbindung zur Datenbank herzustellen und sie abzufragen, Fehlermeldungen erzeugen. Auf die Datenbank kann nicht zugegriffen werden, bis entweder der Shadow wieder verfügbar ist oder der Datenbankadministrator ihn mit der `DROP SHADOW` -Anweisung löscht. `MANUAL` sollte gewählt werden, wenn kontinuierliches Shadowing wichtiger ist als ein unterbrechungsfreier Betrieb der Datenbank.

Optionen für `CREATE SHADOW`

`LENGTH`

Gibt die maximale Größe der primären oder sekundären Schattendatei in Seiten an. Der Wert `LENGTH` hat keinen Einfluss auf die Größe der einzigen Schattendatei oder der letzten, wenn es sich um eine Menge handelt. Die letzte (oder einzige) Datei wächst automatisch weiter, solange es notwendig ist.

`STARTING AT`

Gibt die Schattenseitennummer an, bei der die nächste Schattendatei beginnen soll. Das System beginnt mit dem Hinzufügen neuer Daten zur nächsten Schattendatei, wenn die vorherige Datei bis zur angegebenen Seitenzahl mit Daten gefüllt ist.

Sie können die Größe, die Namen und den Speicherort der Schattendateien überprüfen, indem Sie mit *isql* eine Verbindung zur Datenbank herstellen und den Befehl `SHOW DATABASE;` . ausführen

Wer kann einen Schatten erstellen

Die `CREATE SHADOW`-Anweisung kann ausgeführt werden durch:

- Administratoren

- Benutzer mit dem ALTER DATABASE-Privileg

Beispiele für die Verwendung von CREATE SHADOW

1. Erstellen eines Schattens für die aktuelle Datenbank als “Schattennummer 1”:

```
CREATE SHADOW 1 'g:\data\test.shd';
```

2. Erstellen eines Schattens mit mehreren Dateien für die aktuelle Datenbank als “Schattennummer 2”:

```
CREATE SHADOW 2 'g:\data\test.sh1'
  LENGTH 8000 PAGES
  FILE 'g:\data\test.sh2';
```

Siehe auch

CREATE DATABASE, DROP SHADOW

5.2.2. DROP SHADOW

Verwendet für

Löschen eines Schattens aus der aktuellen Datenbank

Verfügbar in

DSQL, ESQL

Syntax

```
DROP SHADOW sh_num
  [{DELETE | PRESERVE} FILE]
```

Tabelle 23. DROP SHADOW*-Anweisungsparameter*

Parameter	Beschreibung
sh_num	Schattennummer – eine positive Zahl, die den Schattensatz identifiziert

Die DROP SHADOW-Anweisung löscht den angegebenen Schatten für die aktuelle Datenbank. Wenn ein Shadow gelöscht wird, werden alle damit verbundenen Dateien gelöscht und das Shadowing auf die angegebene *sh_num* wird beendet. Die optionale DELETE FILE-Klausel macht dieses Verhalten explizit. Im Gegensatz dazu entfernt die PRESERVE FILE-Klausel den Schatten aus der Datenbank, aber die Datei selbst wird nicht gelöscht.

Wer kann einen Schatten löschen

Die DROP SHADOW-Anweisung kann ausgeführt werden durch:

- Administratoren

- Benutzer mit dem `ALTER DATABASE`-Privileg

Beispiel für `DROP SHADOW`

Löschen von "Schatten mit der Nummer 1".

```
DROP SHADOW 1;
```

Siehe auch

`CREATE SHADOW`

5.3. DOMAIN

`DOMAIN` ist einer der Objekttypen in einer relationalen Datenbank. Eine Domain wird als ein bestimmter Datentyp mit einigen daran angehängten Attributen erstellt. Nachdem es in der Datenbank definiert wurde, kann es wiederholt verwendet werden, um Tabellenspalten, PSQL-Argumente und lokale PSQL-Variablen zu definieren. Diese Objekte erben alle Attribute der Domain. Einige Attribute können bei Bedarf überschrieben werden, wenn das neue Objekt definiert wird.

In diesem Abschnitt wird die Syntax von Anweisungen zum Erstellen, Ändern und Löschen von Domainn beschrieben. Eine detaillierte Beschreibung der Domains und ihrer Verwendung finden Sie in Benutzerdefinierte Datentypen — Domains.

5.3.1. CREATE DOMAIN

Verwendet für

Erstellen einer neuen Domain

Verfügbar in

DSQL, ESQL

Syntax

```
CREATE DOMAIN name [AS] <datatype>
  [DEFAULT {<literal> | NULL | <context_var>}]
  [NOT NULL] [CHECK (<dom_condition>)]
  [COLLATE collation_name]

<datatype> ::=
  <scalar_datatype> | <blob_datatype> | <array_datatype>

<scalar_datatype> ::=
  !! Siehe auch Skalardatentyp-Syntax !!

<blob_datatype> ::=
  !! Siehe auch BLOB-Datentyp-Syntax !!

<array_datatype> ::=
```

```
  !! Siehe auch Array-Datentyp-Syntax !!

<dom_condition> ::=
    <val> <operator> <val>
  | <val> [NOT] BETWEEN <val> AND <val>
  | <val> [NOT] IN ({<val> [, <val> ...] | <select_list>})
  | <val> IS [NOT] NULL
  | <val> IS [NOT] DISTINCT FROM <val>
  | <val> [NOT] CONTAINING <val>
  | <val> [NOT] STARTING [WITH] <val>
  | <val> [NOT] LIKE <val> [ESCAPE <val>]
  | <val> [NOT] SIMILAR TO <val> [ESCAPE <val>]
  | <val> <operator> {ALL | SOME | ANY} (<select_list>)
  | [NOT] EXISTS (<select_expr>)
  | [NOT] SINGULAR (<select_expr>)
  | (<dom_condition>)
  | NOT <dom_condition>
  | <dom_condition> OR <dom_condition>
  | <dom_condition> AND <dom_condition>

<operator> ::=
    <> | != | ^= | ~= | = | < | > | <= | >=
  | !< | ^< | ~< | !> | ^> | ~>

<val> ::=
    VALUE
  | <literal>
  | <context_var>
  | <expression>
  | NULL
  | NEXT VALUE FOR genname
  | GEN_ID(genname, <val>)
  | CAST(<val> AS <cast_type>)
  | (<select_one>)
  | func([<val> [, <val> ...]])

<cast_type> ::= <domain_or_non_array_type> | <array_datatype>

<domain_or_non_array_type> ::=
  !! See Syntax für Skalardatentypen !!
```

Tabelle 24. CREATE DOMAIN-*Anweisungsparameter*

Parameter	Beschreibung
name	Domainname bestehend aus bis zu 31 Zeichen
datatype	SQL-Datentyp
literal	Ein Literalwert, der mit *datatype* kompatibel ist
context_var	Jede Kontextvariable, deren Typ mit *datatype* kompatibel ist

Parameter	Beschreibung
dom_condition	Domain-Bedingung
collation_name	Name einer Kollatierungssequenz, die für *charset_name* gültig ist, wenn sie mit *datatype* versorgt wird oder ansonsten für den Standardzeichensatz der Datenbank gültig ist
select_one	Eine skalare `SELECT`-Anweisung – Auswahl einer Spalte und Rückgabe nur einer Zeile
select_list	Eine `SELECT`-Anweisung, die eine Spalte auswählt und null oder mehr Zeilen zurückgibt
select_expr	Eine `SELECT`-Anweisung, die eine oder mehrere Spalten auswählt und null oder mehr Zeilen zurückgibt
expression	Ein Ausdruck, der in einen Wert aufgelöst wird, der mit *datatype* kompatibel ist
genname	Sequenzname (Generatorname)
func	Interne Funktion oder UDF

Die `CREATE DOMAIN`-Anweisung erstellt eine neue Domain.

Als Domaintyp kann jeder SQL-Datentyp angegeben werden.

Typspezifische Details

Array-Typen

- Wenn die Domain ein Array sein soll, kann der Basistyp ein beliebiger SQL-Datentyp außer `BLOB` und Array sein.
- Die Dimensionen des Arrays sind in eckigen Klammern angegeben. (Im Syntaxblock erscheinen diese Klammern in Anführungszeichen, um sie von den eckigen Klammern zu unterscheiden, die optionale Syntaxelemente kennzeichnen.)
- Für jede Array-Dimension definieren eine oder zwei ganze Zahlen die untere und obere Grenze ihres Indexbereichs:
  - Arrays sind standardmäßig 1-basiert. Die untere Grenze ist implizit und nur die obere Grenze muss angegeben werden. Eine einzelne Zahl kleiner als 1 definiert den Bereich *num*..1 und eine Zahl größer als 1 definiert den Bereich 1..*num*.
  - Zwei durch einen Doppelpunkt getrennte Zahlen (‘:’) und optionales Leerzeichen, das zweite größer als das erste, können verwendet werden, um den Bereich explizit zu definieren. Eine oder beide Grenzen können kleiner als Null sein, solange die obere Grenze größer als die untere ist.
- Wenn das Array mehrere Dimensionen hat, müssen die Bereichsdefinitionen für jede Dimension durch Kommas und optionales Leerzeichen getrennt werden.
- Indizes werden *nur* validiert, wenn tatsächlich ein Array existiert. Das bedeutet, dass keine Fehlermeldungen bezüglich ungültiger Indizes zurückgegeben werden, wenn die Auswahl eines bestimmten Elements nichts zurückgibt oder wenn ein Array-Feld `NULL` ist.

String-Typen

Mit der CHARACTER SET-Klausel können Sie den Zeichensatz für die Typen CHAR, VARCHAR und BLOB (SUB_TYPE TEXT) angeben. Wenn der Zeichensatz nicht angegeben ist, wird der als DEFAULT CHARACTER SET angegebene Zeichensatz der Datenbank verwendet. Wenn kein Zeichensatz angegeben wurde, wird beim Erstellen einer Zeichendomäne standardmäßig der Zeichensatz NONE verwendet.

Mit dem Zeichensatz NONE werden Zeichendaten so gespeichert und abgerufen, wie sie übermittelt wurden. Daten in einer beliebigen Codierung können einer Spalte basierend auf einer solchen Domain hinzugefügt werden, aber es ist unmöglich, diese Daten zu einer Spalte mit einer anderen Codierung hinzuzufügen. Da keine Transliteration zwischen den Quell- und Zielcodierungen durchgeführt wird, können Fehler auftreten.

DEFAULT-Klausel

Mit der optionalen DEFAULT-Klausel können Sie einen Standardwert für die Domain angeben. Dieser Wert wird der Tabellenspalte hinzugefügt, die diese Domain erbt, wenn die INSERT -Anweisung ausgeführt wird, wenn kein Wert dafür in der DML-Anweisung angegeben ist. Lokale Variablen und Argumente in PSQL-Modulen, die auf diese Domain verweisen, werden mit dem Standardwert initialisiert. Verwenden Sie als Standardwert ein Literal eines kompatiblen Typs oder eine Kontextvariable eines kompatiblen Typs.

NOT NULL Constraint

Spalten und Variablen, die auf einer Domain mit der Einschränkung NOT NULL basieren, werden daran gehindert, als NULL geschrieben zu werden, d.h. ein Wert ist *erforderlich*.

Achten Sie beim Anlegen einer Domain darauf, keine Einschränkungen anzugeben, die sich widersprechen würden. Zum Beispiel sind NOT NULL und DEFAULT NULL widersprüchlich.

CHECK Constraint(s)

Die optionale CHECK-Klausel gibt Einschränkungen für die Domain an. Eine Domaineinschränkung gibt Bedingungen an, die von den Werten von Tabellenspalten oder Variablen erfüllt werden müssen, die von der Domain erben. Eine Bedingung muss in Klammern eingeschlossen werden. Eine Bedingung ist ein logischer Ausdruck (auch Prädikat genannt), der die booleschen Ergebnisse TRUE, FALSE und UNKNOWN zurückgeben kann. Eine Bedingung gilt als erfüllt, wenn das Prädikat den Wert TRUE oder „unknown value“ (entspricht NULL) zurückgibt. Liefert das Prädikat FALSE, ist die Annahmebedingung nicht erfüllt.

VALUE Keyword

Das Schlüsselwort VALUE in einer Domaineinschränkung ersetzt die Tabellenspalte, die auf dieser Domain basiert, oder eine Variable in einem PSQL-Modul. Es enthält den Wert, der der Variablen oder der Tabellenspalte zugewiesen wurde. VALUE kann überall in der CHECK-Bedingung verwendet werden, obwohl es normalerweise im linken Teil der Bedingung verwendet wird.

COLLATE

Mit der optionalen COLLATE-Klausel können Sie die Kollatierungssequenz angeben, wenn die

Domain auf einem der String-Datentypen basiert, einschließlich BLOBs mit Textuntertypen. Wenn keine Kollatierungssequenz angegeben ist, ist die Kollatierungssequenz diejenige, die zum Zeitpunkt der Erstellung der Domain für den angegebenen Zeichensatz voreingestellt ist.

Wer kann eine Domain erstellen

Die CREATE DOMAIN-Anweisung kann ausgeführt werden durch:

- Administratoren
- Benutzer mit der Berechtigung CREATE DOMAIN

CREATE DOMAIN-Beispiele

1. Erstellen einer Domain, die Werte über 1.000 annehmen kann, mit einem Standardwert von 10.000.

```
CREATE DOMAIN CUSTNO AS
  INTEGER DEFAULT 10000
  CHECK (VALUE > 1000);
```

2. Erstellen einer Domain, die die Werte 'Yes' und 'No' in dem beim Erstellen der Datenbank angegebenen Standardzeichensatz annehmen kann.

```
CREATE DOMAIN D_BOOLEAN AS
  CHAR(3) CHECK (VALUE IN ('Yes', 'No'));
```

3. Erstellen einer Domain mit dem Zeichensatz UTF8 und der Kollatierungssequenz UNICODE_CI_AI.

```
CREATE DOMAIN FIRSTNAME AS
  VARCHAR(30) CHARACTER SET UTF8
  COLLATE UNICODE_CI_AI;
```

4. Erstellen einer Domain vom Typ DATE, die NULL nicht akzeptiert und das aktuelle Datum als Standardwert verwendet.

```
CREATE DOMAIN D_DATE AS
  DATE DEFAULT CURRENT_DATE
  NOT NULL;
```

5. Erstellen einer Domain, die als Array aus 2 Elementen des Typs NUMERIC(18, 3) definiert ist. Der Array-Startindex ist 1.

```
CREATE DOMAIN D_POINT AS
  NUMERIC(18, 3) [2];
```

Über einen Array-Typ definierte Domainn können nur zum Definieren von Tabellenspalten verwendet werden. Sie können keine Arraydomänen verwenden, um lokale Variablen in PSQL-Modulen zu definieren.

6. Erstellen einer Domain, deren Elemente nur Ländercodes sein können, die in der Tabelle `COUNTRY` definiert sind.

```
CREATE DOMAIN D_COUNTRYCODE AS CHAR(3)
  CHECK (EXISTS(SELECT * FROM COUNTRY
         WHERE COUNTRYCODE = VALUE));
```

Das Beispiel wird nur gegeben, um die Möglichkeit zu zeigen, Prädikate mit Abfragen in der Domainntestbedingung zu verwenden. Es wird nicht empfohlen, diesen Domainnstil in der Praxis zu erstellen, es sei denn, die Nachschlagetabelle enthält Daten, die niemals gelöscht werden.

Siehe auch

`ALTER DOMAIN`, `DROP DOMAIN`

5.3.2. ALTER DOMAIN

Verwendet für

Die aktuellen Attribute einer Domain ändern oder umbenennen

Verfügbar in

DSQL, ESQL

Syntax

```
ALTER DOMAIN domain_name
  [TO new_name]
  [TYPE <datatype>]
  [{SET DEFAULT {<literal> | NULL | <context_var>} | DROP DEFAULT}]
  [{SET | DROP} NOT NULL]
  [{ADD [CONSTRAINT] CHECK (<dom_condition>) | DROP CONSTRAINT}]

<datatype> ::=
   <scalar_datatype> | <blob_datatype>

<scalar_datatype> ::=
  !! Vgl. Syntax für Skalardatentypen !!

<blob_datatype> ::=
  !! Vgl. Syntax für BLOB-Datentypen !!

!! Siehe auch CREATE DOMAIN-Syntax !!
```

Tabelle 25. `ALTER DOMAIN`-*Anweisungsparameter*

Parameter	Beschreibung
new_name	Neuer Name für Domain, bestehend aus bis zu 31 Zeichen
literal	Ein Literalwert, der mit *datatype* kompatibel ist
context_var	Jede Kontextvariable, deren Typ mit *datatype* kompatibel ist

Die `ALTER DOMAIN`-Anweisung ermöglicht Änderungen an den aktuellen Attributen einer Domain, einschließlich ihres Namens. Sie können beliebig viele Domain-Änderungen in einer `ALTER DOMAIN` -Anweisung vornehmen.

`ALTER DOMAIN`-Klausel

`TO name`

Verwenden Sie die `TO`-Klausel, um die Domain umzubenennen, solange keine Abhängigkeiten von der Domain bestehen, d.h. Tabellenspalten, lokale Variablen oder Prozedurargumente, die darauf verweisen.

`SET DEFAULT`

Mit der `SET DEFAULT`-Klausel können Sie einen neuen Standardwert setzen. Wenn die Domain bereits einen Standardwert hat, muss dieser nicht zuerst gelöscht werden – er wird durch den neuen ersetzt.

`DROP DEFAULT`

Verwenden Sie diese Klausel, um einen zuvor angegebenen Standardwert zu löschen und durch `NULL` zu ersetzen.

`SET NOT NULL`

Verwenden Sie diese Klasse, um der Domain eine `NOT NULL`-Einschränkung hinzuzufügen; Spalten oder Parameter dieser Domain werden daran gehindert, als `NULL` geschrieben zu werden, d.h. ein Wert ist *erforderlich*.

Das Hinzufügen einer `NOT NULL`-Einschränkung zu einer vorhandenen Domain unterzieht alle Spalten, die diese Domain verwenden, einer vollständigen Datenvalidierung. Stellen Sie daher sicher, dass die Spalten keine Nullen enthalten, bevor Sie die Änderung vornehmen.

`DROP NOT NULL`

Löschen Sie die Einschränkung `NOT NULL` aus der Domain.

Eine explizite `NOT NULL`-Einschränkung für eine Spalte, die von einer Domain abhängt, hat Vorrang vor der Domain. In dieser Situation wird die Änderung der Domain, um sie auf NULL zu setzen, nicht an die Spalte weitergegeben.

`ADD CONSTRAINT CHECK`

Verwenden Sie die `ADD CONSTRAINT CHECK`-Klausel, um der Domain eine `CHECK`-Beschränkung hinzuzufügen. Wenn die Domain bereits eine `CHECK`-Beschränkung hat, muss sie zuerst mit einer

ALTER DOMAIN-Anweisung gelöscht werden, die eine DROP CONSTRAINT-Klausel enthält.

TYPE

Die TYPE-Klausel wird verwendet, um den Datentyp der Domain in einen anderen, kompatiblen zu ändern. Das System verbietet jede Änderung des Typs, die zu Datenverlust führen könnte. Ein Beispiel wäre, wenn die Anzahl der Zeichen im neuen Typ kleiner wäre als im bestehenden.

Wenn Sie die Attribute einer Domain ändern, kann vorhandener PSQL-Code ungültig werden. Informationen zur Erkennung finden Sie im Artikel *Das RDB$VALID_BLR-Feld* in Anhang A.

Was kann ALTER DOMAIN nicht ändern

- Wenn die Domain als Array deklariert wurde, ist es nicht möglich, ihren Typ oder ihre Dimensionen zu ändern; auch kann kein anderer Typ in einen Array-Typ geändert werden.
- Es gibt keine Möglichkeit, die Standardsortierung zu ändern, ohne die Domain zu löschen und mit den gewünschten Attributen neu zu erstellen.

Wer kann eine Domain ändern

Die ALTER DOMAIN-Anweisung kann ausgeführt werden durch:

- Administratoren
- Der Inhaber der Domain
- Benutzer mit der Berechtigung ALTER ANY DOMAIN

Domainnänderungen können durch Abhängigkeiten von Objekten verhindert werden, für die der Benutzer nicht über ausreichende Berechtigungen verfügt.

ALTER DOMAIN-Beispiele

1. Ändern des Datentyps auf INTEGER und Einstellen oder Ändern des Standardwerts auf 2.000:

```
ALTER DOMAIN CUSTNO
  TYPE INTEGER
  SET DEFAULT 2000;
```

2. Domainnamen ändern

```
ALTER DOMAIN D_BOOLEAN TO D_BOOL;
```

3. Löschen des Standardwerts und Hinzufügen einer Einschränkung für die Domain:

```
ALTER DOMAIN D_DATE
  DROP DEFAULT
```

```
  ADD CONSTRAINT CHECK (VALUE >= date '01.01.2000');
```

4. Ändern der `CHECK`-Beschränkung:

```
ALTER DOMAIN D_DATE
  DROP CONSTRAINT;

ALTER DOMAIN D_DATE
  ADD CONSTRAINT CHECK
    (VALUE BETWEEN date '01.01.1900' AND date '31.12.2100');
```

5. Ändern des Datentyps, um die zulässige Zeichenanzahl zu erhöhen:

```
ALTER DOMAIN FIRSTNAME
  TYPE VARCHAR(50) CHARACTER SET UTF8;
```

6. Hinzufügen einer `NOT NULL`-Einschränkung:

```
ALTER DOMAIN FIRSTNAME
  SET NOT NULL;
```

7. Entfernen einer `NOT NULL`-Einschränkung:

```
ALTER DOMAIN FIRSTNAME
  DROP NOT NULL;
```

Siehe auch

`CREATE DOMAIN`, `DROP DOMAIN`

5.3.3. DROP DOMAIN

Verwendet für

Löschen einer bestehenden Domain

Verfügbar in

DSQL, ESQL

Syntax

```
DROP DOMAIN domain_name
```

Die `DROP DOMAIN`-Anweisung löscht eine in der Datenbank vorhandene Domain. Es ist nicht möglich, eine Domain zu löschen, wenn sie von Datenbanktabellenspalten referenziert oder in einem PSQL-Modul verwendet wird. Um eine verwendete Domain zu löschen, müssen alle Spalten in allen

Tabellen, die auf die Domain verweisen, gelöscht und alle Verweise auf die Domain aus den PSQL-Modulen entfernt werden.

Wer kann eine Domain löschen

Die `DROP DOMAIN`-Anweisung kann ausgeführt werden durch:

- Administratoren
- Der Inhaber der Domain
- Benutzer mit dem `DROP ANY DOMAIN`-Privileg

Example of `DROP DOMAIN`

Löschen der COUNTRYNAME-Domain

```
DROP DOMAIN COUNTRYNAME;
```

Siehe auch

`CREATE DOMAIN`, `ALTER DOMAIN`

5.4. TABLE

Als relationales DBMS speichert Firebird Daten in Tabellen. Eine Tabelle ist eine flache, zweidimensionale Struktur, die eine beliebige Anzahl von Zeilen enthält. Tabellenzeilen werden oft als *records* bezeichnet.

Alle Zeilen einer Tabelle haben die gleiche Struktur und bestehen aus Spalten. Tabellenspalten werden oft als *fields* bezeichnet. Eine Tabelle muss mindestens eine Spalte haben. Jede Spalte enthält einen einzelnen Typ von SQL-Daten.

In diesem Abschnitt wird beschrieben, wie Sie Tabellen in einer Datenbank erstellen, ändern und löschen.

5.4.1. CREATE TABLE

Verwendet für

Erstellen einer neuen Tabelle (Relation)

Verfügbar in

DSQL, ESQL

Syntax

```
CREATE [GLOBAL TEMPORARY] TABLE tablename
  [EXTERNAL [FILE] 'filespec']
  (<col_def> [, {<col_def> | <tconstraint>} ...])
  [ON COMMIT {DELETE | PRESERVE} ROWS]
```

```
<col_def> ::=
    <regular_col_def>
  | <computed_col_def>
  | <identity_col_def>

<regular_col_def> ::=
  colname {<datatype> | domainname}
  [DEFAULT {<literal> | NULL | <context_var>}]
  [<col_constraint> ...]
  [COLLATE collation_name]

<computed_col_def> ::=
  colname [{<datatype> | domainname}]
  {COMPUTED [BY] | GENERATED ALWAYS AS} (<expression>)

<identity_col_def> ::=
  colname {<datatype> | domainname}
  GENERATED BY DEFAULT AS IDENTITY [(START WITH startvalue)]
  [<col_constraint> ...]

<datatype> ::=
    <scalar_datatype> | <blob_datatype> | <array_datatype>

<scalar_datatype> ::=
  !! Siehe auch Skalardatentypensyntax !!

<blob_datatype> ::=
  !! Siehe auch BLOB-Datentypensyntax !!

<array_datatype> ::=
  !! Siehe auch Array-Datentypensyntax !!

<col_constraint> ::=
  [CONSTRAINT constr_name]
    { PRIMARY KEY [<using_index>]
    | UNIQUE      [<using_index>]
    | REFERENCES other_table [(colname)] [<using_index>]
        [ON DELETE {NO ACTION | CASCADE | SET DEFAULT | SET NULL}]
        [ON UPDATE {NO ACTION | CASCADE | SET DEFAULT | SET NULL}]
    | CHECK (<check_condition>)
    | NOT NULL }

<tconstraint> ::=
  [CONSTRAINT constr_name]
    { PRIMARY KEY (<col_list>) [<using_index>]
    | UNIQUE      (<col_list>) [<using_index>]
    | FOREIGN KEY (<col_list>)
        REFERENCES other_table [(<col_list>)] [<using_index>]
        [ON DELETE {NO ACTION | CASCADE | SET DEFAULT | SET NULL}]
        [ON UPDATE {NO ACTION | CASCADE | SET DEFAULT | SET NULL}]
    | CHECK (<check_condition>) }
```

```
<col_list> ::= colname [, colname ...]

<using_index> ::= USING
  [ASC[ENDING] | DESC[ENDING]] INDEX indexname

<check_condition> ::=
    <val> <operator> <val>
  | <val> [NOT] BETWEEN <val> AND <val>
  | <val> [NOT] IN (<val> [, <val> ...] | <select_list>)
  | <val> IS [NOT] NULL
  | <val> IS [NOT] DISTINCT FROM <val>
  | <val> [NOT] CONTAINING <val>
  | <val> [NOT] STARTING [WITH] <val>
  | <val> [NOT] LIKE <val> [ESCAPE <val>]
  | <val> [NOT] SIMILAR TO <val> [ESCAPE <val>]
  | <val> <operator> {ALL | SOME | ANY} (<select_list>)
  | [NOT] EXISTS (<select_expr>)
  | [NOT] SINGULAR (<select_expr>)
  | (<check_condition>)
  | NOT <check_condition>
  | <check_condition> OR <check_condition>
  | <check_condition> AND <check_condition>

<operator> ::=
    <> | != | ^= | ~= | = | < | > | <= | >=
  | !< | ^< | ~< | !> | ^> | ~>

<val> ::=
    colname ['['array_idx [, array_idx ...]']']
  | <literal>
  | <context_var>
  | <expression>
  | NULL
  | NEXT VALUE FOR genname
  | GEN_ID(genname, <val>)
  | CAST(<val> AS <cast_type>)
  | (<select_one>)
  | func([<val> [, <val> ...]])

<cast_type> ::= <domain_or_non_array_type> | <array_datatype>

<domain_or_non_array_type> ::=
  !! Siehe Skalardatentypensyntax !!
```

Tabelle 26. `CREATE TABLE`-*Anweisungsparameter*

Parameter	Beschreibung
tablename	Name (Bezeichner) für die Tabelle. Sie darf bis zu 31 Zeichen lang sein und muss in der Datenbank eindeutig sein.

Parameter	Beschreibung
filespec	Dateispezifikation (nur für externe Tabellen). Vollständiger Dateiname und Pfad in einfachen Anführungszeichen, korrekt für das lokale Dateisystem und auf einem Speichergerät, das physisch mit dem Host-Computer von Firebird verbunden ist.
colname	Name (Bezeichner) für eine Spalte in der Tabelle. Darf bis zu 31 Zeichen lang sein und muss in der Tabelle eindeutig sein.
datatype	SQL-Datentyp
domain_name	Domainname
start_value	Der Anfangswert der Identitätsspalte
col_constraint	Spaltenbeschränkung
tconstraint	Tabellenbeschränkung
constr_name	Der Name (Bezeichner) einer Einschränkung. Kann aus bis zu 31 Zeichen bestehen.
other_table	Der Name der Tabelle, auf die von der Fremdschlüsseleinschränkung verwiesen wird
other_col	Der Name der Spalte in *other_table*, auf die der Fremdschlüssel verweist
literal	Ein Literalwert, der im angegebenen Kontext zulässig ist
context_var	Jede Kontextvariable, deren Datentyp im angegebenen Kontext zulässig ist
check_condition	Die auf eine CHECK-Einschränkung angewendete Bedingung, die entweder als wahr, falsch oder `NULL` aufgelöst wird
collation	Collation
select_one	Eine skalare `SELECT`-Anweisung – Auswahl einer Spalte und Rückgabe nur einer Zeile
select_list	Eine `SELECT`-Anweisung, die eine Spalte auswählt und null oder mehr Zeilen zurückgibt
select_expr	Eine `SELECT`-Anweisung, die eine oder mehrere Spalten auswählt und null oder mehr Zeilen zurückgibt
expression	Ein Ausdruck, der zu einem Wert auflöst, der im gegebenen Kontext zulässig ist
genname	Name der Sequenz (Generator)
func	Interne Funktion oder UDF

Die Anweisung `CREATE TABLE` erstellt eine neue Tabelle. Jeder Benutzer kann es erstellen und sein Name muss unter den Namen aller Tabellen, Ansichten und gespeicherten Prozeduren in der Datenbank eindeutig sein.

Eine Tabelle muss mindestens eine nicht berechnete Spalte enthalten, und die Namen der Spalten müssen in der Tabelle eindeutig sein.

Eine Spalte muss entweder einen expliziten *SQL-Datentyp* haben, den Namen einer *Domäne*, deren Attribute für die Spalte kopiert werden, oder als `COMPUTED BY` einen Ausdruck (ein *berechnetes Feld*) definiert sein.

Eine Tabelle kann eine beliebige Anzahl von Tabelleneinschränkungen haben, einschließlich keiner.

Zeichenspalten

Sie können die Klausel `CHARACTER SET` verwenden, um den Zeichensatz für die Typen `CHAR`, `VARCHAR` und `BLOB` (Textsubtyp) anzugeben. Wenn der Zeichensatz nicht angegeben ist, wird der Standardzeichensatz der Datenbank - zum Zeitpunkt der Erstellung der Spalte - verwendet. Wenn die Datenbank keinen Standardzeichensatz hat, wird der Zeichensatz `NONE` angewendet. In diesem Fall werden die Daten so gespeichert und abgerufen, wie sie übermittelt wurden. Einer solchen Spalte können Daten in einer beliebigen Codierung hinzugefügt werden, es ist jedoch nicht möglich, diese Daten einer Spalte mit einer anderen Codierung hinzuzufügen. Zwischen den Quell- und Zielkodierungen wird keine Transliteration durchgeführt, was zu Fehlern führen kann.

Mit der optionalen `COLLATE`-Klausel können Sie die Kollatierungssequenz für Zeichendatentypen angeben, einschließlich `BLOB SUB_TYPE TEXT`. Wenn keine Kollatierungssequenz angegeben ist, wird die Standardkollatierungssequenz für den angegebenen Zeichensatz - zum Zeitpunkt der Erstellung der Spalte - angewendet.

Einstellen eines `DEFAULT`-Wertes

Mit der optionalen `DEFAULT`-Klausel können Sie den Standardwert für die Tabellenspalte angeben. Dieser Wert wird der Spalte hinzugefügt, wenn eine `INSERT`-Anweisung ausgeführt wird, wenn kein Wert dafür angegeben wurde *und* diese Spalte im `INSERT`-Befehl weggelassen wurde.

Der Standardwert kann ein Literal eines kompatiblen Typs sein, eine Kontextvariable, die mit dem Datentyp der Spalte typkompatibel ist, oder `NULL`, wenn die Spalte dies zulässt. Wenn kein Standardwert explizit angegeben wird, wird `NULL` impliziert.

Ein Ausdruck kann nicht als Standardwert verwendet werden.

Domainenbasierte Spalten

Um eine Spalte zu definieren, können Sie eine zuvor definierte Domäne verwenden. Wenn die Definition einer Spalte auf einer Domäne basiert, kann sie einen neuen Standardwert, zusätzliche `CHECK`-Einschränkungen und eine `COLLATE`-Klausel enthalten, die die in der Domänendefinition angegebenen Werte überschreibt. Die Definition einer solchen Spalte kann zusätzliche Spaltenbeschränkungen enthalten (zB `NOT NULL`), wenn die Domäne sie nicht hat.

Es ist nicht möglich, eine domänenbasierte Spalte zu definieren, die NULL-Werte zulässt, wenn die Domäne mit dem Attribut `NOT NULL` definiert wurde. Wenn Sie eine Domäne haben möchten, die sowohl zum Definieren von nullbaren als auch nicht-nullbaren Spalten und Variablen verwendet werden kann, ist es besser, die Domäne nullable zu definieren und `NOT NULL` in den nachfolgenden Spaltendefinitionen und Variablendeklarationen anzuwenden.

Identitätsspalten (autoinkrement)

Identitätsspalten können mit der `GENERATED BY DEFAULT AS IDENTITY`-Klausel definiert werden. Die Identitätsspalte ist die Spalte, die dem internen Sequenzgenerator zugeordnet ist. Sein Wert wird jedes Mal automatisch gesetzt, wenn er nicht in der `INSERT`-Anweisung angegeben wird. Mit der optionalen `START WITH`-Klausel können Sie einen anderen Anfangswert als 1 angeben.

Falsches `START WITH`-Verhalten

Der SQL-Standard verlangt, dass `START WITH` den ersten zu generierenden Wert angibt. Leider verwendet die aktuelle Implementierung in Firebird stattdessen den angegebenen Wert als Anfangswert des internen Generators, der die Identitätsspalte unterstützt. Das bedeutet, dass es im Moment den Wert **vor** dem ersten generierten Wert angibt.

Dies wird in Firebird 4 behoben, siehe auch CORE-6376.

Regeln

- Der Datentyp einer Identitätsspalte muss ein exakter Zahlentyp mit Nullskala sein. Erlaubte Typen sind somit `SMALLINT`, `INTEGER`, `BIGINT`, `NUMERIC(p[,0])` und `DECIMAL(p[,0])`.
- Eine Identitätsspalte darf keinen `DEFAULT`- oder `COMPUTED`-Wert haben.

- Eine Identitätsspalte kann nicht in eine reguläre Spalte geändert werden. Das Umgekehrte gilt auch. Firebird 4 führt die Option ein, eine Identitätsspalte in eine normale Spalte zu ändern.
- Identitätsspalten sind implizit `NOT NULL` (non-nullable).
- Eindeutigkeit wird nicht automatisch erzwungen. Eine `UNIQUE`- oder `PRIMARY KEY`-Beschränkung ist erforderlich, um die Eindeutigkeit zu garantieren.
- Die Verwendung anderer Methoden zur Generierung von Schlüsselwerten für Identitätsspalten, z. B. durch Trigger-Generator-Code oder indem Benutzern erlaubt wird, sie zu ändern oder hinzuzufügen, wird davon abgeraten, unerwartete Schlüsselverletzungen zu vermeiden.

Berechnete Felder

Berechnete Felder können mit der `COMPUTED [BY]`- oder `GENERATED ALWAYS AS`-Klausel (gemäß SQL:2003-Standard) definiert werden. Sie meinen dasselbe. Die Beschreibung des Datentyps ist für berechnete Felder nicht erforderlich (aber möglich), da das DBMS als Ergebnis der Ausdrucksanalyse den entsprechenden Typ berechnet und speichert. Entsprechende Operationen für die in einem Ausdruck enthaltenen Datentypen müssen genau angegeben werden.

Wenn der Datentyp für ein berechnetes Feld explizit angegeben wird, wird das Berechnungsergebnis in den angegebenen Typ konvertiert. Das bedeutet zum Beispiel, dass das Ergebnis eines numerischen Ausdrucks als String ausgegeben werden könnte.

In einer Abfrage, die eine `COMPUTED BY`-Spalte auswählt, wird der Ausdruck für jede Zeile der ausgewählten Daten ausgewertet.

Anstelle einer berechneten Spalte ist es in manchen Fällen sinnvoll, eine reguläre Spalte zu verwenden, deren Wert in Triggern zum Hinzufügen und Aktualisieren von Daten ausgewertet wird. Dies kann die Leistung beim Einfügen/Aktualisieren von Datensätzen verringern, aber die Leistung der Datenauswahl erhöhen.

Definieren einer Array-Spalte

- Wenn die Spalte ein Array sein soll, kann der Basistyp ein beliebiger SQL-Datentyp außer `BLOB` und Array sein.
- Die Abmessungen des Arrays sind in eckigen Klammern angegeben. (Im Syntax block erscheinen diese Klammern in Anführungszeichen, um sie von den eckigen Klammern zu unterscheiden, die optionale Syntaxelemente kennzeichnen.)
- Für jede Array-Dimension definieren eine oder zwei ganze Zahlen die untere und obere Grenze ihres Indexbereichs:
  - Arrays sind standardmäßig 1-basiert. Die untere Grenze ist implizit und nur die obere Grenze muss angegeben werden. Eine einzelne Zahl kleiner als 1 definiert den Bereich *num*..1 und eine Zahl größer als 1 definiert den Bereich 1..*num*.
  - Zwei durch einen Doppelpunkt getrennte Zahlen (‘:’) und optionales Leerzeichen, das zweite größer als das erste, können verwendet werden, um den Bereich explizit zu definieren. Eine oder beide Grenzen können kleiner als Null sein, solange die obere Grenze größer als die untere ist.
- Wenn das Array mehrere Dimensionen hat, müssen die Bereichsdefinitionen für jede Dimension durch Kommas und optionales Leerzeichen getrennt werden.
- Indizes werden *nur* validiert, wenn tatsächlich ein Array existiert. Das bedeutet, dass keine Fehlermeldungen bezüglich ungültiger Indizes zurückgegeben werden, wenn die Auswahl eines bestimmten Elements nichts zurückgibt oder wenn ein Array-Feld [constant] `NULL` ist.

Constraints

Es können fünf Arten von Einschränkungen angegeben werden. Sie sind:

- Primärschlüssel (`PRIMARY KEY`)
- Eindeutiger Schlüssel (`UNIQUE`)
- Fremdschlüssel (`REFERENCES`)
- `CHECK`-Einschränkung (`CHECK`)
- `NOT NULL`-Einschränkung (`NOT NULL`)

Einschränkungen können auf Spaltenebene (“Spaltenbeschränkungen”) oder auf Tabellenebene (“Tabellenbeschränkungen”) angegeben werden. Einschränkungen auf Tabellenebene sind erforderlich, wenn Schlüssel (eindeutige Einschränkung, Primärschlüssel, Fremdschlüssel) aus mehreren Spalten bestehen und wenn eine `CHECK`-Einschränkung andere Spalten in der Zeile als die definierte Spalte umfasst. Die Einschränkung `NOT NULL` kann nur als Spalteneinschränkung angegeben werden. Die Syntax einiger Einschränkungstypen kann geringfügig abweichen, je nachdem, ob die Einschränkung auf Spalten- oder Tabellenebene definiert ist.

- Eine Einschränkung auf Spaltenebene wird während einer Spaltendefinition angegeben, nachdem alle Spaltenattribute außer `COLLATION` angegeben wurden, und kann nur die in dieser Definition angegebene Spalte betreffen
- Einschränkungen auf Tabellenebene können nur nach den Definitionen der Spalten angegeben werden, die in der Einschränkung verwendet werden.
- Einschränkungen auf Tabellenebene sind eine flexiblere Möglichkeit, Einschränkungen festzulegen, da sie Einschränkungen mit mehreren Spalten berücksichtigen können
- Sie können Einschränkungen auf Spaltenebene und auf Tabellenebene in derselben `CREATE TABLE`-Anweisung mischen

Das System erstellt automatisch den entsprechenden Index für einen Primärschlüssel (`PRIMARY KEY`), einen eindeutigen Schlüssel (`UNIQUE`) und einen Fremdschlüssel (`REFERENCES` für eine Einschränkung auf Spaltenebene, `FOREIGN KEY REFERENCES` für eine auf der Tabellenebene).

Namen für Einschränkungen und ihre Indizes

Einschränkungen auf Spaltenebene und ihre Indizes werden automatisch benannt:

- Der Name der Einschränkung hat die Form `INTEG_n`, wobei *n* eine oder mehrere Ziffern darstellt
- Der Indexname hat die Form `RDB$PRIMARYn` (für einen Primärschlüsselindex), `RDB$FOREIGNn` (für einen Fremdschlüsselindex) oder `RDB$n` (für einen eindeutigen Schlüsselindex). Auch hier steht *n* für eine oder mehrere Ziffern.

Die automatische Benennung von Integritätsbedingungen auf Tabellenebene und ihrer Indizes folgt demselben Muster, es sei denn, die Namen werden explizit angegeben.

Benannte Constraints

Eine Einschränkung kann explizit benannt werden, wenn die `CONSTRAINT`-Klausel für ihre Definition verwendet wird. Während die Klausel `CONSTRAINT` zum Definieren von Einschränkungen auf Spaltenebene optional ist, ist sie für Einschränkungen auf Tabellenebene obligatorisch. Standardmäßig hat der Einschränkungsindex denselben Namen wie die Einschränkung. Wenn für den Constraint-Index ein anderer Name gewünscht wird, kann eine `USING`-Klausel eingefügt werden.

Die `USING`-Klausel

Mit der `USING`-Klausel können Sie einen benutzerdefinierten Namen für den automatisch erstellten Index angeben und optional die Richtung des Index festlegen – entweder aufsteigend (Standard) oder absteigend.

`PRIMARY KEY`

Die Einschränkung `PRIMARY KEY` basiert auf einer oder mehreren *Schlüsselspalten*, wobei für jede Spalte die Einschränkung `NOT NULL` angegeben ist. Die Werte in den Schlüsselspalten in jeder Zeile müssen eindeutig sein. Eine Tabelle kann nur einen Primärschlüssel haben.

- Ein einspaltiger Primärschlüssel kann als Einschränkung auf Spaltenebene oder als Einschränkung auf Tabellenebene definiert werden

- Als Einschränkung auf Tabellenebene muss ein mehrspaltiger Primärschlüssel angegeben werden

Die `UNIQUE`-Einschränkung

Die Einschränkung `UNIQUE` definiert die Anforderung der Eindeutigkeit des Inhalts für die Werte in einem Schlüssel in der gesamten Tabelle. Eine Tabelle kann eine beliebige Anzahl eindeutiger Schlüsseleinschränkungen enthalten.

Wie beim Primärschlüssel kann die Unique-Einschränkung mehrspaltig sein. Wenn dies der Fall ist, muss sie als Einschränkung auf Tabellenebene angegeben werden.

`NULL` in Unique Keys

Die SQL-99-kompatiblen Regeln von Firebird für `UNIQUE`-Beschränkungen erlauben eine oder mehrere `NULL`s in einer Spalte mit einer `UNIQUE`-Beschränkung. Dadurch ist es möglich, eine `UNIQUE` -Beschränkung für eine Spalte zu definieren, die nicht die `NOT NULL`-Beschränkung hat.

Bei `UNIQUE`-Schlüsseln, die sich über mehrere Spalten erstrecken, ist die Logik etwas kompliziert:

- Mehrere Zeilen mit Null in allen Spalten des Schlüssels sind zulässig
- Mehrere Zeilen mit Schlüsseln mit unterschiedlichen Kombinationen von Nullen und Nicht-Null-Werten sind zulässig
- Mehrere Zeilen mit den gleichen Schlüsselspalten null und der Rest mit Werten ungleich null sind erlaubt, sofern sich die Werte in mindestens einer Spalte unterscheiden
- Mehrere Zeilen mit den gleichen Schlüsselspalten null und der Rest mit Werten ungleich null gefüllt, die in jeder Spalte gleich sind, verletzen die Einschränkung

Die Regeln für die Eindeutigkeit lassen sich wie folgt zusammenfassen:

> Im Prinzip werden alle Nullen als verschieden betrachtet. Wenn jedoch zwei Zeilen genau die gleichen Schlüsselspalten haben, die mit Nicht-Null-Werten gefüllt sind, werden die 'NULL'-Spalten ignoriert und die Eindeutigkeit der Nicht-Null-Spalten wird so bestimmt, als ob sie den gesamten Schlüssel bilden würden.

Illustration

```
RECREATE TABLE t( x int, y int, z int, unique(x,y,z));
INSERT INTO t values( NULL, 1, 1 );
INSERT INTO t values( NULL, NULL, 1 );
INSERT INTO t values( NULL, NULL, NULL );
INSERT INTO t values( NULL, NULL, NULL ); -- Permitted
INSERT INTO t values( NULL, NULL, 1 );    -- Not permitted
```

`FOREIGN KEY`

Ein Fremdschlüssel stellt sicher, dass die teilnehmende(n) Spalte(n) nur Werte enthalten können, die auch in der/den referenzierten Spalte(n) der Mastertabelle vorhanden sind. Diese referenzierten Spalten werden oft als *target column* bezeichnet. Sie müssen der Primärschlüssel oder ein eindeutiger Schlüssel in der Zieltabelle sein. Für sie muss keine `NOT NULL`-Beschränkung definiert sein, obwohl sie, wenn sie der Primärschlüssel sind, natürlich diese Einschränkung haben.

Die Fremdschlüsselspalten in der referenzierenden Tabelle selbst erfordern keine `NOT NULL` -Einschränkung.

Ein einspaltiger Fremdschlüssel kann in der Spaltendeklaration mit dem Schlüsselwort `REFERENCES` definiert werden:

```
... ,
  ARTIFACT_ID INTEGER REFERENCES COLLECTION (ARTIFACT_ID),
```

Die Spalte `ARTIFACT_ID` im Beispiel verweist auf eine gleichnamige Spalte in der Tabelle `COLLECTIONS`.

Auf der *Tabellenebene* können sowohl einspaltige als auch mehrspaltige Fremdschlüssel definiert werden. Bei einem mehrspaltigen Fremdschlüssel ist die Deklaration auf Tabellenebene die einzige Option. Diese Methode ermöglicht auch die Bereitstellung eines optionalen Namens für die Einschränkung:

```
...
  CONSTRAINT FK_ARTSOURCE FOREIGN KEY(DEALER_ID, COUNTRY)
    REFERENCES DEALER (DEALER_ID, COUNTRY),
```

Beachten Sie, dass sich die Spaltennamen in der referenzierten Tabelle (“master”) von denen im Fremdschlüssel unterscheiden können.

Wenn keine Zielspalten angegeben sind, verweist der Fremdschlüssel automatisch auf den Primärschlüssel der Zieltabelle.

Fremdschlüsselaktionen

Mit den Unterklauseln `ON UPDATE` und `ON DELETE` ist es möglich, eine Aktion für die betroffene(n) Fremdschlüsselspalte(n) festzulegen, wenn referenzierte Werte in der Mastertabelle geändert werden:

`KEINE AKTION`

(Standard) - Nichts wird getan

`CASCADE`

Die Änderung in der Master-Tabelle wird an die entsprechende(n) Zeile(n) in der Child-Tabelle weitergegeben. Wenn sich ein Schlüsselwert ändert, ändert sich der entsprechende Schlüssel in den untergeordneten Datensätzen auf den neuen Wert; Wenn die Masterzeile gelöscht wird, werden die untergeordneten Datensätze gelöscht.

`SET DEFAULT`

Die Fremdschlüsselspalten in den betroffenen Zeilen werden auf ihre Standardwerte gesetzt *wie sie waren, als die Fremdschlüsseleinschränkung definiert wurde*.

`SET NULL`

Die Fremdschlüsselspalten in den betroffenen Zeilen werden auf `NULL` gesetzt.

Die angegebene Aktion oder die Standardeinstellung `NO ACTION` kann dazu führen, dass eine Fremdschlüsselspalte ungültig wird. Sie könnte beispielsweise einen Wert erhalten, der in der Mastertabelle nicht vorhanden ist, oder er könnte `NULL` werden, während die Spalte eine `NOT NULL` -Einschränkung hat. Solche Bedingungen führen dazu, dass die Operation in der Mastertabelle mit einer Fehlermeldung fehlschlägt.

Beispiel

```
...
  CONSTRAINT FK_ORDERS_CUST
    FOREIGN KEY (CUSTOMER) REFERENCES CUSTOMERS (ID)
      ON UPDATE CASCADE ON DELETE SET NULL
```

`CHECK`-Einschränkung

Die Einschränkung `CHECK` definiert die Bedingung, die die in diese Spalte eingefügten Werte erfüllen müssen. Eine Bedingung ist ein logischer Ausdruck (auch Prädikat genannt), der die Werte TRUE, FALSE und UNKNOWN zurückgeben kann. Eine Bedingung gilt als erfüllt, wenn das Prädikat TRUE oder den Wert UNKNOWN (entspricht `NULL`) zurückgibt. Wenn das Prädikat FALSE zurückgibt, wird der Wert nicht akzeptiert. Diese Bedingung wird zum Einfügen einer neuen Zeile in die Tabelle (die `INSERT`-Anweisung) und zum Aktualisieren des vorhandenen Wertes der Tabellenspalte (die `UPDATE` -Anweisung) und auch für Anweisungen verwendet, bei denen eine dieser Aktionen stattfinden kann (UPDATE ODER EINFÜGEN, MERGE).

Eine `CHECK`-Bedingung für eine domänenbasierte Spalte ersetzt keine vorhandene `CHECK`-Bedingung in der Domäne, sondern wird zu einer Ergänzung dazu. Die Firebird-Engine hat während der Definition keine Möglichkeit zu überprüfen, ob das zusätzliche `CHECK` nicht mit dem vorhandenen kollidiert.

`CHECK`-Einschränkungen — ob auf Tabellen- oder Spaltenebene definiert — beziehen sich auf Tabellenspalten *nach ihren Namen*. Die Verwendung des Schlüsselworts `VALUE` als Platzhalter – wie in den `CHECK`-Einschränkungen der Domäne – ist im Kontext der Definition von Spalteneinschränkungen nicht gültig.

Beispiel

mit zwei Einschränkungen auf Spaltenebene und einer auf Tabellenebene:

```
CREATE TABLE PLACES (
  ...
  LAT DECIMAL(9, 6) CHECK (ABS(LAT) <=  90),
  LON DECIMAL(9, 6) CHECK (ABS(LON) <= 180),
```

```
  ...
  CONSTRAINT CHK_POLES CHECK (ABS(LAT) < 90 OR LON = 0)
);
```

`NOT NULL`-Einschränkung

In Firebird sind Spalten standardmäßig nullable. Die Einschränkung `NOT NULL` gibt an, dass die Spalte nicht `NULL` anstelle eines Werts annehmen kann.

Ein `NOT NULL`-Constraint kann nur als Spalten-Constraint definiert werden, nicht als Tabellen-Constraint.

Wer kann eine Tabelle erstellen

Die `CREATE TABLE`-Anweisung kann ausgeführt werden durch:

- Administratoren
- Benutzer mit dem Privileg `CREATE TABLE`

Der Benutzer, der die Anweisung `CREATE TABLE` ausführt, wird Eigentümer der Tabelle.

`CREATE TABLE`-Beispiele

1. Erstellen der Tabelle "COUNTRY" mit dem als Spalteneinschränkung angegebenen Primärschlüssel.

   ```
   CREATE TABLE COUNTRY (
     COUNTRY COUNTRYNAME NOT NULL PRIMARY KEY,
     CURRENCY VARCHAR(10) NOT NULL
   );
   ```

2. Erstellen der Tabelle `STOCK` mit dem benannten Primärschlüssel, der auf Spaltenebene angegeben ist, und dem benannten eindeutigen Schlüssel, der auf Tabellenebene angegeben ist.

   ```
   CREATE TABLE STOCK (
     MODEL     SMALLINT NOT NULL CONSTRAINT PK_STOCK PRIMARY KEY,
     MODELNAME CHAR(10) NOT NULL,
     ITEMID    INTEGER NOT NULL,
     CONSTRAINT MOD_UNIQUE UNIQUE (MODELNAME, ITEMID)
   );
   ```

3. Erstellen der Tabelle "JOB" mit einer Primärschlüssel-Einschränkung, die sich über zwei Spalten erstreckt, einer Fremdschlüssel-Einschränkung für die Tabelle "COUNTRY" und einer "CHECK"-Einschränkung auf Tabellenebene. Die Tabelle enthält auch ein Array von 5 Elementen.

   ```
   CREATE TABLE JOB (
     JOB_CODE        JOBCODE NOT NULL,
   ```

```
  JOB_GRADE       JOBGRADE NOT NULL,
  JOB_COUNTRY     COUNTRYNAME,
  JOB_TITLE       VARCHAR(25) NOT NULL,
  MIN_SALARY      NUMERIC(18, 2) DEFAULT 0 NOT NULL,
  MAX_SALARY      NUMERIC(18, 2) NOT NULL,
  JOB_REQUIREMENT BLOB SUB_TYPE 1,
  LANGUAGE_REQ    VARCHAR(15) [1:5],
  PRIMARY KEY (JOB_CODE, JOB_GRADE),
  FOREIGN KEY (JOB_COUNTRY) REFERENCES COUNTRY (COUNTRY)
  ON UPDATE CASCADE
  ON DELETE SET NULL,
  CONSTRAINT CHK_SALARY CHECK (MIN_SALARY < MAX_SALARY)
);
```

4. Erstellen der Tabelle "PROJECT" mit Einschränkungen für Primär-, Fremd- und eindeutige Schlüssel mit benutzerdefinierten Indexnamen, die mit der Klausel "USING" angegeben werden.

```
CREATE TABLE PROJECT (
  PROJ_ID     PROJNO NOT NULL,
  PROJ_NAME   VARCHAR(20) NOT NULL UNIQUE USING DESC INDEX IDX_PROJNAME,
  PROJ_DESC   BLOB SUB_TYPE 1,
  TEAM_LEADER EMPNO,
  PRODUCT     PRODTYPE,
  CONSTRAINT PK_PROJECT PRIMARY KEY (PROJ_ID) USING INDEX IDX_PROJ_ID,
  FOREIGN KEY (TEAM_LEADER) REFERENCES EMPLOYEE (EMP_NO)
    USING INDEX IDX_LEADER
);
```

5. Erstellen einer Tabelle mit einer Identitätsspalte

```
create table objects (
  id integer generated by default as identity primary key,
  name varchar(15)
);

insert into objects (name) values ('Table');
insert into objects (id, name) values (10, 'Computer');
insert into objects (name) values ('Book');

select * from objects order by id;

          ID NAME
============ ===============
           1 Table
           2 Book
          10 Computer
```

6. Erstellen der Tabelle "SALARY_HISTORY" mit zwei berechneten Feldern. Das erste wird gemäß

dem SQL:2003-Standard deklariert, während das zweite gemäß der traditionellen Deklaration von berechneten Feldern in Firebird deklariert wird.

```
CREATE TABLE SALARY_HISTORY (
  EMP_NO         EMPNO NOT NULL,
  CHANGE_DATE    TIMESTAMP DEFAULT 'NOW' NOT NULL,
  UPDATER_ID     VARCHAR(20) NOT NULL,
  OLD_SALARY     SALARY NOT NULL,
  PERCENT_CHANGE DOUBLE PRECISION DEFAULT 0 NOT NULL,
  SALARY_CHANGE  GENERATED ALWAYS AS
    (OLD_SALARY * PERCENT_CHANGE / 100),
  NEW_SALARY     COMPUTED BY
    (OLD_SALARY + OLD_SALARY * PERCENT_CHANGE / 100)
);
```

Global Temporary Tables (GTT)

Globale temporäre Tabellen verfügen über persistente Metadaten, ihr Inhalt ist jedoch transaktionsgebunden (Standard) oder verbindungsgebunden. Jede Transaktion oder Verbindung hat ihre eigene private Instanz einer GTT, die von allen anderen isoliert ist. Instanzen werden nur erstellt, wenn und wenn auf die GTT verwiesen wird. Sie werden zerstört, wenn die Transaktion endet oder wenn die Verbindung getrennt wird. Die Metadaten einer GTT können mit `ALTER TABLE` bzw. `DROP TABLE` geändert oder entfernt werden.

Syntax

```
CREATE GLOBAL TEMPORARY TABLE tablename
  (<column_def> [, {<column_def> | <table_constraint>} ...])
  [ON COMMIT {DELETE | PRESERVE} ROWS]
```

Syntax notes

- `ON COMMIT DELETE ROWS` erstellt eine GTT auf Transaktionsebene (Standard), `ON COMMIT PRESERVE ROWS` eine GTT auf Verbindungsebene
- Eine `EXTERNAL [FILE]`-Klausel ist in der Definition einer globalen temporären Tabelle nicht erlaubt

Seit Firebird 3.0 sind GTTs in schreibgeschützten Transaktionen beschreibbar. Die Wirkung ist wie folgt:

Schreibgeschützte Transaktion in der Datenbank mit Lese-/Schreibzugriff

Schreibbar in `ON COMMIT PRESERVE ROWS` und `ON COMMIT DELETE ROWS`

Schreibgeschützte Transaktion in schreibgeschützter Datenbank

Nur in `ON COMMIT DELETE ROWS` beschreibbar

Einschränkungen für GTTs

GTTs können mit allen Funktionen und Utensilien gewöhnlicher Tabellen (Schlüssel, Referenzen, Indizes, Trigger usw.) “aufgeputzt” werden, aber es gibt einige Einschränkungen:

- GTTs und reguläre Tabellen können nicht aufeinander verweisen
- Eine verbindungsgebundene (“PRESERVE ROWS”) GTT kann nicht auf eine transaktionsgebundene (“DELETE ROWS”) GTT verweisen
- Domäneneinschränkungen können keine GTT referenzieren
- Die Zerstörung einer GTT-Instanz am Ende ihres Lebenszyklus führt nicht zum Auslösen von BEFORE/AFTER Delete-Triggern

In einer bestehenden Datenbank ist es nicht immer einfach, eine reguläre Tabelle von einer GTT oder eine GTT auf Transaktionsebene von einer GTT auf Verbindungsebene zu unterscheiden. Verwenden Sie diese Abfrage, um herauszufinden, welche Art von Tabelle Sie betrachten:

```
select t.rdb$type_name
from rdb$relations r
join rdb$types t on r.rdb$relation_type = t.rdb$type
where t.rdb$field_name = 'RDB$RELATION_TYPE'
and r.rdb$relation_name = 'TABLENAME'
```

Für einen Überblick über die Typen aller Relationen in der Datenbank:

```
select r.rdb$relation_name, t.rdb$type_name
from rdb$relations r
join rdb$types t on r.rdb$relation_type = t.rdb$type
where t.rdb$field_name = 'RDB$RELATION_TYPE'
and coalesce (r.rdb$system_flag, 0) = 0
```

Das Feld RDB$TYPE_NAME zeigt PERSISTENT für eine reguläre Tabelle, VIEW für eine Ansicht, GLOBAL_TEMPORARY_PRESERVE für eine verbindungsgebundene GTT und GLOBAL_TEMPORARY_DELETE für eine transaktionsgebundene GTT.

Beispiele für globale temporäre Tabellen

1. Erstellen einer globalen temporären Tabelle mit Verbindungsbereich.

```
CREATE GLOBAL TEMPORARY TABLE MYCONNGTT (
  ID  INTEGER NOT NULL PRIMARY KEY,
  TXT VARCHAR(32),
  TS  TIMESTAMP DEFAULT CURRENT_TIMESTAMP)
ON COMMIT PRESERVE ROWS;
```

2. Erstellen einer transaktionsbezogenen globalen temporären Tabelle, die einen Fremdschlüssel

verwendet, um auf eine verbindungsbezogene globale temporäre Tabelle zu verweisen. Die Unterklausel `ON COMMIT` ist optional, da `DELETE ROWS` die Vorgabe ist.

```
CREATE GLOBAL TEMPORARY TABLE MYTXGTT (
  ID        INTEGER NOT NULL PRIMARY KEY,
  PARENT_ID INTEGER NOT NULL REFERENCES MYCONNGTT(ID),
  TXT       VARCHAR(32),
  TS        TIMESTAMP DEFAULT CURRENT_TIMESTAMP
) ON COMMIT DELETE ROWS;
```

Externe Tabellen

Die optionale `EXTERNAL [FILE]`-Klausel gibt an, dass die Tabelle außerhalb der Datenbank in einer externen Textdatei mit Datensätzen fester Länge gespeichert wird. Die Spalten einer Tabelle, die in einer externen Datei gespeichert sind, können jeden beliebigen Typ haben, außer 'BLOB' oder 'ARRAY', obwohl für die meisten Zwecke nur Spalten des Typs 'CHAR' nützlich wären.

Mit einer in einer externen Datei gespeicherten Tabelle können Sie nur neue Zeilen einfügen (`INSERT`) und die Daten abfragen (`SELECT`). Das Aktualisieren vorhandener Daten (`UPDATE`) und das Löschen von Zeilen (`DELETE`) sind nicht möglich.

Eine Datei, die als externe Tabelle definiert ist, muss sich auf einem Speichergerät befinden, das physisch auf dem Computer vorhanden ist, auf dem der Firebird-Server läuft, und wenn der Parameter *ExternalFileAccess* in der Konfigurationsdatei `firebird.conf` den Wert `Restrict` hat , muss es in einem der dort aufgeführten Verzeichnisse als Argument für `Restrict` liegen. Wenn die Datei noch nicht existiert, erstellt Firebird sie beim ersten Zugriff.

Die Möglichkeit, externe Dateien für eine Tabelle zu verwenden, hängt vom Wert ab, der für den Parameter *ExternalFileAccess* in `firebird.conf` festgelegt wurde:

- Wenn es auf `None` (Standard) gesetzt ist, wird jeder Versuch, auf eine externe Datei zuzugreifen, abgelehnt.
- Die Einstellung `Beschränken` wird empfohlen, um den externen Dateizugriff auf Verzeichnisse einzuschränken, die explizit für diesen Zweck vom Serveradministrator erstellt wurden. Zum Beispiel:
  - `ExternalFileAccess = Restrict externalfiles` beschränkt den Zugriff auf ein Verzeichnis namens `externalfiles` direkt unter dem Firebird-Stammverzeichnis
  - `ExternalFileAccess = d:\databases\outfiles; e:\infiles` beschränkt den Zugriff auf nur diese beiden Verzeichnisse auf dem Windows-Hostserver. Beachten Sie, dass alle Pfade, die eine Netzwerkzuordnung darstellen, nicht funktionieren. Pfade, die in einfache oder doppelte Anführungszeichen eingeschlossen sind, funktionieren ebenfalls nicht.
- Wenn dieser Parameter auf `Full` gesetzt ist, kann auf externe Dateien überall im Host-Dateisystem zugegriffen werden. Dies schafft eine Sicherheitslücke und wird nicht empfohlen.

Externes Dateiformat

Das “row”-Format der externen Tabelle hat eine feste Länge und ist binär. Es gibt keine Feldbegrenzer: Sowohl Feld- als auch Zeilengrenzen werden durch die maximale Größe der Felddefinitionen in Byte bestimmt. Dies ist sowohl bei der Definition der Struktur der externen Tabelle als auch beim Entwurf einer Eingabedatei für eine externe Tabelle wichtig, die Daten aus einer anderen Anwendung importieren soll. Das allgegenwärtige Format “.csv” zum Beispiel ist als Eingabedatei unbrauchbar und kann nicht direkt in eine externe Datei generiert werden.

Der nützlichste Datentyp für die Spalten externer Tabellen ist der Typ "CHAR" mit fester Länge und geeigneter Länge für die zu übertragenden Daten. Datums- und Zahlentypen lassen sich leicht in und aus Strings umwandeln, während die nativen Datentypen – Binärdaten – für externe Anwendungen als nicht zu analysierendes "Alphabetti" erscheinen, es sei denn, die Dateien sollen von einer anderen Firebird-Datenbank gelesen werden.

Natürlich gibt es Möglichkeiten, typisierte Daten zu manipulieren, um Ausgabedateien von Firebird zu erzeugen, die direkt als Eingabedateien für andere Anwendungen gelesen werden können, unter Verwendung von gespeicherten Prozeduren, mit oder ohne Verwendung externer Tabellen. Solche Techniken gehen über den Umfang einer Sprachreferenz hinaus. Hier geben wir einige Richtlinien und Tipps zum Erstellen und Arbeiten mit einfachen Textdateien, da die externe Tabellenfunktion oft als einfache Möglichkeit verwendet wird, transaktionsunabhängige Protokolle zu erstellen oder zu lesen, die offline in einem Texteditor oder Auditing untersucht werden können Anwendung.

Zeilentrennzeichen

Im Allgemeinen sind externe Dateien nützlicher, wenn Zeilen durch ein Trennzeichen in Form einer “newline”-Sequenz getrennt werden, die von Reader-Anwendungen auf der vorgesehenen Plattform erkannt wird. Für die meisten Kontexte unter Windows ist es die Zwei-Byte-'CRLF'-Sequenz, Wagenrücklauf (ASCII-Code dezimal 13) und Zeilenvorschub (ASCII-Code dezimal 10). Auf POSIX ist LF allein üblich; bei einigen MacOSX-Anwendungen kann es LFCR sein. Es gibt verschiedene Möglichkeiten, diese Trennzeichenspalte zu füllen. In unserem Beispiel unten geschieht dies mit einem `BEFORE INSERT` Trigger und der internen Funktion `ASCII_CHAR`.

Beispiel für eine externe Tabelle

In unserem Beispiel definieren wir eine externe Protokolltabelle, die von einem Ausnahmehandler in einer gespeicherten Prozedur oder einem Trigger verwendet werden könnte. Die externe Tabelle wird ausgewählt, weil die Nachrichten von allen behandelten Ausnahmen im Protokoll aufbewahrt werden, selbst wenn die Transaktion, die den Prozess gestartet hat, schließlich aufgrund einer anderen, nicht behandelten Ausnahme zurückgesetzt wird. Zu Demonstrationszwecken hat es nur zwei Datenspalten, einen Zeitstempel und eine Nachricht. Die dritte Spalte speichert das Zeilentrennzeichen:

```
CREATE TABLE ext_log
  EXTERNAL FILE 'd:\externals\log_me.txt' (
  stamp CHAR (24),
  message CHAR(100),
  crlf CHAR(2) -- for a Windows context
```

```
);
COMMIT;
```

Jetzt ein Trigger, um den Zeitstempel und das Zeilentrennzeichen jedes Mal zu schreiben, wenn eine Nachricht in die Datei geschrieben wird:

```
SET TERM ^;
CREATE TRIGGER bi_ext_log FOR ext_log
ACTIVE BEFORE INSERT
AS
BEGIN
  IF (new.stamp is NULL) then
    new.stamp = CAST (CURRENT_TIMESTAMP as CHAR(24));
  new.crlf = ASCII_CHAR(13) || ASCII_CHAR(10);
END ^
COMMIT ^
SET TERM ;^
```

Einfügen einiger Datensätze (was von einem Ausnahmehandler oder einem Shakespeare-Fan hätte erfolgen können):

```
insert into ext_log (message)
values('Shall I compare thee to a summer''s day?');
insert into ext_log (message)
values('Thou art more lovely and more temperate');
```

Die Ausgabe:

```
2015-10-07 15:19:03.4110Shall I compare thee to a summer's day?
2015-10-07 15:19:58.7600Thou art more lovely and more temperate
```

5.4.2. ALTER TABLE

Verwendet für

Ändern der Struktur einer Tabelle.

Verfügbar in

DSQL, ESQL

Syntax

```
ALTER TABLE tablename
  <operation> [, <operation> ...]

<operation> ::=
    ADD <col_def>
```

```
  | ADD <tconstraint>
  | DROP colname
  | DROP CONSTRAINT constr_name
  | ALTER [COLUMN] colname <col_mod>

<col_def> ::=
    <regular_col_def>
  | <computed_col_def>
  | <identity_col_def>

<regular_col_def> ::=
  colname {<datatype> | domainname}
  [DEFAULT {<literal> | NULL | <context_var>}]
  [<col_constraint> ...]
  [COLLATE collation_name]

<computed_col_def> ::=
  colname [{<datatype> | domainname}]
  {COMPUTED [BY] | GENERATED ALWAYS AS} (<expression>)

<identity_col_def> ::=
  colname {<datatype> | domainname}
  GENERATED BY DEFAULT AS IDENTITY [(START WITH startvalue)]
  [<col_constraint> ...]

<col_mod> ::=
    TO newname
  | POSITION newpos
  | <regular_col_mod>
  | <computed_col_mod>
  | <identity_col_mod>

<regular_col_mod> ::=
    TYPE {<datatype> | domainname}
  | SET DEFAULT {<literal> | NULL | <context_var>}
  | DROP DEFAULT
  | {SET | DROP} NOT NULL

<computed_col_mod> ::=
    [TYPE <datatype>] {COMPUTED [BY] | GENERATED ALWAYS AS} (<expression>)

<identity_col_mod> ::=
    RESTART [WITH startvalue]

!! Siehe auch CREATE TABLE-Syntax für weitere Regeln!!
```

Tabelle 27. `ALTER TABLE`-*Anweisungsparameter*

Parameter	Beschreibung
tablename	Name (Bezeichner) der Tabelle

Parameter	Beschreibung
operation	Eine der verfügbaren Operationen, die die Struktur der Tabelle ändern
colname	Name (Bezeichner) für eine Spalte in der Tabelle, max. 31 Zeichen. Muss in der Tabelle eindeutig sein.
domain_name	Domainname
newname	Neuer Name (Bezeichner) für die Spalte, max. 31 Zeichen. Muss in der Tabelle eindeutig sein.
newpos	Die neue Spaltenposition (eine ganze Zahl zwischen 1 und der Anzahl der Spalten in der Tabelle)
start_value	Der erste Wert der Identitätsspalte nach dem Neustart
other_table	Der Name der Tabelle, auf die von der Fremdschlüsseleinschränkung verwiesen wird
literal	Ein Literalwert, der im angegebenen Kontext zulässig ist
context_var	Eine Kontextvariable, deren Typ im angegebenen Kontext zulässig ist
check_condition	Die Bedingung einer `CHECK`-Einschränkung, die erfüllt wird, wenn sie als TRUE oder UNKNOWN/NULL ausgewertet wird
collation	Name einer Kollatierungssequenz, die für *charset_name* gültig ist, wenn sie mit *datatype* versorgt wird oder ansonsten für den Standardzeichensatz der Datenbank gültig ist

Die Anweisung `ALTER TABLE` ändert die Struktur einer bestehenden Tabelle. Mit einer `ALTER TABLE` -Anweisung ist es möglich, mehrere Operationen auszuführen, Spalten und Einschränkungen hinzuzufügen/zu löschen und auch Spaltenspezifikationen zu ändern.

Mehrere Operationen in einer `ALTER TABLE`-Anweisung werden durch Kommas getrennt.

Versionsanzahl-Inkremente

Einige Änderungen in der Struktur einer Tabelle erhöhen den Metadaten-Änderungszähler (“version count”), der jeder Tabelle zugewiesen ist. Die Anzahl der Metadatenänderungen ist für jede Tabelle auf 255 begrenzt. Sobald der Zähler die Grenze von 255 erreicht hat, können Sie keine weiteren Änderungen an der Struktur der Tabelle vornehmen, ohne den Zähler zurückzusetzen.

So setzen Sie den Metadaten-Änderungszähler zurück

Sie müssen die Datenbank mit dem Dienstprogramm *gbak* sichern und wiederherstellen.

Die `ADD`-Klausel

Mit der `ADD`-Klausel können Sie eine neue Spalte oder eine neue Tabelleneinschränkung hinzufügen. Die Syntax zum Definieren der Spalte und die Syntax zum Definieren der Tabelleneinschränkung entsprechen denen, die für die `CREATE TABLE`-Anweisung beschrieben wurden.

Auswirkung auf die Versionsanzahl

- Jedes Mal, wenn eine neue Spalte hinzugefügt wird, wird der Metadatenänderungszähler um eins erhöht
- Das Hinzufügen einer neuen Tabelleneinschränkung erhöht nicht den Metadatenänderungszähler

Zu beachtende Punkte

1. Das Hinzufügen einer Spalte mit einer `NOT NULL`-Einschränkung ohne einen `DEFAULT`-Wert wird - seit Firebird 3.0 - fehlschlagen, wenn die Tabelle bereits Zeilen enthält. Beim Hinzufügen einer Nicht-Nullable-Spalte wird empfohlen, entweder einen Standardwert dafür festzulegen oder sie als Nullable-fähig zu erstellen, die Spalte in vorhandenen Zeilen mit einem Nicht-Null-Wert zu aktualisieren und dann eine `NOT NULL`-Einschränkung hinzuzufügen.
2. Wenn eine neue 'CHECK'-Beschränkung hinzugefügt wird, werden vorhandene Daten nicht auf Übereinstimmung getestet. Es wird empfohlen, vorhandene Daten vorab mit dem neuen 'CHECK'-Ausdruck zu testen.
3. Obwohl das Hinzufügen einer Identitätsspalte unterstützt wird, ist dies nur erfolgreich, wenn die Tabelle leer ist. Das Hinzufügen einer Identitätsspalte schlägt fehl, wenn die Tabelle eine oder mehrere Zeilen enthält.

Die `DROP`-Klausel

Die Klausel `DROP colname` löscht die angegebene Spalte aus der Tabelle. Ein Versuch, eine Spalte zu löschen, schlägt fehl, wenn etwas darauf verweist. Betrachten Sie die folgenden Elemente als Quellen potenzieller Abhängigkeiten:

- Spalten- oder Tabellenbeschränkungen
- Indizes
- gespeicherte Prozeduren und Trigger
- Aufrufe

Auswirkung auf die Versionsanzahl

- Jedes Mal, wenn eine Spalte gelöscht wird, wird der Metadaten-Änderungszähler der Tabelle um eins erhöht.

Die `DROP CONSTRAINT`-Klausel

Die Klausel `DROP CONSTRAINT` löscht die angegebene Einschränkung auf Spalten- oder Tabellenebene.

Eine `PRIMARY KEY`- oder `UNIQUE`-Schlüsseleinschränkung kann nicht gelöscht werden, wenn sie von einer `FOREIGN KEY`-Einschränkung in einer anderen Tabelle referenziert wird. Es wird notwendig sein, diese `FOREIGN KEY`-Beschränkung zu löschen, bevor versucht wird, die `PRIMARY KEY`- oder `UNIQUE` -Schlüssel-Beschränkung, auf die sie verweist, zu löschen.

Auswirkung auf die Versionsanzahl

- Das Löschen einer Spalteneinschränkung oder einer Tabelleneinschränkung erhöht den

Metadatenänderungszähler nicht.

Die `ALTER [COLUMN]`-Klausel

Mit der `ALTER [COLUMN]`-Klausel können Attribute vorhandener Spalten geändert werden, ohne dass die Spalte gelöscht und erneut hinzugefügt werden muss. Erlaubte Modifikationen sind:

- den Namen ändern (hat keinen Einfluss auf den Metadaten-Änderungszähler)
- den Datentyp ändern (erhöht den Metadaten-Änderungszähler um eins)
- die Spaltenposition in der Spaltenliste der Tabelle ändern (hat keinen Einfluss auf den Metadaten-Änderungszähler)
- den Standardspaltenwert löschen (hat keinen Einfluss auf den Metadaten-Änderungszähler)
- einen Standardspaltenwert festlegen oder den vorhandenen Standardwert ändern (hat keinen Einfluss auf den Metadatenänderungszähler)
- Typ und Ausdruck für eine berechnete Spalte ändern (hat keinen Einfluss auf den Metadaten-Änderungszähler)
- Setzen Sie die Einschränkung `NOT NULL` (hat keinen Einfluss auf den Metadaten-Änderungszähler)
- lösche die `NOT NULL`-Beschränkung (hat keinen Einfluss auf den Metadaten-Änderungszähler)

Umbenennen einer Spalte: die `TO`-Klausel

Das Schlüsselwort `TO` mit einem neuen Bezeichner benennt eine vorhandene Spalte um. Die Tabelle darf keine vorhandene Spalte mit demselben Bezeichner aufweisen.

Es ist nicht möglich, den Namen einer Spalte zu ändern, die in einer Einschränkung enthalten ist: `PRIMARY KEY`, `UNIQUE`-Schlüssel, `FOREIGN KEY`, Spaltenbeschränkung oder die `CHECK`-Beschränkung der Tabelle.

Das Umbenennen einer Spalte ist auch nicht zulässig, wenn die Spalte in einem Trigger, einer gespeicherten Prozedur oder einer Ansicht verwendet wird.

Ändern des Datentyps einer Spalte: die `TYPE`-Klausel

Das Schlüsselwort `TYPE` ändert den Datentyp einer existierenden Spalte in einen anderen zulässigen Typ. Eine Typänderung, die zu Datenverlust führen könnte, wird nicht zugelassen. Beispielsweise darf die Anzahl der Zeichen im neuen Typ für eine `CHAR`- oder `VARCHAR`-Spalte nicht kleiner sein als die dafür vorhandene Spezifikation.

Wurde die Spalte als Array deklariert, darf weder der Typ noch die Anzahl der Dimensionen geändert werden.

Der Datentyp einer Spalte, die an einem Fremdschlüssel, Primärschlüssel oder einer eindeutigen Einschränkung beteiligt ist, kann nicht geändert werden.

Ändern der Position einer Spalte: die POSITION-Klausel

Das Schlüsselwort POSITION ändert die Position einer vorhandenen Spalte im fiktiven "von links nach rechts"-Layout des Datensatzes.

Die Nummerierung der Spaltenpositionen beginnt bei 1.

- Wenn eine Position kleiner als 1 angegeben wird, wird eine Fehlermeldung zurückgegeben
- Wenn eine Positionsnummer größer als die Anzahl der Spalten in der Tabelle ist, wird ihre neue Position stillschweigend an die Anzahl der Spalten angepasst.

Die Klauseln DROP DEFAULT und SET DEFAULT

Die optionale DROP DEFAULT-Klausel löscht den Standardwert für die Spalte, wenn er zuvor durch eine CREATE TABLE- oder ALTER TABLE-Anweisung dort abgelegt wurde.

- Wenn die Spalte auf einer Domäne mit einem Standardwert basiert, wird der Standardwert auf den Domänenstandard zurückgesetzt
- Ein Ausführungsfehler wird ausgelöst, wenn versucht wird, den Standardwert einer Spalte zu löschen, die keinen Standardwert hat oder deren Standardwert domänenbasiert ist

Die optionale SET DEFAULT-Klausel setzt einen Standardwert für die Spalte. Wenn die Spalte bereits einen Standardwert hat, wird dieser durch den neuen ersetzt. Der auf eine Spalte angewendete Standardwert überschreibt immer einen von einer Domäne geerbten Wert.

Die Klauseln SET NOT NULL und DROP NOT NULL

Die SET NOT NULL-Klausel fügt einer vorhandenen Tabellenspalte eine NOT NULL-Einschränkung hinzu. Im Gegensatz zur Definition in CREATE TABLE ist die Angabe eines Constraint-Namens nicht möglich.

Das erfolgreiche Hinzufügen der NOT NULL-Einschränkung unterliegt einer vollständigen Datenvalidierung für die Tabelle. Stellen Sie daher sicher, dass die Spalte keine Nullen enthält, bevor Sie die Änderung vornehmen.

Eine explizite NOT NULL-Einschränkung für domänenbasierte Spalten überschreibt Domäneneinstellungen. In diesem Szenario erstreckt sich das Ändern der Domäne in NULL-Zulässigkeit nicht auf eine Tabellenspalte.

Das Löschen der NOT NULL-Beschränkung aus der Spalte, wenn ihr Typ eine Domäne ist, die auch eine NOT NULL-Beschränkung hat, hat keine beobachtbaren Auswirkungen, bis die NOT NULL -Beschränkung ebenfalls aus der Domäne gelöscht wird.

Die Klauseln COMPUTED [BY] oder GENERATED ALWAYS AS

Der einer berechneten Spalte zugrunde liegende Datentyp und Ausdruck können mit einer COMPUTED [BY]- oder GENERATED ALWAYS AS-Klausel in der ALTER TABLE ALTER [COLUMN]-Anweisung geändert werden. Das Konvertieren einer regulären Spalte in eine berechnete und umgekehrt ist nicht zulässig.

Identitätsspalten ändern

Für Identitätsspalten (`GENERATED BY DEFAULT AS IDENTITY`) ist es möglich, die zum Generieren von Identitätswerten verwendete Sequenz neu zu starten. Wenn nur die `RESTART`-Klausel angegeben wird, wird die Sequenz auf den bei `CREATE TABLE` angegebenen Anfangswert zurückgesetzt. Wenn die optionale `WITH start_value`-Klausel angegeben wird, wird die Sequenz mit dem angegebenen Wert neu gestartet.

Es ist nicht möglich, eine vorhandene Spalte in eine Identitätsspalte oder eine Identitätsspalte in eine normale Spalte umzuwandeln. Firebird 4 wird die Möglichkeit einführen, eine Identitätsspalte in eine normale Spalte umzuwandeln.

Der Neustart unterliegt derzeit einem Fehler: der erste nach einem Neustart generierte Wert ist 1 (eins) höher als der konfigurierte Anfangswert (oder der durch `WITH` angegebene Wert) .Siehe auch Identitätsspalten (autoinkrement).

Attribute, die nicht geändert werden können

Die folgenden Änderungen werden nicht unterstützt:

- Ändern der Sortierung einer Zeichentypspalte

Wer kann eine Tabelle ändern?

Die `ALTER TABLE`-Anweisung kann ausgeführt werden durch:

- Administratoren
- Der Besitzer der Tabelle
- Benutzer mit der Berechtigung `ALTER ANY TABLE`

Beispiele für die Verwendung von `ALTER TABLE`

1. Hinzufügen der Spalte "CAPITAL" zur Tabelle "COUNTRY".

```
ALTER TABLE COUNTRY
  ADD CAPITAL VARCHAR(25);
```

2. Hinzufügen der Spalte "CAPITAL" mit den Einschränkungen "NOT NULL" und "UNIQUE" und Löschen der Spalte "CURRENCY".

```
ALTER TABLE COUNTRY
  ADD CAPITAL VARCHAR(25) NOT NULL UNIQUE,
  DROP CURRENCY;
```

3. Hinzufügen der Prüfbedingung `CHK_SALARY` und eines Fremdschlüssels zur Tabelle `JOB`.

```
ALTER TABLE JOB
```

```
  ADD CONSTRAINT CHK_SALARY CHECK (MIN_SALARY < MAX_SALARY),
  ADD FOREIGN KEY (JOB_COUNTRY) REFERENCES COUNTRY (COUNTRY);
```

4. Festlegen des Standardwerts für das Feld "MODEL", Ändern des Typs der Spalte "ITEMID" und Umbenennen der Spalte MODELNAME.

```
ALTER TABLE STOCK
  ALTER COLUMN MODEL SET DEFAULT 1,
  ALTER COLUMN ITEMID TYPE BIGINT,
  ALTER COLUMN MODELNAME TO NAME;
```

5. Neustart der Sequenz einer Identitätsspalte.

```
ALTER TABLE objects
  ALTER ID RESTART WITH 100;
```

6. Ändern der berechneten Spalten `NEW_SALARY` und `SALARY_CHANGE`.

```
ALTER TABLE SALARY_HISTORY
  ALTER NEW_SALARY GENERATED ALWAYS AS
    (OLD_SALARY + OLD_SALARY * PERCENT_CHANGE / 100),
  ALTER SALARY_CHANGE COMPUTED BY
    (OLD_SALARY * PERCENT_CHANGE / 100);
```

Siehe auch

`CREATE TABLE`, `DROP TABLE`, `CREATE DOMAIN`

5.4.3. DROP TABLE

Verwendet für

Löschen (Löschen) einer Tabelle

Verfügbar in

DSQL, ESQL

Syntax

```
DROP TABLE tablename
```

Tabelle 28. `DROP TABLE`*-Anweisungsparameter*

Parameter	Beschreibung
tablename	Name (Bezeichner) der Tabelle

Die Anweisung `DROP TABLE` löscht (löscht) eine vorhandene Tabelle. Wenn die Tabelle

Abhängigkeiten aufweist, schlägt die Anweisung DROP TABLE mit einem Ausführungsfehler fehl.

Wenn eine Tabelle gelöscht wird, werden auch alle ihre Trigger und Indizes gelöscht.

Wer kann eine Tabelle löschen?

Die DROP TABLE-Anweisung kann ausgeführt werden durch:

- Administratoren
- Der Besitzer der Tabelle
- Benutzer mit dem Privileg DROP ANY TABLE

Beispiel für DROP TABLE

Löschen der 'COUNTRY'-Tabelle.

```
DROP TABLE COUNTRY;
```

Siehe auch

CREATE TABLE, ALTER TABLE, RECREATE TABLE

5.4.4. RECREATE TABLE

Verwendet für

Erstellen einer neuen Tabelle (Relation) oder Wiederherstellen einer bestehenden Tabelle

Verfügbar in

DSQL

Syntax

```
RECREATE [GLOBAL TEMPORARY] TABLE tablename
  [EXTERNAL [FILE] 'filespec']
  (<col_def> [, {<col_def> | <tconstraint>} ...])
  [ON COMMIT {DELETE | PRESERVE} ROWS]
```

Siehe Abschnitt CREATE TABLE für die vollständige Syntax von CREATE TABLE und Beschreibungen zur Definition von Tabellen, Spalten und Einschränkungen.

RECREATE TABLE erstellt oder erstellt eine Tabelle neu. Existiert bereits eine Tabelle mit diesem Namen, versucht die Anweisung RECREATE TABLE, sie zu löschen und eine neue zu erstellen. Vorhandene Abhängigkeiten verhindern die Ausführung der Anweisung.

Beispiel für RECREATE TABLE

Erstellen oder Neuerstellen der Tabelle 'COUNTRY'.

```
RECREATE TABLE COUNTRY (
```

```
  COUNTRY COUNTRYNAME NOT NULL PRIMARY KEY,
  CURRENCY VARCHAR(10) NOT NULL
);
```

Siehe auch

CREATE TABLE, DROP TABLE

5.5. INDEX

Ein Index ist ein Datenbankobjekt, das zum schnelleren Abrufen von Daten aus einer Tabelle oder zum schnelleren Sortieren in einer Abfrage verwendet wird. Indizes werden auch verwendet, um die referenziellen Integritätsbedingungen PRIMARY KEY, FOREIGN KEY und UNIQUE zu erzwingen.

In diesem Abschnitt wird beschrieben, wie Sie Indizes erstellen, aktivieren und deaktivieren, löschen und Statistiken dazu sammeln (Selektivität neu berechnen).

5.5.1. CREATE INDEX

Verwendet für

Erstellen eines Index für eine Tabelle

Verfügbar in

DSQL, ESQL

Syntax

```
CREATE [UNIQUE] [ASC[ENDING] | DESC[ENDING]]
  INDEX indexname ON tablename
  {(col [, col …]) | COMPUTED BY (<expression>)}
```

Tabelle 29. CREATE INDEX*-Anweisungsparameter*

Parameter	Beschreibung
indexname	Indexname. Es kann aus bis zu 31 Zeichen bestehen
tablename	Der Name der Tabelle, für die der Index erstellt werden soll
col	Name einer Spalte in der Tabelle. Spalten der Typen BLOB und ARRAY sowie berechnete Felder können nicht in einem Index verwendet werden
expression	Der Ausdruck, der die Werte für einen berechneten Index berechnet, auch als “Ausdrucksindex” bekannt

Die CREATE INDEX-Anweisung erstellt einen Index für eine Tabelle, mit dem das Suchen, Sortieren und Gruppieren beschleunigt werden kann. Indizes werden beim Definieren von Constraints wie Primärschlüssel-, Fremdschlüssel- oder Unique-Constraints automatisch erstellt.

Ein Index kann auf dem Inhalt von Spalten jedes Datentyps mit Ausnahme von 'BLOB' und Arrays erstellt werden. Der Name (Bezeichner) eines Index muss unter allen Indexnamen eindeutig sein.

Schlüsselindizes

Wenn einer Tabelle oder Spalte ein Primärschlüssel, Fremdschlüssel oder eine eindeutige Einschränkung hinzugefügt wird, wird automatisch ein Index mit demselben Namen ohne ausdrückliche Anweisung des Designers erstellt. Der Index `PK_COUNTRY` wird beispielsweise automatisch erstellt, wenn Sie die folgende Anweisung ausführen und festschreiben:

```
ALTER TABLE COUNTRY ADD CONSTRAINT PK_COUNTRY
  PRIMARY KEY (ID);
```

Wer kann einen Index erstellen?

Die `CREATE INDEX`-Anweisung kann ausgeführt werden durch:

- Administratoren
- Der Besitzer der Tabelle
- Benutzer mit dem `ALTER ANY TABLE`-Privileg

Eindeutige Indizes

Die Angabe des Schlüsselworts `UNIQUE` in der Anweisung zur Indexerstellung erstellt einen Index, in dem die Eindeutigkeit in der gesamten Tabelle erzwungen wird. Der Index wird als „eindeutiger Index“ bezeichnet. Ein eindeutiger Index ist keine Einschränkung.

Eindeutige Indizes dürfen keine doppelten Schlüsselwerte (oder doppelte Schlüsselwertkombinationen im Fall von *zusammengesetzten* oder mehrspaltigen oder mehrsegmentigen) Indizes enthalten. Doppelte `NULL`s sind gemäß dem SQL:99-Standard sowohl in Einzelsegment- als auch in Mehrfachsegment-Indizes erlaubt.

Indexrichtung

Alle Indizes in Firebird sind unidirektional. Ein Index kann vom niedrigsten Wert zum höchsten (aufsteigende Reihenfolge) oder vom höchsten zum niedrigsten Wert (absteigende Reihenfolge) aufgebaut werden. Die Schlüsselwörter `ASC[ENDING]` und `DESC[ENDING]` werden verwendet, um die Richtung des Index anzugeben. Die Standardindexreihenfolge ist `ASC[ENDING]`. Es ist durchaus zulässig, sowohl einen aufsteigenden als auch einen absteigenden Index für dieselbe Spalte oder denselben Schlüsselsatz zu definieren.

Ein absteigender Index kann für eine Spalte nützlich sein, die nach den hohen Werten (""neuest", Maximum usw.) gesucht wird.

Firebird verwendet B-tree-Indizes, die bidirektional sind. Aufgrund technischer Einschränkungen verwendet Firebird jedoch einen Index nur in eine Richtung.

Siehe auch Firebird für den Datenbankexperten: Episode 3 – Auf Datenträgerkonsistenz

Berechnete (Ausdrucks-)Indizes

Beim Erstellen eines Index können Sie die `COMPUTED BY`-Klausel verwenden, um einen Ausdruck anstelle einer oder mehrerer Spalten anzugeben. Berechnete Indizes werden in Abfragen verwendet, bei denen die Bedingung in einer `WHERE`-, `ORDER BY`- oder `GROUP BY`-Klausel genau mit dem Ausdruck in der Indexdefinition übereinstimmt. Der Ausdruck in einem berechneten Index kann mehrere Spalten in der Tabelle umfassen.

> Sie können tatsächlich einen berechneten Index für ein berechnetes Feld erstellen, aber ein solcher Index wird niemals verwendet.

Beschränkungen für Indizes

Für Indizes gelten bestimmte Grenzen.

Die maximale Länge eines Schlüssels in einem Index ist auf ¼ der Seitengröße begrenzt.

Maximale Indizes pro Tabelle

Die Anzahl der Indizes, die für jede Tabelle untergebracht werden können, ist begrenzt. Das tatsächliche Maximum für eine bestimmte Tabelle hängt von der Seitengröße und der Anzahl der Spalten in den Indizes ab.

Tabelle 30. Maximale Indizes pro Tabelle

Seitengröße (Page size)	Anzahl der Indizes abhängig von der Spaltenanzahl		
	Einspaltig	Zweispaltig	Dreispaltig
4096	203	145	113
8192	408	291	227
16384	818	584	454

Zeichenindexbeschränkungen

Die maximale Länge der indizierten Zeichenfolge beträgt 9 Byte weniger als die maximale Schlüssellänge. Die maximale Länge der indexierbaren Zeichenfolge hängt von der Seitengröße und dem Zeichensatz ab.

Tabelle 31. Maximale indexierbare (VAR)CHAR-Länge

Seitengröße (Page size)	Maximale Länge der indizierbaren Zeichenfolge nach Zeichensatztyp			
	1 Byte/Zeichen	2 Bytes/Zeichen	3 Bytes/Zeichen	4 Bytes/Zeichen
4096	1015	507	338	253
8192	2039	1019	679	509
16384	4087	2043	1362	1021

Beispiele für die Verwendung von CREATE INDEX

1. Erstellen eines Index für die Spalte UPDATER_ID in der Tabelle SALARY_HISTORY

```
CREATE INDEX IDX_UPDATER
  ON SALARY_HISTORY (UPDATER_ID);
```

2. Erstellen eines Index mit in absteigender Reihenfolge sortierten Schlüsseln für die Spalte CHANGE_DATE in der Tabelle SALARY_HISTORY

```
CREATE DESCENDING INDEX IDX_CHANGE
  ON SALARY_HISTORY (CHANGE_DATE);
```

3. Erstellen eines Multi-Segment-Index für die Spalten ORDER_STATUS, PAID in der Tabelle SALES

```
CREATE INDEX IDX_SALESTAT
  ON SALES (ORDER_STATUS, PAID);
```

4. Erstellen eines Index, der keine doppelten Werte für die Spalte NAME in der Tabelle COUNTRY zulässt

```
CREATE UNIQUE INDEX UNQ_COUNTRY_NAME
  ON COUNTRY (NAME);
```

5. Erstellen eines berechneten Index für die Tabelle PERSONS

```
CREATE INDEX IDX_NAME_UPPER ON PERSONS
  COMPUTED BY (UPPER (NAME));
```

   Ein Index wie dieser kann für eine Suche ohne Beachtung der Groß-/Kleinschreibung verwendet werden:

```
SELECT *
FROM PERSONS
WHERE UPPER(NAME) STARTING WITH UPPER('Iv');
```

Siehe auch

ALTER INDEX, DROP INDEX

5.5.2. ALTER INDEX

Verwendet für

Aktivieren oder Deaktivieren eines Indexes; Neuerstellung eines Index

Verfügbar in

DSQL, ESQL

Syntax

```
ALTER INDEX indexname {ACTIVE | INACTIVE}
```

Tabelle 32. `ALTER INDEX`-Anweisungsparameter

Parameter	Beschreibung
indexname	Indexname

Die `ALTER INDEX`-Anweisung aktiviert oder deaktiviert einen Index. Diese Anweisung bietet keine Möglichkeit, irgendwelche Attribute des Indexes zu ändern.

`INAKTIV`

Mit der Option `INACTIVE` wird der Index vom aktiven in den inaktiven Zustand geschaltet. Die Wirkung ist ähnlich wie bei der `DROP INDEX`-Anweisung, außer dass die Indexdefinition in der Datenbank verbleibt. Das Ändern eines Einschränkungsindex in den inaktiven Zustand ist nicht zulässig.

Ein aktiver Index kann deaktiviert werden, wenn keine Abfragen mit diesem Index vorbereitet sind; andernfalls wird ein Fehler “object in use” zurückgegeben.

Die Aktivierung eines inaktiven Index ist ebenfalls sicher. Wenn jedoch aktive Transaktionen vorhanden sind, die die Tabelle ändern, schlägt die Transaktion mit der Anweisung `ALTER INDEX` fehl, wenn sie das Attribut `NOWAIT` besitzt. Wenn sich die Transaktion im `WAIT`-Modus befindet, wartet sie auf den Abschluss gleichzeitiger Transaktionen.

Auf der anderen Seite der Medaille, wenn unser `ALTER INDEX` erfolgreich ist und beginnt, den Index bei `COMMIT` neu aufzubauen, werden andere Transaktionen, die diese Tabelle ändern, fehlschlagen oder warten, entsprechend ihren `WAIT`/`NO WAIT`-Attributen. Genauso verhält es sich mit `CREATE INDEX`.

Wofür ist es nützlich?

Es kann sinnvoll sein, einen Index in den inaktiven Zustand zu versetzen, während ein großer Satz von Datensätzen in der Tabelle, die den Index besitzt, eingefügt, aktualisiert oder gelöscht wird.

`AKTIV`

Mit der Option 'ACTIVE' wird der Index, wenn er sich im inaktiven Zustand befindet, in den aktiven Zustand geschaltet und das System baut den Index neu auf.

Wofür ist es nützlich?

Auch wenn der Index *active* ist, wenn `ALTER INDEX ⋯ ACTIVE` ausgeführt wird, wird der Index neu aufgebaut. Das Neuerstellen von Indizes kann ein nützliches Stück Haushaltsführung sein, um gelegentlich die Indizes einer großen Tabelle in einer Datenbank zu verwalten, die häufig eingefügt,

aktualisiert oder gelöscht wird, aber selten wiederhergestellt wird.

Wer kann einen Index ändern?

Die ALTER INDEX-Anweisung kann ausgeführt werden durch:

- Administratoren
- Der Besitzer der Tabelle
- Benutzer mit dem ALTER ANY TABLE-Privileg

Verwendung von ALTER INDEX für einen Einschränkungsindex

Das Ändern des Index eines PRIMARY KEY, FOREIGN KEY oder UNIQUE Constraints in INACTIVE ist nicht erlaubt. ALTER INDEX ⋯ ACTIVE funktioniert jedoch bei Constraint-Indizes genauso gut wie bei anderen, als Werkzeug zum Neuaufbau von Indizes.

ALTER INDEX-Beispiele

1. Deaktivieren des IDX_UPDATER-Index

```
ALTER INDEX IDX_UPDATER INACTIVE;
```

2. Den IDX_UPDATER-Index zurück in den aktiven Zustand schalten und neu aufbauen

```
ALTER INDEX IDX_UPDATER ACTIVE;
```

Siehe auch

CREATE INDEX, DROP INDEX, SET STATISTICS

5.5.3. DROP INDEX

Verwendet für

Einen Index löschen (löschen)

Verfügbar in

DSQL, ESQL

Syntax

```
DROP INDEX indexname
```

Tabelle 33. DROP INDEX-Anweisungsparameter

Parameter	Beschreibung
indexname	Indexname

Die DROP INDEX-Anweisung löscht (löscht) den benannten Index aus der Datenbank.

Ein Einschränkungsindex kann nicht mit DROP INDEX gelöscht werden. Constraint-Indizes werden während der Ausführung des Befehls ALTER TABLE … DROP CONSTRAINT … gelöscht.

Wer kann einen Index löschen?

Die DROP INDEX-Anweisung kann ausgeführt werden durch:

- Administratoren
- Der Besitzer der Tabelle
- Benutzer mit dem ALTER ANY TABLE-Privileg

DROP INDEX-Beispiel

Löschen des IDX_UPDATER-*Index*

```
DROP INDEX IDX_UPDATER;
```

Siehe auch

CREATE INDEX, ALTER INDEX

5.5.4. SET STATISTICS

Verwendet für

Neuberechnung der Selektivität eines Index

Verfügbar in

DSQL, ESQL

Syntax

```
SET STATISTICS INDEX indexname
```

Tabelle 34. SET STATISTICS-*Anweisungsparameter*

Parameter	Beschreibung
indexname	Indexname

Die Anweisung SET STATISTICS berechnet die Selektivität des angegebenen Index neu.

Wer kann Indexstatistiken aktualisieren?

Die Anweisung SET STATISTICS kann ausgeführt werden durch:

- Administratoren

- Der Besitzer der Tabelle
- Benutzer mit dem `ALTER ANY TABLE`-Privileg

Indexselektivität

Die Selektivität eines Index ergibt sich aus der Auswertung der Anzahl der Zeilen, die bei einer Suche nach jedem Indexwert ausgewählt werden können. Ein eindeutiger Index hat die maximale Selektivität, da es unmöglich ist, mehr als eine Zeile für jeden Wert eines Indexschlüssels auszuwählen, wenn dieser verwendet wird. Die Selektivität eines Index auf dem neuesten Stand zu halten ist wichtig für die Auswahl des Optimierers bei der Suche nach dem optimalsten Abfrageplan.

Indexstatistiken in Firebird werden als Reaktion auf große Mengen von Einfügungen, Aktualisierungen oder Löschungen nicht automatisch neu berechnet. Es kann von Vorteil sein, die Selektivität eines Index nach solchen Operationen neu zu berechnen, da die Selektivität dazu neigt, veraltet zu werden.

Die Anweisungen `CREATE INDEX` und `ALTER INDEX ACTIVE` speichern beide Indexstatistiken, die vollständig dem Inhalt des neu erstellten Index entsprechen.

Es kann unter gleichzeitiger Last ohne Beschädigungsrisiko ausgeführt werden. Beachten Sie jedoch, dass die neu berechneten Statistiken bei gleichzeitiger Belastung veraltet sein können, sobald `SET STATISTICS` beendet ist.

Beispiel für die Verwendung von SET STATISTICS

Neuberechnung der Selektivität des Indexes `IDX_UPDATER`

```
SET STATISTICS INDEX IDX_UPDATER;
```

Siehe auch

`CREATE INDEX`, `ALTER INDEX`

5.6. VIEW

Eine Ansicht (View) ist eine virtuelle Tabelle, die eigentlich eine gespeicherte und benannte `SELECT` -Abfrage zum Abrufen von Daten beliebiger Komplexität ist. Daten können aus einer oder mehreren Tabellen, aus anderen Ansichten und auch aus auswählbaren gespeicherten Prozeduren abgerufen werden.

Im Gegensatz zu regulären Tabellen in relationalen Datenbanken ist eine Ansicht kein unabhängiger Datensatz, der in der Datenbank gespeichert ist. Das Ergebnis wird bei Auswahl der Ansicht dynamisch als Datensatz erstellt.

Die Metadaten einer View stehen dem Prozess zur Verfügung, der den Binärcode für Stored Procedures und Trigger generiert, als wären es konkrete Tabellen, die persistente Daten speichern.

5.6.1. CREATE VIEW

Verwendet für

Erstellen einer Ansicht

Verfügbar in

DSQL

Syntax

```
CREATE VIEW viewname [<full_column_list>]
  AS <select_statement>
  [WITH CHECK OPTION]

<full_column_list> ::= (colname [, colname ...])
```

Tabelle 35. CREATE VIEW-Anweisungsparameter

Parameter	Beschreibung
viewname	Name der Ansicht (View), maximal 31 Zeichen
select_statement	SELECT-Anweisung
full_column_list	Die Liste der Spalten in der Ansicht
colname	Spaltennamen anzeigen. Doppelte Spaltennamen sind nicht zulässig.

Die Anweisung `CREATE VIEW` erstellt eine neue Ansicht. Der Bezeichner (Name) einer Ansicht muss unter den Namen aller Ansichten, Tabellen und gespeicherten Prozeduren in der Datenbank eindeutig sein.

Auf den Namen der neuen Ansicht kann die Liste der Spaltennamen folgen, die beim Aufrufen der Ansicht an den Aufrufer zurückgegeben werden sollen. Namen in der Liste müssen sich nicht auf die Namen der Spalten in den Basistabellen beziehen, von denen sie abgeleitet sind.

Wenn die Ansichtsspaltenliste weggelassen wird, verwendet das System die Spaltennamen und/oder Aliase aus der `SELECT`-Anweisung. Wenn doppelte Namen oder ausdrucksabgeleitete Spalten ohne Alias das Abrufen einer gültigen Liste unmöglich machen, schlägt die Erstellung der Ansicht mit einem Fehler fehl.

Die Anzahl der Spalten in der Liste der View muss genau mit der Anzahl der Spalten in der Auswahlliste der zugrunde liegenden SELECT-Anweisung in der View-Definition übereinstimmen.

Zusätzliche Punkte

- Wenn die vollständige Liste der Spalten angegeben ist, macht es keinen Sinn, Aliase in der `SELECT`-Anweisung anzugeben, da die Namen in der Spaltenliste diese überschreiben
- Die Spaltenliste ist optional, wenn alle Spalten im `SELECT` explizit benannt und in der Auswahlliste eindeutig sind

Aktualisierbare Ansichten

Eine Ansicht kann aktualisierbar oder schreibgeschützt sein. Wenn eine View aktualisierbar ist, können die beim Aufruf dieser View abgerufenen Daten durch die DML-Anweisungen `INSERT`, `UPDATE`, `DELETE`, `UPDATE OR INSERT` oder `MERGE` geändert werden. In einer aktualisierbaren Ansicht vorgenommene Änderungen werden auf die zugrunde liegende(n) Tabelle(n) angewendet.

Eine schreibgeschützte Ansicht kann mithilfe von Triggern aktualisierbar gemacht werden. Sobald Trigger für eine Ansicht definiert wurden, werden an sie gesendete Änderungen nie automatisch in die zugrunde liegende Tabelle geschrieben, selbst wenn die Ansicht von Anfang an aktualisierbar war. Es liegt in der Verantwortung des Programmierers sicherzustellen, dass die Trigger die Basistabellen nach Bedarf aktualisieren (oder aus ihnen löschen oder einfügen).

Eine Ansicht ist automatisch aktualisierbar, wenn alle folgenden Bedingungen erfüllt sind:

- die `SELECT`-Anweisung fragt nur eine Tabelle oder eine aktualisierbare Ansicht ab
- die `SELECT`-Anweisung ruft keine gespeicherten Prozeduren auf
- jede nicht in der Ansichtsdefinition vorhandene Spalte der Basistabelle (oder Basisansicht) erfüllt eine der folgenden Bedingungen:
  - es ist nullable
  - es hat einen Nicht-`NULL`-Standardwert
  - es hat einen Trigger, der einen zulässigen Wert liefert
- die `SELECT`-Anweisung enthält keine Felder, die von Unterabfragen oder anderen Ausdrücken abgeleitet sind
- die `SELECT`-Anweisung enthält keine Felder, die durch Aggregatfunktionen (`MIN`, `MAX`, `AVG`, `SUM`, `COUNT`, `LIST` usw.), statistische Funktionen (`CORR`, `COVAR_POP`, `COVAR_SAMP`, etc.), lineare Regressionsfunktionen (`REGR_AVGX`, `REGR_AVGY`, etc.) oder jede Art von Fensterfunktion
- die `SELECT`-Anweisung enthält keine `ORDER BY`-, `GROUP BY`- oder `HAVING`-Klausel
- die `SELECT`-Anweisung enthält weder das Schlüsselwort `DISTINCT` noch zeilenbeschränkende Schlüsselwörter wie `ROWS`, `FIRST`, `SKIP`, `OFFSET` oder `FETCH`

`WITH CHECK OPTION`

Die optionale `WITH CHECK OPTION`-Klausel erfordert eine aktualisierbare Ansicht, um zu prüfen, ob neue oder aktualisierte Daten die in der `WHERE`-Klausel der `SELECT`-Anweisung angegebene Bedingung erfüllen. Bei jedem Versuch, einen neuen Datensatz einzufügen oder einen bestehenden zu aktualisieren, wird geprüft, ob der neue oder aktualisierte Datensatz die 'WHERE'-Kriterien erfüllen würde. Wenn sie die Prüfung nicht bestehen, wird die Operation nicht ausgeführt und eine entsprechende Fehlermeldung zurückgegeben.

`WITH CHECK OPTION` kann nur in einer `CREATE VIEW`-Anweisung angegeben werden, in der eine `WHERE`-Klausel vorhanden ist, um die Ausgabe der `SELECT`-Hauptanweisung einzuschränken. Andernfalls wird eine Fehlermeldung zurückgegeben.

Bitte beachten Sie:

Wenn `WITH CHECK OPTION` verwendet wird, prüft die Engine die Eingabe gegen die

> WHERE-Klausel, bevor sie irgendetwas an die Basisrelation übergibt. Wenn die Prüfung der Eingabe fehlschlägt, werden daher keine Standardklauseln oder Trigger für die Basisrelation, die möglicherweise zur Korrektur der Eingabe entworfen wurden, in Aktion treten.
>
> Darüber hinaus werden View-Felder, die in der INSERT-Anweisung weggelassen wurden, als NULLs an die Basisrelation übergeben, unabhängig davon, ob sie in der WHERE-Klausel vorhanden oder nicht vorhanden sind. Daher werden die für solche Felder definierten Basistabellen-Standardwerte nicht angewendet. Trigger hingegen werden wie erwartet ausgelöst und funktionieren.
>
> Bei Ansichten, die nicht über WITH CHECK OPTION verfügen, werden Felder, die in der INSERT-Anweisung weggelassen wurden, überhaupt nicht an die Basisrelation übergeben, sodass alle Standardwerte angewendet werden.

Wer kann eine Ansicht erstellen?

Die CREATE VIEW-Anweisung kann ausgeführt werden durch:

- Administratoren
- Benutzer mit dem Privileg CREATE VIEW

Der Ersteller einer Ansicht wird ihr Eigentümer.

Um eine Ansicht zu erstellen, benötigt ein Nicht-Administrator-Benutzer außerdem mindestens 'SELECT'-Zugriff auf die zugrunde liegende(n) Tabelle(n) und/oder Ansicht(en) und das 'EXECUTE'-Privileg für alle beteiligten auswählbaren gespeicherten Prozeduren.

Um Einfügungen, Aktualisierungen und Löschungen über die Ansicht zu ermöglichen, muss der Ersteller/Eigentümer auch die entsprechenden INSERT, UPDATE und DELETE-Rechte für das/die zugrunde liegende(n) Objekt(e) besitzen.

Anderen Benutzern Berechtigungen für die Ansicht zu erteilen ist nur möglich, wenn der Ansichtsbesitzer diese Berechtigungen für die zugrunde liegenden Objekte hat WITH GRANT OPTION. Dies ist immer dann der Fall, wenn der View-Eigentümer auch der Eigentümer der zugrunde liegenden Objekte ist.

===Beispiele für das Erstellen von Ansichten

1. Erstellen einer Ansicht, die die Spalten JOB_CODE und JOB_TITLE nur für die Jobs zurückgibt, bei denen MAX_SALARY weniger als 15.000 USD beträgt.

```
CREATE VIEW ENTRY_LEVEL_JOBS AS
SELECT JOB_CODE, JOB_TITLE
FROM JOB
WHERE MAX_SALARY < 15000;
```

2. Erstellen einer Ansicht, die die Spalten JOB_CODE und JOB_TITLE nur für Jobs zurückgibt, bei denen MAX_SALARY weniger als 15.000 USD beträgt. Immer wenn ein neuer Datensatz eingefügt

oder ein vorhandener Datensatz aktualisiert wird, wird die Bedingung `MAX_SALARY < 15000` geprüft. Wenn die Bedingung nicht wahr ist, wird die Einfüge-/Aktualisierungsoperation abgelehnt.

```
CREATE VIEW ENTRY_LEVEL_JOBS AS
SELECT JOB_CODE, JOB_TITLE
FROM JOB
WHERE MAX_SALARY < 15000
WITH CHECK OPTION;
```

3. Erstellen einer Ansicht mit einer expliziten Spaltenliste.

```
CREATE VIEW PRICE_WITH_MARKUP (
  CODE_PRICE,
  COST,
  COST_WITH_MARKUP
) AS
SELECT
  CODE_PRICE,
  COST,
  COST * 1.1
FROM PRICE;
```

4. Erstellen einer View mit Hilfe von Aliasen für Felder in der `SELECT`-Anweisung (das gleiche Ergebnis wie in Beispiel 3).

```
CREATE VIEW PRICE_WITH_MARKUP AS
SELECT
  CODE_PRICE,
  COST,
  COST * 1.1 AS COST_WITH_MARKUP
FROM PRICE;
```

5. Erstellen einer schreibgeschützten Ansicht basierend auf zwei Tabellen und einer gespeicherten Prozedur.

```
CREATE VIEW GOODS_PRICE AS
SELECT
  goods.name AS goodsname,
  price.cost AS cost,
  b.quantity AS quantity
FROM
  goods
  JOIN price ON goods.code_goods = price.code_goods
  LEFT JOIN sp_get_balance(goods.code_goods) b ON 1 = 1;
```

Siehe auch

`ALTER VIEW`, `CREATE OR ALTER VIEW`, `RECREATE VIEW`, `DROP VIEW`

5.6.2. ALTER VIEW

Verwendet für

Ändern einer vorhandenen Ansicht

Verfügbar in

DSQL

Syntax

```
ALTER VIEW viewname [<full_column_list>]
    AS <select_statement>
    [WITH CHECK OPTION]

<full_column_list> ::= (colname [, colname ...])
```

Tabelle 36. `ALTER VIEW`-Anweisungsparameter

Parameter	Beschreibung
viewname	Name einer existierenden Ansicht
select_statement	SELECT-Anweisung
full_column_list	Die Liste der Spalten in der Ansicht
colname	Spaltennamen anzeigen. Doppelte Spaltennamen sind nicht zulässig.

Verwenden Sie die Anweisung `ALTER VIEW`, um die Definition einer bestehenden Ansicht zu ändern. Berechtigungen für Ansichten bleiben erhalten und Abhängigkeiten werden nicht beeinflusst.

Die Syntax der `ALTER VIEW`-Anweisung entspricht vollständig der von `CREATE VIEW`.

Seien Sie vorsichtig, wenn Sie die Anzahl der Spalten in einer Ansicht ändern. Vorhandener Anwendungscode und PSQL-Module, die auf die Ansicht zugreifen, können ungültig werden. Informationen zum Erkennen dieser Art von Problem in gespeicherten Prozeduren und Triggern finden Sie unter *Das RDB$VALID_BLR-Feld* im Anhang.

Wer kann eine Ansicht ändern?

Die `ALTER VIEW`-Anweisung kann ausgeführt werden durch:

- Administratoren
- Der Besitzer der Ansicht
- Benutzer mit der Berechtigung `ALTER ANY VIEW`

Beispiel mit `ALTER VIEW`

Ändern der Ansicht `PRICE_WITH_MARKUP`

```
ALTER VIEW PRICE_WITH_MARKUP (
  CODE_PRICE,
  COST,
  COST_WITH_MARKUP
) AS
SELECT
  CODE_PRICE,
  COST,
  COST * 1.15
FROM PRICE;
```

Siehe auch

`CREATE VIEW`, `CREATE OR ALTER VIEW`, `RECREATE VIEW`

5.6.3. `CREATE OR ALTER VIEW`

Verwendet für

Erstellen einer neuen Ansicht oder Ändern einer vorhandenen Ansicht.

Verfügbar in

DSQL

Syntax

```
CREATE OR ALTER VIEW viewname [<full_column_list>]
  AS <select_statement>
  [WITH CHECK OPTION]

<full_column_list> ::= (colname [, colname ...])
```

Tabelle 37. `CREATE OR ALTER VIEW`*-Anweisungsparameter*

Parameter	Beschreibung
viewname	Name einer Ansicht, die möglicherweise nicht vorhanden ist
select_statement	SELECT-Anweisung
full_column_list	Die Liste der Spalten in der Ansicht
colname	Spaltennamen anzeigen. Doppelte Spaltennamen sind nicht zulässig.

Verwenden Sie die Anweisung `CREATE OR ALTER VIEW`, um die Definition einer bestehenden Ansicht zu ändern oder sie zu erstellen, falls sie nicht existiert. Berechtigungen für eine vorhandene Ansicht bleiben erhalten und Abhängigkeiten werden nicht beeinflusst.

Die Syntax der `CREATE OR ALTER VIEW`-Anweisung entspricht vollständig der von `CREATE VIEW`.

Beispiel für CREATE OR ALTER VIEW

Erstellen der neuen Ansicht PRICE_WITH_MARKUP-*Ansicht oder Ändern, wenn sie bereits vorhanden ist*

```
CREATE OR ALTER VIEW PRICE_WITH_MARKUP (
  CODE_PRICE,
  COST,
  COST_WITH_MARKUP
) AS
SELECT
  CODE_PRICE,
  COST,
  COST * 1.15
FROM PRICE;
```

Siehe auch

CREATE VIEW, ALTER VIEW, RECREATE VIEW

5.6.4. DROP VIEW

Verwendet für

Löschen einer Ansicht

Verfügbar in

DSQL

Syntax

```
DROP VIEW viewname
```

Tabelle 38. DROP VIEW-*Anweisungsparameter*

Parameter	Beschreibung
viewname	Name der Ansicht

Die DROP VIEW-Anweisung löscht (löscht) eine vorhandene Ansicht. Die Anweisung schlägt fehl, wenn die Ansicht Abhängigkeiten aufweist.

Wer kann eine Ansicht löschen?

Die DROP VIEW-Anweisung kann ausgeführt werden durch:

- Administratoren
- Der Besitzer der Ansicht
- Benutzer mit dem Privileg DROP ANY VIEW

Beispiel

Löschen der Ansicht `PRICE_WITH_MARKUP`

```
DROP VIEW PRICE_WITH_MARKUP;
```

Siehe auch

`CREATE VIEW`, `RECREATE VIEW`, `CREATE OR ALTER VIEW`

5.6.5. RECREATE VIEW

Verwendet für

Erstellen einer neuen Ansicht oder Neuerstellen einer vorhandenen Ansicht

Verfügbar in

DSQL

Syntax

```
RECREATE VIEW viewname [<full_column_list>]
  AS <select_statement>
  [WITH CHECK OPTION]

<full_column_list> ::= (colname [, colname ...])
```

Tabelle 39. `RECREATE VIEW`*-Anweisungsparameter*

Parameter	Beschreibung
viewname	Name der Ansicht (View), maximal 31 Zeichen
select_statement	SELECT-Anweisung
full_column_list	Die Liste der Spalten in der Ansicht
colname	Spaltennamen anzeigen. Doppelte Spaltennamen sind nicht zulässig.

Erstellt eine Ansicht oder erstellt sie neu. Wenn bereits eine Ansicht mit diesem Namen vorhanden ist, versucht die Engine, sie zu löschen, bevor die neue Instanz erstellt wird. Wenn die vorhandene Ansicht nicht gelöscht werden kann, z. B. aufgrund von Abhängigkeiten oder unzureichenden Rechten, schlägt `RECREATE VIEW` mit einem Fehler fehl.

Beispiel für `RECREATE VIEW`

Neue Ansicht `PRICE_WITH_MARKUP`*-Ansicht erstellen oder neu erstellen, falls bereits vorhanden*

```
RECREATE VIEW PRICE_WITH_MARKUP (
  CODE_PRICE,
  COST,
  COST_WITH_MARKUP
) AS
SELECT
```

```
  CODE_PRICE,
  COST,
  COST * 1.15
FROM PRICE;
```

Siehe auch

`CREATE VIEW`, `DROP VIEW`, `CREATE OR ALTER VIEW`

5.7. TRIGGER

Ein Trigger ist eine spezielle Art einer gespeicherten Prozedur, die nicht direkt aufgerufen wird, sondern ausgeführt wird, wenn ein angegebenes Ereignis in der zugeordneten Tabelle oder Sicht auftritt. Ein DML-Trigger ist spezifisch für eine und nur eine Beziehung (Tabelle oder Ansicht) und eine Phase im Timing des Ereignisses (*BEFORE* oder *AFTER*). Es kann für ein bestimmtes Ereignis (insert, update, delete) oder für eine Kombination von zwei oder drei dieser Ereignisse ausgeführt werden.

Es gibt zwei andere Formen von Auslösern:

1. ein “Datenbank-Trigger” kann angegeben werden, um zu Beginn oder am Ende einer Benutzersitzung (Verbindung) oder einer Benutzertransaktion auszulösen.
2. ein “DDL-Trigger” kann angegeben werden, um vor oder nach der Ausführung einer oder mehrerer Typen von DDL-Anweisungen auszulösen.

5.7.1. CREATE TRIGGER

Verwendet für

Einen neuen Trigger erstellen

Verfügbar in

DSQL, ESQL

Syntax

```
CREATE TRIGGER trigname
  { <relation_trigger_legacy>
  | <relation_trigger_sql2003>
  | <database_trigger>
  | <ddl_trigger> }
  <module-body>

<module-body> ::=
  !! Siehe auch Syntax des Modul-Bodys !!

<relation_trigger_legacy> ::=
  FOR {tablename | viewname}
  [ACTIVE | INACTIVE]
  {BEFORE | AFTER} <mutation_list>
```

```
  [POSITION number]

<relation_trigger_sql2003> ::=
  [ACTIVE | INACTIVE]
  {BEFORE | AFTER} <mutation_list>
  [POSITION number]
  ON {tablename | viewname}

<database_trigger> ::=
  [ACTIVE | INACTIVE] ON <db_event>
  [POSITION number]

<ddl_trigger> ::=
  [ACTIVE | INACTIVE]
  {BEFORE | AFTER} <ddl_event>
  [POSITION number]

<mutation_list> ::=
  <mutation> [OR <mutation> [OR <mutation>]]

<mutation> ::= INSERT | UPDATE | DELETE

<db_event> ::=
    CONNECT | DISCONNECT
  | TRANSACTION {START | COMMIT | ROLLBACK}

<ddl_event> ::=
    ANY DDL STATEMENT
  | <ddl_event_item> [{OR <ddl_event_item>} ...]

<ddl_event_item> ::=
    {CREATE | ALTER | DROP} TABLE
  | {CREATE | ALTER | DROP} PROCEDURE
  | {CREATE | ALTER | DROP} FUNCTION
  | {CREATE | ALTER | DROP} TRIGGER
  | {CREATE | ALTER | DROP} EXCEPTION
  | {CREATE | ALTER | DROP} VIEW
  | {CREATE | ALTER | DROP} DOMAIN
  | {CREATE | ALTER | DROP} ROLE
  | {CREATE | ALTER | DROP} SEQUENCE
  | {CREATE | ALTER | DROP} USER
  | {CREATE | ALTER | DROP} INDEX
  | {CREATE | DROP} COLLATION
  | ALTER CHARACTER SET
  | {CREATE | ALTER | DROP} PACKAGE
  | {CREATE | DROP} PACKAGE BODY
  | {CREATE | ALTER | DROP} MAPPING
```

Tabelle 40. `CREATE TRIGGER`-*Anweisgungsparameter*

Parameter	Beschreibung
trigname	Triggername bestehend aus bis zu 31 Zeichen. Er muss unter allen Triggernamen in der Datenbank eindeutig sein.
relation_trigger_legacy	Legacy-Stil der Trigger-Deklaration für einen Relations-Trigger
relation_trigger_sql2003	Relations-Trigger-Deklaration gemäß SQL:2003-Standard
database_trigger	Datenbank-Trigger-Deklaration
tablename	Name der Tabelle, mit der der Relationstrigger verknüpft ist
viewname	Name der Ansicht, mit der der Relationstrigger verknüpft ist
mutation_list	Liste der Beziehungsereignisse (Tabellenansicht |)
number	Position des Abzugs in der Schussfolge. Von 0 bis 32.767
db_event	Verbindungs- oder Transaktionsereignis
ddl_event	Liste der Metadatenänderungsereignisse
ddl_event_item	Eines der Metadatenänderungsereignisse

Die Anweisung `CREATE TRIGGER` wird verwendet, um einen neuen Trigger zu erstellen. Ein Trigger kann entweder für ein *Relation (Tabelle | Ansicht) Ereignis* (oder eine Kombination von Ereignissen), für ein *Datenbankereignis* oder für ein *DDL Ereignis* erstellt werden.

`CREATE TRIGGER` ist zusammen mit seinen assoziierten `ALTER TRIGGER`, `CREATE OR ALTER TRIGGER` und `RECREATE TRIGGER` eine *zusammengesetzte Anweisung*, bestehend aus einem Header und einem Body. Der Header gibt den Namen des Triggers, den Namen der Beziehung (bei einem DML-Trigger), die Phase des Triggers, die Ereignisse, auf die er angewendet wird, und die Position an, um eine Reihenfolge zwischen den Triggern zu bestimmen.

Der Triggerrumpf besteht aus optionalen Deklarationen lokaler Variablen und benannten Cursors gefolgt von einer oder mehreren Anweisungen oder Anweisungsblöcken, die alle in einem äußeren Block eingeschlossen sind, der mit dem Schlüsselwort `BEGIN` beginnt und mit dem Schlüsselwort `END` endet. Deklarationen und eingebettete Anweisungen werden mit Semikolons (‘;’) abgeschlossen.

Der Name des Triggers muss unter allen Triggernamen eindeutig sein.

Statement-Terminatoren

Einige SQL-Anweisungseditoren – insbesondere das mit Firebird gelieferte Dienstprogramm *isql* und möglicherweise einige Editoren von Drittanbietern – verwenden eine interne Konvention, die erfordert, dass alle Anweisungen mit einem Semikolon abgeschlossen werden. Dies führt beim Codieren in diesen Umgebungen zu einem Konflikt mit der PSQL-Syntax. Wenn Sie dieses Problem und seine Lösung nicht kennen, lesen Sie bitte die Details im PSQL-Kapitel im Abschnitt Terminator in *isql* umschalten.

Externe UDR-Trigger

Ein Trigger kann sich auch in einem externen Modul befinden. In diesem Fall spezifiziert `CREATE TRIGGER` anstelle eines Trigger-Bodys die Position des Triggers im externen Modul mit der `EXTERNAL`

-Klausel. Die optionale NAME-Klausel spezifiziert den Namen des externen Moduls, den Namen des Triggers innerhalb des Moduls und – optional – benutzerdefinierte Informationen. Die erforderliche ENGINE-Klausel gibt den Namen der UDR-Engine an, die die Kommunikation zwischen Firebird und dem externen Modul handhabt. Die optionale AS-Klausel akzeptiert ein String-Literal "body", das von der Engine oder dem Modul für verschiedene Zwecke verwendet werden kann.

DML-Triggers (auf Tabellen oder Ansichten)

DML- oder "relation"-Trigger werden auf Zeilen-(Datensatz-)Ebene ausgeführt, jedes Mal, wenn sich das Zeilenbild ändert. Ein Trigger kann entweder 'AKTIV' oder 'INAKTIV' sein. Es werden nur aktive Trigger ausgeführt. Trigger werden standardmäßig als 'AKTIV' erstellt.

Wer kann einen DML-Trigger erstellen?

DML-Trigger können erstellt werden durch:

- Administratoren
- Der Besitzer der Tabelle (oder Ansicht)
- Benutzer mit dem ALTER ANY TABLE- oder — für eine Ansicht — ALTER ANY VIEW-Privileg

Formulare der Erklärung

Firebird unterstützt zwei Deklarationsformen für Relations-Trigger:

- Die ursprüngliche, veraltete Syntax
- Das SQL:2003 standardkonforme Formular (empfohlen)

Das mit SQL:2003 standardkonforme Formular wird empfohlen.

Ein Relationstrigger spezifiziert — unter anderem — eine *Phase* und ein oder mehrere *Ereignisse*.

Phase

Phase betrifft das Timing des Triggers in Bezug auf das Change-of-State-Ereignis in der Datenzeile:

- Ein BEFORE-Trigger wird ausgelöst, bevor die angegebene Datenbankoperation (insert, update oder delete) ausgeführt wird
- Ein 'AFTER'-Trigger wird ausgelöst, nachdem die Datenbankoperation abgeschlossen ist

Zeilenereignis

Eine Relations-Trigger-Definition spezifiziert mindestens eine der DML-Operationen 'INSERT', 'UPDATE' und 'DELETE', um ein oder mehrere Ereignisse anzugeben, bei denen der Trigger ausgelöst werden soll. Werden mehrere Operationen angegeben, müssen diese durch das Schlüsselwort OR getrennt werden. Keine Operation darf mehr als einmal erfolgen.

Innerhalb des Anweisungsblocks die booleschen Kontextvariablen INSERTING, UPDATING und DELETING kann verwendet werden, um zu testen, welche Operation gerade ausgeführt wird.

Auslösereihenfolge der Auslöser

Das Schlüsselwort `POSITION` ermöglicht die Angabe einer optionalen Ausführungsreihenfolge ("firing order") für eine Reihe von Triggern, die die gleiche Phase und das gleiche Ereignis wie ihr Ziel haben. Die Standardposition ist 0. Wenn keine Positionen angegeben sind oder mehrere Trigger eine einzige Positionsnummer haben, werden die Trigger in alphabetischer Reihenfolge ihrer Namen ausgeführt.

Variablendeklarationen

Der optionale Deklarationsabschnitt unter dem Schlüsselwort `AS` in der Kopfzeile des Triggers dient zum Definieren von Variablen und benannten Cursorn, die lokal für den Trigger sind. Weitere Informationen finden Sie unter `DECLARE VARIABLE` und `DECLARE CURSOR` im Prozedurales SQL Kapitel.

Der Trigger-Body

Die lokalen Deklarationen (sofern vorhanden) sind der letzte Teil des Header-Abschnitts eines Triggers. Es folgt der Triggerrumpf, wobei ein oder mehrere Blöcke von PSQL-Anweisungen in eine Struktur eingeschlossen werden, die mit dem Schlüsselwort `BEGIN` beginnt und mit dem Schlüsselwort `END` endet.

Nur der Eigentümer der Ansicht oder Tabelle und Administratoren haben die Berechtigung, `CREATE TRIGGER` zu verwenden.

Beispiele für `CREATE TRIGGER` für Tabellen und Ansichten

1. Erstellen eines Triggers in "legacy"-Form, der ausgelöst wird, bevor ein neuer Datensatz in die Tabelle `CUSTOMER` eingefügt wird.

```
CREATE TRIGGER SET_CUST_NO FOR CUSTOMER
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
  IF (NEW.CUST_NO IS NULL) THEN
    NEW.CUST_NO = GEN_ID(CUST_NO_GEN, 1);
END
```

2. Erstellen einer Triggerauslösung vor dem Einfügen eines neuen Datensatzes in die `CUSTOMER` -Tabelle in SQL:2003-Standard-konformer Form.

```
CREATE TRIGGER set_cust_no
ACTIVE BEFORE INSERT POSITION 0 ON customer
AS
BEGIN
  IF (NEW.cust_no IS NULL) THEN
    NEW.cust_no = GEN_ID(cust_no_gen, 1);
END
```

3. Erstellen eines Triggers, der nach dem Einfügen, Aktualisieren oder Löschen eines Datensatzes

in der CUSTOMER-Tabelle ausgelöst wird.

```
CREATE TRIGGER TR_CUST_LOG
ACTIVE AFTER INSERT OR UPDATE OR DELETE POSITION 10
ON CUSTOMER
AS
BEGIN
  INSERT INTO CHANGE_LOG (LOG_ID,
                          ID_TABLE,
                          TABLE_NAME,
                          MUTATION)
  VALUES (NEXT VALUE FOR SEQ_CHANGE_LOG,
          OLD.CUST_NO,
          'CUSTOMER',
          CASE
            WHEN INSERTING THEN 'INSERT'
            WHEN UPDATING  THEN 'UPDATE'
            WHEN DELETING  THEN 'DELETE'
          END);
END
```

Datenbank-Trigger

Trigger können so definiert werden, dass sie bei “Datenbankereignissen” ausgelöst werden, was sich wirklich auf eine Mischung aus Ereignissen bezieht, die über den Umfang einer Sitzung (Verbindung) und Ereignissen, die über den Umfang einer einzelnen Transaktion hinweg wirken:

- CONNECT
- DISCONNECT
- TRANSACTION START
- TRANSACTION COMMIT
- TRANSACTION ROLLBACK

DDL-Trigger sind eine Unterart von Datenbank-Triggern, die in einem separaten Abschnitt behandelt werden.

Wer kann einen Datenbank-Trigger erstellen?

Datenbank-Trigger können erstellt werden durch:

- Administratoren
- Benutzer mit dem ALTER DATABASE-Privileg

Ausführung von Datenbank-Triggern und Ausnahmebehandlung

Die Trigger CONNECT und DISCONNECT werden in einer eigens dafür erstellten Transaktion ausgeführt. Diese Transaktion verwendet die Standardisolationsstufe, d. h. Snapshot (Parallelität), Schreiben und Warten. Wenn alles gut geht, wird die Transaktion festgeschrieben. Nicht abgefangene

Ausnahmen führen zu einem Rollback der Transaktion und

- bei einem CONNECT-Trigger wird die Verbindung dann unterbrochen und die Ausnahme wird an den Client zurückgegeben
- Bei einem DISCONNECT-Trigger werden Ausnahmen nicht gemeldet. Die Verbindung ist wie beabsichtigt unterbrochen

TRANSACTION-Trigger werden innerhalb der Transaktion ausgeführt, deren Start, Commit oder Rollback sie hervorruft. Die nach einer nicht abgefangenen Ausnahme ausgeführte Aktion hängt vom Ereignis ab:

- Bei einem TRANSACTION START-Trigger wird die Ausnahme an den Client gemeldet und die Transaktion wird zurückgesetzt
- Bei einem TRANSACTION COMMIT Trigger wird die Ausnahme gemeldet, die bisherigen Aktionen des Triggers werden rückgängig gemacht und der Commit wird abgebrochen
- Bei einem TRANSACTION ROLLBACK-Trigger wird die Ausnahme nicht gemeldet und die Transaktion wie vorgesehen zurückgesetzt.

Fallstricke

Offensichtlich gibt es keine direkte Möglichkeit zu wissen, ob ein DISCONNECT- oder TRANSACTION ROLLBACK-Trigger eine Ausnahme verursacht hat. Daraus folgt auch, dass die Verbindung zur Datenbank nicht zustande kommen kann, wenn ein CONNECT-Trigger eine Ausnahme auslöst und eine Transaktion auch nicht starten kann, wenn ein TRANSACTION START-Trigger eine auslöst. Beide Phänomene sperren Sie effektiv aus Ihrer Datenbank, bis Sie mit unterdrückten Datenbank-Triggern dort hineinkommen und den fehlerhaften Code beheben.

Unterdrücken von Datenbank-Triggern

Einige Firebird-Befehlszeilentools wurden mit Schaltern geliefert, mit denen ein Administrator das automatische Auslösen von Datenbank-Triggern unterdrücken kann. Bisher sind das:

```
gbak -nodbtriggers
isql -nodbtriggers
nbackup -T
```

Zweiphasen-Commit

In einem zweiphasigen Commit-Szenario löst TRANSACTION COMMIT das Auslösen in der Vorbereitungsphase aus, nicht beim Commit.

Einige Vorbehalte

1. Die Verwendung der Anweisung IN AUTONOMOUS TRANSACTION DO in den Datenbankereignis-Triggern in Bezug auf Transaktionen (TRANSACTION START, TRANSACTION ROLLBACK, TRANSACTION COMMIT) kann dazu führen, dass die autonome Transaktion in eine Endlosschleife gerät
2. Die Ereignistrigger DISCONNECT und TRANSACTION ROLLBACK werden nicht ausgeführt, wenn Clients

über Monitoring-Tabellen getrennt werden (`DELETE FROM MON$ATTACHMENTS`)

Nur der Datenbankbesitzer und Administratoren haben die Berechtigung, Datenbank-Trigger zu erstellen.

Beispiele für `CREATE TRIGGER` für „Datenbank-Trigger“

1. Erstellen eines Triggers für das Ereignis der Verbindung mit der Datenbank, der die Anmeldung von Benutzern am System protokolliert. Der Trigger wird als inaktiv angelegt.

```
CREATE TRIGGER tr_log_connect
INACTIVE ON CONNECT POSITION 0
AS
BEGIN
  INSERT INTO LOG_CONNECT (ID,
                           USERNAME,
                           ATIME)
  VALUES (NEXT VALUE FOR SEQ_LOG_CONNECT,
          CURRENT_USER,
          CURRENT_TIMESTAMP);
END
```

2. Erstellen eines Auslösers für das Ereignis einer Verbindung mit der Datenbank, der es keinem Benutzer außer SYSDBA erlaubt, sich außerhalb der Geschäftszeiten anzumelden.

```
CREATE EXCEPTION E_INCORRECT_WORKTIME 'The working day has not started yet.';

CREATE TRIGGER TR_LIMIT_WORKTIME ACTIVE
ON CONNECT POSITION 1
AS
BEGIN
  IF ((CURRENT_USER <> 'SYSDBA') AND
      NOT (CURRENT_TIME BETWEEN time '9:00' AND time '17:00')) THEN
    EXCEPTION E_INCORRECT_WORKTIME;
END
```

DDL-Trigger

DDL-Trigger ermöglichen Einschränkungen für Benutzer, die versuchen, ein DDL-Objekt zu erstellen, zu ändern oder zu löschen. Ihr anderer Zweck besteht darin, ein Metadatenänderungsprotokoll zu führen.

DDL-Trigger lösen bei bestimmten Metadatenänderungsereignissen in einer bestimmten Phase aus. `BEFORE`-Trigger werden vor Änderungen an Systemtabellen ausgeführt. `AFTER`-Trigger werden nach Änderungen in Systemtabellen ausgeführt.

Der Ereignistyp `[BEFORE | AFTER]` eines DDL-Triggers kann nicht geändert werden.

In gewisser Weise sind DDL-Trigger ein Untertyp von Datenbank-Triggern.

Wer kann einen DDL-Trigger erstellen?

DDL-Trigger können erstellt werden durch:

- Administratoren
- Benutzer mit dem `ALTER DATABASE`-Privileg

Unterdrücken von DDL-Triggern

Ein DDL-Trigger ist eine Art Datenbank-Trigger. Siehe auch Unterdrücken von Datenbank-Triggern wie man Datenbank- und DDL-Trigger unterdrückt.

Beispiele für DDL-Trigger

1. So können Sie einen DDL-Trigger verwenden, um ein konsistentes Benennungsschema zu erzwingen. In diesem Fall sollten die Namen der gespeicherten Prozeduren mit dem Präfix "`SP_`" beginnen:

```
set auto on;
create exception e_invalid_sp_name 'Invalid SP name (should start with SP_)';

set term !;

create trigger trig_ddl_sp before CREATE PROCEDURE
as
begin
  if (rdb$get_context('DDL_TRIGGER', 'OBJECT_NAME') not starting 'SP_') then
    exception e_invalid_sp_name;
end!
```

Test

```
create procedure sp_test
as
begin
end!

create procedure test
as
begin
end!

-- Der letzte Befehl löst diese Ausnahme aus und die Prozedur TEST wird nicht
erstellt:
-- Statement failed, SQLSTATE = 42000
-- exception 1
-- -E_INVALID_SP_NAME
```

```
-- -Invalid SP name (should start with SP_)
-- -At trigger 'TRIG_DDL_SP' line: 4, col: 5

set term ;!
```

2. Implementieren Sie benutzerdefinierte DDL-Sicherheit, indem Sie in diesem Fall die Ausführung von DDL-Befehlen auf bestimmte Benutzer beschränken:

```
create exception e_access_denied 'Access denied';

set term !;

create trigger trig_ddl before any ddl statement
as
begin
  if (current_user <> 'SUPER_USER') then
    exception e_access_denied;
end!
```

Test

```
create procedure sp_test
as
begin
end!

-- Der letzte Befehl löst diese Ausnahme aus und die Prozedur SP_TEST wird nicht
erstellt
-- Statement failed, SQLSTATE = 42000
-- exception 1
-- -E_ACCESS_DENIED
-- -Access denied
-- -At trigger 'TRIG_DDL' line: 4, col: 5

set term ;!
```

Firebird hat Berechtigungen zum Ausführen von DDL-Anweisungen, daher sollte das Schreiben eines DDL-Triggers dafür der letzte Ausweg sein, wenn der gleiche Effekt nicht mit Berechtigungen erzielt werden kann.

3. Verwenden eines Triggers, um DDL-Aktionen und -Versuche zu protokollieren:

```
create sequence ddl_seq;

create table ddl_log (
  id bigint not null primary key,
  moment timestamp not null,
```

```
  user_name varchar(31) not null,
  event_type varchar(25) not null,
  object_type varchar(25) not null,
  ddl_event varchar(25) not null,
  object_name varchar(31) not null,
  sql_text blob sub_type text not null,
  ok char(1) not null
);

set term !;

create trigger trig_ddl_log_before before any ddl statement
as
  declare id type of column ddl_log.id;
begin
  -- Wir nehmen die Änderungen in einer AUTONOMEN TRANSAKTION vor. Wenn also eine
Ausnahme auftritt und
  -- der Befehl nicht ausgeführt wurde, bleibt das Protokoll erhalten.
  in autonomous transaction do
  begin
    insert into ddl_log (id, moment, user_name, event_type, object_type,
                         ddl_event, object_name, sql_text, ok)
      values (next value for ddl_seq, current_timestamp, current_user,
              rdb$get_context('DDL_TRIGGER', 'EVENT_TYPE'),
              rdb$get_context('DDL_TRIGGER', 'OBJECT_TYPE'),
              rdb$get_context('DDL_TRIGGER', 'DDL_EVENT'),
              rdb$get_context('DDL_TRIGGER', 'OBJECT_NAME'),
              rdb$get_context('DDL_TRIGGER', 'SQL_TEXT'),
              'N')
      returning id into id;
    rdb$set_context('USER_SESSION', 'trig_ddl_log_id', id);
  end
end!
```

Der obige Trigger wird für diesen DDL-Befehl ausgelöst. Es ist eine gute Idee, `-nodbtriggers` zu verwenden, wenn Sie mit ihnen arbeiten!

```
create trigger trig_ddl_log_after after any ddl statement
as
begin
  -- Hier benötigen wir eine AUTONOME TRANSACTION, da die ursprüngliche Transaktion
den Datensatz
  -- nicht sehen wird, der in den BEFORE-Trigger der
  -- autonomen Transaktion eingefügt wurde, wenn die Benutzertransaktion nicht READ
COMMITTED ist.
  in autonomous transaction do
     update ddl_log set ok = 'Y'
     where id = rdb$get_context('USER_SESSION', 'trig_ddl_log_id');
end!
```

```
commit!

set term ;!

-- Löschen Sie den Datensatz über trig_ddl_log_after
delete from ddl_log;
commit;
```

Test

```
-- Dies wird einmalig protokolliert
-- (da T1 nicht existierte, fungiert RECREATE als CREATE) mit OK = Y.
recreate table t1 (
  n1 integer,
  n2 integer
);

-- Dies schlägt fehl, da T1 bereits existiert, also ist OK N.
create table t1 (
  n1 integer,
  n2 integer
);

-- T2 existiert nicht. Es wird kein Protokoll geben.
drop table t2;

-- Dies wird zweimal protokolliert
-- (da T1 existiert, fungiert RECREATE als DROP und CREATE) mit OK = Y.
recreate table t1 (
  n integer
);

commit;
```

```
select id, ddl_event, object_name, sql_text, ok
  from ddl_log order by id;

 ID DDL_EVENT                OBJECT_NAME                     SQL_TEXT OK
=== ======================== ======================= ================ ======
  2 CREATE TABLE             T1                                 80:3 Y
==================================================
SQL_TEXT:
recreate table t1 (
    n1 integer,
    n2 integer
)
==================================================
  3 CREATE TABLE             T1                                 80:2 N
```

```
================================================
SQL_TEXT:
create table t1 (
    n1 integer,
    n2 integer
)
================================================
  4 DROP TABLE                T1                                80:6 Y
================================================
SQL_TEXT:
recreate table t1 (
    n integer
)
================================================
  5 CREATE TABLE              T1                                80:9 Y
================================================
SQL_TEXT:
recreate table t1 (
    n integer
)
================================================
```

Siehe auch

`ALTER TRIGGER`, `CREATE OR ALTER TRIGGER`, `RECREATE TRIGGER`, `DROP TRIGGER`, DDL-Trigger im Kapitel *Procedural SQL (PSQL)-Anweisungen*

5.7.2. ALTER TRIGGER

Verwendet für

Ändern und Deaktivieren eines bestehenden Triggers

Verfügbar in

DSQL, ESQL

Syntax

```
ALTER TRIGGER trigname
  [ACTIVE | INACTIVE]
  [{BEFORE | AFTER} <mutation_list>]
  [POSITION number]
  [<module-body>]

!! Vgl. auch die Syntax CREATE TRIGGER für weitere Regeln!!
```

Die `ALTER TRIGGER`-Anweisung erlaubt nur bestimmte Änderungen am Header und Body eines Triggers.

Zulässige Änderungen an Triggern

- Status (`ACTIVE | INACTIVE`)
- Phase (`BEFORE | AFTER`) (bei DML-Triggern)
- Ereignisse (bei DML-Triggern)
- Position in der Ausführungsfolge
- Änderungen am Code im Trigger-Body

Wenn ein Element nicht angegeben wird, bleibt es unverändert.

Ein DML-Trigger kann nicht in einen Datenbank- (oder DDL-)Trigger geändert werden.

Es ist nicht möglich, das/die Ereignis(e) oder die Phase eines Datenbank- (oder DDL-)Triggers zu ändern.

Merken Sie sich

Das Schlüsselwort `BEFORE` weist an, dass der Trigger ausgeführt wird, bevor das zugehörige Ereignis eintritt; das Schlüsselwort `AFTER` weist an, dass es nach dem Ereignis ausgeführt wird.

Mehrere DML-Ereignisse – `INSERT`, `UPDATE`, `DELETE` – können in einem einzigen Trigger abgedeckt werden. Die Ereignisse sollten mit dem Schlüsselwort `OR` getrennt werden. Kein Ereignis sollte mehr als einmal erwähnt werden.

Das Schlüsselwort `POSITION` ermöglicht die Angabe einer optionalen Ausführungsreihenfolge (“firing order”) für eine Reihe von Triggern, die die gleiche Phase und das gleiche Ereignis wie ihr Ziel haben. Die Standardposition ist 0. Wenn keine Positionen angegeben sind oder mehrere Trigger eine einzige Positionsnummer haben, werden die Trigger in alphabetischer Reihenfolge ihrer Namen ausgeführt.====

Wer kann einen Trigger ändern?

DML-Trigger können geändert werden durch:

- Administratoren
- Der Besitzer der Tabelle (oder Ansicht)
- Benutzer mit dem `ALTER ANY TABLE`- oder — für eine Ansicht — `ALTER ANY VIEW`-Privileg

Datenbank- und DDL-Trigger können geändert werden durch:

- Administratoren
- Benutzer mit dem `ALTER DATABASE`-Privileg

Beispiele mit ALTER TRIGGER

1. Den Trigger `set_cust_no` deaktivieren (in den inaktiven Zustand schalten).

```
ALTER TRIGGER set_cust_no INACTIVE;
```

2. Ändern der Position der Zündreihenfolge des Triggers `set_cust_no`.

```
ALTER TRIGGER set_cust_no POSITION 14;
```

3. Den Trigger `TR_CUST_LOG` in den inaktiven Zustand schalten und die Ereignisliste ändern.

```
ALTER TRIGGER TR_CUST_LOG
INACTIVE AFTER INSERT OR UPDATE;
```

4. Den `tr_log_connect Trigger` in den aktiven Status schalten, seine Position und seinen Körper ändern.

```
ALTER TRIGGER tr_log_connect
ACTIVE POSITION 1
AS
BEGIN
  INSERT INTO LOG_CONNECT (ID,
                           USERNAME,
                           ROLENAME,
                           ATIME)
  VALUES (NEXT VALUE FOR SEQ_LOG_CONNECT,
          CURRENT_USER,
          CURRENT_ROLE,
          CURRENT_TIMESTAMP);
END
```

Siehe auch

`CREATE TRIGGER`, `CREATE OR ALTER TRIGGER`, `RECREATE TRIGGER`, `DROP TRIGGER`

5.7.3. CREATE OR ALTER TRIGGER

Verwendet für

Erstellen eines neuen Triggers oder Ändern eines bestehenden Triggers

Verfügbar in

DSQL

Syntax

```
CREATE OR ALTER TRIGGER trigname
```

```
  { <relation_trigger_legacy>
  | <relation_trigger_sql2003>
  | <database_trigger>
  | <ddl_trigger> }
  <module-body>

!!Vgl. auch die Syntax CREATE TRIGGER für weitere Regeln !!
```

Die Anweisung `CREATE OR ALTER TRIGGER` erstellt einen neuen Trigger, falls dieser nicht existiert; andernfalls ändert und kompiliert es sie mit den intakten Privilegien und unberührten Abhängigkeiten.

Beispiel für `CREATE OR ALTER TRIGGER`

Neuen Trigger erstellen, wenn er nicht existiert, oder ihn ändern, falls vorhanden

```
CREATE OR ALTER TRIGGER set_cust_no
ACTIVE BEFORE INSERT POSITION 0 ON customer
AS
BEGIN
  IF (NEW.cust_no IS NULL) THEN
    NEW.cust_no = GEN_ID(cust_no_gen, 1);
END
```

Siehe auch

`CREATE TRIGGER`, `ALTER TRIGGER`, `RECREATE TRIGGER`

5.7.4. DROP TRIGGER

Verwendet für

Löschen eines vorhandenen Triggers

Verfügbar in

DSQL, ESQL

Syntax

```
DROP TRIGGER trigname
```

Tabelle 41. `DROP TRIGGER`-Anweisgungsparameter

Parameter	Beschreibung
trigname	Triggername

Die Anweisung `DROP TRIGGER` verwirft (löscht) einen vorhandenen Trigger.

Wer kann einen Trigger fallen lassen?

DML-Trigger können gelöscht werden durch:

- Administratoren
- Der Besitzer der Tabelle (oder Ansicht)
- Benutzer mit dem `ALTER ANY TABLE`- oder — für eine Ansicht — `ALTER ANY VIEW`-Privileg

Datenbank- und DDL-Trigger können gelöscht werden durch:

- Administratoren
- Benutzer mit dem `ALTER DATABASE`-Privileg

Beispiel für `DROP TRIGGER`

Löschen des Triggers `set_cust_no`

```
DROP TRIGGER set_cust_no;
```

Siehe auch

`CREATE TRIGGER`, `RECREATE TRIGGER`

5.7.5. `RECREATE TRIGGER`

Verwendet für

Erstellen eines neuen Triggers oder Neuerstellen eines vorhandenen Triggers

Verfügbar in

DSQL

Syntax

```
RECREATE TRIGGER trigname
  { <relation_trigger_legacy>
  | <relation_trigger_sql2003>
  | <database_trigger>
  | <ddl_trigger> }
  <module-body>

!! Vgl. auch die Syntax CREATE TRIGGER für weitere Regeln !!
```

Die Anweisung `RECREATE TRIGGER` erstellt einen neuen Trigger, wenn kein Trigger mit dem angegebenen Namen existiert; andernfalls versucht die Anweisung `RECREATE TRIGGER`, den vorhandenen Trigger zu löschen und einen neuen zu erstellen. Die Operation schlägt bei `COMMIT` fehl, wenn der Trigger verwendet wird.

Beachten Sie, dass Abhängigkeitsfehler erst in der `COMMIT`-Phase dieser Operation

> erkannt werden.

Beispiel für RECREATE TRIGGER

Erstellen oder erneutes Erstellen des Triggers set_cust_no.

```
RECREATE TRIGGER set_cust_no
ACTIVE BEFORE INSERT POSITION 0 ON customer
AS
BEGIN
  IF (NEW.cust_no IS NULL) THEN
    NEW.cust_no = GEN_ID(cust_no_gen, 1);
END
```

Siehe auch

CREATE TRIGGER, DROP TRIGGER, CREATE OR ALTER TRIGGER

5.8. PROCEDURE

Eine Stored Procedure ist ein Softwaremodul, das von einem Client, einer anderen Prozedur, Funktion, ausführbaren Block oder Trigger aufgerufen werden kann. Gespeicherte Prozeduren, gespeicherte Funktionen, ausführbare Blöcke und Trigger werden in prozeduralem SQL (PSQL) geschrieben. Die meisten SQL-Anweisungen sind auch in PSQL verfügbar, manchmal mit einigen Einschränkungen oder Erweiterungen, bemerkenswerte Einschränkungen sind DDL- und Transaktionssteuerungsanweisungen.

Gespeicherte Prozeduren können viele Eingabe- und Ausgabeparameter haben.

5.8.1. CREATE PROCEDURE

Verwendet für

Erstellen einer neuen gespeicherten Prozedur

Verfügbar in

DSQL, ESQL

Syntax

```
CREATE PROCEDURE procname [ ( [ <in_params> ] ) ]
  [RETURNS (<out_params>)]
  <module-body>

<module-body> ::=
  !! Siehe auch Syntax des Modul-Bodys !!

<in_params> ::= <inparam> [, <inparam> ...]

<inparam> ::= <param_decl> [{= | DEFAULT} <value>]
```

```
<out_params> ::= <outparam> [, <outparam> ...]

<outparam> ::= <param_decl>

<value> ::= {<literal> | NULL | <context_var>}

<param_decl> ::= paramname <domain_or_non_array_type> [NOT NULL]
  [COLLATE collation]

<type> ::=
    <datatype>
  | [TYPE OF] domain
  | TYPE OF COLUMN rel.col

<domain_or_non_array_type> ::=
  !! Siehe auch Syntax für Skalardatentypen !!
```

Tabelle 42. `CREATE PROCEDURE`-*Anweisungsparameter*

Parameter	Beschreibung
procname	Name der gespeicherten Prozedur, bestehend aus bis zu 31 Zeichen. Muss unter allen Tabellen-, Ansichts- und Prozedurnamen in der Datenbank eindeutig sein
inparam	Beschreibung der Eingabeparameter
outparam	Beschreibung der Ausgangsparameter
literal	Ein Literalwert, der mit dem Datentyp des Parameters zuweisungskompatibel ist
context_var	Jede Kontextvariable, deren Typ mit dem Datentyp des Parameters kompatibel ist
paramname	Der Name eines Eingabe- oder Ausgabeparameters der Prozedur. Er kann aus bis zu 31 Zeichen bestehen. Der Name des Parameters muss unter den Eingabe- und Ausgabeparametern der Prozedur und ihrer lokalen Variablen eindeutig sein
collation	Sortierreihenfolge

Die Anweisung `CREATE PROCEDURE` erstellt eine neue gespeicherte Prozedur. Der Name der Prozedur muss unter den Namen aller gespeicherten Prozeduren, Tabellen und Ansichten in der Datenbank eindeutig sein.

`CREATE PROCEDURE` ist eine *zusammengesetzte Anweisung*, bestehend aus einem Header und einem Body. Der Header gibt den Namen der Prozedur an und deklariert Eingabeparameter und gegebenenfalls Ausgabeparameter, die von der Prozedur zurückgegeben werden sollen.

Der Prozedurrumpf besteht aus Deklarationen für alle lokalen Variablen und benannten Cursors, die von der Prozedur verwendet werden, gefolgt von einer oder mehreren Anweisungen oder Anweisungsblöcken, die alle in einem äußeren Block eingeschlossen sind, der mit dem

Schlüsselwort `BEGIN` beginnt und mit . endet das Schlüsselwort `END`. Deklarationen und eingebettete Anweisungen werden mit Semikolons (‘;’) abgeschlossen.

Statement-Terminatoren

Einige Editoren für SQL-Anweisungen – insbesondere das Dienstprogramm *isql*, das mit Firebird geliefert wird, und möglicherweise einige Editoren von Drittanbietern – verwenden eine interne Konvention, die erfordert, dass alle Anweisungen mit einem Semikolon abgeschlossen werden. Dies führt beim Codieren in diesen Umgebungen zu einem Konflikt mit der PSQL-Syntax. Wenn Sie dieses Problem und seine Lösung nicht kennen, lesen Sie bitte die Details im PSQL-Kapitel im Abschnitt Umschalten des Terminators in *isql*.

Parameter

Jeder Parameter hat einen Datentyp. Der `NOT NULL`-Constraint kann auch für jeden Parameter angegeben werden, um zu verhindern, dass `NULL` übergeben oder ihm zugewiesen wird.

Eine Kollatierungssequenz kann für String-Typ-Parameter mit der `COLLATE`-Klausel angegeben werden.

Eingabeparameter

Eingabeparameter werden als Liste in Klammern nach dem Namen der Funktion angezeigt. Sie werden als Wert an die Prozedur übergeben, sodass Änderungen innerhalb der Prozedur keine Auswirkungen auf die Parameter im Aufrufer haben. Eingabeparameter können Standardwerte haben. Parameter mit angegebenen Standardwerten müssen am Ende der Parameterliste hinzugefügt werden.

Ausgabeparameter

Die optionale `RETURNS`-Klausel dient zum Angeben einer in Klammern gesetzten Liste von Ausgabeparametern für die gespeicherte Prozedur.

Variablen-, Cursor- und Sub-Routine-Deklarationen

Der optionale Deklarationsabschnitt, der sich am Anfang des Hauptteils der Prozedurdefinition befindet, definiert Variablen (einschließlich Cursors) und Unterroutinen lokal für die Prozedur. Lokale Variablendeklarationen folgen den gleichen Regeln wie Parameter bezüglich der Angabe des Datentyps. Weitere Informationen finden Sie im PSQL-Kapitel für `DECLARE VARIABLE`, `` DECLARE CURSOR` ``, `DECLARE FUNCTION` und `DECLARE PROCEDURE`.

Externe UDR-Prozeduren

Eine gespeicherte Prozedur kann sich auch in einem externen Modul befinden. In diesem Fall spezifiziert `CREATE PROCEDURE` anstelle eines Prozedurrumpfs die Position der Prozedur im externen Modul mit der `EXTERNAL`-Klausel. Die optionale `NAME`-Klausel spezifiziert den Namen des externen Moduls, den Namen der Prozedur innerhalb des Moduls und – optional – benutzerdefinierte Informationen. Die erforderliche `ENGINE`-Klausel gibt den Namen der UDR-Engine an, die die Kommunikation zwischen Firebird und dem externen Modul handhabt. Die optionale `AS`-Klausel akzeptiert ein String-Literal “body”, das von der Engine oder dem Modul für verschiedene Zwecke verwendet werden kann.

Wer kann ein Verfahren erstellen

Die CREATE PROCEDURE-Anweisung kann ausgeführt werden durch:

- Administratoren
- Benutzer mit dem Privileg CREATE PROCEDURE

Der Benutzer, der die Anweisung CREATE PROCEDURE ausführt, wird Eigentümer der Tabelle.

Beispiele

1. Erstellen einer gespeicherten Prozedur, die einen Datensatz in die BREED-Tabelle einfügt und den Code des eingefügten Datensatzes zurückgibt:

```
CREATE PROCEDURE ADD_BREED (
  NAME D_BREEDNAME, /* Domain attributes are inherited */
  NAME_EN TYPE OF D_BREEDNAME, /* Only the domain type is inherited */
  SHORTNAME TYPE OF COLUMN BREED.SHORTNAME,
    /* The table column type is inherited */
  REMARK VARCHAR(120) CHARACTER SET WIN1251 COLLATE PXW_CYRL,
  CODE_ANIMAL INT NOT NULL DEFAULT 1
)
RETURNS (
  CODE_BREED INT
)
AS
BEGIN
  INSERT INTO BREED (
    CODE_ANIMAL, NAME, NAME_EN, SHORTNAME, REMARK)
  VALUES (
    :CODE_ANIMAL, :NAME, :NAME_EN, :SHORTNAME, :REMARK)
  RETURNING CODE_BREED INTO CODE_BREED;
END
```

2. Erstellen einer auswählbaren gespeicherten Prozedur, die Daten für Adressetiketten generiert (aus employee.fdb):

```
CREATE PROCEDURE mail_label (cust_no INTEGER)
RETURNS (line1 CHAR(40), line2 CHAR(40), line3 CHAR(40),
         line4 CHAR(40), line5 CHAR(40), line6 CHAR(40))
AS
  DECLARE VARIABLE customer VARCHAR(25);
  DECLARE VARIABLE first_name VARCHAR(15);
  DECLARE VARIABLE last_name VARCHAR(20);
  DECLARE VARIABLE addr1 VARCHAR(30);
  DECLARE VARIABLE addr2 VARCHAR(30);
  DECLARE VARIABLE city VARCHAR(25);
  DECLARE VARIABLE state VARCHAR(15);
  DECLARE VARIABLE country VARCHAR(15);
```

```
  DECLARE VARIABLE postcode VARCHAR(12);
  DECLARE VARIABLE cnt INTEGER;
BEGIN
  line1 = '';
  line2 = '';
  line3 = '';
  line4 = '';
  line5 = '';
  line6 = '';

  SELECT customer, contact_first, contact_last, address_line1,
    address_line2, city, state_province, country, postal_code
  FROM CUSTOMER
  WHERE cust_no = :cust_no
  INTO :customer, :first_name, :last_name, :addr1, :addr2,
    :city, :state, :country, :postcode;

  IF (customer IS NOT NULL) THEN
    line1 = customer;
  IF (first_name IS NOT NULL) THEN
    line2 = first_name || ' ' || last_name;
  ELSE
    line2 = last_name;
  IF (addr1 IS NOT NULL) THEN
    line3 = addr1;
  IF (addr2 IS NOT NULL) THEN
    line4 = addr2;

  IF (country = 'USA') THEN
  BEGIN
    IF (city IS NOT NULL) THEN
      line5 = city || ', ' || state || '  ' || postcode;
    ELSE
      line5 = state || '  ' || postcode;
  END
  ELSE
  BEGIN
    IF (city IS NOT NULL) THEN
      line5 = city || ', ' || state;
    ELSE
      line5 = state;
    line6 = country || '    ' || postcode;
  END

  SUSPEND; -- die Anweisung, die eine Ausgabezeile an den Puffer sendet
           -- und die Prozedur "selektierbar" macht
END
```

Siehe auch

CREATE OR ALTER PROCEDURE, ALTER PROCEDURE, RECREATE PROCEDURE, DROP PROCEDURE

5.8.2. ALTER PROCEDURE

Verwendet für

Ändern einer vorhandenen gespeicherten Prozedur

Verfügbar in

DSQL, ESQL

Syntax

```
ALTER PROCEDURE procname [ ( [ <in_params> ] ) ]
  [RETURNS (<out_params>)]
  <module-body>

!! Vgl. auch die Syntax CREATE PROCEDURE für weitere Regeln !!
```

Die ALTER PROCEDURE-Anweisung ermöglicht die folgenden Änderungen an einer Stored-Procedure-Definition:

- der Satz und die Eigenschaften der Eingabe- und Ausgabeparameter
- lokale Variablen
- Code im Hauptteil der gespeicherten Prozedur

Nachdem ALTER PROCEDURE ausgeführt wurde, bleiben bestehende Privilegien intakt und Abhängigkeiten werden nicht beeinflusst.

Achten Sie darauf, die Anzahl und den Typ der Eingabe- und Ausgabeparameter in gespeicherten Prozeduren zu ändern. Vorhandener Anwendungscode und Prozeduren und Trigger, die ihn aufrufen, könnten ungültig werden, da die neue Beschreibung der Parameter nicht mit dem alten Aufrufformat kompatibel ist. Informationen zur Behebung einer solchen Situation finden Sie im Artikel Das RDB$VALID_BLR-Feld im Anhang.

Wer kann ein Verfahren ändern

Die Anweisung ALTER PROCEDURE kann ausgeführt werden durch:

- Administratoren
- Der Besitzer der gespeicherten Prozedur
- Benutzer mit der Berechtigung ALTER ANY PROCEDURE

ALTER PROCEDURE-Beispiel

Ändern der gespeicherten Prozedur GET_EMP_PROJ.

```
ALTER PROCEDURE GET_EMP_PROJ (
  EMP_NO SMALLINT)
RETURNS (
```

```
  PROJ_ID VARCHAR(20))
AS
BEGIN
  FOR SELECT
      PROJ_ID
    FROM
      EMPLOYEE_PROJECT
    WHERE
      EMP_NO = :emp_no
    INTO :proj_id
  DO
    SUSPEND;
END
```

Siehe auch

`CREATE PROCEDURE`, `CREATE OR ALTER PROCEDURE`, `RECREATE PROCEDURE`, `DROP PROCEDURE`

5.8.3. CREATE OR ALTER PROCEDURE

Verwendet für

Erstellen einer neuen gespeicherten Prozedur oder Ändern einer vorhandenen Prozedur

Verfügbar in

DSQL

Syntax

```
CREATE OR ALTER PROCEDURE procname [ ( [ <in_params> ] ) ]
  [RETURNS (<out_params>)]
  <module-body>

!! Vgl. auch die Syntax CREATE PROCEDURE für weitere Regeln !!
```

Die Anweisung "CREATE OR ALTER PROCEDURE" erstellt eine neue gespeicherte Prozedur oder ändert eine vorhandene. Wenn die gespeicherte Prozedur nicht existiert, wird sie durch transparentes Aufrufen einer `CREATE PROCEDURE`-Anweisung erstellt. Wenn die Prozedur bereits existiert, wird sie geändert und kompiliert, ohne ihre bestehenden Privilegien und Abhängigkeiten zu beeinträchtigen.

`CREATE OR ALTER PROCEDURE`-Beispiel

Erstellen oder Ändern der Prozedur `GET_EMP_PROJ`.

```
CREATE OR ALTER PROCEDURE GET_EMP_PROJ (
    EMP_NO SMALLINT)
RETURNS (
    PROJ_ID VARCHAR(20))
AS
BEGIN
```

```
  FOR SELECT
      PROJ_ID
    FROM
      EMPLOYEE_PROJECT
    WHERE
      EMP_NO = :emp_no
    INTO :proj_id
  DO
    SUSPEND;
END
```

Siehe auch

`CREATE PROCEDURE`, `ALTER PROCEDURE`, `RECREATE PROCEDURE`

5.8.4. DROP PROCEDURE

Verwendet für

Löschen einer gespeicherten Prozedur

Verfügbar in

DSQL, ESQL

Syntax

```
DROP PROCEDURE procname
```

Tabelle 43. `DROP PROCEDURE`*-Anweisungsparameter*

Parameter	Beschreibung
procname	Name einer vorhandenen gespeicherten Prozedur

Die Anweisung `DROP PROCEDURE` löscht eine vorhandene gespeicherte Prozedur. Wenn die gespeicherte Prozedur Abhängigkeiten aufweist, schlägt der Versuch, sie zu löschen, fehl und der entsprechende Fehler wird ausgegeben.

Wer kann ein Verfahren abbrechen

Die Anweisung `ALTER PROCEDURE` kann ausgeführt werden durch:

- Administratoren
- Der Besitzer der gespeicherten Prozedur
- Benutzer mit dem Privileg `DROP ANY PROCEDURE`

`DROP PROCEDURE`-Beispiel

Löschen der gespeicherten Prozedur `GET_EMP_PROJ`.

```
DROP PROCEDURE GET_EMP_PROJ;
```

Siehe auch

`CREATE PROCEDURE`, `RECREATE PROCEDURE`

5.8.5. RECREATE PROCEDURE

Verwendet für

Erstellen einer neuen gespeicherten Prozedur oder Neuerstellen einer vorhandenen Prozedur

Verfügbar in

DSQL

Syntax

```
RECREATE PROCEDURE procname [ ( [ <in_params> ] ) ]
  [RETURNS (<out_params>)]
  <module-body>

!! Vgl. auch die Syntax CREATE PROCEDURE für weitere Regeln !!
```

Die Anweisung 'RECREATE PROCEDURE' erstellt eine neue gespeicherte Prozedur oder erstellt eine vorhandene neu. Wenn es bereits eine Prozedur mit diesem Namen gibt, versucht die Engine, diese zu löschen und eine neue zu erstellen. Das Neuerstellen einer vorhandenen Prozedur schlägt bei der `COMMIT`-Anforderung fehl, wenn die Prozedur Abhängigkeiten hat.

Beachten Sie, dass Abhängigkeitsfehler erst in der `COMMIT`-Phase dieser Operation erkannt werden.

Nachdem eine Prozedur erfolgreich neu erstellt wurde, werden die Berechtigungen zum Ausführen der gespeicherten Prozedur und die Berechtigungen der gespeicherten Prozedur selbst gelöscht.

`RECREATE PROCEDURE`-Beispiel

Erstellen der neuen gespeicherten Prozedur `GET_EMP_PROJ` *oder Neuerstellen der vorhandenen gespeicherten Prozedur* `GET_EMP_PROJ`.

```
RECREATE PROCEDURE GET_EMP_PROJ (
  EMP_NO SMALLINT)
RETURNS (
  PROJ_ID VARCHAR(20))
AS
BEGIN
  FOR SELECT
      PROJ_ID
    FROM
```

```
      EMPLOYEE_PROJECT
    WHERE
      EMP_NO = :emp_no
    INTO :proj_id
  DO
    SUSPEND;
END
```

Siehe auch

`CREATE PROCEDURE`, `DROP PROCEDURE`, `CREATE OR ALTER PROCEDURE`

5.9. FUNCTION

Eine gespeicherte Funktion ist eine benutzerdefinierte Funktion, die in den Metadaten einer Datenbank gespeichert ist und auf dem Server ausgeführt wird. Gespeicherte Funktionen können von gespeicherten Prozeduren, gespeicherten Funktionen (einschließlich der Funktion selbst), Triggern und Client-Programmen aufgerufen werden. Wenn sich eine gespeicherte Funktion selbst aufruft, wird eine solche gespeicherte Funktion als rekursive Funktion bezeichnet.

Im Gegensatz zu gespeicherten Prozeduren geben gespeicherte Funktionen immer einen einzelnen Skalarwert zurück. Um einen Wert aus einer gespeicherten Funktion zurückzugeben, verwenden Sie die `RETURN`-Anweisung, die die Funktion sofort beendet.

Siehe auch

`EXTERNAL FUNCTION`

5.9.1. CREATE FUNCTION

Verwendet für

Erstellen einer neuen gespeicherten Funktion

Verfügbar in

DSQL

Syntax

```
CREATE FUNCTION funcname [ ( [ <in_params> ] ) ]
  RETURNS <domain_or_non_array_type> [COLLATE collation]
  [DETERMINISTIC]
  <module-body>

<module-body> ::=
  !! Vgl. Syntax des Modulrumpfes !!

<in_params> ::= <inparam> [, <inparam> ... ]

<inparam> ::= <param-decl> [ { = | DEFAULT } <value> ]

<value> ::= { <literal> | NULL | <context-var> }
```

```
<param-decl> ::= paramname <domain_or_non_array_type> [NOT NULL]
  [COLLATE collation]

<domain_or_non_array_type> ::=
  !! Vgl. Skalardatentyp-Syntax !!
```

Tabelle 44. `CREATE FUNCTION`-*Anweisungsparameter*

Parameter	Beschreibung
funcname	Gespeicherter Funktionsname bestehend aus bis zu 31 Zeichen. Muss unter allen Funktionsnamen in der Datenbank eindeutig sein.
inparam	Beschreibung der Eingabeparameter
collation	Sortierreihenfolge
literal	Ein Literalwert, der mit dem Datentyp des Parameters zuweisungskompatibel ist
context-var	Jede Kontextvariable, deren Typ mit dem Datentyp des Parameters kompatibel ist
paramname	Der Name eines Eingabeparameters der Funktion. Er kann aus bis zu 31 Zeichen bestehen. Der Name des Parameters muss unter den Eingabeparametern der Funktion und ihren lokalen Variablen eindeutig sein.

Die Anweisung `CREATE FUNCTION` erstellt eine neue gespeicherte Funktion. Der gespeicherte Funktionsname muss unter den Namen aller gespeicherten und externen (alten) Funktionen eindeutig sein, mit Ausnahme von Unterfunktionen oder Funktionen in Paketen. Bei Unterfunktionen oder Funktionen in Paketen muss der Name innerhalb ihres Moduls (Paket, Stored Procedure, Stored Function, Trigger) eindeutig sein.

Es ist ratsam, Funktionsnamen zwischen globalen gespeicherten Funktionen und gespeicherten Funktionen in Paketen nicht wiederzuverwenden, obwohl dies zulässig ist. Momentan ist es nicht möglich, eine Funktion oder Prozedur aus dem globalen Namensraum innerhalb eines Pakets aufzurufen, wenn dieses Paket eine Funktion oder Prozedur mit demselben Namen definiert. In dieser Situation wird die Funktion oder Prozedur des Pakets aufgerufen.

`CREATE FUNCTION` ist eine zusammengesetzte Anweisung mit einem Header und einem Body. Der Header definiert den Namen der gespeicherten Funktion und deklariert Eingabeparameter und Rückgabetyp.

Der Funktionsrumpf besteht aus optionalen Deklarationen von lokalen Variablen, benannten Cursorn und Unterprogrammen (Unterfunktionen und Unterprozeduren) und einer oder mehreren Anweisungen oder Anweisungsblöcken, eingeschlossen in einen äußeren Block, der mit dem Schlüsselwort `BEGIN` beginnt und endet mit dem Schlüsselwort `END`. Deklarationen und Anweisungen innerhalb des Funktionsrumpfs müssen mit einem Semikolon (‘;’) abgeschlossen werden.

Statement-Terminatoren

Einige SQL-Anweisungseditoren – insbesondere das mit Firebird gelieferte Dienstprogramm *isql* und möglicherweise einige Editoren von Drittanbietern – verwenden eine interne Konvention, die erfordert, dass alle Anweisungen mit einem Semikolon abgeschlossen werden. Dies führt beim Codieren in diesen Umgebungen zu einem Konflikt mit der PSQL-Syntax. Wenn Sie dieses Problem und seine Lösung nicht kennen, lesen Sie bitte die Details im PSQL-Kapitel im Abschnitt Terminator in *isql* umschalten.

Parameter

Jeder Parameter hat einen Datentyp.

Eine Kollatierungssequenz kann für String-Typ-Parameter mit der `COLLATE`-Klausel angegeben werden.

Eingabeparameter

Eingabeparameter werden als Liste in Klammern nach dem Namen der Funktion angezeigt. Sie werden als Wert an die Funktion übergeben, sodass Änderungen innerhalb der Funktion keine Auswirkungen auf die Parameter im Aufrufer haben. Die `NOT NULL`-Einschränkung kann auch für jeden Eingabeparameter angegeben werden, um zu verhindern, dass `NULL` übergeben oder zugewiesen wird. Eingabeparameter können Standardwerte haben. Parameter mit angegebenen Standardwerten müssen am Ende der Parameterliste hinzugefügt werden.

Ausgabeparameter

Die `RETURNS`-Klausel gibt den Rückgabetyp der gespeicherten Funktion an. Wenn eine Funktion einen String-Wert zurückgibt, ist es möglich, die Sortierung mit der `COLLATE`-Klausel anzugeben. Als Rückgabetyp können Sie einen Datentyp, einen Domänennamen, den Typ einer Domäne (mit `TYPE OF`) oder den Typ einer Spalte einer Tabelle oder View (mit `TYPE OF COLUMN`) angeben.

Deterministische Funktionen

Die optionale `DETERMINISTIC`-Klausel gibt an, dass die Funktion deterministisch ist. Deterministische Funktionen geben immer das gleiche Ergebnis für den gleichen Satz von Eingaben zurück. Nicht-deterministische Funktionen können für jeden Aufruf unterschiedliche Ergebnisse zurückgeben, sogar für denselben Satz von Eingaben. Wenn eine Funktion als deterministisch angegeben ist, wird eine solche Funktion möglicherweise nicht erneut aufgerufen, wenn sie bereits einmal mit den angegebenen Eingaben aufgerufen wurde, sondern übernimmt das Ergebnis aus einem Metadaten-Cache.

Aktuelle Versionen von Firebird speichern Ergebnisse deterministischer Funktionen nicht wirklich.

Die Angabe der `DETERMINISTIC`-Klausel ist eigentlich so etwas wie ein “Versprechen”, dass die Funktion für gleiche Eingaben dasselbe zurückgibt. Im Moment wird eine deterministische Funktion als Invariante betrachtet und funktioniert wie andere Invarianten. Das heißt, sie werden auf der aktuellen Ausführungsebene einer bestimmten Anweisung berechnet und zwischengespeichert.

Dies lässt sich leicht an einem Beispiel demonstrieren:

```
CREATE FUNCTION FN_T
RETURNS DOUBLE PRECISION DETERMINISTIC
AS
BEGIN
  RETURN rand();
END;

-- die Funktion wird zweimal ausgewertet und gibt 2 verschiedene Werte
zurück
SELECT fn_t() FROM rdb$database
UNION ALL
SELECT fn_t() FROM rdb$database;

-- die Funktion wird einmal ausgewertet und gibt 2 identische Werte
zurück
WITH t (n) AS (
  SELECT 1 FROM rdb$database
  UNION ALL
  SELECT 2 FROM rdb$database
)
SELECT n, fn_t() FROM t;
```

Variablen-, Cursor- und Sub-Routine-Deklarationen

Der optionale Deklarationsabschnitt, der sich am Anfang des Hauptteils der Funktionsdefinition befindet, definiert Variablen (einschließlich Cursors) und funktionslokale Unterroutinen. Lokale Variablendeklarationen folgen den gleichen Regeln wie Parameter bezüglich der Angabe des Datentyps. Weitere Informationen finden Sie im PSQL-Kapitel für `DECLARE VARIABLE`, `DECLARE CURSOR`, `DECLARE FUNCTION` und `DECLARE PROCEDURE`.

Funktionsrumpf

Auf den Header-Abschnitt folgt der Funktionsrumpf, der aus einer oder mehreren PSQL-Anweisungen besteht, die zwischen den äußeren Schlüsselwörtern `BEGIN` und `END` eingeschlossen sind. Mehrere `BEGIN ⋯ END`-Blöcke von beendeten Anweisungen können in den Prozedurrumpf eingebettet werden.

Externe UDR-Funktionen

Eine gespeicherte Funktion kann sich auch in einem externen Modul befinden. In diesem Fall spezifiziert `CREATE FUNCTION` anstelle eines Funktionsrumpfs die Position der Funktion im externen Modul mit der `EXTERNAL`-Klausel. Die optionale `NAME`-Klausel spezifiziert den Namen des externen Moduls, den Namen der Funktion innerhalb des Moduls und – optional – benutzerdefinierte Informationen. Die erforderliche `ENGINE`-Klausel gibt den Namen der UDR-Engine an, die die Kommunikation zwischen Firebird und dem externen Modul handhabt. Die optionale `AS`-Klausel akzeptiert ein String-Literal “body”, das von der Engine oder dem Modul für verschiedene Zwecke verwendet werden kann.

Externe UDR (User Defined Routine)-Funktionen, die mit `CREATE FUNCTION … EXTERNAL …` erstellt wurden, sollten nicht mit älteren UDFs (User Defined Functions) verwechselt werden, die mit `DECLARE EXTERNAL FUNCTION` deklariert wurden.

UDFs sind veraltet und ein Erbe früherer Firebird-Funktionen. Ihre Fähigkeiten sind den Fähigkeiten der neuen Art von externen UDR-Funktionen deutlich unterlegen.

Wer kann eine Funktion erstellen?

Die `CREATE FUNCTION`-Anweisung kann ausgeführt werden durch:

- Administratoren
- Benutzer mit dem Privileg `CREATE FUNCTION`TION

Der Benutzer, der die gespeicherte Funktion erstellt hat, wird deren Eigentümer.

`CREATE FUNCTION`-Beispiele

1. Erstellen einer gespeicherten Funktion

```
CREATE FUNCTION ADD_INT (A INT, B INT DEFAULT 0)
RETURNS INT
AS
BEGIN
  RETURN A + B;
END
```

   Aufruf einer Auswahl:

```
SELECT ADD_INT(2, 3) AS R FROM RDB$DATABASE
```

   Aufruf innerhalb von PSQL-Code, der zweite optionale Parameter wird nicht angegeben:

```
MY_VAR = ADD_INT(A);
```

2. Erstellen einer deterministischen gespeicherten Funktion

```
CREATE FUNCTION FN_E()
RETURNS DOUBLE PRECISION DETERMINISTIC
AS
BEGIN
  RETURN EXP(1);
END
```

3. Erstellen einer gespeicherten Funktion mit Parametern vom Typ Tabellenspalte

   Gibt den Namen eines Typs nach Feldname und Wert zurück

```
CREATE FUNCTION GET_MNEMONIC (
  AFIELD_NAME TYPE OF COLUMN RDB$TYPES.RDB$FIELD_NAME,
  ATYPE TYPE OF COLUMN RDB$TYPES.RDB$TYPE)
RETURNS TYPE OF COLUMN RDB$TYPES.RDB$TYPE_NAME
AS
BEGIN
  RETURN (SELECT RDB$TYPE_NAME
          FROM RDB$TYPES
          WHERE RDB$FIELD_NAME = :AFIELD_NAME
          AND RDB$TYPE = :ATYPE);
END
```

4. Erstellen einer extern gespeicherten Funktion

   Erstellen Sie eine Funktion, die sich in einem externen Modul (UDR) befindet. Die Funktionsimplementierung befindet sich im externen Modul `udrcpp_example`. Der Name der Funktion innerhalb des Moduls ist `wait_event`.

```
CREATE FUNCTION wait_event (
  event_name varchar (31) CHARACTER SET ascii
) RETURNS INTEGER
EXTERNAL NAME 'udrcpp_example!Wait_event'
ENGINE udr
```

5. Erstellen einer gespeicherten Funktion mit einer Unterfunktion

   Erstellen einer Funktion zum Konvertieren einer Zahl in das Hexadezimalformat.

```
CREATE FUNCTION INT_TO_HEX (
  ANumber BIGINT ,
  AByte_Per_Number SMALLINT = 8)
RETURNS CHAR (66)
AS
DECLARE VARIABLE xMod SMALLINT ;
DECLARE VARIABLE xResult VARCHAR (64);
DECLARE FUNCTION TO_HEX (ANum SMALLINT ) RETURNS CHAR
  AS
  BEGIN
    RETURN CASE ANum
      WHEN 0 THEN '0'
      WHEN 1 THEN '1'
      WHEN 2 THEN '2'
      WHEN 3 THEN '3'
      WHEN 4 THEN '4'
```

```
        WHEN 5 THEN '5'
        WHEN 6 THEN '6'
        WHEN 7 THEN '7'
        WHEN 8 THEN '8'
        WHEN 9 THEN '9'
        WHEN 10 THEN 'A'
        WHEN 11 THEN 'B'
        WHEN 12 THEN 'C'
        WHEN 13 THEN 'D'
        WHEN 14 THEN 'E'
        WHEN 15 THEN 'F'
        ELSE NULL
      END;
    END
  BEGIN
    xMod = MOD (ANumber, 16);
    ANumber = ANumber / 16;
    xResult = TO_HEX (xMod);
    WHILE (ANUMBER> 0) DO
    BEGIN
      xMod = MOD (ANumber, 16);
      ANumber = ANumber / 16;
      xResult = TO_HEX (xMod) || xResult;
    END
    RETURN '0x' || LPAD (xResult, AByte_Per_Number * 2, '0' );
  END
```

Siehe auch

CREATE OR ALTER FUNCTION, ALTER FUNCTION, RECREATE FUNCTION, DROP FUNCTION, DECLARE EXTERNAL FUNCTION

5.9.2. ALTER FUNCTION

Verwendet für

Ändern einer vorhandenen gespeicherten Funktion

Verfügbar in

DSQL

Syntax

```
ALTER FUNCTION funcname
  [ ( [ <in_params> ] ) ]
  RETURNS <domain_or_non_array_type> [COLLATE collation]
  [DETERMINISTIC]
  <module-body>

!! Vgl. Syntax CREATE FUNCTION für weitere Regeln !!
```

Die ALTER FUNCTION-Anweisung erlaubt die folgenden Änderungen an einer gespeicherten Funktionsdefinition:

- der Satz und die Eigenschaften des Eingangs- und Ausgangstyps
- lokale Variablen, benannte Cursor und Unterprogramme
- Code im Hauptteil der gespeicherten Prozedur

Für externe Funktionen (UDR) können Sie den Einstiegspunkt und den Engine-Namen ändern. Für ältere externe Funktionen, die mit DECLARE EXTERNAL FUNCTION deklariert wurden – auch als UDFs bekannt – ist es nicht möglich, in PSQL zu konvertieren und umgekehrt.

Nachdem ALTER FUNCTION ausgeführt wurde, bleiben bestehende Privilegien intakt und Abhängigkeiten werden nicht beeinflusst.

Achten Sie darauf, die Anzahl und den Typ der Eingabeparameter und den Ausgabetyp einer gespeicherten Funktion zu ändern. Vorhandener Anwendungscode und Prozeduren, Funktionen und Trigger, die ihn aufrufen, könnten ungültig werden, weil die neue Beschreibung der Parameter nicht mit dem alten Aufrufformat kompatibel ist. Informationen zur Fehlerbehebung in einer solchen Situation finden Sie im Artikel Das RDB$VALID_BLR-Feld im Anhang.

Wer kann eine Funktion ändern

Die ALTER FUNCTION-Anweisung kann ausgeführt werden durch:

- Administratoren
- Inhaber der gespeicherten Funktion
- Benutzer mit der Berechtigung ALTER ANY FUNCTION

Beispiele für ALTER FUNCTION

Ändern einer gespeicherten Funktion

```
ALTER FUNCTION ADD_INT(A INT, B INT, C INT)
RETURNS INT
AS
BEGIN
  RETURN A + B + C;
END
```

Siehe auch

CREATE FUNCTION, CREATE OR ALTER FUNCTION, RECREATE FUNCTION, DROP FUNCTION

5.9.3. CREATE OR ALTER FUNCTION

Verwendet für

Erstellen einer neuen oder Ändern einer vorhandenen gespeicherten Funktion

Verfügbar in

DSQL

Syntax

```
CREATE OR ALTER FUNCTION funcname
  [ ( [ <in_params> ] ) ]
  RETURNS <domain_or_non_array_type> [COLLATE collation]
  [DETERMINISTIC]
  <module-body>

!! Vgl. Syntax CREATE FUNCTION für weitere Regeln !!
```

Die Anweisung `CREATE OR ALTER FUNCTION` erstellt eine neue gespeicherte Funktion oder ändert eine vorhandene. Wenn die gespeicherte Funktion nicht existiert, wird sie durch transparentes Aufrufen einer `CREATE FUNCTION`-Anweisung erstellt. Wenn die Funktion bereits existiert, wird sie geändert und kompiliert (durch `ALTER FUNCTION`), ohne ihre bestehenden Privilegien und Abhängigkeiten zu beeinträchtigen.

Beispiele für `CREATE OR ALTER FUNCTION`

Erstellen Sie eine neue oder ändern Sie eine vorhandene gespeicherte Funktion

```
CREATE OR ALTER FUNCTION ADD_INT(A INT, B INT DEFAULT 0)
RETURNS INT
AS
BEGIN
  RETURN A + B;
END
```

Siehe auch

`CREATE FUNCTION`, `ALTER FUNCTION`, `DROP FUNCTION`

5.9.4. DROP FUNCTION

Verwendet für

Löschen einer gespeicherten Funktion

Verfügbar in

DSQL

Syntax

```
DROP FUNCTION funcname
```

Tabelle 45. `DROP FUNCTION`-Anweisungsparameter

Parameter	Beschreibung
funcname	Gespeicherter Funktionsname bestehend aus bis zu 31 Zeichen. Muss unter allen Funktionsnamen in der Datenbank eindeutig sein.

Die DROP FUNCTION-Anweisung löscht eine vorhandene gespeicherte Funktion. Wenn die gespeicherte Funktion Abhängigkeiten aufweist, schlägt der Versuch, sie zu löschen, fehl und der entsprechende Fehler wird ausgegeben.

Wer kann eine Funktion löschen?

Die DROP FUNCTION-Anweisung kann ausgeführt werden durch:

- Administratoren
- Inhaber der gespeicherten Funktion
- Benutzer mit dem Privileg DROP ANY FUNCTION

Beispiele für DROP FUNCTION

```
DROP FUNCTION ADD_INT;
```

Siehe auch

CREATE FUNCTION, CREATE OR ALTER FUNCTION, RECREATE FUNCTION

5.9.5. RECREATE FUNCTION

Verwendet für

Erstellen einer neuen gespeicherten Funktion oder Neuerstellen einer vorhandenen Funktion

Verfügbar in

DSQL

Syntax

```
RECREATE FUNCTION funcname
  [ ( [ <in_params> ] ) ]
  RETURNS <domain_or_non_array_type> [COLLATE collation]
  [DETERMINISTIC]
  <module-body>

!! Vgl. Syntax CREATE FUNCTION für weitere Regeln !!
```

Die Anweisung 'RECREATE FUNCTION' erstellt eine neue gespeicherte Funktion oder erstellt eine vorhandene neu. Wenn es bereits eine Funktion mit diesem Namen gibt, versucht die Engine, sie zu löschen und dann eine neue zu erstellen. Das Neuerstellen einer vorhandenen Funktion schlägt bei COMMIT fehl, wenn die Funktion Abhängigkeiten hat.

Beachten Sie, dass Abhängigkeitsfehler erst in der COMMIT-Phase dieser Operation erkannt werden.

Nachdem eine Prozedur erfolgreich neu erstellt wurde, werden vorhandene Berechtigungen zum Ausführen der gespeicherten Funktion und der Berechtigungen der gespeicherten Funktion selbst werden verworfen.

Beispiele für RECREATE FUNCTION

Erstellen oder Wiederherstellen einer gespeicherten Funktion

```
RECREATE FUNCTION ADD_INT(A INT, B INT DEFAULT 0)
RETURNS INT
AS
BEGIN
  RETURN A + B;
EN
```

Siehe auch

CREATE FUNCTION, DROP FUNCTION

5.10. EXTERNAL FUNCTION

Externe Funktionen, auch bekannt als “User-Defined Functions” (UDFs) sind Programme, die in einer externen Programmiersprache geschrieben und in dynamisch geladenen Bibliotheken gespeichert sind. Einmal in einer Datenbank deklariert, stehen sie in dynamischen und prozeduralen Anweisungen zur Verfügung, als wären sie in der Sprache SQL implementiert.

Externe Funktionen erweitern die Möglichkeiten der Datenverarbeitung mit SQL erheblich. Um einer Datenbank eine Funktion zur Verfügung zu stellen, wird sie mit der Anweisung DECLARE EXTERNAL FUNCTION deklariert.

Die Bibliothek, die eine Funktion enthält, wird geladen, wenn eine darin enthaltene Funktion aufgerufen wird.

Externe Funktionen, die als EXTERNAL FUNCTION deklariert wurden, sind ein Erbe früherer Versionen von Firebird. Ihre Fähigkeiten sind den Fähigkeiten des neuen Typs externer Funktionen, UDR (User-Defined Routine), unterlegen. Solche Funktionen werden als CREATE FUNCTION ⋯ EXTERNAL ⋯ deklariert. Siehe CREATE FUNCTION für Details.

Externe Funktionen können in mehr als einer Bibliothek enthalten sein — oder “Modul”, wie es in der Syntax genannt wird.

UDFs sind grundsätzlich unsicher. Wir empfehlen, ihre Verwendung nach Möglichkeit zu vermeiden und UDFs in Ihrer Datenbankkonfiguration zu deaktivieren (UdfAccess = None in firebird.conf). Wenn Sie nativen Code aus Ihrer

> Datenbank aufrufen müssen, verwenden Sie stattdessen eine externe UDR-Engine.

Siehe auch

FUNCTION

5.10.1. DECLARE EXTERNAL FUNCTION

Verwendet für

Deklarieren einer benutzerdefinierten Funktion (UDF) zur Datenbank

Verfügbar in

DSQL, ESQL

Syntax

```
DECLARE EXTERNAL FUNCTION funcname
  [{ <arg_desc_list> | ( <arg_desc_list> ) }]
  RETURNS { <return_value> | ( <return_value> ) }
  ENTRY_POINT 'entry_point' MODULE_NAME 'library_name'

<arg_desc_list> ::=
  <arg_type_decl> [, <arg_type_decl> ...]

<arg_type_decl> ::=
  <udf_data_type> [BY {DESCRIPTOR | SCALAR_ARRAY} | NULL]

<udf_data_type> ::=
    <scalar_datatype>
  | BLOB
  | CSTRING(length) [ CHARACTER SET charset ]

<scalar_datatype> ::=
  !! Vgl. Syntax für Skalardatentypen !!

<return_value> ::=
  { <udf_data_type> | PARAMETER param_num }
  [{ BY VALUE | BY DESCRIPTOR [FREE_IT] | FREE_IT }]
```

Tabelle 46. DECLARE EXTERNAL FUNCTION-*Anweisungsparameter*

Parameter	Beschreibung
funcname	Funktionsname in der Datenbank. Er kann aus bis zu 31 Zeichen bestehen. Er sollte unter allen internen und externen Funktionsnamen in der Datenbank eindeutig sein und muss nicht mit dem Namen identisch sein, der über ENTRY_POINT aus der UDF-Bibliothek exportiert wird.
entry_point	Der exportierte Name der Funktion

Parameter	Beschreibung
library_name	Der Name des Moduls (`MODULE_NAME`), aus dem die Funktion exportiert wird. Dies ist der Name der Datei ohne die Dateierweiterung “.dll” oder “.so”.
length	Die maximale Länge einer nullterminierten Zeichenfolge, angegeben in Byte
charset	Zeichensatz des CSTRING
param_num	Die Nummer des Eingabeparameters, nummeriert von 1 in der Liste der Eingabeparameter in der Deklaration, die den Datentyp beschreibt, der von der Funktion zurückgegeben wird

Die Anweisung `DECLARE EXTERNAL FUNCTION` stellt eine benutzerdefinierte Funktion in der Datenbank zur Verfügung. UDF-Deklarationen müssen in *jeder Datenbank* vorgenommen werden, die sie verwenden wird. Es müssen keine UDFs deklariert werden, die niemals verwendet werden.

Der Name der externen Funktion muss unter allen Funktionsnamen eindeutig sein. Er kann sich vom exportierten Namen der Funktion unterscheiden, wie im Argument `ENTRY_POINT` angegeben.

`DECLARE EXTERNAL FUNCTION`-Eingabeparameter

Die Eingabeparameter der Funktion folgen dem Namen der Funktion und werden durch Kommas getrennt. Für jeden Parameter ist ein SQL-Datentyp angegeben. Arrays können nicht als Funktionsparameter verwendet werden. Zusätzlich zu den SQL-Typen steht der Typ `CSTRING` zur Angabe eines nullterminierten Strings mit einer maximalen Länge von `LENGTH` Bytes zur Verfügung. Es gibt mehrere Mechanismen, um einen Parameter von der Firebird-Engine an eine externe Funktion zu übergeben. Jeder dieser Mechanismen wird unten diskutiert.

Standardmäßig werden Eingabeparameter *per Referenz* übergeben. Es gibt keine separate Klausel, die explizit angibt, dass Parameter als Referenz übergeben werden.

Wenn ein `NULL`-Wert als Referenz übergeben wird, wird dieser in das Äquivalent von Null umgewandelt, zum Beispiel eine Zahl `0` oder eine leere Zeichenfolge (“''”). Wenn nach einem Parameter das Schlüsselwort `NULL` angegeben wird, wird bei der Übergabe von `NULL`-Werten der Nullzeiger an die externe Funktion übergeben.

Das Deklarieren einer Funktion mit dem Schlüsselwort `NULL` garantiert nicht, dass die Funktion einen `NULL`-Eingabeparameter korrekt behandelt. Jede Funktion muss geschrieben oder umgeschrieben werden, um `NULL`-Werte korrekt zu behandeln. Verwenden Sie immer die vom Entwickler bereitgestellte Funktionsdeklaration.

Wenn `BY DESCRIPTOR` angegeben ist, wird der Eingabeparameter vom Deskriptor übergeben. In diesem Fall erhält der UDF-Parameter einen Zeiger auf eine interne Struktur, die als Deskriptor bekannt ist. Der Deskriptor enthält Informationen über Datentyp, Untertyp, Genauigkeit, Zeichensatz und Kollation, Skalierung, einen Zeiger auf die Daten selbst und einige Flags, einschließlich des `NULL`-Indikators. Diese Deklaration funktioniert nur, wenn die externe Funktion mit einem Handle geschrieben wird.

Wenn ein Funktionsparameter per Deskriptor übergeben wird, wird der übergebene Wert nicht in den deklarierten Datentyp umgewandelt.

Die Klausel `BY SCALAR_ARRAY` wird verwendet, wenn Arrays als Eingabeparameter übergeben werden. Im Gegensatz zu anderen Typen können Sie kein Array aus einer UDF zurückgeben.

Klauseln und Schlüsselwörter

`RETURNS`-Klausel

(Erforderlich) gibt den von der Funktion zurückgegebenen Ausgabeparameter an. Eine Funktion ist skalar, sie gibt einen Wert (Ausgabeparameter) zurück. Der Ausgabeparameter kann einen beliebigen SQL-Typ (außer einem Array oder einem Array-Element) oder ein nullterminierter String (`CSTRING`) sein. Der Ausgabeparameter kann als Referenz (Standard), als Deskriptor oder als Wert übergeben werden. Wenn die Klausel `BY DESCRIPTOR` angegeben ist, wird der Ausgabeparameter vom Deskriptor übergeben. Wenn die Klausel `BY VALUE` angegeben ist, wird der Ausgabeparameter als Wert übergeben.

`PARAMETER`-Schlüsselwort

gibt an, dass die Funktion den Wert des Parameters unter der Nummer *param_num* zurückgibt. Es ist notwendig, wenn Sie einen Wert vom Datentyp `BLOB` zurückgeben müssen.

`FREE_IT`-Schlüsselwort

bedeutet, dass der zum Speichern des Rückgabewerts zugewiesene Speicher freigegeben wird, nachdem die Funktion ausgeführt wurde. Es wird nur verwendet, wenn der Speicher im UDF dynamisch allokiert wurde. In einem solchen UDF muss der Speicher mit Hilfe der Funktion `ib_util_malloc` aus dem Modul `ib_util` allokiert werden, eine Voraussetzung für die Kompatibilität mit den im Firebird-Code verwendeten Funktionen und im Code der ausgelieferten UDF-Module zum Zuweisen und Freigeben von Speicher.

`ENTRY_POINT`-Klausel

gibt den Namen des Einstiegspunkts (den Namen der importierten Funktion) an, wie er aus dem Modul exportiert wurde.

`MODULE_NAME`-Klausel

definiert den Namen des Moduls, in dem sich die exportierte Funktion befindet. Der Link zum Modul sollte nicht der vollständige Pfad und die Erweiterung der Datei sein, wenn dies vermieden werden kann. Wenn sich das Modul am Standardspeicherort (im `../UDF` -Unterverzeichnis des Firebird-Server-Roots) oder an einem explizit in `firebird.conf` konfigurierten Speicherort befindet, erleichtert es das Verschieben der Datenbank zwischen verschiedene Plattformen. Der Parameter `UDFAccess` in der Datei firebird.conf ermöglicht die Konfiguration von Zugriffsbeschränkungen auf externe Funktionsmodule.

Jeder mit der Datenbank verbundene Benutzer kann eine externe Funktion (UDF) deklarieren.

Wer kann eine externe Funktion erstellen?

Die Anweisung `DECLARE EXTERNAL FUNCTION` kann ausgeführt werden durch:

- Administratoren

- Benutzer mit dem Privileg `CREATE FUNCTION`

Der Benutzer, der die Funktion erstellt hat, wird ihr Besitzer.

Beispiele für die Verwendung von `DECLARE EXTERNAL FUNCTION`

1. Deklarieren der externen Funktion `addDay` im Modul `fbudf`. Die Eingabe- und Ausgabeparameter werden als Referenz übergeben.

```
DECLARE EXTERNAL FUNCTION addDay
  TIMESTAMP, INT
  RETURNS TIMESTAMP
  ENTRY_POINT 'addDay' MODULE_NAME 'fbudf';
```

2. Deklarieren der externen Funktion `invl` im Modul `fbudf`. Die Eingabe- und Ausgabeparameter werden per Deskriptor übergeben.

```
DECLARE EXTERNAL FUNCTION invl
  INT BY DESCRIPTOR, INT BY DESCRIPTOR
  RETURNS INT BY DESCRIPTOR
  ENTRY_POINT 'idNvl' MODULE_NAME 'fbudf';
```

3. Deklarieren der externen Funktion `isLeapYear` im Modul `fbudf`. Der Eingabeparameter wird als Referenz übergeben, während der Ausgabeparameter als Wert übergeben wird.

```
DECLARE EXTERNAL FUNCTION isLeapYear
  TIMESTAMP
  RETURNS INT BY VALUE
  ENTRY_POINT 'isLeapYear' MODULE_NAME 'fbudf';
```

4. Deklarieren der externen Funktion `i64Truncate` im Modul `fbudf`. Die Eingabe- und Ausgabeparameter werden per Deskriptor übergeben. Als Rückgabewert wird der zweite Parameter der Funktion verwendet.

```
DECLARE EXTERNAL FUNCTION i64Truncate
  NUMERIC(18) BY DESCRIPTOR, NUMERIC(18) BY DESCRIPTOR
  RETURNS PARAMETER 2
  ENTRY_POINT 'fbtruncate' MODULE_NAME 'fbudf';
```

Siehe auch

`ALTER EXTERNAL FUNCTION`, `DROP EXTERNAL FUNCTION`, `CREATE FUNCTION`

5.10.2. ALTER EXTERNAL FUNCTION

Verwendet für

Ändern des Einstiegspunkts und/oder des Modulnamens für eine benutzerdefinierte Funktion (UDF)

Verfügbar in

DSQL

Syntax

```
ALTER EXTERNAL FUNCTION funcname
  [ENTRY_POINT 'new_entry_point']
  [MODULE_NAME 'new_library_name']
```

Tabelle 47. `ALTER EXTERNAL FUNCTION`-*Anweisungsparameter*

Parameter	Beschreibung
funcname	Funktionsname in der Datenbank
new_entry_point	Der neue exportierte Name der Funktion
new_library_name	Der neue Name des Moduls (`MODULE_NAME` aus dem die Funktion exportiert wird). Dies ist der Name der Datei ohne die Dateierweiterung ".dll" oder ".so".

Die Anweisung `ALTER EXTERNAL FUNCTION` ändert den Einstiegspunkt und/oder den Modulnamen für eine benutzerdefinierte Funktion (UDF). Vorhandene Abhängigkeiten bleiben erhalten, nachdem die Anweisung ausgeführt wird, die die Änderung(en) enthält.

Die `ENTRY_POINT`-Klausel

dient zur Angabe des neuen Einstiegspunkts (der Name der Funktion, wie er aus dem Modul exportiert wurde).

Die `MODULE_NAME`-Klausel

dient zur Angabe des neuen Namens des Moduls, in dem sich die exportierte Funktion befindet.

Jeder mit der Datenbank verbundene Benutzer kann den Einstiegspunkt und den Modulnamen ändern.

Wer kann eine externe Funktion ändern?

Die Anweisung `ALTER EXTERNAL FUNCTION` kann ausgeführt werden durch:

- Administratoren
- Inhaber der externen Funktion
- Benutzer mit der Berechtigung `ALTER ANY FUNCTION`

Beispiele zur Verwendung `ALTER EXTERNAL FUNCTION`

Ändern des Einstiegspunkts für eine externe Funktion

```
ALTER EXTERNAL FUNCTION invl ENTRY_POINT 'intNvl';
```

Ändern des Modulnamens für eine externe Funktion

```
ALTER EXTERNAL FUNCTION invl MODULE_NAME 'fbudf2';
```

Siehe auch

`DECLARE EXTERNAL FUNCTION`, `DROP EXTERNAL FUNCTION`

5.10.3. DROP EXTERNAL FUNCTION

Verwendet für

Entfernen einer benutzerdefinierten Funktion (UDF) aus einer Datenbank

Verfügbar in

DSQL, ESQL

Syntax

```
DROP EXTERNAL FUNCTION funcname
```

Tabelle 48. `DROP EXTERNAL FUNCTION`*-Anweisungsparameter*

Parameter	Beschreibung
funcname	Funktionsname in der Datenbank

Die Anweisung `DROP EXTERNAL FUNCTION` löscht die Deklaration einer benutzerdefinierten Funktion aus der Datenbank. Wenn Abhängigkeiten von der externen Funktion bestehen, schlägt die Anweisung fehl und der entsprechende Fehler wird ausgegeben.

Jeder mit der Datenbank verbundene Benutzer kann die Deklaration einer internen Funktion löschen.

Wer kann eine externe Funktion löschen?

Die Anweisung `DROP EXTERNAL FUNCTION` kann ausgeführt werden durch:

- Administratoren
- Inhaber der externen Funktion
- Benutzer mit dem Privileg `DROP ANY FUNCTION`

Beispiel für `DROP EXTERNAL FUNCTION`

Löschen der Deklaration der Funktion `addDay`.

```
DROP EXTERNAL FUNCTION addDay;
```

Siehe auch

`DECLARE EXTERNAL FUNCTION`

5.11. PACKAGE

Ein Paket ist eine Gruppe von Prozeduren und Funktionen, die als eine Einheit verwaltet werden.

5.11.1. CREATE PACKAGE

Verwendet für

Paket-Header deklarieren

Verfügbar in

DSQL

Syntax

```
CREATE PACKAGE package_name
AS
BEGIN
  [ <package_item> ... ]
END

<package_item> ::=
    <function_decl>;
  | <procedure_decl>;

<function_decl> ::=
  FUNCTION funcname [ ( [ <in_params> ] ) ]
  RETURNS <domain_or_non_array_type> [COLLATE collation]
  [DETERMINISTIC]

<procedure_decl> ::=
  PROCEDURE procname [ ( [ <in_params> ] ) ]
  [RETURNS (<out_params>)]

<in_params> ::= <inparam> [, <inparam> ... ]

<inparam> ::= <param_decl> [ { = | DEFAULT } <value> ]

<out_params> ::= <outparam> [, <outparam> ...]

<outparam> ::= <param_decl>

<value> ::= { literal | NULL | context_var }
```

```
<param-decl> ::= paramname <domain_or_non_array_type> [NOT NULL]
  [COLLATE collation]

<domain_or_non_array_type> ::=
  !! Siehe auch Skalardatentypen-Syntax !!
```

Tabelle 49. `CREATE PACKAGE`-*Anweisungsparameter*

Parameter	Beschreibung
package_name	Paketname bestehend aus bis zu 31 Zeichen. Der Paketname muss unter allen Paketnamen eindeutig sein.
function_decl	Funktionsdeklaration
procedure_decl	Prozedurdeklaration
func_name	Funktionsname bestehend aus bis zu 31 Zeichen. Der Funktionsname muss innerhalb des Pakets eindeutig sein.
proc_name	Prozedurname bestehend aus bis zu 31 Zeichen. Der Funktionsname muss innerhalb des Pakets eindeutig sein.
collation	Sortierreihenfolge
inparam	Deklaration der Eingabeparameter
outparam	Deklaration der Ausgabeparameter
literal	Ein Literalwert, der mit dem Datentyp des Parameters zuweisungskompatibel ist
context_var	Jede Kontextvariable, die mit dem Datentyp des Parameters zuweisungskompatibel ist
paramname	Der Name eines Eingabeparameters einer Prozedur oder Funktion oder eines Ausgabeparameters einer Prozedur. Er kann aus bis zu 31 Zeichen bestehen. Der Name des Parameters muss unter den Eingabe- und Ausgabeparametern der Prozedur oder Funktion eindeutig sein.

Die Anweisung `CREATE PACKAGE` erstellt einen neuen Paket-Header. Im Paketheader deklarierte Routinen (Prozeduren und Funktionen) sind außerhalb des Pakets unter Verwendung des vollständigen Bezeichners (*package_name.proc_name* oder *package_name.func_name*) verfügbar. Routinen, die nur im Paketrumpf definiert sind – aber nicht im Paketkopf – sind außerhalb des Pakets nicht sichtbar.

Paketprozedur- und Funktionsnamen können globale Routinen überschatten

Wenn ein Paketheader oder Paketrumpf eine Prozedur oder Funktion mit demselben Namen wie eine gespeicherte Prozedur oder Funktion im globalen Namespace deklariert, ist es nicht möglich, diese globale Prozedur oder Funktion aus dem Paketrumpf aufzurufen. In diesem Fall wird immer die Prozedur oder Funktion des Pakets aufgerufen.

Aus diesem Grund wird empfohlen, dass sich die Namen von gespeicherten

> Prozeduren und Funktionen in Paketen nicht mit Namen von gespeicherten Prozeduren und Funktionen im globalen Namespace überschneiden.

Statement-Terminatoren

Einige SQL-Anweisungseditoren – insbesondere das mit Firebird gelieferte Dienstprogramm *isql* und möglicherweise einige Editoren von Drittanbietern – verwenden eine interne Konvention, die erfordert, dass alle Anweisungen mit einem Semikolon abgeschlossen werden. Dies führt beim Codieren in diesen Umgebungen zu einem Konflikt mit der PSQL-Syntax. Wenn Sie dieses Problem und seine Lösung nicht kennen, lesen Sie bitte die Details im PSQL-Kapitel im Abschnitt Umschalten des Terminators in *isql*.

Verfahrens- und Funktionsparameter

Ausführliche Informationen zu Parametern für gespeicherte Prozeduren finden Sie unter Parameter in `CREATE PROCEDURE`.

Einzelheiten zu Funktionsparametern finden Sie unter Parameter in `CREATE FUNCTION`.

Wer kann ein Paket erstellen

Die `CREATE PACKAGE`-Anweisung kann ausgeführt werden durch:

- Administratoren
- Benutzer mit dem Privileg `CREATE PACKAGE`

Der Benutzer, der den Paketheader erstellt hat, wird sein Besitzer.

Beispiele für `CREATE PACKAGE`

Erstellen Sie einen Paket-Header

```
CREATE PACKAGE APP_VAR
AS
BEGIN
  FUNCTION GET_DATEBEGIN() RETURNS DATE DETERMINISTIC;
  FUNCTION GET_DATEEND() RETURNS DATE DETERMINISTIC;
  PROCEDURE SET_DATERANGE(ADATEBEGIN DATE,
      ADATEEND DATE DEFAULT CURRENT_DATE);
END
```

Siehe auch

`CREATE PACKAGE BODY`, `RECREATE PACKAGE BODY`, `ALTER PACKAGE`, `DROP PACKAGE`, `RECREATE PACKAGE`

5.11.2. `ALTER PACKAGE`

Verwendet für

Ändern des Paketheaders

Verfügbar in

DSQL

Syntax

```
ALTER PACKAGE package_name
AS
BEGIN
  [ <package_item> ... ]
END

!! Vgl. Syntax CREATE PACKAGE für weitere Regeln!!
```

Die `ALTER PACKAGE`-Anweisung modifiziert den Paket-Header. Es kann verwendet werden, um die Anzahl und Definition von Prozeduren und Funktionen einschließlich ihrer Ein- und Ausgabeparameter zu ändern. Der Quelltext und die kompilierte Form des Paketkörpers werden jedoch beibehalten, obwohl der Körper nach der Änderung des Paketheaders möglicherweise inkompatibel ist. Die Gültigkeit eines Paketkörpers für den definierten Header wird in der Spalte `RDB$PACKAGES.RDB$VALID_BODY_FLAG` gespeichert.

Wer kann ein Paket ändern

Die `ALTER PACKAGE`-Anweisung kann ausgeführt werden durch:

- Administratoren
- Der Besitzer des Pakets
- Benutzer mit der Berechtigung `ALTER ANY PACKAGE`

Beispiel für `ALTER PACKAGE`

Ändern eines Paketheaders

```
ALTER PACKAGE APP_VAR
AS
BEGIN
  FUNCTION GET_DATEBEGIN() RETURNS DATE DETERMINISTIC;
  FUNCTION GET_DATEEND() RETURNS DATE DETERMINISTIC;
  PROCEDURE SET_DATERANGE(ADATEBEGIN DATE,
      ADATEEND DATE DEFAULT CURRENT_DATE);
END
```

Siehe auch

`CREATE PACKAGE`, `DROP PACKAGE`, `ALTER PACKAGE BODY`, `RECREATE PACKAGE BODY`

5.11.3. CREATE OR ALTER PACKAGE

Verwendet für

Erstellen eines neuen oder Ändern eines bestehenden Paket-Headers

Verfügbar in

DSQL

Syntax

```
CREATE OR ALTER PACKAGE package_name
AS
BEGIN
  [ <package_item> ... ]
END

!! Siehe auch Syntax CREATE PACKAGE für weitere Regeln!!
```

Die Anweisung `CREATE OR ALTER PACKAGE` erstellt ein neues Paket oder ändert einen vorhandenen Paket-Header. Existiert der Paket-Header nicht, wird er mit `CREATE PACKAGE` erstellt. Wenn es bereits existiert, wird es mit `ALTER PACKAGE` modifiziert, während bestehende Privilegien und Abhängigkeiten beibehalten werden.

Beispiel für `CREATE OR ALTER PACKAGE`

Erstellen eines neuen oder Ändern eines vorhandenen Paketheaders

```
CREATE OR ALTER PACKAGE APP_VAR
AS
BEGIN
  FUNCTION GET_DATEBEGIN() RETURNS DATE DETERMINISTIC;
  FUNCTION GET_DATEEND() RETURNS DATE DETERMINISTIC;
  PROCEDURE SET_DATERANGE(ADATEBEGIN DATE,
      ADATEEND DATE DEFAULT CURRENT_DATE);
END
```

Siehe auch

`CREATE PACKAGE`, `ALTER PACKAGE`, `RECREATE PACKAGE`, `ALTER PACKAGE BODY`, `RECREATE PACKAGE BODY`

5.11.4. DROP PACKAGE

Verwendet für

Einen Paket-Header löschen

Verfügbar in

DSQL

Syntax

```
DROP PACKAGE package_name
```

Tabelle 50. `DROP PACKAGE`-Anweisungsparameter

Parameter	Beschreibung
package_name	Paketname

Die DROP PACKAGE-Anweisung löscht einen vorhandenen Paket-Header. Wenn ein Paketkörper vorhanden ist, wird er zusammen mit dem Paketkopf gelöscht. Wenn noch Abhängigkeiten vom Paket bestehen, wird ein Fehler ausgegeben.

Wer kann ein Paket abgeben

Die DROP PACKAGE-Anweisung kann ausgeführt werden durch:

- Administratoren
- Der Besitzer des Pakets
- Benutzer mit der Berechtigung DROP ANY PACKAGE

Beispiel für DROP PACKAGE

Einen Paket-Header löschen

```
DROP PACKAGE APP_VAR
```

Siehe auch

CREATE PACKAGE, DROP PACKAGE BODY

5.11.5. RECREATE PACKAGE

Verwendet für

Erstellen eines neuen oder erneuten Erstellens eines vorhandenen Paketheaders

Verfügbar in

DSQL

Syntax

```
RECREATE PACKAGE package_name
AS
BEGIN
  [ <package_item> ... ]
END

!! Siehe auch Syntax CREATE PACKAGE für weitere Regeln!!
```

Die Anweisung RECREATE PACKAGE erstellt ein neues Paket oder erstellt einen vorhandenen Paket-Header neu. Wenn bereits ein Paketheader mit demselben Namen vorhanden ist, wird dieser durch diese Anweisung zuerst gelöscht und dann ein neuer Paketheader erstellt. Es ist nicht möglich, den Paketheader neu zu erstellen, wenn noch Abhängigkeiten von dem vorhandenen Paket bestehen oder wenn der Hauptteil des Pakets vorhanden ist. Bestehende Privilegien des Pakets selbst werden

nicht beibehalten, ebenso wenig Privilegien zum Ausführen der Prozeduren oder Funktionen des Pakets.

Beispiel für RECREATE PACKAGE

Erstellen eines neuen oder erneuten Erstellens eines vorhandenen Paketheaders

```
RECREATE PACKAGE APP_VAR
AS
BEGIN
  FUNCTION GET_DATEBEGIN() RETURNS DATE DETERMINISTIC;
  FUNCTION GET_DATEEND() RETURNS DATE DETERMINISTIC;
  PROCEDURE SET_DATERANGE(ADATEBEGIN DATE,
      ADATEEND DATE DEFAULT CURRENT_DATE);
END
```

Siehe auch

CREATE PACKAGE, DROP PACKAGE, CREATE PACKAGE BODY, RECREATE PACKAGE BODY

5.12. PACKAGE BODY

5.12.1. CREATE PACKAGE BODY

Verwendet für

Erstellen des Paketrumpfes

Verfügbar in

DSQL

Syntax

```
CREATE PACKAGE BODY name
AS
BEGIN
  [ <package_item> ... ]
  [ <package_body_item> ... ]
END

<package_item> ::=
  !! Siehe auch CREATE PACKAGE-Syntax !!

<package_body_item> ::=
  <function_impl> |
  <procedure_impl>

<function_impl> ::=
  FUNCTION funcname [ ( [ <in_params> ] ) ]
  RETURNS <domain_or_non_array_type> [COLLATE collation]
  [DETERMINISTIC]
```

```
  <module-body>

<procedure_impl> ::=
  PROCEDURE procname [ ( [ <in_params> ] ) ]
  [RETURNS (<out_params>)]
  <module-body>

<module-body> ::=
  !! Siehe auch Syntax des Modul-Bodys !!

<in_params> ::=
  !! Siehe auch CREATE PACKAGE-Syntax !!
  !! Siehe auch die Regeln weiter unten !!

<out_params> ::=
  !! Siehe auch CREATE PACKAGE-Syntax !!

<domain_or_non_array_type> ::=
  !! Siehe auch Syntax der Skalardatentypen !!
```

Tabelle 51. `CREATE PACKAGE BODY`-*Anweisungsparameter*

Parameter	Beschreibung
package_name	Paketname bestehend aus bis zu 31 Zeichen. Der Paketname muss unter allen Paketnamen eindeutig sein.
function_impl	Funktionsimplementierung. Im Wesentlichen eine `CREATE FUNCTION` -Anweisung ohne `CREATE`.
procedure_impl	Verfahrensimplementierung Im Wesentlichen eine `CREATE PROCEDURE` -Anweisung ohne `CREATE`.
func_name	Funktionsname bestehend aus bis zu 31 Zeichen. Der Funktionsname muss innerhalb des Pakets eindeutig sein.
collation	Sortierreihenfolge
proc_name	Prozedurname bestehend aus bis zu 31 Zeichen. Der Funktionsname muss innerhalb des Pakets eindeutig sein.

Die Anweisung `CREATE PACKAGE BODY` erstellt einen neuen Paketkörper. Der Paketkörper kann erst erstellt werden, nachdem der Paketkopf erstellt wurde. Wenn kein Paketheader mit dem Namen *package_name* vorhanden ist, wird ein entsprechender Fehler ausgegeben.

Alle im Paketkopf deklarierten Prozeduren und Funktionen müssen im Paketrumpf implementiert werden. Zusätzliche Prozeduren und Funktionen dürfen nur im Paketrumpf definiert und implementiert werden. Prozeduren und Funktionen, die im Paketrumpf definiert, aber nicht im Paketkopf definiert sind, sind außerhalb des Paketrumpfs nicht sichtbar.

Die Namen von Prozeduren und Funktionen, die im Paketrumpf definiert sind, müssen unter den Namen von Prozeduren und Funktionen, die im Paketkopf definiert und im Paketrumpf implementiert sind, eindeutig sein.

Paketprozedur- und Funktionsnamen können globale Routinen überschatten

Wenn ein Paketheader oder Paketrumpf eine Prozedur oder Funktion mit demselben Namen wie eine gespeicherte Prozedur oder Funktion im globalen Namespace deklariert, ist es nicht möglich, diese globale Prozedur oder Funktion aus dem Paketrumpf aufzurufen. In diesem Fall wird immer die Prozedur oder Funktion des Pakets aufgerufen.

Aus diesem Grund wird empfohlen, dass sich die Namen von gespeicherten Prozeduren und Funktionen in Paketen nicht mit Namen von gespeicherten Prozeduren und Funktionen im globalen Namespace überschneiden.

Regeln

- Im Paketrumpf müssen alle Prozeduren und Funktionen mit derselben Signatur implementiert werden, die im Header und am Anfang des Paketrumpfs deklariert ist
- Die Standardwerte für Prozedur- oder Funktionsparameter können nicht überschrieben werden (wie im Paketkopf oder in <*package_item*> angegeben). Dies bedeutet, dass Standardwerte nur in <*package_body_item*> für Prozeduren oder Funktionen definiert werden können, die nicht im Paketkopf oder früher im Paketrumpf definiert wurden.

UDF-Deklarationen (`DECLARE EXTERNAL FUNCTION`) werden für Pakete nicht unterstützt. Verwenden Sie stattdessen UDR.

Wer kann einen Paketkörper erstellen

Die Anweisung `CREATE PACKAGE BODY` kann ausgeführt werden durch:

- Administratoren
- Der Besitzer des Pakets
- Benutzer mit der Berechtigung `ALTER ANY PACKAGE`

Beispiel für `CREATE PACKAGE BODY`

Erstellen des Paketkörpers

```
CREATE PACKAGE BODY APP_VAR
AS
BEGIN
  - Gibt das Startdatum der Periode zurück
  FUNCTION GET_DATEBEGIN() RETURNS DATE DETERMINISTIC
  AS
  BEGIN
    RETURN RDB$GET_CONTEXT('USER_SESSION', 'DATEBEGIN');
  END
  - Gibt das Enddatum des Zeitraums zurück
  FUNCTION GET_DATEEND() RETURNS DATE DETERMINISTIC
  AS
  BEGIN
    RETURN RDB$GET_CONTEXT('USER_SESSION', 'DATEEND');
```

```
  END
  - Legt den Datumsbereich des Arbeitszeitraums fest
  PROCEDURE SET_DATERANGE(ADATEBEGIN DATE, ADATEEND DATE)
  AS
  BEGIN
    RDB$SET_CONTEXT('USER_SESSION', 'DATEBEGIN', ADATEBEGIN);
    RDB$SET_CONTEXT('USER_SESSION', 'DATEEND', ADATEEND);
  END
END
```

Siehe auch

`ALTER PACKAGE BODY`, `DROP PACKAGE BODY`, `RECREATE PACKAGE BODY`, `CREATE PACKAGE`

5.12.2. ALTER PACKAGE BODY

Verwendet für

Ändern des Paketrumpfes

Verfügbar in

DSQL

Syntax

```
ALTER PACKAGE BODY name
AS
BEGIN
  [ <package_item> ... ]
  [ <package_body_item> ... ]
END

!! Siehe auch Syntax CREATE PACKAGE BODY für weitere Regeln !!
```

Die Anweisung `ALTER PACKAGE BODY` modifiziert den Paketrumpf. Es kann verwendet werden, um die Definition und Implementierung von Prozeduren und Funktionen des Paketkörpers zu ändern.

Siehe `CREATE PACKAGE BODY` für weitere Details.

Wer kann einen Paketkörper ändern?

Die Anweisung `ALTER PACKAGE BODY` kann ausgeführt werden durch:

- Administratoren
- Der Besitzer des Pakets
- Benutzer mit der Berechtigung `ALTER ANY PACKAGE`

Beispiel für `ALTER PACKAGE BODY`

Ändern des Paketkörpers

```
ALTER PACKAGE BODY APP_VAR
AS
BEGIN
  - Gibt das Startdatum der Periode zurück
  FUNCTION GET_DATEBEGIN() RETURNS DATE DETERMINISTIC
  AS
  BEGIN
    RETURN RDB$GET_CONTEXT('USER_SESSION', 'DATEBEGIN');
  END
  - Gibt das Enddatum des Zeitraums zurück
  FUNCTION GET_DATEEND() RETURNS DATE DETERMINISTIC
  AS
  BEGIN
    RETURN RDB$GET_CONTEXT('USER_SESSION', 'DATEEND');
  END
  - Legt den Datumsbereich des Arbeitszeitraums fest
  PROCEDURE SET_DATERANGE(ADATEBEGIN DATE, ADATEEND DATE)
  AS
  BEGIN
    RDB$SET_CONTEXT('USER_SESSION', 'DATEBEGIN', ADATEBEGIN);
    RDB$SET_CONTEXT('USER_SESSION', 'DATEEND', ADATEEND);
  END
END
```

Siehe auch

`CREATE PACKAGE BODY`, `DROP PACKAGE BODY`, `RECREATE PACKAGE BODY`, `ALTER PACKAGE`

5.12.3. DROP PACKAGE BODY

Verwendet für

Löschen des Paketrumpfes

Verfügbar in

DSQL

Syntax

```
DROP PACKAGE package_name
```

Tabelle 52. `DROP PACKAGE BODY`-Anweisungsparameter

Parameter	Beschreibung
package_name	Paketname

Die Anweisung `DROP PACKAGE BODY` löscht den Paketkörper.

Wer kann einen Paketkörper fallen lassen?

Die DROP PACKAGE BODY-Anweisung kann ausgeführt werden durch:

- Administratoren
- Der Besitzer des Pakets
- Benutzer mit der Berechtigung ALTER ANY PACKAGE

Beispiel für DROP PACKAGE BODY

Löschen des Paketrumpfes

```
DROP PACKAGE BODY APP_VAR;
```

Siehe auch

CREATE PACKAGE BODY, ALTER PACKAGE BODY, DROP PACKAGE

5.12.4. RECREATE PACKAGE BODY

Verwendet für

Erstellen eines neuen oder erneuten Erstellens eines vorhandenen Paketrumpfes

Verfügbar in

DSQL

Syntax

```
RECREATE PACKAGE BODY name
AS
BEGIN
  [ <package_item> ... ]
  [ <package_body_item> ... ]
END

!! Siehe auch Syntax CREATE PACKAGE BODY für weitere Regeln !!
```

Die Anweisung RECREATE PACKAGE BODY erstellt einen neuen oder erstellt einen bestehenden Paketkörper neu. Wenn bereits ein Paketkörper mit demselben Namen vorhanden ist, versucht die Anweisung, ihn zu löschen und dann einen neuen Paketkörper zu erstellen. Nach der Neuerstellung des Paketkörpers bleiben die Berechtigungen des Pakets und seiner Routinen erhalten.

Siehe CREATE PACKAGE BODY für weitere Details.

Beispiele für RECREATE PACKAGE BODY

Neuerstellen des Paketrumpfes

```
RECREATE PACKAGE BODY APP_VAR
AS
BEGIN
  - Gibt das Startdatum der Periode zurück
  FUNCTION GET_DATEBEGIN() RETURNS DATE DETERMINISTIC
  AS
  BEGIN
    RETURN RDB$GET_CONTEXT('USER_SESSION', 'DATEBEGIN');
  END
  - Gibt das Enddatum des Zeitraums zurück
  FUNCTION GET_DATEEND() RETURNS DATE DETERMINISTIC
  AS
  BEGIN
    RETURN RDB$GET_CONTEXT('USER_SESSION', 'DATEEND');
  END
  - Legt den Datumsbereich des Arbeitszeitraums fest
  PROCEDURE SET_DATERANGE(ADATEBEGIN DATE, ADATEEND DATE)
  AS
  BEGIN
    RDB$SET_CONTEXT('USER_SESSION', 'DATEBEGIN', ADATEBEGIN);
    RDB$SET_CONTEXT('USER_SESSION', 'DATEEND', ADATEEND);
  END
END
```

Siehe auch

`CREATE PACKAGE BODY`, `ALTER PACKAGE BODY`, `DROP PACKAGE BODY`, `ALTER PACKAGE`

5.13. FILTER

Ein 'BLOB FILTER' ist ein Datenbankobjekt, das ein besonderer Typ einer externen Funktion ist, mit dem einzigen Zweck, ein 'BLOB'-Objekt in einem Format zu übernehmen und dieses in ein anderes Format umzuwandeln. Die Formate der `BLOB`-Objekte werden mit benutzerdefinierten `BLOB` -Subtypen spezifiziert.

Externe Funktionen zum Konvertieren von 'BLOB'-Typen werden in dynamischen Bibliotheken gespeichert und bei Bedarf geladen.

Weitere Informationen zu 'BLOB'-Subtypen finden Sie unter Binärdatentypen.

5.13.1. DECLARE FILTER

Verwendet für

Deklarieren eines 'BLOB'-Filters für die Datenbank

Verfügbar in

DSQL, ESQL

Syntax

```
DECLARE FILTER filtername
  INPUT_TYPE <sub_type> OUTPUT_TYPE <sub_type>
  ENTRY_POINT 'function_name' MODULE_NAME 'library_name'

<sub_type> ::= number | <mnemonic>

<mnemonic> ::=
    BINARY | TEXT | BLR | ACL | RANGES
  | SUMMARY | FORMAT | TRANSACTION_DESCRIPTION
  | EXTERNAL_FILE_DESCRIPTION | user_defined
```

Tabelle 53. DECLARE FILTER-Anweisungsparameter

Parameter	Beschreibung
filtername	Filtername in der Datenbank. Er kann aus bis zu 31 Zeichen bestehen. Es muss nicht der gleiche Name sein wie der Name, der über ENTRY_POINT aus der Filterbibliothek exportiert wurde.
sub_type	BLOB-Untertyp
number	BLOB-Untertypnummer (muss negativ sein)
mnemonic	'BLOB'-Untertyp mnemonischer Name`
function_name	Der exportierte Name (Einstiegspunkt) der Funktion
library_name	Der Name des Moduls, in dem sich der Filter befindet
user_defined	Benutzerdefinierter mnemonischer Name des 'BLOB'-Untertyps

Die Anweisung DECLARE FILTER stellt der Datenbank einen BLOB-Filter zur Verfügung. Der Name des 'BLOB'-Filters muss unter den Namen der 'BLOB'-Filter eindeutig sein.

Angeben der Untertypen

Die Untertypen können als Untertypnummer oder als Untertyp-Mnemonikname angegeben werden. Benutzerdefinierte Untertypen müssen durch negative Zahlen (von -1 bis -32.768) dargestellt werden. Ein Versuch, mehr als einen 'BLOB'-Filter mit derselben Kombination der Eingabe- und Ausgabetypen zu deklarieren, schlägt mit einem Fehler fehl.

INPUT_TYPE

Klausel, die den BLOB-Subtyp des zu konvertierenden Objekts definiert

OUTPUT_TYPE

-Klausel, die den 'BLOB'-Untertyp des zu erstellenden Objekts definiert.

Mnemonische Namen können für benutzerdefinierte 'BLOB'-Subtypen definiert und manuell in die Systemtabelle 'RDB$TYPES'-Systemtabelle eingefügt werden:

```
INSERT INTO RDB$TYPES (RDB$FIELD_NAME, RDB$TYPE, RDB$TYPE_NAME)
```

```
VALUES ('RDB$FIELD_SUB_TYPE', -33, 'MIDI');
```

Nachdem die Transaktion festgeschrieben wurde, können die mnemonischen Namen in Deklarationen verwendet werden, wenn Sie neue Filter erstellen.

Der Wert der Spalte RDB$FIELD_NAME muss immer 'RDB$FIELD_SUB_TYPE' sein. Wenn mnemonische Namen in Großbuchstaben definiert wurden, können sie bei der Deklaration eines Filters ohne Beachtung der Groß-/Kleinschreibung und ohne Anführungszeichen verwendet werden, wobei die Regeln für andere Objektnamen beachtet werden.

Warnung

Ab Firebird 3.0 können die Systemtabellen von Benutzern nicht mehr geschrieben werden. Das Einfügen von benutzerdefinierten Typen in RDB$TYPES ist jedoch weiterhin möglich. Firebird 4 führt ein Systemprivileg CREATE_USER_TYPES ein, um benutzerdefinierte Untertypen zu erstellen.

Parameter

ENTRY_POINT
: Klausel, die den Namen des Einstiegspunkts (den Namen der importierten Funktion) im Modul definiert.

MODULE_NAME
: Die Klausel, die den Namen des Moduls definiert, in dem sich die exportierte Funktion befindet. Standardmäßig müssen sich Module im UDF-Ordner des Stammverzeichnisses auf dem Server befinden. Der Parameter UDFAccess in firebird.conf ermöglicht das Bearbeiten von Zugriffsbeschränkungen auf Filterbibliotheken.

Jeder mit der Datenbank verbundene Benutzer kann einen BLOB-Filter deklarieren.

Wer kann einen 'BLOB'-Filter erstellen?

Die DECLARE FILTER-Anweisung kann ausgeführt werden durch:

- Administratoren
- Benutzer mit der Berechtigung CREATE FILTER

Der Benutzer, der die Anweisung DECLARE FILTER ausführt, wird Eigentümer des Filters.

Beispiele für DECLARE FILTER

1. Erstellen eines 'BLOB'-Filters mit Untertypnummern.

```
DECLARE FILTER DESC_FILTER
  INPUT_TYPE 1
  OUTPUT_TYPE -4
  ENTRY_POINT 'desc_filter'
```

```
  MODULE_NAME 'FILTERLIB';
```

2. Erstellen eines 'BLOB'-Filters unter Verwendung von mnemonischen Subtypnamen.

```
DECLARE FILTER FUNNEL
  INPUT_TYPE blr OUTPUT_TYPE text
  ENTRY_POINT 'blr2asc' MODULE_NAME 'myfilterlib';
```

Siehe auch

`DROP FILTER`

5.13.2. DROP FILTER

Verwendet für

Entfernen einer 'BLOB'-Filterdeklaration aus der Datenbank

Verfügbar in

DSQL, ESQL

Syntax

```
DROP FILTER filtername
```

Tabelle 54. `DROP FILTER`-Anweisungsparameter

Parameter	Beschreibung
filtername	Filtername in der Datenbank

Die `DROP FILTER`-Anweisung entfernt die Deklaration eines `BLOB`-Filters aus der Datenbank. Das Entfernen eines 'BLOB'-Filters aus einer Datenbank macht ihn für die Verwendung in dieser Datenbank nicht verfügbar. Die dynamische Bibliothek, in der sich die Konvertierungsfunktion befindet, bleibt intakt und das Entfernen aus einer Datenbank wirkt sich nicht auf andere Datenbanken aus, in denen noch derselbe 'BLOB'-Filter deklariert ist.

Wer kann einen 'BLOB'-Filter fallen lassen?

Die `DROP FILTER`-Anweisung kann ausgeführt werden durch:

- Administratoren
- Der Besitzer des Filters
- Benutzer mit der Berechtigung `DROP ANY FILTER`

`DROP FILTER`-Beispiel

Löschen eines 'BLOB'-Filters.

```
DROP FILTER DESC_FILTER;
```

Siehe auch

`DECLARE FILTER`

5.14. SEQUENCE (GENERATOR)

Eine Sequenz oder ein Generator ist ein Datenbankobjekt, das verwendet wird, um eindeutige Zahlenwerte zum Füllen einer Reihe zu erhalten. “Sequence” ist der SQL-konforme Begriff für dasselbe, was in Firebird traditionell als “Generator” bekannt war. Firebird hat Syntax für beide Begriffe.

Sequenzen (oder Generatoren) werden immer als 64-Bit-Ganzzahlen gespeichert, unabhängig vom SQL-Dialekt der Datenbank.

Wenn ein Client mit Dialekt 1 verbunden ist, sendet der Server Sequenzwerte als 32-Bit-Ganzzahlen an ihn. Die Übergabe eines Sequenzwerts an ein 32-Bit-Feld oder eine 32-Bit-Variable verursacht keine Fehler, solange der aktuelle Wert der Sequenz die Grenzen einer 32-Bit-Zahl nicht überschreitet. Sobald jedoch der Sequenzwert diese Grenze überschreitet, erzeugt eine Datenbank in Dialekt 3 einen Fehler. Eine Datenbank in Dialekt 1 schneidet die Werte ständig ab, was die Einzigartigkeit der Serie beeinträchtigt.

In diesem Abschnitt wird beschrieben, wie Sie Sequenzen erstellen, ändern, einstellen und löschen.

5.14.1. CREATE SEQUENCE

Verwendet für

Erstellen einer neuen `SEQUENCE` (`GENERATOR`)

Verfügbar in

DSQL, ESQL

Syntax

```
CREATE {SEQUENCE | GENERATOR} seq_name
  [START WITH start_value]
  [INCREMENT [BY] increment]
```

Tabelle 55. `CREATE SEQUENCE`-Anweisungsparameter

Parameter	Beschreibung
seq_name	Name der Sequenz (Generator). Diese kann aus bis zu 31 Zeichen bestehen
start_value	Anfangswert der Sequenz

Parameter	Beschreibung
increment	Erhöhen der Sequenz (bei Verwendung von `NEXT VALUE FOR seq_name`); kann nicht `0` sein

Die Anweisungen `CREATE SEQUENCE` und `CREATE GENERATOR` sind synonym – beide erzeugen eine neue Sequenz. Beide können verwendet werden, aber `CREATE SEQUENCE` wird empfohlen, da dies die im SQL-Standard definierte Syntax ist.

Wenn eine Sequenz erstellt wird, wird ihr Wert auf den Wert gesetzt, der in der Option `START WITH` -Klausel angegeben ist. Wenn keine `START WITH`-Klausel vorhanden ist, wird die Sequenz auf 0 gesetzt.

Mit der optionalen `INCREMENT [BY]`-Klausel können Sie ein Inkrement für den Ausdruck `NEXT VALUE FOR seq_name` angeben. Standardmäßig ist das Inkrement 1 (eins). Die Schrittweite kann nicht auf 0 (Null) gesetzt werden. Stattdessen kann die Funktion `GEN_ID(seq_name, <step>)` aufgerufen werden, um die Serie um eine andere ganze Zahl zu “step”. Das durch `INCREMENT [BY]` angegebene Inkrement wird nicht für `GEN_ID` verwendet.

Bug mit `START WITH` und `INCREMENT [BY]`

Der SQL-Standard gibt an, dass die `START WITH`-Klausel den Anfangswert angeben soll, der beim ersten Aufruf von `NEXT VALUE FOR seq_name` generiert wird, aber Firebird verwendet sie stattdessen, um den aktuellen Wert der Sequenz zu setzen. Als Ergebnis generiert der erste Aufruf von `NEXT VALUE FOR seq_name` fälschlicherweise den Wert `start_value + increment`.

Das Erstellen einer Sequenz ohne eine `START WITH`-Klausel entspricht derzeit der Angabe von `START WITH 0`, während es `START WITH 1` entsprechen sollte.

Dies wird in Firebird 4 behoben, siehe auch CORE-6084

Nicht standardmäßiges Verhalten bei negativen Inkrementen

Der SQL-Standard legt fest, dass Sequenzen mit negativem Inkrement beim Maximalwert der Sequenz (2^{63} - 1) beginnen und herunterzählen sollen. Firebird tut dies nicht und beginnt stattdessen bei `0 + Inkrement`.

Dies kann sich in einer zukünftigen Firebird-Version ändern.

Wer kann eine Sequenz erstellen?

Die Anweisung `CREATE SEQUENCE` (`CREATE GENERATOR`) kann ausgeführt werden durch:

- Administratoren
- Benutzer mit dem Privileg `CREATE SEQUENCE` (`CREATE GENERATOR`)

Der Benutzer, der die Anweisung `CREATE SEQUENCE` (`CREATE GENERATOR`) ausführt, wird ihr Eigentümer.

Beispiel für CREATE SEQUENCE

1. Erstellen der Sequenz EMP_NO_GEN mit CREATE SEQUENCE.

```
CREATE SEQUENCE EMP_NO_GEN;
```

2. Erstellen der Sequenz EMP_NO_GEN mit CREATE GENERATOR.

```
CREATE GENERATOR EMP_NO_GEN;
```

3. Erstellen der Sequenz EMP_NO_GEN mit einem Anfangswert von 5 und einem Inkrement von 1. Siehe auch Bug mit START WITH und INCREMENT [BY].

```
CREATE SEQUENCE EMP_NO_GEN START WITH 5;
```

4. Erstellen der Sequenz EMP_NO_GEN mit einem Anfangswert von 1 und einem Inkrement von 10. Siehe auch Bug mit START WITH und INCREMENT [BY].

```
CREATE SEQUENCE EMP_NO_GEN INCREMENT BY 10;
```

5. Erstellen der Sequenz EMP_NO_GEN mit einem Anfangswert von 5 und einem Inkrement von 10. Siehe auch Bug mit START WITH und INCREMENT [BY].

```
CREATE SEQUENCE EMP_NO_GEN START WITH 5 INCREMENT BY 10;
```

Siehe auch

ALTER SEQUENCE, CREATE OR ALTER SEQUENCE, DROP SEQUENCE, RECREATE SEQUENCE, SET GENERATOR, NEXT VALUE FOR, GEN_ID()-Funktion

5.14.2. ALTER SEQUENCE

Verwendet für

Den Wert einer Sequenz oder eines Generators auf einen bestimmten Wert setzen

Verfügbar in

DSQL

Syntax

```
ALTER {SEQUENCE | GENERATOR} seq_name
  [RESTART [WITH newvalue]]
  [INCREMENT [BY] increment]
```

Tabelle 56. ALTER SEQUENCE-*Anweisungsparameter*

Parameter	Beschreibung
seq_name	Name der Sequenz (Generator)
newvalue	Neuer Sequenz-(Generator-)Wert. Eine 64-Bit-Ganzzahl von -2^{63} bis $2^{63}-1$.
increment	Erhöhen der Sequenz (bei Verwendung von `NEXT VALUE FOR seq_name`); kann nicht `0` sein

Die `ALTER SEQUENCE`-Anweisung setzt den aktuellen Wert einer Sequenz oder eines Generators auf den angegebenen Wert und/oder ändert das Inkrement der Sequenz.

Mit der `RESTART WITH newvalue`-Klausel können Sie den Wert einer Sequenz festlegen. Die `RESTART` -Klausel (ohne `WITH`) startet die Sequenz mit dem Anfangswert neu, der mit der `START WITH`-Klausel konfiguriert wurde, als die Sequenz erstellt wurde.

Bugs mit `RESTART`

Der Anfangswert (entweder in den Metadaten gespeichert oder in der `WITH`-Klausel angegeben) wird verwendet, um den aktuellen Wert der Sequenz festzulegen, anstatt den nächsten Wert, der vom SQL-Standard gefordert wird, zu generieren. Siehe Hinweis Bug mit `START WITH` und `INCREMENT [BY]` für weitere Informationen.

Außerdem startet `RESTART WITH newvalue` nicht nur die Sequenz mit dem angegebenen Wert neu, sondern speichert auch *newvalue* als neuen Anfangswert der Sequenz. Dies bedeutet, dass ein nachfolgender `ALTER SEQUENCE RESTART` auch *newvalue* verwendet. Dieses Verhalten entspricht nicht dem im SQL-Standard angegebenen Verhalten.

Dieser Fehler wird in Firebird 4 behoben, siehe auch CORE-6386

Eine falsche Verwendung der `ALTER SEQUENCE`-Anweisung (Änderung des aktuellen Werts der Sequenz oder des Generators) kann wahrscheinlich die logische Integrität der Daten verletzen.

Mit `INCREMENT [BY]` können Sie das Sequenzinkrement für den `NEXT VALUE FOR`-Ausdruck ändern.

Das Ändern des Inkrementwerts wird für alle Abfragen wirksam, die nach dem Festschreiben der Transaktion ausgeführt werden. Prozeduren, die zum ersten Mal nach dem Ändern des Commits aufgerufen werden, verwenden den neuen Wert, wenn sie `NEXT VALUE FOR` verwenden. Prozeduren, die bereits verwendet (und im Metadaten-Cache zwischengespeichert wurden) verwenden weiterhin das alte Inkrement. Möglicherweise müssen Sie alle Verbindungen zur Datenbank schließen, damit der Metadatencache gelöscht und das neue Inkrement verwendet werden kann. Prozeduren, die `NEXT VALUE FOR` verwenden, müssen nicht neu kompiliert werden, um das neue Inkrement zu sehen. Prozeduren, die `GEN_ID(gen, expression)` verwenden, sind nicht betroffen, wenn das Inkrement geändert wird.

Wer kann eine Sequenz ändern?

Die Anweisung ALTER SEQUENCE (ALTER GENERATOR) kann ausgeführt werden durch:

- Administratoren
- Der Besitzer der Sequenz
- Benutzer mit dem Privileg ALTER ANY SEQUENCE (ALTER ANY GENERATOR)

Beispiele für ALTER SEQUENCE

1. Setzen des Werts der Sequenz EMP_NO_GEN auf 145.

```
ALTER SEQUENCE EMP_NO_GEN RESTART WITH 145;
```

2. Zurücksetzen des Basiswerts der Sequenz EMP_NO_GEN auf den in den Metadaten gespeicherten Initialwert

```
ALTER SEQUENCE EMP_NO_GEN RESTART;
```

3. Ändern der Schrittweite der Sequenz EMP_NO_GEN auf 10

```
ALTER SEQUENCE EMP_NO_GEN INCREMENT BY 10;
```

Siehe auch

SET GENERATOR, CREATE SEQUENCE, CREATE OR ALTER SEQUENCE, DROP SEQUENCE, RECREATE SEQUENCE, NEXT VALUE FOR, GEN_ID()-Funktion

5.14.3. CREATE OR ALTER SEQUENCE

Verwendet für

Erstellen einer neuen oder Ändern einer bestehenden Sequenz

Verfügbar in

DSQL, ESQL

Syntax

```
CREATE OR ALTER {SEQUENCE | GENERATOR} seq_name
  {RESTART | START WITH start_value}
  [INCREMENT [BY] increment]
```

Tabelle 57. CREATE OR ALTER SEQUENCE-Anweisungsparameter

Parameter	Beschreibung
seq_name	Name der Sequenz (Generator). Diese kann aus bis zu 31 Zeichen bestehen
start_value	Anfangswert der Sequenz
increment	Erhöhen der Sequenz (bei Verwendung von `NEXT VALUE FOR seq_name`); kann nicht `0` sein

Wenn die Sequenz nicht existiert, wird sie erstellt. Eine bestehende Sequenz wird geändert:

- Wenn `RESTART` angegeben ist, wird die Sequenz mit dem in den Metadaten gespeicherten Anfangswert neu gestartet
- Wenn die `START WITH`-Klausel angegeben ist, wird *start_value* als Initialwert in den Metadaten gespeichert und die Sequenz wird neu gestartet
- Wenn die `INCREMENT [BY]`-Klausel angegeben ist, wird *increment* als Inkrement in den Metadaten gespeichert und für nachfolgende Aufrufe von `NEXT VALUE FOR` verwendet

Beispiel für `SEQUENZ ERSTELLEN ODER ÄNDERN`

Erstelle eine neue oder modifiziere eine bestehende Sequenz `EMP_NO_GEN`

```
CREATE OR ALTER SEQUENCE EMP_NO_GEN
  START WITH 10
  INCREMENT BY 1
```

Siehe auch

`CREATE SEQUENCE`, `ALTER SEQUENCE`, `DROP SEQUENCE`, `RECREATE SEQUENCE`, `SET GENERATOR`, `NEXT VALUE FOR`, `GEN_ID()-Funktion`

5.14.4. DROP SEQUENCE

Verwendet für

Löschen einer Sequenz `SEQUENCE (GENERATOR)`

Verfügbar in

DSQL, ESQL

Syntax

```
DROP {SEQUENCE | GENERATOR} seq_name
```

Tabelle 58. `DROP SEQUENCE`*-Anweisungsparameter*

Parameter	Beschreibung
seq_name	Name der Sequenz (Generator). Diese kann aus bis zu 31 Zeichen bestehen

Die Anweisungen `DROP SEQUENCE` und `DROP GENERATOR` sind äquivalent: beide löschen eine existierende Sequenz (Generator). Beides ist gültig, aber `DROP SEQUENCE` wird empfohlen, da es im SQL-Standard definiert ist.

Die Anweisungen schlagen fehl, wenn die Sequenz (Generator) Abhängigkeiten hat.

Wer kann eine Sequenz löschen?

Die Anweisung `DROP SEQUENCE` (`DROP GENERATOR`) kann ausgeführt werden durch:

- Administratoren
- Der Besitzer der Sequenz
- Benutzer mit dem Privileg `DROP ANY SEQUENCE` (`DROP ANY GENERATOR`)

Beispiel für `DROP SEQUENCE`

Löschen der `EMP_NO_GEN`*-Reihe:*

```
DROP SEQUENCE EMP_NO_GEN;
```

Siehe auch

`CREATE SEQUENCE`, `CREATE OR ALTER SEQUENCE`, `RECREATE SEQUENCE`

5.14.5. `RECREATE SEQUENCE`

Verwendet für

Sequenz erstellen oder neu erstellen (Generator)

Verfügbar in

DSQL, ESQL

Syntax

```
RECREATE {SEQUENCE | GENERATOR} seq_name
  [START WITH start_value]
  [INCREMENT [BY] increment]
```

Tabelle 59. `RECREATE SEQUENCE`*-Anweisungsparameter*

Parameter	Beschreibung
seq_name	Name der Sequenz (Generator). Diese kann aus bis zu 31 Zeichen bestehen
start_value	Anfangswert der Sequenz
increment	Erhöhen der Sequenz (bei Verwendung von `NEXT VALUE FOR seq_name`); kann nicht `0` sein

Siehe `CREATE SEQUENCE` für die vollständige Syntax von `CREATE SEQUENCE` und Beschreibungen zur

Definition einer Sequenz und ihrer Optionen.

`RECREATE SEQUENCE` erstellt oder erstellt eine Sequenz neu. Existiert bereits eine Sequenz mit diesem Namen, versucht die `RECREATE SEQUENCE`-Anweisung, sie zu löschen und eine neue zu erstellen. Vorhandene Abhängigkeiten verhindern die Ausführung der Anweisung.

Beispiel für `RECREATE SEQUENCE`

Neuerstellen der Sequenz `EMP_NO_GEN`

```
RECREATE SEQUENCE EMP_NO_GEN
  START WITH 10
  INCREMENT BY 2;
```

Siehe auch

`CREATE SEQUENCE`, `ALTER SEQUENCE`, `CREATE OR ALTER SEQUENCE`, `DROP SEQUENCE`, `SET GENERATOR`, `NEXT VALUE FOR`, `GEN_ID()-Funktion`

5.14.6. SET GENERATOR

Verwendet für

Den Wert einer Sequenz oder eines Generators auf einen bestimmten Wert setzen

Verfügbar in

DSQL, ESQL

Syntax

```
SET GENERATOR seq_name TO new_val
```

Tabelle 60. `SET GENERATOR`*-Anweisungsparameter*

Parameter	Beschreibung
seq_name	Name des Generators (Sequenz)
new_val	Neuer Sequenz-(Generator-)Wert. Eine 64-Bit-Ganzzahl von -2^{63} bis $2^{63}-1$.

Die Anweisung `SET GENERATOR` setzt den aktuellen Wert einer Sequenz oder eines Generators auf den angegebenen Wert.

Obwohl `SET GENERATOR` als veraltet gilt, wird es aus Gründen der Abwärtskompatibilität beibehalten. Die Verwendung der standardkonformen `ALTER SEQUENCE` wird empfohlen.

Wer kann einen `SET GENERATOR` verwenden?

Die `SET GENERATOR`-Anweisung kann ausgeführt werden durch:

- Administratoren

- Der Besitzer der Sequenz (Generator)
- Benutzer mit dem Privileg `ALTER ANY SEQUENCE` (`ALTER ANY GENERATOR`)

Beispiel für `SET GENERATOR`

Wert der Sequenz `EMP_NO_GEN` *auf 145 setzen:*

```
SET GENERATOR EMP_NO_GEN TO 145;
```

Das gleiche kann mit `ALTER SEQUENCE` gemacht werden:

```
ALTER SEQUENCE EMP_NO_GEN RESTART WITH 145;
```

Siehe auch

`ALTER SEQUENCE`, `CREATE SEQUENCE`, `CREATE OR ALTER SEQUENCE`, `DROP SEQUENCE`, `NEXT VALUE FOR`, `GEN_ID()`-Funktion

5.15. EXCEPTION

In diesem Abschnitt wird beschrieben, wie Sie *benutzerdefinierte Ausnahmen* zur Verwendung in Fehlerhandlern in PSQL-Modulen erstellen, ändern und löschen.

5.15.1. CREATE EXCEPTION

Verwendet für

Erstellen einer neuen Ausnahme zur Verwendung in PSQL-Modulen

Verfügbar in

DSQL, ESQL

Syntax

```
CREATE EXCEPTION exception_name '<message>'

<message> ::= <message-part> [<message-part> ...]

<message-part> ::=
    <text>
  | @<slot>

<slot> ::= one of 1..9
```

Tabelle 61. `CREATE EXCEPTION`*-Anweisungsparameter*

Parameter	Beschreibung
exception_name	Ausnahmename. Die maximale Länge beträgt 31 Zeichen

Parameter	Beschreibung
message	Standardfehlermeldung. Die maximale Länge beträgt 1.021 Zeichen
text	Text beliebiger Zeichen
slot	Slotnummer eines Parameters. Die Nummerierung beginnt bei 1. Die maximale Steckplatznummer ist 9.

Die Anweisung `CREATE EXCEPTION` erzeugt eine neue Ausnahme zur Verwendung in PSQL-Modulen. Existiert eine gleichnamige Ausnahme, schlägt die Anweisung mit einer entsprechenden Fehlermeldung fehl.

Der Ausnahmename ist ein Standardbezeichner. In einer Dialect 3-Datenbank kann es in doppelte Anführungszeichen eingeschlossen werden, um die Groß-/Kleinschreibung zu berücksichtigen und bei Bedarf Zeichen zu verwenden, die in regulären Bezeichnern nicht gültig sind. Weitere Informationen finden Sie unter Bezeichner.

Die Standardnachricht wird im Zeichensatz `NONE` gespeichert, d.h. in Zeichen eines beliebigen Einzelbyte-Zeichensatzes. Der Text kann im PSQL-Code überschrieben werden, wenn die Ausnahme ausgelöst wird.

Die Fehlermeldung kann “Parameter-Slots” enthalten, die beim Auslösen der Ausnahme gefüllt werden können.

Wenn die *message* eine Parameter-Slot-Nummer enthält, die größer als 9 ist, werden die zweite und die nachfolgenden Ziffern als Literaltext behandelt. Zum Beispiel wird `@10` als Slot 1 interpretiert, gefolgt von einem Literal ‘0’.

Benutzerdefinierte Ausnahmen werden in der Systemtabelle `RDB$EXCEPTIONS` gespeichert.

Wer kann eine Ausnahme erstellen

Die `CREATE EXCEPTION`-Anweisung kann ausgeführt werden durch:

- Administratoren
- Benutzer mit dem Privileg `CREATE EXCEPTION`‘

Der Benutzer, der die Anweisung `CREATE EXCEPTION` ausführt, wird Eigentümer der Ausnahme.

`CREATE EXCEPTION`-Beispiele

Erstellen einer Ausnahme namens `E_LARGE_VALUE`

```
CREATE EXCEPTION E_LARGE_VALUE
  'The value is out of range';
```

Erstellen einer parametrisierten Ausnahme `E_INVALID_VALUE`

```
CREATE EXCEPTION E_INVALID_VALUE
  'Invalid value @1 for field @2';
```

Tips

Die Gruppierung von `CREATE EXCEPTION`-Anweisungen in Systemaktualisierungsskripten vereinfacht die Arbeit mit ihnen und deren Dokumentation. Ein System von Präfixen zur Benennung und Kategorisierung von Ausnahmegruppen wird empfohlen.

Siehe auch

`ALTER EXCEPTION`, `CREATE OR ALTER EXCEPTION`, `DROP EXCEPTION`, `RECREATE EXCEPTION`

5.15.2. ALTER EXCEPTION

Verwendet für

Ändern der von einer benutzerdefinierten Ausnahme zurückgegebenen Nachricht

Verfügbar in

DSQL, ESQL

Syntax

```
ALTER EXCEPTION exception_name '<message>'

!! Vgl. auch CREATE EXCEPTION für weitere Regeln !!
```

Die Anweisung `ALTER EXCEPTION` kann jederzeit verwendet werden, um den Standardtext der Nachricht zu ändern.

Wer kann eine Ausnahme ändern?

Die Anweisung `ALTER EXCEPTION` kann ausgeführt werden durch:

- Administratoren
- Der Inhaber der Ausnahme
- Benutzer mit der Berechtigung `ALTER ANY EXCEPTION`

`ALTER EXCEPTION`-Beispiele

Ändern der Standardnachricht für die Ausnahme `E_LARGE_VALUE`

```
ALTER EXCEPTION E_LARGE_VALUE
  'The value exceeds the prescribed limit of 32,765 bytes';
```

Siehe auch

CREATE EXCEPTION, CREATE OR ALTER EXCEPTION, DROP EXCEPTION, RECREATE EXCEPTION

5.15.3. CREATE OR ALTER EXCEPTION

Verwendet für

Ändern der von einer benutzerdefinierten Ausnahme zurückgegebenen Nachricht, falls die Ausnahme vorhanden ist; andernfalls eine neue Ausnahme erstellen

Verfügbar in

DSQL

Syntax

```
CREATE OR ALTER EXCEPTION exception_name '<message>'

!! Vgl. Syntax von CREATE EXCEPTION für weitere Regeln !!
```

Die Anweisung CREATE OR ALTER EXCEPTION wird verwendet, um die angegebene Ausnahme zu erstellen, falls sie nicht existiert, oder um den Text der von ihr zurückgegebenen Fehlermeldung zu ändern, wenn sie bereits existiert. Wenn eine bestehende Ausnahme durch diese Anweisung geändert wird, bleiben alle bestehenden Abhängigkeiten erhalten.

CREATE OR ALTER EXCEPTION-Beispiel

Ändern der Nachricht für die Ausnahme E_LARGE_VALUE

```
CREATE OR ALTER EXCEPTION E_LARGE_VALUE
  'The value is higher than the permitted range 0 to 32,765';
```

Siehe auch

CREATE EXCEPTION, ALTER EXCEPTION, RECREATE EXCEPTION

5.15.4. DROP EXCEPTION

Verwendet für

Löschen einer benutzerdefinierten Ausnahme

Verfügbar in

DSQL, ESQL

Syntax

```
DROP EXCEPTION exception_name
```

Tabelle 62. DROP EXCEPTION-*Anweisungsparameter*

Parameter	Beschreibung
exception_name	Exception name

Die Anweisung `DROP EXCEPTION` dient zum Löschen einer Ausnahme. Alle Abhängigkeiten von der Ausnahme führen dazu, dass die Anweisung fehlschlägt und die Ausnahme nicht gelöscht wird.

Wer kann eine Ausnahme fallen lassen?

Die `DROP EXCEPTION`-Anweisung kann ausgeführt werden durch:

- Administratoren
- Der Inhaber der Ausnahme
- Benutzer mit dem Privileg `DROP ANY EXCEPTION`

`DROP EXCEPTION`-Beispiele

Ausnahme `E_LARGE_VALUE` *wird gelöscht*

```
DROP EXCEPTION E_LARGE_VALUE;
```

Siehe auch

`CREATE EXCEPTION`, `RECREATE EXCEPTION`

5.15.5. `RECREATE EXCEPTION`

Verwendet für

Erstellen einer neuen benutzerdefinierten Ausnahme oder Neuerstellen einer vorhandenen Ausnahme

Verfügbar in

DSQL

Syntax

```
RECREATE EXCEPTION exception_name '<message>'

!! Vgl. Syntax CREATE EXCEPTION für weitere Regeln !!
```

Die Anweisung `RECREATE EXCEPTION` erzeugt eine neue Ausnahme zur Verwendung in PSQL-Modulen. Wenn bereits eine Ausnahme mit demselben Namen existiert, versucht die Anweisung `RECREATE EXCEPTION`, sie zu löschen und eine neue zu erstellen. Wenn Abhängigkeiten von der bestehenden Ausnahme bestehen, schlägt der Löschversuch fehl und `RECREATE EXCEPTION` wird nicht ausgeführt.

`RECREATE EXCEPTION`-Beispiel

Ausnahme `E_LARGE_VALUE` *neu erstellen*

```
RECREATE EXCEPTION E_LARGE_VALUE
  'The value exceeds its limit';
```

Siehe auch

`CREATE EXCEPTION`, `DROP EXCEPTION`, `CREATE OR ALTER EXCEPTION`

5.16. COLLATION

In SQL sind Textzeichenfolgen sortierbare Objekte. Das bedeutet, dass sie Ordnungsregeln wie die alphabetische Reihenfolge befolgen. Vergleichsoperationen können auf solche Textzeichenfolgen angewendet werden (z. B. “kleiner als” oder “größer als”), bei denen der Vergleich eine bestimmte Sortierreihenfolge oder Kollatierung anwenden muss. Der Ausdruck “`'a' < 'b'`” bedeutet beispielsweise, dass ‘`'a'`’ in der Kollatierung vor ‘`'b'`’ steht. Der Ausdruck “`'c' > 'b'`” bedeutet, dass ‘`'c'`’ in der Kollation auf ‘`'b'`’ folgt. Textstrings mit mehr als einem Zeichen werden durch sequentielle Zeichenvergleiche sortiert: Zuerst werden die ersten Zeichen der beiden Strings verglichen, dann die zweiten Zeichen usw., bis ein Unterschied zwischen den beiden Strings gefunden wird. Dieser Unterschied definiert die Sortierreihenfolge.

Eine `COLLATION` ist das Schemaobjekt, das eine Kollatierung (oder Sortierreihenfolge) definiert.

5.16.1. CREATE COLLATION

Verwendet für

Erstellen einer neuen Kollatierung für einen unterstützten Zeichensatz, der für die Datenbank verfügbar ist

Verfügbar in

DSQL

Syntax

```
CREATE COLLATION collname
    FOR charset
    [FROM {basecoll | EXTERNAL ('extname')}]
    [NO PAD | PAD SPACE]
    [CASE [IN]SENSITIVE]
    [ACCENT [IN]SENSITIVE]
    ['<specific-attributes>']

<specific-attributes> ::= <attribute> [; <attribute> ...]

<attribute> ::= attrname=attrvalue
```

Tabelle 63. `CREATE COLLATION`*-Anweisungsparameter*

Parameter	Beschreibung
collname	Der für die neue Sortierung zu verwendende Name. Die maximale Länge beträgt 31 Zeichen
charset	Ein in der Datenbank vorhandener Zeichensatz
basecoll	Eine Kollation, die bereits in der Datenbank vorhanden ist
extname	Der in der Datei `.conf` verwendete Kollatierungsname

Die `CREATE COLLATION`-Anweisung "erzeugt" nichts, ihr Zweck besteht darin, einer Datenbank eine Kollation bekannt zu machen. Die Kollatierung muss bereits auf dem System vorhanden sein, normalerweise in einer Bibliotheksdatei, und muss ordnungsgemäß in einer `.conf`-Datei im `intl` -Unterverzeichnis der Firebird-Installation registriert sein.

Die Kollation kann alternativ auf einer basieren, die bereits in der Datenbank vorhanden ist.

Wie die Engine die Kollation erkennt

Die optionale `FROM'-Klausel gibt die Basiskollation an, die verwendet wird, um eine neue Kollation abzuleiten. Diese Kollation muss bereits in der Datenbank vorhanden sein. Wenn das Schlüsselwort `EXTERNAL` angegeben wird, scannt Firebird die `.conf`-Dateien in `$fbroot/intl/`, wobei *extname* genau mit dem Namen in der Konfigurationsdatei übereinstimmen muss (Groß-/Kleinschreibung beachten).

Wenn keine `FROM`-Klausel vorhanden ist, durchsucht Firebird die `.conf`-Datei(en) im `intl` -Unterverzeichnis nach einer Kollatierung mit dem in `CREATE COLLATION` angegebenen Kollatierungsnamen. Mit anderen Worten, das Weglassen der `FROM basecoll`-Klausel entspricht der Angabe von `FROM EXTERNAL ('collname')`.

Beim — in einfachen Anführungszeichen angegebenen — *extname* muss die Groß-/Kleinschreibung beachtet werden und genau mit dem Kollatierungsnamen in der Datei `.conf` übereinstimmen. Bei den Parametern *collname*, *charset* und *basecoll* wird die Groß-/Kleinschreibung nicht beachtet, es sei denn, sie stehen in doppelten Anführungszeichen.

Beim Erstellen einer Sortierung können Sie angeben, ob nachgestellte Leerzeichen in den Vergleich einbezogen werden. Bei Angabe der `NO PAD`-Klausel werden abschließende Leerzeichen beim Vergleich berücksichtigt. Wenn die `PAD SPACE`-Klausel angegeben ist, werden nachgestellte Leerzeichen beim Vergleich ignoriert.

Mit der optionalen `CASE`-Klausel können Sie angeben, ob beim Vergleich die Groß-/Kleinschreibung beachtet wird oder nicht.

Mit der optionalen `ACCENT`-Klausel können Sie angeben, ob der Vergleich akzentsensitiv oder akzentunabhängig ist (zB wenn ‘`'e'`’ und ‘`'é'`’ als gleich betrachtet werden oder ungleich).

Spezifische Attribute

Die `CREATE COLLATION`-Anweisung kann auch spezielle Attribute enthalten, um die Kollatierung zu konfigurieren. Die verfügbaren spezifischen Attribute sind in der folgenden Tabelle aufgeführt. Nicht alle spezifischen Attribute gelten für jede Sortierung. Wenn das Attribut nicht auf die

Sortierung anwendbar ist, aber beim Erstellen angegeben wird, wird kein Fehler verursacht.

Bei spezifische Attributnamen muss die Groß-/Kleinschreibung beachtet werden.

In der Tabelle gibt “1 bpc” an, dass ein Attribut für Kollationen von Zeichensätzen mit 1 Byte pro Zeichen gültig ist (sogenannte schmale Zeichensätze), und “UNI” für “Unicode Collations”.

Tabelle 64. Spezifische Sortierattribute

Atrribute	Werte	Gültig für	Hinweis
DISABLE-COMPRESSIONS	0, 1	1 bpc	Deaktiviert Kompressionen (auch bekannt als Kontraktionen). Kompressionen bewirken, dass bestimmte Zeichenfolgen als atomare Einheiten sortiert werden, z.B. Spanisch c+h als einzelnes Zeichen ch
DISABLE-EXPANSIONS	0, 1	1 bpc	Deaktiviert Erweiterungen. Erweiterungen bewirken, dass bestimmte Zeichen (z.B. Ligaturen oder umlaute Vokale) als Zeichenfolgen behandelt und entsprechend sortiert werden
ICU-VERSION	Standard oder M.m	UNI	Gibt die zu verwendende ICU-Bibliotheksversion an. Gültige Werte sind diejenigen, die im entsprechenden <intl_module>-Element in intl/fbintl.conf definiert sind. Format: entweder das Stringliteral “default” oder eine Major+Minor Versionsnummer wie “3.0” (beide ohne Anführungszeichen).
LOCALE	xx_YY	UNI	Gibt das Kollatierungsgebietsschema an. Erfordert eine vollständige Version der ICU-Bibliotheken. Format: ein Gebietsschema-String wie “du_NL” (ohne Anführungszeichen)
MULTI-LEVEL	0, 1	1 bpc	Verwendet mehr als eine Sortierebene
NUMERIC-SORT	0, 1	UNI	Behandelt zusammenhängende Gruppen von Dezimalziffern in der Zeichenfolge als atomare Einheiten und sortiert sie numerisch. (Dies wird auch als natürliche Sortierung bezeichnet)
SPECIALS-FIRST	0, 1	1 bpc	Ordnet Sonderzeichen (Leerzeichen, Symbole usw.) vor alphanumerischen Zeichen an

Wenn Sie Ihrer Datenbank einen neuen Zeichensatz mit seiner Standardsortierung hinzufügen möchten, deklarieren Sie die gespeicherte Prozedur

> `sp_register_character_set(name, max_bytes_per_character)`, die sich in `misc/intl.sql` im Firebird-Installationsverzeichnis befindet, und führen Sie sie aus.
>
> Damit dies funktioniert, muss der Zeichensatz auf dem System vorhanden und in einer `.conf`-Datei im `intl`-Unterverzeichnis registriert sein.

Wer kann eine Kollation erstellen

Die `CREATE COLLATION`-Anweisung kann ausgeführt werden durch:

- Administratoren
- Benutzer mit dem Privileg `CREATE COLLATION`ATION

Der Benutzer, der die `CREATE COLLATION`-Anweisung ausführt, wird Eigentümer der Kollation.

Beispiele zur Nutzung von CREATE COLLATION

1. Erstellen einer Kollatierung mit dem Namen aus der Datei `fbintl.conf` (Groß-/Kleinschreibung beachten)

```
CREATE COLLATION ISO8859_1_UNICODE FOR ISO8859_1;
```

2. Erstellen einer Kollatierung unter Verwendung eines speziellen (benutzerdefinierten) Namens (der “externe” Name muss vollständig mit dem Namen in der Datei `fbintl.conf` übereinstimmen)

```
CREATE COLLATION LAT_UNI
  FOR ISO8859_1
  FROM EXTERNAL ('ISO8859_1_UNICODE');
```

3. Erstellen einer Sortierung ohne Beachtung der Groß-/Kleinschreibung basierend auf einer bereits in der Datenbank vorhandenen.

```
CREATE COLLATION ES_ES_NOPAD_CI
  FOR ISO8859_1
  FROM ES_ES
  NO PAD
  CASE INSENSITIVE;
```

4. Erstellen einer Sortierung ohne Beachtung der Groß-/Kleinschreibung basierend auf einer bereits in der Datenbank vorhandenen Sortierung mit bestimmten Attributen

```
CREATE COLLATION ES_ES_CI_COMPR
  FOR ISO8859_1
  FROM ES_ES
  CASE INSENSITIVE
```

```
  'DISABLE-COMPRESSIONS=0';
```

5. Erstellen einer Sortierung ohne Beachtung der Groß-/Kleinschreibung nach dem Wert von Zahlen (die sogenannte natürliche Sortierung)

```
CREATE COLLATION nums_coll FOR UTF8
  FROM UNICODE
  CASE INSENSITIVE 'NUMERIC-SORT=1';

CREATE DOMAIN dm_nums AS varchar(20)
  CHARACTER SET UTF8 COLLATE nums_coll; -- Original-(Hersteller-)Nummern

CREATE TABLE wares(id int primary key, articul dm_nums ...);
```

Siehe auch

`DROP COLLATION`

5.16.2. DROP COLLATION

Verwendet für

Eine Kollation aus der Datenbank entfernen

Verfügbar in

DSQL

Syntax

```
DROP COLLATION collname
```

Tabelle 65. `DROP COLLATION`-Anweisungsparameter

Parameter	Beschreibung
collname	Der Name der Kollation

Die Anweisung `DROP COLLATION` entfernt die angegebene Kollatierung aus der Datenbank, falls vorhanden. Wenn die angegebene Sortierung nicht vorhanden ist, wird ein Fehler ausgegeben.

Wenn Sie einen ganzen Zeichensatz mit all seinen Kollatierungen aus der Datenbank entfernen möchten, deklarieren Sie die gespeicherte Prozedur `sp_unregister_character_set(name)` aus dem `misc/intl.sql`-Unterverzeichnis der Firebird-Installation und führen Sie sie aus.

Wer kann eine Kollation abgeben

Die `Drop COLLATION`-Anweisung kann ausgeführt werden durch:

- Administratoren

- Der Besitzer der Kollation
- Benutzer mit dem Privileg `DROP ANY COLLATION`

Beispiele für `DROP COLLATION`

Löschen der Kollatierung `ES_ES_NOPAD_CI`.

```
DROP COLLATION ES_ES_NOPAD_CI;
```

Siehe auch
`CREATE COLLATION`

5.17. CHARACTER SET

5.17.1. ALTER CHARACTER SET

Verwendet für

Festlegen der Standardsortierung für einen Zeichensatz

Verfügbar in

DSQL

Syntax

```
ALTER CHARACTER SET charset
  SET DEFAULT COLLATION collation
```

Tabelle 66. `ALTER CHARACTER SET`*-Anweisungsparameter*

Parameter	Beschreibung
charset	Zeichensatzkennung
collation	Der Name der Kollation

Die Anweisung `ALTER CHARACTER SET` ändert die Standardsortierung für den angegebenen Zeichensatz. Sie wirkt sich auf die zukünftige Verwendung des Zeichensatzes aus, außer in Fällen, in denen die `COLLATE`-Klausel explizit überschrieben wird. In diesem Fall bleibt die Kollatierungsreihenfolge bestehender Domänen, Spalten und PSQL-Variablen nach der Änderung der Standardkollation des zugrunde liegenden Zeichensatzes erhalten.

Wenn Sie die Standardsortierung für den Datenbankzeichensatz ändern (der beim Erstellen der Datenbank definierte), wird die Standardsortierung für die Datenbank geändert.

Wenn Sie die Standardsortierung für den während der Verbindung angegebenen Zeichensatz ändern, werden Zeichenfolgenkonstanten gemäß dem neuen Sortierungswert interpretiert, außer in den Fällen, in denen der Zeichensatz und/oder die Sortierung überschrieben wurden.

Wer kann einen Zeichensatz ändern?

Die Anweisung `ALTER CHARACTER SET` kann ausgeführt werden durch:

- Administratoren
- Benutzer mit der Berechtigung `ALTER ANY CHARACTER SET`

`ALTER CHARACTER SET`-Beispiel

Festlegen der Standardsortierung `UNICODE_CI_AI` *für die* `UTF8`*-Codierung*

```
ALTER CHARACTER SET UTF8
  SET DEFAULT COLLATION UNICODE_CI_AI;
```

5.18. COMMENT

Datenbankobjekte und eine Datenbank selbst können mit Kommentaren versehen werden. Es ist ein bequemer Mechanismus, um die Entwicklung und Wartung einer Datenbank zu dokumentieren. Kommentare, die mit `COMMENT ON` erstellt wurden, überleben eine *gbak*-Sicherung und -Wiederherstellung.

5.18.1. `COMMENT ON`

Verwendet für

Metadaten dokumentieren

Verfügbar in

DSQL

Syntax

```
COMMENT ON <object> IS {'sometext' | NULL}

<object> ::=
    {DATABASE | SCHEMA}
  | <basic-type> objectname
  | COLUMN relationname.fieldname
  | [{PROCEDURE | FUNCTION}] PARAMETER
      [packagename.]routinename.paramname
  | {PROCEDURE | [EXTERNAL] FUNCTION}
      [package_name.]routinename

<basic-type> ::=
    CHARACTER SET | COLLATION | DOMAIN
  | EXCEPTION     | FILTER    | GENERATOR
  | INDEX         | PACKAGE   | ROLE
  | SEQUENCE      | TABLE     | TRIGGER
  | USER          | VIEW
```

Tabelle 67. COMMENT ON-Anweisungsparameter

Parameter	Beschreibung
sometext	Kommentartext
basic-type	Metadaten-Objekttyp
objectname	Name des Metadatenobjekts
relationname	Name der Tabelle oder Ansicht
fieldname	Name der Spalte
package_name	Name des Pakets
routinename	Name der gespeicherten Prozedur oder Funktion
paramname	Name einer gespeicherten Prozedur oder eines Funktionsparameters

Die Anweisung COMMENT ON fügt Kommentare für Datenbankobjekte (Metadaten) hinzu. Kommentare werden in der Spalte RDB$DESCRIPTION der entsprechenden Systemtabellen gespeichert. Clientanwendungen können Kommentare aus diesen Feldern anzeigen.

1. Wenn Sie einen leeren Kommentar hinzufügen (“''”), wird dieser als NULL in der Datenbank gespeichert.
2. Die COMMENT ON USER-Anweisung erstellt nur Kommentare zu Benutzern, die vom Standardbenutzermanager verwaltet werden (das erste Plugin, das in der Konfigurationsoption UserManager aufgeführt ist). Siehe auch CORE-6479.
3. Kommentare zu Benutzern werden in der Sicherheitsdatenbank gespeichert.
4. SCHEMA ist derzeit ein Synonym für DATABASE; dies kann sich in einer zukünftigen Version ändern, daher empfehlen wir immer DATABASE zu verwenden

Kommentare zu Benutzern sind für diesen Benutzer über die virtuelle Tabelle SEC$USERS sichtbar.

Wer kann einen Kommentar hinzufügen

Die COMMENT ON-Anweisung kann ausgeführt werden durch:

- Administratoren
- Der Besitzer des Objekts, das kommentiert wird
- Benutzer mit der Berechtigung ALTER ANY object_type, wobei *object_type* der Typ des kommentierten Objekts ist (z. B. PROCEDURE)

Beispiele mit COMMENT ON

1. Kommentar zur aktuellen Datenbank hinzufügen

```
COMMENT ON DATABASE IS 'It is a test (''my.fdb'') database';
```

2. Kommentar für die Tabelle `METALS` hinzufügen

```
COMMENT ON TABLE METALS IS 'Metal directory';
```

3. Hinzufügen eines Kommentars für das Feld `ISALLOY` in der Tabelle `METALS`

```
COMMENT ON COLUMN METALS.ISALLOY IS '0 = fine metal, 1 = alloy';
```

4. Kommentar zu einem Parameter hinzufügen

```
COMMENT ON PARAMETER ADD_EMP_PROJ.EMP_NO IS 'Employee ID';
```

5. Hinzufügen eines Kommentars für ein Paket, seine Prozeduren und Funktionen sowie deren Parameter

```
COMMENT ON PACKAGE APP_VAR IS 'Application Variables';

COMMENT ON FUNCTION APP_VAR.GET_DATEBEGIN
IS 'Returns the start date of the period';

COMMENT ON PROCEDURE APP_VAR.SET_DATERANGE
IS 'Set date range';

COMMENT ON
PROCEDURE PARAMETER APP_VAR.SET_DATERANGE.ADATEBEGIN
IS 'Start Date';
```

Kapitel 6. Data Manipulation-Statements (DML)

DML – Data Manipulation Language – ist die Teilmenge von SQL, die von Anwendungen und prozeduralen Modulen verwendet wird, um Daten zu extrahieren und zu ändern. Die Extraktion zum Lesen von Daten, sowohl roh als auch manipuliert, wird mit der `SELECT`-Anweisung erreicht. `INSERT` dient zum Hinzufügen neuer Daten und `DELETE` zum Löschen nicht mehr benötigter Daten. `UPDATE`, `MERGE` und `UPDATE OR INSERT` ändern alle Daten auf verschiedene Weise.

6.1. SELECT

Verwendet für

Abfrage von Daten

Verfügbar in

DSQL, ESQL, PSQL

Global syntax

```
[WITH [RECURSIVE] <cte> [, <cte> ...]]
SELECT
  [FIRST m] [SKIP n]
  [{DISTINCT | ALL}] <columns>
FROM
  <source> [[AS] alias]
  [<joins>]
[WHERE <condition>]
[GROUP BY <grouping-list>
[HAVING <aggregate-condition>]]
[PLAN <plan-expr>]
[UNION [{DISTINCT | ALL}] <other-select>]
[ORDER BY <ordering-list>]
[{ ROWS <m> [TO <n>]
 | [OFFSET n {ROW | ROWS}]
   [FETCH {FIRST | NEXT} [m] {ROW | ROWS} ONLY]
}]
[FOR UPDATE [OF <columns>]]
[WITH LOCK]
[INTO <variables>]

<variables> ::= [:]varname [, [:]varname ...]
```

Die `SELECT`-Anweisung ruft Daten aus der Datenbank ab und übergibt sie an die Anwendung oder die einschließende SQL-Anweisung. Daten werden in null oder mehr *rows* zurückgegeben, die jeweils eine oder mehrere *columns* oder *fields* enthalten. Die Summe der zurückgegebenen Zeilen ist der *result set* der Anweisung.

Die einzigen obligatorischen Teile der SELECT-Anweisung sind:

- Das Schlüsselwort SELECT, gefolgt von einer Spaltenliste. Dieser Teil gibt an, *was* Sie abrufen möchten.
- Das Schlüsselwort FROM, gefolgt von einem auswählbaren Objekt. Dies teilt der Engine mit, *wo* Sie sie *von* beziehen möchten.

In seiner einfachsten Form ruft SELECT eine Reihe von Spalten aus einer einzelnen Tabelle oder Ansicht ab, wie folgt:

```
select id, name, address
  from contacts
```

Oder, um alle Spalten abzurufen:

```
select * from sales
```

In der Praxis wird eine SELECT-Anweisung normalerweise mit einer WHERE-Klausel ausgeführt, die die zurückgegebenen Zeilen begrenzt. Die Ergebnismenge kann durch eine ORDER BY-Klausel sortiert werden, und FIRST … SKIP, OFFSET … FETCH oder ROWS können die Anzahl der zurückgegebenen Zeilen weiter begrenzen und können - zum Beispiel - sein für die Paginierung verwendet.

Die Spaltenliste kann statt nur Spaltennamen alle Arten von Ausdrücken enthalten, und die Quelle muss keine Tabelle oder Sicht sein: Sie kann auch eine abgeleitete Tabelle, ein allgemeiner Tabellenausdruck (CTE) oder eine auswählbare gespeicherte Prozedur (SP) sein. Mehrere Quellen können in einem JOIN kombiniert werden und mehrere Ergebnismengen können in einer UNION kombiniert werden.

Die folgenden Abschnitte behandeln die verfügbaren SELECT-Unterklauseln und ihre Verwendung im Detail.

6.1.1. FIRST, SKIP

Verwendet für

Abrufen eines Zeilenabschnitts aus einer geordneten Menge

Verfügbar in

DSQL, PSQL

Syntax

```
SELECT
  [FIRST <m>] [SKIP <n>]
  FROM ...
  ...

<m>, <n>  ::=
```

```
    <integer-literal>
  | <query-parameter>
  | (<integer-expression>)
```

Tabelle 68. Argumente für die Klauseln `FIRST` *und* `SKIP`

Argument	Beschreibung
integer-literal	Ganzzahlliteral
query-parameter	Platzhalter für Abfrageparameter. ? in DSQL und `:paramname` in PSQL
integer-expression	Ausdruck, der einen ganzzahligen Wert zurückgibt

`FIRST` und `SKIP` sind Nicht-Standard-Syntax

`FIRST` und `SKIP` sind Firebird-spezifische Klauseln. Verwenden Sie nach Möglichkeit die SQL-Standardsyntax `OFFSET, FETCH`.

`FIRST` begrenzt die Ausgabe einer Abfrage auf die ersten *m* Zeilen. `SKIP` unterdrückt die angegebenen *n* Zeilen, bevor die Ausgabe gestartet wird.

`FIRST` und `SKIP` sind beide optional. Bei gemeinsamer Verwendung wie in "`FIRST m SKIP n`" werden die *n* obersten Zeilen der Ausgabemenge verworfen und die ersten *m* Zeilen der restlichen Menge zurückgegeben.

Eigenschaften von `FIRST` und `SKIP`

- Jedes Argument für `FIRST` und `SKIP`, das kein Integer-Literal oder ein SQL-Parameter ist, muss in Klammern eingeschlossen werden. Dies impliziert, dass ein Unterabfrageausdruck in *zwei* Klammerpaare eingeschlossen werden muss.
- `SKIP 0` ist erlaubt, aber völlig sinnlos.
- `FIRST 0` ist ebenfalls erlaubt und gibt eine leere Menge zurück.
- Negative `SKIP` und/oder `FIRST` Werte führen zu einem Fehler.
- Wenn ein `SKIP` nach dem Ende des Datensatzes landet, wird ein leerer Satz zurückgegeben.
- Wenn die Anzahl der Zeilen im Datensatz (oder der Rest nach einem `SKIP`) kleiner ist als der Wert des *m*-Arguments, das für `FIRST` bereitgestellt wurde, wird diese kleinere Anzahl von Zeilen zurückgegeben. Dies sind gültige Ergebnisse, keine Fehlerbedingungen.

Beispiele für FIRST/SKIP

1. Die folgende Abfrage gibt die ersten 10 Namen aus der Tabelle "People" zurück:

```
select first 10 id, name from People
  order by name asc
```

2. Die folgende Abfrage gibt alles zurück, *aber* die ersten 10 Namen:

```
select skip 10 id, name from People
  order by name asc
```

3. Und dieser gibt die letzten 10 Zeilen zurück. Beachten Sie die doppelten Klammern:

```
select skip ((select count(*) - 10 from People))
  id, name from People
  order by name asc
```

4. Diese Abfrage gibt die Zeilen 81 bis 100 der People-Tabelle zurück:

```
select first 20 skip 80 id, name from People
  order by name asc
```

Siehe auch

OFFSET, FETCH, ROWS

6.1.2. Die SELECT-Spaltenliste

Die Spaltenliste enthält einen oder mehrere durch Kommas getrennte Wertausdrücke. Jeder Ausdruck stellt einen Wert für eine Ausgabespalte bereit. Alternativ kann * (“Hole Sternchen” oder “Hole alle”) verwendet werden, um für alle Spalten in einer Relation (d.h. einer Tabelle, View oder auswählbaren Stored Procedure) zu stehen. .Syntax

```
SELECT
  [...]
  [{DISTINCT | ALL}] <output-column> [, <output-column> ...]
  [...]
  FROM ...

<output-column> ::=
  { [<qualifier>.]*
  | <value-expression> [COLLATE collation] [[AS] alias] }

<value-expression> ::=
  { [<qualifier>.]table-column
  | [<qualifier>.]view-column
  | [<qualifier>.]selectable-SP-outparm
  | <literal>
  | <context-variable>
  | <function-call>
  | <single-value-subselect>
  | <CASE-construct>
  | any other expression returning a single
    value of a Firebird data type or NULL }
```

```
<qualifier> ::= a relation name or alias
```

Tabelle 69. Argumente für die `SELECT`*-Spaltenliste*

Argument	Beschreibung
qualifier	Name der Relation (View, Stored Procedure, abgeleitete Tabelle); oder ein Alias dafür
collation	Nur für zeichenartige Spalten: ein vorhandener und für den Zeichensatz der Daten gültiger Kollatierungsname
alias	Spalten- oder Feldalias
table-column	Name einer Tabellenspalte
view-column	Name einer Ansichtsspalte
selectable-SP-outparm	Deklarierter Name eines Ausgabeparameters einer auswählbaren gespeicherten Prozedur
constant	Eine Konstante
context-variable	Kontextvariable
function-call	Skalar-, Aggregat- oder Fensterfunktionsausdruck
single-value-subselect	Eine Unterabfrage, die einen Skalarwert zurückgibt (Singleton)
CASE-construct	CASE-Konstrukt, das Bedingungen für einen Rückgabewert setzt
other-single-value-expr	Jeder andere Ausdruck, der einen einzelnen Wert eines Firebird-Datentyps zurückgibt; oder NULL

Es ist immer gültig, einen Spaltennamen (oder "*") mit dem Namen oder Alias der Tabelle, Ansicht oder auswählbaren SP, zu der er gehört, zu qualifizieren, gefolgt von einem Punkt ('.'). Beispiel: `Beziehungsname.Spaltenname`, `Beziehungsname.*`, `Alias.Spaltenname`, `Alias.*`. Qualifizierend ist *erforderlich*, wenn der Spaltenname in mehr als einer Relation vorkommt, die an einem Join teilnimmt. Das Qualifizieren von "*" ist immer obligatorisch, wenn es nicht das einzige Element in der Spaltenliste ist.

Aliase verbergen den ursprünglichen Beziehungsnamen: Sobald eine Tabelle, Ansicht oder Prozedur mit einem Alias versehen wurde, kann nur der Alias als Qualifizierer während der gesamten Abfrage verwendet werden. Der Beziehungsname selbst wird nicht mehr verfügbar.

Der Spaltenliste kann optional eines der Schlüsselwörter `DISTINCT` oder `ALL` vorangestellt werden:

- `DISTINCT` filtert alle doppelten Zeilen heraus. Das heißt, wenn zwei oder mehr Zeilen in jeder entsprechenden Spalte die gleichen Werte haben, wird nur eine davon in die Ergebnismenge aufgenommen
- `ALL` ist die Vorgabe: es gibt alle Zeilen zurück, einschließlich der Duplikate. `ALL` wird selten verwendet; es wird zur Einhaltung des SQL-Standards unterstützt.

Eine `COLLATE`-Klausel ändert das Aussehen der Spalte als solche nicht. Wenn die angegebene

Sortierung jedoch die Groß-/Kleinschreibung oder die Akzentempfindlichkeit der Spalte ändert, kann dies Folgendes beeinflussen:

- Die Reihenfolge, wenn auch eine `ORDER BY`-Klausel vorhanden ist und diese Spalte betrifft
- Gruppierung, wenn die Spalte Teil einer `GROUP BY`-Klausel ist
- Die abgerufenen Zeilen (und damit die Gesamtzahl der Zeilen in der Ergebnismenge), wenn `DISTINCT` verwendet wird

Beispiele für `SELECT`-Abfragen mit verschiedenen Arten von Spaltenlisten

Ein einfaches `SELECT`, das nur Spaltennamen verwendet:

```
select cust_id, cust_name, phone
  from customers
  where city = 'London'
```

Eine Abfrage mit einem Verkettungsausdruck und einem Funktionsaufruf in der Spaltenliste:

```
select 'Mr./Mrs. ' || lastname, street, zip, upper(city)
  from contacts
  where date_last_purchase(id) = current_date
```

Eine Abfrage mit zwei Unterauswahlen:

```
select p.fullname,
  (select name from classes c where c.id = p.class) as class,
  (select name from mentors m where m.id = p.mentor) as mentor
from pupils p
```

Die folgende Abfrage bewirkt dasselbe wie die vorherige, indem Joins anstelle von Subselects verwendet werden:

```
select p.fullname,
  c.name as class,
  m.name as mentor
  join classes c on c.id = p.class
from pupils p
  join mentors m on m.id = p.mentor
```

Diese Abfrage verwendet ein `CASE`-Konstrukt, um den richtigen Titel zu ermitteln, z.B. beim Senden von E-Mails an eine Person:

```
select case upper(sex)
    when 'F' then 'Mrs.'
    when 'M' then 'Mr.'
```

```
    else ''
  end as title,
  lastname,
  address
from employees
```

Abfrage über eine Fensterfunktion. Sortiert Mitarbeiter nach Gehalt.

```
SELECT
  id,
  salary,
  name ,
  DENSE_RANK() OVER (ORDER BY salary) AS EMP_RANK
FROM employees
ORDER BY salary;
```

Abfrage einer auswählbaren gespeicherten Prozedur:

```
select * from interesting_transactions(2010, 3, 'S')
  order by amount
```

Auswählen aus Spalten einer abgeleiteten Tabelle. Eine abgeleitete Tabelle ist eine SELECT-Anweisung in Klammern, deren Ergebnismenge in einer einschließenden Abfrage verwendet wird, als wäre es eine reguläre Tabelle oder Ansicht. Die abgeleitete Tabelle ist hier fett gedruckt:

```
select fieldcount,
  count(relation) as num_tables
from (select r.rdb$relation_name as relation,
        count(*) as fieldcount
      from rdb$relations r
        join rdb$relation_fields rf
          on rf.rdb$relation_name = r.rdb$relation_name
      group by relation)
group by fieldcount
```

Abfrage der Uhrzeit über eine Kontextvariable (`CURRENT_TIME`):

```
select current_time from rdb$database
```

Für diejenigen, die mit `RDB$DATABASE` nicht vertraut sind: Dies ist eine Systemtabelle, die in allen Firebird-Datenbanken vorhanden ist und garantiert genau eine Zeile enthält. Obwohl es nicht für diesen Zweck erstellt wurde, ist es unter Firebird-Programmierern zur Standardpraxis geworden, aus dieser Tabelle auszuwählen, wenn Sie “from Nothing” auswählen möchten, dh wenn Sie Daten benötigen, die nicht an eine Tabelle oder Ansicht gebunden sind, kann aber allein aus den Ausdrücken in den Ausgabespalten abgeleitet werden. Ein anderes Beispiel ist:

```
select power(12, 2) as twelve_squared, power(12, 3) as twelve_cubed
  from rdb$database
```

Schließlich ein Beispiel, in dem Sie einige aussagekräftige Informationen aus RDB$DATABASE selbst auswählen:

```
select rdb$character_set_name from rdb$database
```

Wie Sie vielleicht erraten haben, erhalten Sie dadurch den Standardzeichensatz der Datenbank.

Siehe auch

Funktionen, Aggregatfunktionen, Window-Funktionen>, <<fblangref30-contextvars-de,Kontextvariablen, CASE, Unterabfragen

6.1.3. Die FROM-Klausel

Die FROM-Klausel gibt die Quelle(n) an, aus der die Daten abgerufen werden sollen. In seiner einfachsten Form ist dies nur eine einzelne Tabelle oder Ansicht. Die Quelle kann jedoch auch eine auswählbare gespeicherte Prozedur, eine abgeleitete Tabelle oder ein allgemeiner Tabellenausdruck sein. Mehrere Quellen können mit verschiedenen Arten von Joins kombiniert werden.

Dieser Abschnitt konzentriert sich auf Single-Source-Selects. Joins werden in einem der folgenden Abschnitte behandelt.

Syntax

```
SELECT
  ...
  FROM <source>
  [<joins>]
  [...]

<source> ::=
  { table
  | view
  | selectable-stored-procedure [(<args>)]
  | <derived-table>
  | <common-table-expression>
  } [[AS] alias]

<derived-table> ::=
  (<select-statement>) [[AS] alias] [(<column-aliases>)]

<common-table-expression> ::=
  WITH [RECURSIVE] <cte-def> [, <cte-def> ...]
  <select-statement>
```

```
<cte-def> ::= name [(<column-aliases>)] AS (<select-statement>)

<column-aliases> ::= column-alias [, column-alias ...]
```

Tabelle 70. Argumente für die `FROM`*-Klausel*

Argument	Beschreibung
table	Name einer Tabelle
view	Name einer Ansicht
selectable-stored-procedure	Name einer auswählbaren Stored Procedure
args	Selektierbare Argumente für gespeicherte Prozeduren
derived-table	Abgeleiteter Tabellenabfrageausdruck
cte-def	Common Table Expression (CTE)-Definition, einschließlich eines “ad hoc”-Namens
select-statement	Beliebige SELECT-Anweisung
column-aliases	Alias für eine Spalte in einer Beziehung, CTE oder abgeleiteten Tabelle
name	Der “ad hoc”-Name für einen CTE
alias	Der Alias einer Datenquelle (Tabelle, Sicht, Prozedur, CTE, abgeleitete Tabelle)

Auswählen mit `FROM` in einer Tabelle oder Ansicht

Bei der Auswahl aus einer einzelnen Tabelle oder Ansicht erfordert die `FROM`-Klausel nur den Namen. Ein Alias kann nützlich oder sogar notwendig sein, wenn es Unterabfragen gibt, die sich auf die Haupt-select-Anweisung beziehen (wie sie es oft tun — Unterabfragen wie diese werden als *korrelierte Unterabfragen* bezeichnet).

Beispiele

```
select id, name, sex, age from actors
where state = 'Ohio'
```

```
select * from birds
where type = 'flightless'
order by family, genus, species
```

```
select firstname,
  middlename,
  lastname,
  date_of_birth,
  (select name from schools s where p.school = s.id) schoolname
```

```
from pupils p
where year_started = '2012'
order by schoolname, date_of_birth
```

Mischen Sie niemals Spaltennamen mit Spaltenaliasen!

Wenn Sie einen Alias für eine Tabelle oder einen View angeben, müssen Sie diesen Alias immer anstelle des Tabellennamens verwenden, wenn Sie die Spalten der Relation abfragen (und überall dort, wo Sie sonst auf Spalten verweisen, z. `GROUP BY`- und `WHERE`-Klauseln).

Richtige Verwendung:

```
SELECT PEARS
FROM FRUIT;

SELECT FRUIT.PEARS
FROM FRUIT;

SELECT PEARS
FROM FRUIT F;

SELECT F.PEARS
FROM FRUIT F;
```

Falsche Verwendung:

```
SELECT FRUIT.PEARS
FROM FRUIT F;
```

Auswählen von `FROM` einer gespeicherten Prozedur

Eine *auswählbare gespeicherte Prozedur* ist eine Prozedur, die:

- enthält mindestens einen Ausgabeparameter und
- verwendet das Schlüsselwort `SUSPEND`, damit der Aufrufer die Ausgabezeilen einzeln abrufen kann, genau wie bei der Auswahl aus einer Tabelle oder Ansicht.

Die Ausgabeparameter einer auswählbaren gespeicherten Prozedur entsprechen den Spalten einer regulären Tabelle.

Die Auswahl aus einer gespeicherten Prozedur ohne Eingabeparameter entspricht der Auswahl aus einer Tabelle oder Ansicht:

```
select * from suspicious_transactions
  where assignee = 'John'
```

Alle erforderlichen Eingabeparameter müssen nach dem Prozedurnamen in Klammern angegeben werden:

```
select name, az, alt from visible_stars('Brugge', current_date, '22:30')
  where alt >= 20
  order by az, alt
```

Werte für optionale Parameter (dh Parameter, für die Standardwerte definiert wurden) können weggelassen oder bereitgestellt werden. Wenn Sie sie jedoch nur teilweise bereitstellen, müssen sich die weggelassenen Parameter alle am Ende befinden.

Angenommen, die Prozedur `visible_stars` aus dem vorherigen Beispiel hat zwei optionale Parameter: `min_magn` (`numeric(3,1)`) und `spectral_class` (`varchar(12)`), sind die folgenden Abfragen gültig:

```
select name, az, alt
from visible_stars('Brugge', current_date, '22:30');

select name, az, alt
from visible_stars('Brugge', current_date, '22:30', 4.0);

select name, az, alt
from visible_stars('Brugge', current_date, '22:30', 4.0, 'G');
```

Dies ist jedoch nicht der Fall, da die Parameterliste ein “hole” enthält:

```
select name, az, alt
from visible_stars('Brugge', current_date, '22:30', 'G');
```

Ein Alias für eine auswählbare gespeicherte Prozedur wird *nach* der Parameterliste angegeben:

```
select
  number,
  (select name from contestants c where c.number = gw.number)
from get_winners('#34517', 'AMS') gw
```

Wenn Sie auf einen Ausgabeparameter (“column”) verweisen, indem Sie ihn mit dem vollständigen Prozedurnamen qualifizieren, sollte der Prozeduralias weggelassen werden:

```
select
  number,
  (select name from contestants c where c.number = get_winners.number)
from get_winners('#34517', 'AMS')
```

Siehe auch

Stored Procedures, `CREATE PROCEDURE`

Abfragen einer abgeleiteten Tabelle mittels `FROM`

Eine abgeleitete Tabelle ist eine gültige `SELECT`-Anweisung in Klammern, optional gefolgt von einem Tabellenalias und/oder Spaltenaliasen. Die Ergebnismenge der Anweisung fungiert als virtuelle Tabelle, die die einschließende Anweisung abfragen kann.

Syntax

```
(<select-query>)
  [[AS] derived-table-alias]
  [(<derived-column-aliases>)]

<derived-column-aliases> := column-alias [, column-alias ...]
```

Der von diesem "`SELECT FROM (SELECT FROM..)`"-Stil der Anweisung zurückgegebene Datensatz ist eine virtuelle Tabelle, die innerhalb der einschließenden Anweisung abgefragt werden kann, als wäre es eine reguläre Tabelle oder Ansicht.

Beispiel using a derived table

Die abgeleitete Tabelle in der folgenden Abfrage gibt die Liste der Tabellennamen in der Datenbank und die Anzahl der Spalten in jeder Tabelle zurück. Eine "Drill-Down"-Abfrage für die abgeleitete Tabelle gibt die Anzahl der Felder und die Anzahl der Tabellen mit jeder Feldanzahl zurück:

```
SELECT
  FIELDCOUNT,
  COUNT(RELATION) AS NUM_TABLES
FROM (SELECT
        R.RDB$RELATION_NAME RELATION,
        COUNT(*) AS FIELDCOUNT
      FROM RDB$RELATIONS R
        JOIN RDB$RELATION_FIELDS RF
        ON RF.RDB$RELATION_NAME = R.RDB$RELATION_NAME
        GROUP BY RELATION)
GROUP BY FIELDCOUNT
```

Ein triviales Beispiel, das zeigt, wie der Alias einer abgeleiteten Tabelle und die Liste der Spaltenaliase (beide optional) verwendet werden können:

```
SELECT
  DBINFO.DESCR, DBINFO.DEF_CHARSET
FROM (SELECT *
      FROM RDB$DATABASE) DBINFO
        (DESCR, REL_ID, SEC_CLASS, DEF_CHARSET)
```

Mehr über abgeleitete Tabellen

Abgeleitete Tabellen können

- verschachtelt sein
- Gewerkschaften sein und in Gewerkschaften verwendet werden können
- enthalten Aggregatfunktionen, Unterabfragen und Joins
- in Aggregatfunktionen, Unterabfragen und Joins verwendet werden
- Aufrufe an auswählbare gespeicherte Prozeduren oder Abfragen an diese sein
- haben `WHERE`, `ORDER BY` und `GROUP BY` Klauseln, `FIRST`/`SKIP` oder `ROWS` Direktiven, et al.

Außerdem,

- Jede Spalte in einer abgeleiteten Tabelle muss einen Namen haben. Wenn es keinen Namen hat, z. B. wenn es sich um eine Konstante oder einen Laufzeitausdruck handelt, sollte ihm ein Alias zugewiesen werden, entweder auf reguläre Weise oder durch Einfügen in die Liste der Spaltenaliase in der Spezifikation der abgeleiteten Tabelle.
  - *Die Liste der Spaltenaliase ist optional, aber falls vorhanden, muss sie für jede Spalte in der abgeleiteten Tabelle einen Alias enthalten*
- Der Optimierer kann abgeleitete Tabellen sehr effektiv verarbeiten. Wenn jedoch eine abgeleitete Tabelle in einen Inner Join eingeschlossen ist und eine Unterabfrage enthält, kann der Optimierer keine Join-Reihenfolge verwenden.

Ein nützlicheres Beispiel

Angenommen, wir haben eine Tabelle `COEFFS`, die die Koeffizienten einer Reihe von quadratischen Gleichungen enthält, die wir lösen müssen. Es wurde wie folgt definiert:

```
create table coeffs (
  a double precision not null,
  b double precision not null,
  c double precision not null,
  constraint chk_a_not_zero check (a <> 0)
)
```

Abhängig von den Werten von 'a', 'b' und 'c' kann jede Gleichung null, eine oder zwei Lösungen haben. Es ist möglich, diese Lösungen mit einer einstufigen Abfrage der Tabelle `COEFFS` zu finden, aber der Code sieht ziemlich unordentlich aus und mehrere Werte (wie die Diskriminante) müssen mehrmals pro Zeile berechnet werden. Eine abgeleitete Tabelle kann hier helfen, die Dinge sauber zu halten:

```
select
  iif (D >= 0, (-b - sqrt(D)) / denom, null) sol_1,
  iif (D >  0, (-b + sqrt(D)) / denom, null) sol_2
  from
```

```
    (select b, b*b - 4*a*c, 2*a from coeffs) (b, D, denom)
```

Wenn wir die Koeffizienten neben den Lösungen anzeigen möchten (was möglicherweise keine schlechte Idee ist), können wir die Abfrage wie folgt ändern:

```
select
  a, b, c,
  iif (D >= 0, (-b - sqrt(D)) / denom, null) sol_1,
  iif (D >  0, (-b + sqrt(D)) / denom, null) sol_2
  from
    (select a, b, c, b*b - 4*a*c as D, 2*a as denom
     from coeffs)
```

Beachten Sie, dass, während die erste Abfrage eine Spaltenaliasliste für die abgeleitete Tabelle verwendet, die zweite bei Bedarf intern Aliase hinzufügt. Beide Methoden funktionieren, solange jede Spalte garantiert einen Namen hat.

Alle Spalten in der abgeleiteten Tabelle werden so oft ausgewertet, wie sie in der Hauptabfrage angegeben sind. Dies ist wichtig, da es bei der Verwendung nichtdeterministischer Funktionen zu unerwarteten Ergebnissen führen kann. Das Folgende zeigt ein Beispiel dafür.

```
SELECT
  UUID_TO_CHAR(X) AS C1,
  UUID_TO_CHAR(X) AS C2,
  UUID_TO_CHAR(X) AS C3
FROM (SELECT GEN_UUID() AS X
      FROM RDB$DATABASE) T;
```

Das Ergebnis, wenn diese Abfrage drei verschiedene Werte erzeugt:

```
C1  80AAECED-65CD-4C2F-90AB-5D548C3C7279
C2  C1214CD3-423C-406D-B5BD-95BF432ED3E3
C3  EB176C10-F754-4689-8B84-64B666381154
```

Um ein einzelnes Ergebnis der Funktion `GEN_UUID` sicherzustellen, können Sie die folgende Methode verwenden:

```
SELECT
  UUID_TO_CHAR(X) AS C1,
  UUID_TO_CHAR(X) AS C2,
  UUID_TO_CHAR(X) AS C3
FROM (SELECT GEN_UUID() AS X
      FROM RDB$DATABASE
      UNION ALL
```

```
       SELECT NULL FROM RDB$DATABASE WHERE 1 = 0) T;
```

Diese Abfrage erzeugt ein einzelnes Ergebnis für alle drei Spalten:

```
C1  80AAECED-65CD-4C2F-90AB-5D548C3C7279
C2  80AAECED-65CD-4C2F-90AB-5D548C3C7279
C3  80AAECED-65CD-4C2F-90AB-5D548C3C7279
```

Eine alternative Lösung besteht darin, die Abfrage 'GEN_UUID' in eine Unterabfrage einzuschließen:

```
SELECT
  UUID_TO_CHAR(X) AS C1,
  UUID_TO_CHAR(X) AS C2,
  UUID_TO_CHAR(X) AS C3
FROM (SELECT
        (SELECT GEN_UUID() FROM RDB$DATABASE) AS X
      FROM RDB$DATABASE) T;
```

Dies ist ein Artefakt der aktuellen Implementierung. Dieses Verhalten kann sich in einer zukünftigen Firebird-Version ändern.

Abfragen einer Common Table Expression (CTE) mittels `FROM`

Ein allgemeiner Tabellenausdruck – oder *CTE* – ist eine komplexere Variante der abgeleiteten Tabelle, aber auch leistungsfähiger. Eine Präambel, die mit dem Schlüsselwort `WITH` beginnt, definiert eine oder mehrere benannte *CTE*s, jede mit einer optionalen Spalten-Alias-Liste. Die Hauptabfrage, die der Präambel folgt, kann dann auf diese *CTE*s zugreifen, als wären es reguläre Tabellen oder Ansichten. Die *CTE*s verlassen den Gültigkeitsbereich, sobald die Hauptabfrage vollständig ausgeführt wurde.

Eine vollständige Diskussion der *CTE*s finden Sie im Abschnitt Common Table Expressions (“`WITH` ⋯ `AS` ⋯ `SELECT`”).

Das Folgende ist eine Umschreibung unseres abgeleiteten Tabellenbeispiels als *CTE*:

```
with vars (b, D, denom) as (
  select b, b*b - 4*a*c, 2*a from coeffs
)
select
  iif (D >= 0, (-b - sqrt(D)) / denom, null) sol_1,
  iif (D >  0, (-b + sqrt(D)) / denom, null) sol_2
from vars
```

Abgesehen davon, dass die Berechnungen, die zuerst durchgeführt werden müssen, jetzt am Anfang stehen, ist dies keine große Verbesserung gegenüber der abgeleiteten Tabellenversion.

Allerdings können wir jetzt auch die doppelte Berechnung von `sqrt(D)` für jede Zeile eliminieren:

```
with vars (b, D, denom) as (
  select b, b*b - 4*a*c, 2*a from coeffs
),
vars2 (b, D, denom, sqrtD) as (
  select b, D, denom, iif (D >= 0, sqrt(D), null) from vars
)
select
  iif (D >= 0, (-b - sqrtD) / denom, null) sol_1,
  iif (D >  0, (-b + sqrtD) / denom, null) sol_2
from vars2
```

Der Code ist jetzt etwas komplizierter, kann aber effizienter ausgeführt werden (je nachdem, was länger dauert: Ausführen der `SQRT`-Funktion oder Übergabe der Werte von `b`, `D` und `denom` durch einen zusätzlichen *CTE*) . Übrigens hätten wir das auch mit abgeleiteten Tabellen machen können, aber das würde eine Verschachtelung erfordern.

Alle Spalten im CTE werden so oft ausgewertet, wie sie in der Hauptabfrage angegeben sind. Dies ist wichtig, da es bei der Verwendung nichtdeterministischer Funktionen zu unerwarteten Ergebnissen führen kann. Das Folgende zeigt ein Beispiel dafür.

```
WITH T (X) AS (
  SELECT GEN_UUID()
  FROM RDB$DATABASE)
SELECT
  UUID_TO_CHAR(X) as c1,
  UUID_TO_CHAR(X) as c2,
  UUID_TO_CHAR(X) as c3
FROM T
```

Das Ergebnis, wenn diese Abfrage drei verschiedene Werte erzeugt:

```
C1  80AAECED-65CD-4C2F-90AB-5D548C3C7279
C2  C1214CD3-423C-406D-B5BD-95BF432ED3E3
C3  EB176C10-F754-4689-8B84-64B666381154
```

Um ein einzelnes Ergebnis der Funktion `GEN_UUID` sicherzustellen, können Sie die folgende Methode verwenden:

```
WITH T (X) AS (
  SELECT GEN_UUID()
  FROM RDB$DATABASE
  UNION ALL
  SELECT NULL FROM RDB$DATABASE WHERE 1 = 0)
```

```
SELECT
  UUID_TO_CHAR(X) as c1,
  UUID_TO_CHAR(X) as c2,
  UUID_TO_CHAR(X) as c3
FROM T;
```

Diese Abfrage erzeugt ein einzelnes Ergebnis für alle drei Spalten:

```
C1  80AAECED-65CD-4C2F-90AB-5D548C3C7279
C2  80AAECED-65CD-4C2F-90AB-5D548C3C7279
C3  80AAECED-65CD-4C2F-90AB-5D548C3C7279
```

Eine alternative Lösung besteht darin, die Abfrage 'GEN_UUID' in eine Unterabfrage einzuschließen:

```
WITH T (X) AS (
  SELECT (SELECT GEN_UUID() FROM RDB$DATABASE)
  FROM RDB$DATABASE)
SELECT
  UUID_TO_CHAR(X) as c1,
  UUID_TO_CHAR(X) as c2,
  UUID_TO_CHAR(X) as c3
FROM T;
```

Dies ist ein Artefakt der aktuellen Implementierung. Dieses Verhalten kann sich in einer zukünftigen Firebird-Version ändern.

Siehe auch

Common Table Expressions (“`WITH ⋯ AS ⋯ SELECT`”).

6.1.4. Joins

Joins kombinieren Daten aus zwei Quellen zu einem einzigen Satz. Dies erfolgt zeilenweise und beinhaltet normalerweise die Überprüfung einer *Join-Bedingung*, um zu bestimmen, welche Zeilen zusammengeführt und im resultierenden Dataset erscheinen sollen. Es gibt verschiedene Typen (`INNER`, `OUTER`) und Klassen (qualifiziert, natürlich usw.) von Joins, jede mit ihrer eigenen Syntax und eigenen Regeln.

Da Joins verkettet werden können, können die an einem Join beteiligten Datasets selbst verbundene Sets sein.

Syntax

```
SELECT
   ...
   FROM <source>
   [<joins>]
```

```
   [...]

<source> ::=
  { table
  | view
  | selectable-stored-procedure [(<args>)]
  | <derived-table>
  | <common-table-expression>
  } [[AS] alias]

<joins> ::= <join> [<join> ...]

<join> ::=
    [<join-type>] JOIN <source> <join-condition>
  | NATURAL [<join-type>] JOIN <source>
  | {CROSS JOIN | ,} <source>

<join-type> ::= INNER | {LEFT | RIGHT | FULL} [OUTER]

<join-condition> ::= ON <condition> | USING (<column-list>)
```

Tabelle 71. Argumente für die `JOIN`*-Klausel*

Argument	Beschreibung
table	Name einer Tabelle
view	Name einer Ansicht
selectable-stored-procedure	Name einer auswählbaren Stored Procedure
args	Wählbare Eingabeparameter für gespeicherte Prozeduren
derived-table	Verweis, namentlich, auf eine abgeleitete Tabelle
common-table-expression	Verweis nach Name auf einen allgemeinen Tabellenausdruck (CTE)
alias	Ein Alias für eine Datenquelle (Tabelle, Sicht, Prozedur, CTE, abgeleitete Tabelle)
condition	Join-Bedingung (Kriterium)
column-list	Die Liste der Spalten, die für einen Equi-Join verwendet werden

Inner vs. Outer Joins

Ein Join kombiniert immer Datenzeilen aus zwei Sätzen (normalerweise als linker Satz und rechter Satz bezeichnet). Standardmäßig gelangen nur Zeilen in die Ergebnismenge, die die Join-Bedingung erfüllen (d. h. die mindestens einer Zeile in der anderen Menge entsprechen, wenn die Join-Bedingung angewendet wird). Dieser Standard-Join-Typ wird als *inner join* bezeichnet. Angenommen, wir haben die folgenden zwei Tabellen:

Tabelle A

ID	S
87	Just some text
235	Silence

Tabelle B

CODE	X
-23	56.7735
87	416.0

Wenn wir diese Tabellen wie folgt verbinden:

```
select *
  from A
  join B on A.id = B.code;
```

dann ist die Ergebnismenge:

ID	S	CODE	X
87	Just some text	87	416.0

Die erste Reihe von `A` wurde mit der zweiten Reihe von `B` verbunden, weil sie zusammen die Bedingung "`A.id = B.code`" erfüllten. Die anderen Zeilen aus den Quelltabellen haben keine Übereinstimmung in der entgegengesetzten Menge und werden daher nicht in den Join aufgenommen. Denken Sie daran, dies ist ein `INNER`-Join. Wir können diese Tatsache explizit machen, indem wir schreiben:

```
select *
  from A
  inner join B on A.id = B.code;
```

Da jedoch `INNER` die Vorgabe ist, wird es normalerweise weggelassen.

Es ist durchaus möglich, dass eine Reihe im linken Satz mit mehreren Reihen im rechten Satz übereinstimmt oder umgekehrt. In diesem Fall sind alle diese Kombinationen enthalten, und wir können Ergebnisse erhalten wie:

ID	S	CODE	X
87	Just some text	87	416.0
87	Just some text	87	-1.0
-23	Don't know	-23	56.7735
-23	Still don't know	-23	56.7735
-23	I give up	-23	56.7735

Manchmal möchten (oder müssen) wir *alle* Zeilen einer oder beider Quellen in der verbundenen Menge erscheinen, unabhängig davon, ob sie mit einem Datensatz in der anderen Quelle übereinstimmen. Hier kommen Outer Joins ins Spiel. Ein 'LEFT' Outer Join enthält alle Datensätze aus dem linken Satz, aber nur übereinstimmende Datensätze aus dem rechten Satz. Bei einem `RIGHT` Outer Join ist es umgekehrt. `FULL` Outer Joins beinhalten alle Datensätze aus beiden Sets. In allen Outer Joins werden die "Löcher" (die Stellen, an denen ein eingeschlossener Quelldatensatz keine Übereinstimmung im anderen Satz hat) mit `NULL` aufgefüllt.

Um einen Outer Join zu erstellen, müssen Sie `LEFT`, `RIGHT` oder `FULL` angeben, optional gefolgt vom Schlüsselwort `OUTER`.

Unten sind die Ergebnisse der verschiedenen Outer Joins, wenn sie auf unsere ursprünglichen Tabellen `A` und `B` angewendet werden:

```
select *
  from A
  left [outer] join B on A.id = B.code;
```

ID	S	CODE	X
87	Just some text	87	416.0
235	Silence	*<null>*	*<null>*

```
select *
  from A
  right [outer] join B on A.id = B.code
```

ID	S	CODE	X
<null>	*<null>*	-23	56.7735
87	Just some text	87	416.0

```
select *
  from A
  full [outer] join B on A.id = B.code
```

ID	S	CODE	X
<null>	*<null>*	-23	56.7735
87	Just some text	87	416.0
235	Silence	*<null>*	*<null>*

Qualifizierte joins

Qualifizierte Joins geben Bedingungen für das Kombinieren von Zeilen an. Dies geschieht entweder

explizit in einer ON-Klausel oder implizit in einer USING-Klausel.

Syntax

```
<qualified-join> ::= [<join-type>] JOIN <source> <join-condition>

<join-type> ::= INNER | {LEFT | RIGHT | FULL} [OUTER]

<join-condition> ::= ON <condition> | USING (<column-list>)
```

Joins mit expliziter Bedingung

Die meisten qualifizierten Joins haben eine ON-Klausel mit einer expliziten Bedingung, die jeder gültige boolesche Ausdruck sein kann, aber normalerweise einen Vergleich zwischen den beiden beteiligten Quellen beinhaltet.

Sehr oft ist die Bedingung ein Gleichheitstest (oder eine Reihe von AND-verknüpften Gleichheitstests) mit dem Operator "=". Joins wie diese heißen *equi-joins*. (Die Beispiele im Abschnitt über innere und äußere Verknüpfungen waren alle Gleichverknüpfungen.)

Beispiele für Joins mit einer expliziten Bedingung:

```
/* Wählen Sie alle Detroit-Kunden aus, die einen Kauf getätigt haben
    2013, zusammen mit den Kaufdetails: */
select * from customers c
  join sales s on s.cust_id = c.id
  where c.city = 'Detroit' and s.year = 2013;
```

```
/* Wie oben, aber auch nicht kaufende Kunden: */
select * from customers c
  left join sales s on s.cust_id = c.id
  where c.city = 'Detroit' and s.year = 2013;
```

```
/* Wählen Sie für jeden Mann die Frauen aus, die größer sind als er.
    Männer, für die es keine solche Frau gibt, werden nicht berücksichtigt. */
select m.fullname as man, f.fullname as woman
  from males m
  join females f on f.height > m.height;
```

```
/* Wählen Sie alle Schüler mit ihrer Klasse und ihrem Mentor aus.
    Auch Schüler ohne Mentor werden einbezogen.
    Schüler ohne Klasse werden nicht berücksichtigt. */
select p.firstname, p.middlename, p.lastname,
       c.name, m.name
  from pupils p
  join classes c on c.id = p.class
```

```
  left join mentors m on m.id = p.mentor;
```

Joins mit benannten Spalten

Equi-Joins vergleichen häufig Spalten mit dem gleichen Namen in beiden Tabellen. Wenn dies der Fall ist, können wir auch den zweiten Typ eines qualifizierten Joins verwenden: den *benannten Spalten join*.

Benannte Spalten-Joins werden in Dialekt-1-Datenbanken nicht unterstützt.

Benannte Spalten-Joins haben eine `USING`-Klausel, die nur die Spaltennamen angibt. Also stattdessen:

```
select * from flotsam f
  join jetsam j
  on f.sea = j.sea
  and f.ship = j.ship;
```

wir können auch schreiben:

```
select * from flotsam
  join jetsam using (sea, ship)
```

was deutlich kürzer ist. Die Ergebnismenge ist jedoch etwas anders — zumindest bei Verwendung von "`SELECT *`":

- Der Join mit expliziter Bedingung — mit der `ON`-Klausel — enthält jede der Spalten `SEA` und `SHIP` zweimal: einmal aus der Tabelle `FLOTSAM` und einmal aus der Tabelle `JETSAM`. Offensichtlich haben sie die gleichen Werte.
- Der Join mit benannten Spalten – mit der `USING`-Klausel – enthält diese Spalten nur einmal.

Wenn Sie alle Spalten in der Ergebnismenge der benannten Spalten verknüpfen möchten, richten Sie Ihre Abfrage wie folgt ein:

```
select f.*, j.*
  from flotsam f
  join jetsam j using (sea, ship);
```

Dadurch erhalten Sie genau die gleiche Ergebnismenge wie beim Join mit expliziter Bedingung.

Für einen `OUTER` benannten Spalten-Join gibt es eine zusätzliche Wendung, wenn "`SELECT *`" oder ein nicht qualifizierter Spaltenname aus der `USING`-Liste verwendet wird:

Wenn eine Zeile aus einem Quellsatz keine Übereinstimmung im anderen hat, aber aufgrund der Direktiven `LEFT`, `RIGHT` oder `FULL` trotzdem eingeschlossen werden muss, erhält die zusammengeführte Spalte in der verbundenen Menge das Nicht- `NULL`-Wert. Das ist fair genug, aber

jetzt können Sie nicht sagen, ob dieser Wert aus dem linken Satz, dem rechten Satz oder beiden stammt. Dies kann besonders täuschen, wenn der Wert aus dem rechten Satz stammt, da "*" immer kombinierte Spalten im linken Teil anzeigt — auch bei einem `RIGHT`-Join.

Ob dies ein Problem ist oder nicht, hängt von der Situation ab. Wenn dies der Fall ist, verwenden Sie den oben gezeigten Ansatz "`a.*, b.*`", wobei `a` und `b` die Namen oder Aliase der beiden Quellen sind. Oder noch besser, vermeiden Sie "*" in Ihren ernsthaften Abfragen und qualifizieren Sie alle Spaltennamen in verbundenen Mengen. Dies hat den zusätzlichen Vorteil, dass Sie sich überlegen müssen, welche Daten Sie woher abrufen möchten.

Es liegt in Ihrer Verantwortung, sicherzustellen, dass die Spaltennamen in der `USING`-Liste von kompatiblen Typen zwischen den beiden Quellen sind. Wenn die Typen kompatibel, aber nicht gleich sind, konvertiert die Engine sie in den Typ mit dem breitesten Wertebereich, bevor die Werte verglichen werden. Dies ist auch der Datentyp der zusammengeführten Spalte, der in der Ergebnismenge angezeigt wird, wenn "`SELECT *`" oder der nicht qualifizierte Spaltenname verwendet wird. Qualifizierte Spalten hingegen behalten immer ihren ursprünglichen Datentyp.

Wenn Sie beim Zusammenführen nach benannten Spalten eine Join-Spalte in der `WHERE`-Klausel verwenden, verwenden Sie immer den qualifizierten Spaltennamen, andernfalls wird kein Index für diese Spalte verwendet.

```
SELECT 1 FROM t1 a JOIN t2 b USING (x) WHERE x = 0;

-- PLAN JOIN (A NATURAL , B INDEX (RDB$2))
```

Jedoch:

```
SELECT 1 FROM t1 a JOIN t2 b USING (x) WHERE a.x = 0;
-- PLAN JOIN (A INDEX (RDB$1), B INDEX (RDB$2))

SELECT 1 FROM t1 a JOIN t2 b USING (x) WHERE b.x = 0;
-- PLAN JOIN (A INDEX (RDB$1), B INDEX (RDB$2))
```

Tatsache ist, dass die nicht spezifizierte Spalte in diesem Fall implizit durch `COALESCE(a.x, b.x) ersetzt wird. Dieser clevere Trick wird verwendet, um Spaltennamen eindeutig zu machen, stört aber auch die Verwendung des Indexes.

Natural Joins

Um die Idee des benannten Spalten-Joins noch einen Schritt weiter zu gehen, führt ein *natural join* einen automatischen Equi-Join für alle Spalten mit dem gleichen Namen in der linken und rechten Tabelle durch. Die Datentypen dieser Spalten müssen kompatibel sein.

Natural-Joins werden in Dialekt-1-Datenbanken nicht unterstützt.

Syntax

```
<natural-join> ::= NATURAL [<join-type>] JOIN <source>

<join-type> ::= INNER | {LEFT | RIGHT | FULL} [OUTER]
```

Gegeben seien diese beiden Tabellen:

```
create table TA (
  a bigint,
  s varchar(12),
  ins_date date
);
```

```
create table TB (
  a bigint,
  descr varchar(12),
  x float,
  ins_date date
);
```

Ein natürlicher Join von `TA` und `TB` würde die Spalten `a` und `ins_date` beinhalten, und die folgenden beiden Anweisungen hätten den gleichen Effekt:

```
select * from TA
  natural join TB;
```

```
select * from TA
  join TB using (a, ins_date);
```

Wie alle Joins sind natürliche Joins standardmäßig innere Joins, aber Sie können sie in äußere Joins umwandeln, indem Sie `LEFT`, `RIGHT` oder `FULL` vor dem `JOIN`-Schlüsselwort angeben.

Gibt es in den beiden Quellbeziehungen keine gleichnamigen Spalten, wird ein `CROSS JOIN` ausgeführt. Wir kommen in einer Minute zu dieser Art von Join.

Cross Joins

Ein Cross-Join erzeugt das Full-Set-Produkt der beiden Datenquellen. Dies bedeutet, dass jede Zeile in der linken Quelle erfolgreich mit jeder Zeile in der rechten Quelle abgeglichen wird.

Syntax

```
<cross-join> ::= {CROSS JOIN | ,} <source>
```

Bitte beachten Sie, dass die Kommasyntax veraltet ist! Es wird nur unterstützt, um die Funktionsfähigkeit des Legacy-Codes aufrechtzuerhalten, und kann in einer zukünftigen Version verschwinden.

Das Kreuzverknüpfen zweier Mengen ist äquivalent dazu, sie auf einer Tautologie zu verbinden (eine Bedingung, die immer wahr ist). Die folgenden beiden Aussagen haben die gleiche Wirkung:

```
select * from TA
  cross join TB;
```

```
select * from TA
  join TB on 1 = 1;
```

Cross-Joins sind Inner-Joins, da sie nur übereinstimmende Datensätze enthalten – es kommt einfach vor, dass *jeder* Datensatz übereinstimmt! Ein Outer-Cross-Join, falls vorhanden, würde dem Ergebnis nichts hinzufügen, da die hinzugefügten Outer-Joins nicht übereinstimmende Datensätze sind und diese in Cross-Joins nicht vorhanden sind.

Cross-Joins sind selten sinnvoll, außer wenn Sie alle möglichen Kombinationen von zwei oder mehr Variablen auflisten möchten. Angenommen, Sie verkaufen ein Produkt in verschiedenen Größen, Farben und Materialien. Wenn diese Variablen jeweils in einer eigenen Tabelle aufgeführt sind, würde diese Abfrage alle Kombinationen zurückgeben:

```
select m.name, s.size, c.name
  from materials m
  cross join sizes s
  cross join colors c;
```

Implizite Joins

Im SQL:89-Standard wurden die an einem Join beteiligten Tabellen als durch Kommas getrennte Liste in der `FROM`-Klausel angegeben (mit anderen Worten, ein Cross Join). Die Join-Bedingungen wurden dann neben anderen Suchbegriffen in der `WHERE`-Klausel angegeben. Diese Art von Join wird als impliziter Join bezeichnet.

Ein Beispiel für einen impliziten Join:

```
/*
 * Eine Auswahl aller Detroit-Kunden, die
 * einen Einkauf getätigt haben
 */
SELECT *
FROM customers c, sales s
WHERE s.cust_id = c.id AND c.city = 'Detroit'
```

Die implizite Join-Syntax ist veraltet und wird möglicherweise in einer zukünftigen Version entfernt. Wir empfehlen, die zuvor gezeigte explizite Join-Syntax zu verwenden.

Explizite und implizite Verknüpfungen mischen

Das Mischen von expliziten und impliziten Joins wird nicht empfohlen, ist jedoch zulässig. Einige Arten des Mischens werden jedoch von Firebird nicht unterstützt.

Die folgende Abfrage gibt beispielsweise den Fehler “Spalte gehört nicht zur referenzierten Tabelle” aus.

```
SELECT *
FROM TA, TB
JOIN TC ON TA.COL1 = TC.COL1
WHERE TA.COL2 = TB.COL2
```

Das liegt daran, dass der explizite Join die Tabelle `TA` nicht sehen kann. Die nächste Abfrage wird jedoch ohne Fehler abgeschlossen, da die Einschränkung nicht verletzt wird.

```
SELECT *
FROM TA, TB
JOIN TC ON TB.COL1 = TC.COL1
WHERE TA.COL2 = TB.COL2
```

Ein Hinweis zu Gleichheit

Dieser Hinweis zu Gleichheits- und Ungleichheitsoperatoren gilt überall in Firebirds SQL-Sprache, nicht nur in `JOIN`-Bedingungen.

Der Operator “=”, der explizit in vielen bedingten Joins und implizit in benannten Spalten-Joins und natürlichen Joins verwendet wird, gleicht nur Werte mit Werten ab. Nach dem SQL-Standard ist `NULL` kein Wert und daher sind zwei `NULL` weder gleich noch ungleich. Wenn `NULL`s in einem Join miteinander übereinstimmen müssen, verwenden Sie den `IS NOT DISTINCT FROM`-Operator. Dieser Operator gibt true zurück, wenn die Operanden den gleichen Wert *oder* haben, wenn beide `NULL` sind.

```
select *
  from A join B
  on A.id is not distinct from B.code;
```

Ebenso in den — extrem seltenen — Fällen, in denen Sie bei *in*equality beitreten möchten, verwenden Sie `IS DISTINCT FROM`, nicht “<>”, wenn `NULL` als anders betrachtet werden soll Wert und zwei `NULL`s als gleich betrachtet:

```
select *
  from A join B
  on A.id is distinct from B.code;
```

Mehrdeutige Feldnamen in Joins

Firebird weist nicht qualifizierte Feldnamen in einer Abfrage zurück, wenn diese Feldnamen in mehr als einem an einem Join beteiligten Dataset vorhanden sind. Dies gilt sogar für innere Equi-Joins, bei denen der Feldname in der `ON`-Klausel wie folgt vorkommt:

```
select a, b, c
  from TA
  join TB on TA.a = TB.a;
```

Von dieser Regel gibt es eine Ausnahme: Bei Named-Column-Joins und Natural-Joins darf der unqualifizierte Feldname einer am Matching-Prozess beteiligten Spalte legal verwendet werden und bezieht sich auf die gleichnamige zusammengeführte Spalte. Bei Joins mit benannten Spalten sind dies die Spalten, die in der `USING`-Klausel aufgelistet sind. Bei natürlichen Verknüpfungen sind dies die Spalten, die in beiden Beziehungen denselben Namen haben. Beachten Sie aber bitte noch einmal, dass, insbesondere bei Outer-Joins, ein einfacher `colname` nicht immer gleich `links.colname` oder `right.colname` ist. Typen können unterschiedlich sein und eine der qualifizierten Spalten kann `NULL` sein, während die andere nicht ist. In diesem Fall kann der Wert in der zusammengeführten, nicht qualifizierten Spalte die Tatsache maskieren, dass einer der Quellwerte fehlt.

Joins mit gespeicherten Prozeduren

Wenn ein Join mit einer Stored Procedure durchgeführt wird, die nicht über Eingabeparameter mit anderen Datenströmen korreliert ist, gibt es keine Merkwürdigkeiten. Wenn Korrelation im Spiel ist, offenbart sich eine unangenehme Eigenart. Das Problem ist, dass sich der Optimierer jede Möglichkeit verweigert, die Zusammenhänge der Eingabeparameter der Prozedur aus den Feldern in den anderen Streams zu ermitteln:

```
SELECT *
FROM MY_TAB
JOIN MY_PROC(MY_TAB.F) ON 1 = 1;
```

Hier wird die Prozedur ausgeführt, bevor ein einzelner Datensatz aus der Tabelle `MY_TAB` abgerufen wurde. Der Fehler `isc_no_cur_rec error` (*no current record for fetch operation*) wird ausgelöst und unterbricht die Ausführung.

Die Lösung besteht darin, eine Syntax zu verwenden, die die Join-Reihenfolge *explizit* angibt:

```
SELECT *
FROM MY_TAB
LEFT JOIN MY_PROC(MY_TAB.F) ON 1 = 1;
```

Dies erzwingt, dass die Tabelle vor dem Vorgang gelesen wird und alles funktioniert ordnungsgemäß.

Diese Eigenart wurde im Optimierer als Fehler erkannt und wird in der nächsten Version von Firebird behoben.

6.1.5. Die WHERE-Klausel

Die WHERE-Klausel dient dazu, die zurückgegebenen Zeilen auf diejenigen zu beschränken, die den Aufrufer interessieren. Die Bedingung, die dem Schlüsselwort WHERE folgt, kann eine einfache Prüfung wie "AMOUNT = 3" sein oder ein vielschichtiger, verschachtelter Ausdruck mit Unterauswahlen, Prädikaten, Funktionsaufrufen, mathematischen und logischen Operatoren, Kontexvariablen und mehr.

Die Bedingung in der WHERE-Klausel wird oft als *Suchbedingung*, als *Suchausdruck* oder einfach als *Suche* bezeichnet.

In DSQL und ESQL kann der Suchausdruck Parameter enthalten. Dies ist sinnvoll, wenn eine Abfrage mit unterschiedlichen Eingabewerten mehrmals wiederholt werden muss. In der SQL-Zeichenfolge, die an den Server übergeben wird, werden Fragezeichen als Platzhalter für die Parameter verwendet. Sie werden *positionale Parameter* genannt, weil sie nur durch ihre Position im String unterschieden werden können. Konnektivitätsbibliotheken unterstützen oft *named parameters* der Form :id, :amount, :a usw. Diese sind benutzerfreundlicher; die Bibliothek kümmert sich um die Übersetzung der benannten Parameter in Positionsparameter, bevor die Anweisung an den Server übergeben wird.

Die Suchbedingung kann auch lokale (PSQL) oder Host- (ESQL) Variablennamen enthalten, denen ein Doppelpunkt vorangestellt ist.

Syntax

```
SELECT ...
  FROM ...
  [...]
  WHERE <search-condition>
  [...]
```

Tabelle 72. WHERE-Argumente

Parameter	Beschreibung
search-condition	Ein boolescher Ausdruck, der TRUE, FALSE oder möglicherweise UNKNOWN (NULL) zurückgibt.

Nur die Zeilen, für die die Suchbedingung 'TRUE' ergibt, werden in die Ergebnismenge aufgenommen. Seien Sie vorsichtig mit möglichen NULL-Ergebnissen: Wenn Sie einen NULL-Ausdruck mit NOT negieren, ist das Ergebnis immer noch NULL und die Zeile wird nicht passieren. Dies wird in einem der folgenden Beispiele demonstriert.

Beispiele

```
select genus, species from mammals
  where family = 'Felidae'
  order by genus;
```

```
select * from persons
  where birthyear in (1880, 1881)
     or birthyear between 1891 and 1898;
```

```
select name, street, borough, phone
  from schools s
  where exists (select * from pupils p where p.school = s.id)
  order by borough, street;
```

```
select * from employees
  where salary >= 10000 and position <> 'Manager';
```

```
select name from wrestlers
  where region = 'Europe'
    and weight > all (select weight from shot_putters
                      where region = 'Africa');
```

```
select id, name from players
  where team_id = (select id from teams where name = 'Buffaloes');
```

```
select sum (population) from towns
  where name like '%dam'
  and province containing 'land';
```

```
select password from usertable
  where username = current_user;
```

Das folgende Beispiel zeigt, was passieren kann, wenn die Suchbedingung `NULL` ergibt.

Angenommen, Sie haben eine Tabelle mit den Namen einiger Kinder und der Anzahl der Murmeln (engl. marbles), die sie besitzen. Zu einem bestimmten Zeitpunkt enthält die Tabelle diese Daten:

CHILD	MARBLES
Anita	23

CHILD	MARBLES
Bob E.	12
Chris	<*null*>
Deirdre	1
Eve	17
Fritz	0
Gerry	21
Hadassah	<*null*>
Isaac	6

Beachten Sie zunächst den Unterschied zwischen `NULL` und 0: Fritz hat *bekannt* überhaupt keine Murmeln, Chris' und Hadassah's Murmeln sind unbekannt.

Wenn Sie nun diese SQL-Anweisung ausgeben:

```
select list(child) from marbletable where marbles > 10;
```

Sie erhalten die Namen Anita, Bob E., Eve und Gerry. Diese Kinder haben alle mehr als 10 Murmeln.

Wenn Sie den Ausdruck negieren:

```
select list(child) from marbletable where not marbles > 10
```

Deirdre, Fritz und Isaac sind an der Reihe, die Liste zu füllen. Chris und Hadassah sind nicht enthalten, da sie nicht *bekannt* haben, dass sie zehn Murmeln oder weniger haben. Sollten Sie diese letzte Abfrage ändern in:

```
select list(child) from marbletable where marbles <= 10;
```

das Ergebnis bleibt gleich, da der Ausdruck `NULL <= 10` `UNKNOWN` ergibt. Dies ist nicht dasselbe wie `TRUE`, daher werden Chris und Hadassah nicht aufgeführt. Wenn Sie möchten, dass sie mit den "armen"-Kindern aufgelistet werden, ändern Sie die Abfrage in:

```
select list(child) from marbletable
where marbles <= 10 or marbles is null;
```

Jetzt wird die Suchbedingung für Chris und Hadassah wahr, da "`marbles is null`" in ihrem Fall offensichtlich `TRUE` zurückgibt. Tatsächlich kann die Suchbedingung jetzt für niemanden `NULL` sein.

Zuletzt zwei Beispiele für `SELECT`-Abfragen mit Parametern in der Suche. Es hängt von der Anwendung ab, wie Sie Abfrageparameter definieren sollten und ob dies überhaupt möglich ist. Beachten Sie, dass Abfragen wie diese nicht sofort ausgeführt werden können: Sie müssen zuerst

vorbereitet werden. Nachdem eine parametrisierte Abfrage erstellt wurde, kann der Benutzer (oder der aufrufende Code) Werte für die Parameter bereitstellen und mehrmals ausführen lassen, wobei vor jedem Aufruf neue Werte eingegeben werden. Wie die Werte eingegeben und die Ausführung gestartet wird, bleibt der Anwendung überlassen. In einer GUI-Umgebung gibt der Benutzer typischerweise die Parameterwerte in ein oder mehrere Textfelder ein und klickt dann auf eine Schaltfläche "Ausführen", "Ausführen" oder "Aktualisieren".

```
select name, address, phone frome stores
  where city = ? and class = ?;
```

```
select * from pants
  where model = :model and size = :size and color = :col;
```

Die letzte Abfrage kann nicht direkt an die Engine übergeben werden; die Anwendung muss es zuerst in das andere Format konvertieren und benannte Parameter Positionsparametern zuordnen.

6.1.6. Die `GROUP BY`-Klausel

`GROUP BY` führt Ausgabezeilen, die dieselbe Kombination von Werten in ihrer Elementliste haben, zu einer einzigen Zeile zusammen. Aggregatfunktionen in der Auswahlliste werden auf jede Gruppe einzeln und nicht auf den gesamten Datensatz angewendet.

Wenn die Auswahlliste nur Aggregatspalten enthält oder allgemeiner Spalten, deren Werte nicht von einzelnen Zeilen in der zugrunde liegenden Menge abhängen, ist `GROUP BY` optional. Wenn es weggelassen wird, besteht die endgültige Ergebnismenge von aus einer einzelnen Zeile (vorausgesetzt, dass mindestens eine aggregierte Spalte vorhanden ist).

Wenn die Auswahlliste sowohl Aggregatspalten als auch Spalten enthält, deren Werte pro Zeile variieren können, wird die `GROUP BY`-Klausel obligatorisch.

Syntax

```
SELECT ... FROM ...
  GROUP BY <grouping-item> [, <grouping-item> ...]
  [HAVING <grouped-row-condition>]
  ...

<grouping-item> ::=
    <non-aggr-select-item>
  | <non-aggr-expression>

<non-aggr-select-item> ::=
    column-copy
  | column-alias
  | column-position
```

Tabelle 73. Argumente für die `GROUP BY`-Klausel

Argument	Beschreibung
non-aggr-expression	Jeder nicht aggregierende Ausdruck, der nicht in der SELECT-Liste enthalten ist, d. h. nicht ausgewählte Spalten aus dem Quellsatz oder Ausdrücke, die überhaupt nicht von den Daten im Satz abhängen
column-copy	Eine wörtliche Kopie aus der SELECT-Liste eines Ausdrucks, der keine Aggregatfunktion enthält
column-alias	Der Alias aus der SELECT-Liste eines Ausdrucks (Spalte), der keine Aggregatfunktion enthält
column-position	Die Positionsnummer in der SELECT-Liste eines Ausdrucks (Spalte), der keine Aggregatfunktion enthält

Als allgemeine Faustregel gilt, dass jedes nicht aggregierte Element in der SELECT-Liste auch in der GROUP BY-Liste enthalten sein muss. Sie können dies auf drei Arten tun:

1. Durch wörtliches Kopieren des Artikels aus der Auswahlliste, z.B. “`class`” oder “`'D:' || upper(doccode)`”.
2. Durch Angabe des Spaltenalias, falls vorhanden.
3. Durch Angabe der Spaltenposition als Ganzzahl *literal* zwischen 1 und der Anzahl der Spalten. Ganzzahlwerte, die aus Ausdrücken oder Parameterersetzungen resultieren, sind einfach unveränderlich und werden als solche in der Gruppierung verwendet. Sie haben jedoch keine Auswirkung, da ihr Wert für jede Zeile gleich ist.

Wenn Sie nach einer Spaltenposition gruppieren, wird der Ausdruck an dieser Position intern aus der Auswahlliste kopiert. Wenn es sich um eine Unterabfrage handelt, wird diese Unterabfrage in der Gruppierungsphase erneut ausgeführt. Das heißt, das Gruppieren nach der Spaltenposition, anstatt den Unterabfrageausdruck in der Gruppierungsklausel zu duplizieren, spart Tastenanschläge und Bytes, aber es ist keine Möglichkeit, Verarbeitungszyklen zu sparen!

Zusätzlich zu den erforderlichen Elementen kann die Gruppierungsliste auch Folgendes enthalten:

- Spalten aus der Quelltabelle, die nicht in der Auswahlliste enthalten sind, oder nicht aggregierte Ausdrücke, die auf solchen Spalten basieren. Das Hinzufügen solcher Spalten kann die Gruppen weiter unterteilen. Da sich diese Spalten jedoch nicht in der Auswahlliste befinden, können Sie nicht erkennen, welche aggregierte Zeile welchem Wert in der Spalte entspricht. Wenn Sie also an diesen Informationen interessiert sind, nehmen Sie im Allgemeinen auch die Spalte oder den Ausdruck in die Auswahlliste auf — was Sie zu der Regel zurückbringt: “Jede nicht aggregierte Spalte in der Auswahlliste muss auch in der Gruppierungsliste”.
- Ausdrücke, die nicht von den Daten in der zugrunde liegenden Menge abhängig sind, z. Konstanten, Kontextvariablen, einwertige nicht korrelierte Unterauswahlen usw. Dies wird nur der Vollständigkeit halber erwähnt, da das Hinzufügen solcher Elemente völlig sinnlos ist: Sie beeinflussen die Gruppierung überhaupt nicht. “Harmlose aber nutzlose” Elemente wie diese können auch in der Auswahlliste vorkommen, ohne in die Gruppierungsliste kopiert zu werden.

Beispiele

Wenn die Auswahlliste nur aggregierte Spalten enthält, ist `GROUP BY` nicht obligatorisch:

```
select count(*), avg(age) from students
  where sex = 'M';
```

Dadurch wird eine einzelne Zeile zurückgegeben, die die Anzahl der männlichen Studenten und ihr Durchschnittsalter auflistet. Das Hinzufügen von Ausdrücken, die nicht von Werten in einzelnen Zeilen der Tabelle `STUDENTS` abhängen, ändert daran nichts:

```
select count(*), avg(age), current_date from students
  where sex = 'M';
```

Die Zeile enthält jetzt eine zusätzliche Spalte mit dem aktuellen Datum, aber ansonsten hat sich nichts Wesentliches geändert. Eine `GROUP BY`-Klausel ist weiterhin nicht erforderlich.

In beiden obigen Beispielen ist es jedoch *erlaubt*. Das ist vollkommen gültig:

```
select count(*), avg(age) from students
  where sex = 'M'
  group by class;
```

Dadurch wird für jede Klasse mit Jungen eine Zeile zurückgegeben, in der die Anzahl der Jungen und ihr Durchschnittsalter in dieser bestimmten Klasse aufgeführt sind. (Wenn Sie auch das Feld `current_date` belassen, wird dieser Wert in jeder Zeile wiederholt, was nicht sehr aufregend ist.)

Die obige Abfrage hat jedoch einen großen Nachteil: Sie gibt Ihnen Informationen über die verschiedenen Klassen, aber sie sagt Ihnen nicht, welche Zeile für welche Klasse gilt. Um diese zusätzlichen Informationen zu erhalten, muss die nicht aggregierte Spalte "CLASS" zur Auswahlliste hinzugefügt werden:

```
select class, count(*), avg(age) from students
  where sex = 'M'
  group by class;
```

Jetzt haben wir eine nützliche Abfrage. Beachten Sie, dass das Hinzufügen der Spalte `CLASS` auch die `GROUP BY`-Klausel obligatorisch macht. Wir können diese Klausel nicht mehr löschen, es sei denn, wir entfernen auch `CLASS` aus der Spaltenliste.

Die Ausgabe unserer letzten Abfrage kann etwa so aussehen:

CLASS	COUNT	AVG
2A	12	13.5
2B	9	13.9

CLASS	COUNT	AVG
3A	11	14.6
3B	12	14.4
…	…	…

Die Überschriften “COUNT” und “AVG” sind wenig aussagekräftig. In einem einfachen Fall wie diesem kommen Sie vielleicht damit durch, aber im Allgemeinen sollten Sie Aggregatspalten einen aussagekräftigen Namen geben, indem Sie sie mit einem Alias versehen:

```
select class,
       count(*) as num_boys,
       avg(age) as boys_avg_age
  from students
  where sex = 'M'
  group by class;
```

Wie Sie sich vielleicht an der formalen Syntax der Spaltenliste erinnern, ist das Schlüsselwort `AS` optional.

Das Hinzufügen weiterer nicht-aggregierter (oder besser: zeilenabhängiger) Spalten erfordert auch das Hinzufügen dieser zur `GROUP BY`-Klausel. Zum Beispiel möchten Sie vielleicht die oben genannten Informationen auch für Mädchen sehen; und vielleicht möchten Sie auch zwischen Internats- und Tagesschülern unterscheiden:

```
select class,
       sex,
       boarding_type,
       count(*) as number,
       avg(age) as avg_age
  from students
  group by class, sex, boarding_type;
```

Dies kann zu folgendem Ergebnis führen:

CLASS	SEX	BOARDING_TYPE	NUMBER	AVG_AGE
2A	F	BOARDING	9	13.3
2A	F	DAY	6	13.5
2A	M	BOARDING	7	13.6
2A	M	DAY	5	13.4
2B	F	BOARDING	11	13.7
2B	F	DAY	5	13.7
2B	M	BOARDING	6	13.8

CLASS	SEX	BOARDING_TYPE	NUMBER	AVG_AGE
…	…	…	…	…

Jede Zeile in der Ergebnismenge entspricht einer bestimmten Kombination der Spalten `CLASS`, `SEX` und `BOARDING_TYPE`. Die aggregierten Ergebnisse – Anzahl und Durchschnittsalter – werden für jede dieser eher spezifischen Gruppen einzeln angegeben. In einer Abfrage wie dieser sehen Sie keine Gesamtsumme für Jungen als Ganzes oder Tagesschüler als Ganzes. Das ist der Kompromiss: Je mehr nicht aggregierte Spalten Sie hinzufügen, desto mehr können Sie sehr spezifische Gruppen lokalisieren, aber desto mehr verlieren Sie auch den Überblick. Natürlich können Sie die "gröberen" Aggregate weiterhin durch separate Abfragen erhalten.

`HAVING`

So wie eine 'WHERE'-Klausel die Zeilen in einem Datensatz auf diejenigen beschränkt, die die Suchbedingung erfüllen, so erlegt die 'HAVING'-Unterklausel Beschränkungen für die aggregierten Zeilen in einer gruppierten Menge auf. `HAVING` ist optional und kann nur in Verbindung mit `GROUP BY` verwendet werden.

Die Bedingung(en) in der `HAVING`-Klausel können sich beziehen auf:

- Jede aggregierte Spalte in der Auswahlliste. Dies ist der am häufigsten verwendete Fall.
- Jeder aggregierte Ausdruck, der nicht in der Auswahlliste enthalten ist, aber im Kontext der Abfrage zulässig ist. Dies ist manchmal auch nützlich.
- Jede Spalte in der `GROUP BY`-Liste. Obwohl es legal ist, ist es effizienter, diese nicht aggregierten Daten zu einem früheren Zeitpunkt zu filtern: in der WHERE-Klausel.
- Jeder Ausdruck, dessen Wert nicht vom Inhalt des Datasets abhängt (wie eine Konstante oder eine Kontextvariable). Dies ist gültig, aber völlig sinnlos, da es entweder die gesamte Menge unterdrückt oder unberührt lässt, basierend auf Bedingungen, die nichts mit der Menge selbst zu tun haben.

Eine `HAVING`-Klausel kann *nicht* enthalten:

- Nicht aggregierte Spaltenausdrücke, die nicht in der `GROUP BY`-Liste enthalten sind.
- Spaltenpositionen. Eine ganze Zahl in der `HAVING`-Klausel ist nur eine ganze Zahl.
- Spaltenaliase – nicht einmal, wenn sie in der `GROUP BY`-Klausel vorkommen!

Beispiele

Aufbauend auf unseren früheren Beispielen könnte dies verwendet werden, um kleine Schülergruppen zu überspringen:

```
select class,
       count(*) as num_boys,
       avg(age) as boys_avg_age
  from students
  where sex = 'M'
  group by class
```

```
  having count(*) >= 5;
```

So wählen Sie nur Gruppen mit einer Mindestaltersspanne aus:

```
select class,
       count(*) as num_boys,
       avg(age) as boys_avg_age
  from students
  where sex = 'M'
  group by class
  having max(age) - min(age) > 1.2;
```

Beachten Sie, dass Sie, wenn Sie wirklich an diesen Informationen interessiert sind, normalerweise `min(age)` und `max(age)` einschließen würden – oder den Ausdruck „`` `max(age) - min(age)` `` – auch in der Auswahlliste!

Um nur 3. Klassen einzubeziehen:

```
select class,
       count(*) as num_boys,
       avg(age) as boys_avg_age
  from students
  where sex = 'M'
  group by class
  having class starting with '3';
```

Besser wäre es, diese Bedingung in die WHERE-Klausel zu verschieben:

```
select class,
       count(*) as num_boys,
       avg(age) as boys_avg_age
  from students
  where sex = 'M' and class starting with '3'
  group by class;
```

6.1.7. Die `PLAN`-Klausel

Die `PLAN`-Klausel ermöglicht es dem Benutzer, einen Datenabrufplan zu übermitteln und damit den Plan zu überschreiben, den der Optimierer automatisch generiert hätte.

Syntax

```
PLAN <plan-expr>

<plan-expr> ::=
    (<plan-item> [, <plan-item> ...])
  | <sorted-item>
```

```
  | <joined-item>
  | <merged-item>
  | <hash-item>

<sorted-item> ::= SORT (<plan-item>)

<joined-item> ::=
  JOIN (<plan-item>, <plan-item> [, <plan-item> ...])

<merged-item> ::=
  [SORT] MERGE (<sorted-item>, <sorted-item> [, <sorted-item> ...])

<hash-item> ::=
  HASH (<plan-item>, <plan-item> [, <plan-item> ...])

<plan-item> ::= <basic-item> | <plan-expr>

<basic-item> ::=
  <relation> { NATURAL
             | INDEX (<indexlist>)
             | ORDER index [INDEX (<indexlist>)] }

<relation> ::= table | view [table]

<indexlist> ::= index [, index ...]
```

Tabelle 74. Argumente für die PLAN*-Klausel*

Argument	Beschreibung
table	Tabellenname oder sein Alias
view	Ansichtsname
index	Indexname

Jedes Mal, wenn ein Benutzer eine Abfrage an die Firebird-Engine sendet, berechnet der Optimierer eine Datenabrufstrategie. Die meisten Firebird-Clients können diesen Abrufplan für den Benutzer sichtbar machen. In Firebirds eigenem Dienstprogramm `isql` geschieht dies mit dem Befehl `SET PLAN ON`. Wenn Sie Abfragepläne untersuchen, anstatt Abfragen auszuführen, zeigt `SET PLANONLY ON` den Plan an, ohne die Abfrage auszuführen. Verwenden Sie `SET PLANONLY OFF`, um die Abfrage auszuführen und den Plan anzuzeigen.

Einen detaillierteren Plan erhalten Sie, wenn Sie einen erweiterten Plan aktivieren. In *isql* kann dies mit `SET EXPLAIN ON` erfolgen. Der erweiterte Plan zeigt detailliertere Informationen über die vom Optimierer verwendeten Zugriffsmethoden an, kann jedoch nicht in die PLAN-Klausel einer Anweisung aufgenommen werden. Die Beschreibung des erweiterten Plans geht über den Rahmen dieser Sprachreferenz hinaus.

In den meisten Situationen können Sie darauf vertrauen, dass Firebird den optimalen Abfrageplan

für Sie auswählt. Wenn Sie jedoch komplizierte Abfragen haben, deren Leistung nicht ausreicht, kann es sich durchaus lohnen, den Plan zu prüfen und zu prüfen, ob Sie ihn verbessern können.

Einfache Pläne

Die einfachsten Pläne bestehen nur aus einem Relationsnamen gefolgt von einer Abrufmethode. Zum Beispiel für eine unsortierte Einzeltabellenauswahl ohne `WHERE`-Klausel:

```
select * from students
  plan (students natural);
```

Erweiterter Plan:

```
Select Expression
  -> Table "STUDENTS" Full Scan
```

Wenn es eine `WHERE`- oder eine `HAVING`-Klausel gibt, können Sie den Index angeben, der für die Suche nach Übereinstimmungen verwendet werden soll:

```
select * from students
  where class = '3C'
  plan (students index (ix_stud_class));
```

Erweiterter Plan:

```
Select Expression
  -> Filter
    -> Table "STUDENTS" Access By ID
      -> Bitmap
        -> Index "IX_STUD_CLASS" Range Scan (full match)
```

Die Direktive `INDEX` wird auch für Join-Bedingungen verwendet (wird etwas später besprochen). Es kann eine durch Kommas getrennte Liste von Indizes enthalten.

`ORDER` gibt den Index zum Sortieren der Menge an, wenn eine `ORDER BY`- oder `GROUP BY`-Klausel vorhanden ist:

```
select * from students
  plan (students order pk_students)
  order by id;
```

Erweiterter plan:

```
Select Expression
```

```
  -> Table "STUDENTS" Access By ID
    -> Index "PK_STUDENTS" Full Scan
```

`ORDER` und `INDEX` können kombiniert werden:

```
select * from students
  where class >= '3'
  plan (students order pk_students index (ix_stud_class))
  order by id;
```

Erweiterter Plan:

```
Select Expression
  -> Filter
    -> Table "STUDENTS" Access By ID
      -> Index "PK_STUDENTS" Full Scan
        -> Bitmap
          -> Index "IX_STUD_CLASS" Range Scan (lower bound: 1/1)
```

Es ist vollkommen in Ordnung, wenn `ORDER` und `INDEX` denselben Index angeben:

```
select * from students
  where class >= '3'
  plan (students order ix_stud_class index (ix_stud_class))
  order by class;
```

Erweiterter Plan:

```
Select Expression
  -> Filter
    -> Table "STUDENTS" Access By ID
      -> Index "IX_STUD_CLASS" Range Scan (lower bound: 1/1)
        -> Bitmap
          -> Index "IX_STUD_CLASS" Range Scan (lower bound: 1/1)
```

Um Sets zu sortieren, wenn kein verwendbarer Index verfügbar ist (oder wenn Sie seine Verwendung unterdrücken möchten), lassen Sie `ORDER` weg und stellen Sie dem Planausdruck `SORT` voran:

```
select * from students
  plan sort (students natural)
  order by name;
```

Erweiterter Plan:

```
Select Expression
  -> Sort (record length: 128, key length: 56)
    -> Table "STUDENTS" Full Scan
```

Oder wenn ein Index für die Suche verwendet wird:

```
select * from students
  where class >= '3'
  plan sort (students index (ix_stud_class))
  order by name;
```

Erweiterter Plan:

```
elect Expression
  -> Sort (record length: 136, key length: 56)
    -> Filter
      -> Table "STUDENTS" Access By ID
        -> Bitmap
          -> Index "IX_STUD_CLASS" Range Scan (lower bound: 1/1)
```

Beachten Sie, dass `SORT` im Gegensatz zu `ORDER` außerhalb der Klammern steht. Dies spiegelt die Tatsache wider, dass die Datenzeilen ungeordnet abgerufen und anschließend von der Engine sortiert werden.

Geben Sie bei der Auswahl aus einer Ansicht die Ansicht und die betreffende Tabelle an. Wenn Sie beispielsweise eine Ansicht `FRESHMEN` haben, die nur die Erstsemester auswählt:

```
select * from freshmen
  plan (freshmen students natural);
```

Erweiterter Plan:

```
Select Expression
  -> Table "STUDENTS" as "FRESHMEN" Full Scan
```

Oder zum Beispiel:

```
select * from freshmen
  where id > 10
  plan sort (freshmen students index (pk_students))
  order by name desc;
```

Erweiterter Plan:

```
Select Expression
  -> Sort (record length: 144, key length: 24)
    -> Filter
      -> Table "STUDENTS" as "FRESHMEN" Access By ID
        -> Bitmap
          -> Index "PK_STUDENTS" Range Scan (lower bound: 1/1)
```

Wenn eine Tabelle oder Ansicht mit einem Alias versehen wurde, muss der Alias, nicht der ursprüngliche Name, in der PLAN-Klausel verwendet werden.

Zusammengesetzte Pläne

Bei einem Join können Sie den Index angeben, der für den Abgleich verwendet werden soll. Sie müssen auch die JOIN-Direktive für die beiden Streams im Plan verwenden:

```
select s.id, s.name, s.class, c.mentor
  from students s
  join classes c on c.name = s.class
  plan join (s natural, c index (pk_classes));
```

Erweiterter Plan:

```
Select Expression
  -> Nested Loop Join (inner)
    -> Table "STUDENTS" as "S" Full Scan
    -> Filter
      -> Table "CLASSES" as "C" Access By ID
        -> Bitmap
          -> Index "PK_CLASSES" Unique Scan
```

Dieselbe Verknüpfung, sortiert nach einer indizierten Spalte:

```
select s.id, s.name, s.class, c.mentor
  from students s
  join classes c on c.name = s.class
  plan join (s order pk_students, c index (pk_classes))
  order by s.id;
```

Erweiterter Plan:

```
Select Expression
  -> Nested Loop Join (inner)
    -> Table "STUDENTS" as "S" Access By ID
      -> Index "PK_STUDENTS" Full Scan
    -> Filter
```

```
      -> Table "CLASSES" as "C" Access By ID
        -> Bitmap
          -> Index "PK_CLASSES" Unique Scan
```

Und für eine nicht indizierte Spalte:

```
select s.id, s.name, s.class, c.mentor
  from students s
  join classes c on c.name = s.class
  plan sort (join (s natural, c index (pk_classes)))
  order by s.name;
```

Erweiterter Plan:

```
Select Expression
  -> Sort (record length: 152, key length: 12)
    -> Nested Loop Join (inner)
      -> Table "STUDENTS" as "S" Full Scan
      -> Filter
        -> Table "CLASSES" as "C" Access By ID
          -> Bitmap
            -> Index "PK_CLASSES" Unique Scan
```

Mit einer hinzugefügten Suchbedingung:

```
select s.id, s.name, s.class, c.mentor
  from students s
  join classes c on c.name = s.class
  where s.class <= '2'
  plan sort (join (s index (fk_student_class), c index (pk_classes)))
  order by s.name;
```

Erweiterter Plan:

```
Select Expression
  -> Sort (record length: 152, key length: 12)
    -> Nested Loop Join (inner)
      -> Filter
        -> Table "STUDENTS" as "S" Access By ID
          -> Bitmap
            -> Index "FK_STUDENT_CLASS" Range Scan (lower bound: 1/1)
      -> Filter
        -> Table "CLASSES" as "C" Access By ID
          -> Bitmap
            -> Index "PK_CLASSES" Unique Scan
```

Als Left Outer Join:

```
select s.id, s.name, s.class, c.mentor
  from classes c
  left join students s on c.name = s.class
  where s.class <= '2'
  plan sort (join (c natural, s index (fk_student_class)))
  order by s.name;
```

Erweiterter Plan:

```
Select Expression
  -> Sort (record length: 192, key length: 56)
    -> Filter
      -> Nested Loop Join (outer)
        -> Table "CLASSES" as "C" Full Scan
        -> Filter
          -> Table "STUDENTS" as "S" Access By ID
            -> Bitmap
              -> Index "FK_STUDENT_CLASS" Range Scan (full match)
```

Wenn keine Indizes verfügbar sind, die der Join-Bedingung entsprechen (oder wenn Sie sie nicht verwenden möchten), können Sie die Streams mit der Methode `HASH` oder `MERGE` verbinden.

Um eine Verbindung mit der `HASH`-Methode im Plan herzustellen, wird die `HASH`-Direktive anstelle der `JOIN`-Direktive verwendet. In diesem Fall wird der kleinere (sekundäre) Strom vollständig in einem internen Puffer materialisiert. Beim Lesen dieses sekundären Streams wird eine Hash-Funktion angewendet und ein Paar *{Hash, Zeiger auf Puffer}* in eine Hash-Tabelle geschrieben. Dann wird der primäre Stream gelesen und sein Hash-Schlüssel wird gegen die Hash-Tabelle getestet.

```
select *
  from students s
  join classes c on c.cookie = s.cookie
  plan hash (c natural, s natural)
```

Erweiterter Plan:

```
Select Expression
  -> Filter
    -> Hash Join (inner)
      -> Table "STUDENTS" as "S" Full Scan
      -> Record Buffer (record length: 145)
        -> Table "CLASSES" as "C" Full Scan
```

Für einen 'MERGE'-Join muss der Plan zuerst beide Streams in deren Join-Spalte(n) sortieren und

dann zusammenführen. Dies wird mit der SORT-Direktive (die wir bereits gesehen haben) und MERGE statt JOIN erreicht:

```
select * from students s
  join classes c on c.cookie = s.cookie
  plan merge (sort (c natural), sort (s natural));
```

Das Hinzufügen einer ORDER BY-Klausel bedeutet, dass das Ergebnis der Zusammenführung ebenfalls sortiert werden muss:

```
select * from students s
  join classes c on c.cookie = s.cookie
  plan sort (merge (sort (c natural), sort (s natural)))
  order by c.name, s.id;
```

Schließlich fügen wir eine Suchbedingung für zwei indizierbare Spalten der Tabelle STUDENTS hinzu:

```
select * from students s
  join classes c on c.cookie = s.cookie
  where s.id < 10 and s.class <= '2'
  plan sort (merge (sort (c natural),
                    sort (s index (pk_students, fk_student_class))))
  order by c.name, s.id;
```

Wie aus der formalen Syntaxdefinition hervorgeht, können JOINs und MERGEs im Plan mehr als zwei Streams kombinieren. Außerdem kann jeder Planausdruck als Planelement in einem umfassenden Plan verwendet werden. Dies bedeutet, dass Pläne bestimmter komplizierter Abfragen verschiedene Verschachtelungsebenen haben können.

Schließlich können Sie statt MERGE auch SORT MERGE schreiben. Da dies absolut keinen Unterschied macht und zu Verwirrung mit “real” SORT-Direktiven führen kann (die einen Unterschied machen), ist es wahrscheinlich am besten, beim einfachen MERGE zu bleiben.

Neben dem Plan für die Hauptabfrage können Sie für jede Unterabfrage einen Plan angeben. Die folgende Abfrage mit mehreren Plänen funktioniert beispielsweise:

```
select *
from color
where exists (
  select *
  from hors
  where horse.code_color = color.code_color
  plan (horse index (fk_horse_color)))
plan (color natural)
```

Gelegentlich akzeptiert der Optimierer einen Plan und folgt ihm dann nicht,

obwohl er ihn nicht als ungültig zurückweist. Ein solches Beispiel war

```
MERGE (unsorted stream, unsorted stream)
```

Es ist ratsam, einen solchen Plan als “veraltet” zu behandeln.

6.1.8. UNION

Die UNION-Klausel verkettet zwei oder mehr Datensätze und erhöht somit die Anzahl der Zeilen, aber nicht die Anzahl der Spalten. Datensätze, die an einer UNION teilnehmen, müssen die gleiche Anzahl von Spalten haben und die Spalten an den entsprechenden Positionen müssen vom gleichen Typ sein. Abgesehen davon können sie völlig unabhängig sein.

Standardmäßig unterdrückt eine Vereinigung doppelte Zeilen. UNION ALL zeigt alle Zeilen, einschließlich aller Duplikate. Das optionale Schlüsselwort DISTINCT macht das Standardverhalten explizit.

Syntax

```
<union> ::=
  <individual-select>
  UNION [{DISTINCT | ALL}]
  <individual-select>
  [
    [UNION [{DISTINCT | ALL}]
    <individual-select>
    ...
  ]
  [<union-wide-clauses>]

<individual-select> ::=
  SELECT
  [TRANSACTION name]
  [FIRST m] [SKIP n]
  [{DISTINCT | ALL}] <columns>
  [INTO <host-varlist>]
  FROM <source> [[AS] alias]
  [<joins>]
  [WHERE <condition>]
  [GROUP BY <grouping-list>
  [HAVING <aggregate-condition>]]
  [PLAN <plan-expr>]

<union-wide-clauses> ::=
  [ORDER BY <ordering-list>]
  [{ ROWS <m> [TO <n>]
   | [OFFSET n {ROW | ROWS}]
     [FETCH {FIRST | NEXT} [m] {ROW | ROWS} ONLY]
  }]
```

```
  [FOR UPDATE [OF <columns>]]
  [WITH LOCK]
  [INTO <PSQL-varlist>]
```

Unions beziehen ihre Spaltennamen aus der *ersten* Auswahlabfrage. Wenn Sie Union-Spalten mit einem Alias versehen möchten, tun Sie dies in der Spaltenliste des obersten `SELECT`. Aliase in anderen teilnehmenden Selects sind erlaubt und können sogar nützlich sein, werden aber nicht auf Gewerkschaftsebene verbreitet.

Wenn eine Union eine `ORDER BY`-Klausel hat, sind die einzigen zulässigen Sortierelemente Integer-Literale, die 1-basierte Spaltenpositionen angeben, optional gefolgt von einem `ASC`/`DESC` und/oder einem `NULLS {FIRST | LAST}`-Anweisung. Dies impliziert auch, dass Sie eine Union nicht nach etwas sortieren können, das keine Spalte in der Union ist. (Sie können es jedoch in eine abgeleitete Tabelle einschließen, die Ihnen alle üblichen Sortieroptionen zurückgibt.)

Unions sind in Unterabfragen jeglicher Art erlaubt und können selbst Unterabfragen enthalten. Sie können auch Joins enthalten und an einem Join teilnehmen, wenn sie in eine abgeleitete Tabelle eingeschlossen sind.

Beispiele

Diese Abfrage präsentiert Informationen aus verschiedenen Musiksammlungen in einem Datensatz unter Verwendung von Unions:

```
select id, title, artist, length, 'CD' as medium
  from cds
union
select id, title, artist, length, 'LP'
  from records
union
select id, title, artist, length, 'MC'
  from cassettes
order by 3, 2  -- artist, title;
```

Wenn `id`, `title`, `artist` und `length` die einzigen beteiligten Felder in den Tabellen sind, kann die Abfrage auch so geschrieben werden:

```
select c.*, 'CD' as medium
  from cds c
union
select r.*, 'LP'
  from records r
union
select c.*, 'MC'
  from cassettes c
order by 3, 2  -- artist, title;
```

Die Qualifizierung der “Sterne” ist hier notwendig, da sie nicht das einzige Element in der

Spaltenliste sind. Beachten Sie, dass die Aliase “c” in der ersten und dritten Auswahl nicht miteinander in Konflikt geraten: Ihre Gültigkeitsbereiche sind nicht unionsweit, sondern gelten nur für ihre jeweiligen select-Abfragen.

Die nächste Abfrage ruft Namen und Telefonnummern von Übersetzern und Korrektoren ab. Übersetzer, die auch als Korrektoren tätig sind, erscheinen nur einmal in der Ergebnismenge, sofern ihre Telefonnummer in beiden Tabellen gleich ist. Das gleiche Ergebnis kann ohne `DISTINCT` erreicht werden. Mit `ALL` würden diese Personen zweimal erscheinen.

```
select name, phone from translators
  union distinct
select name, telephone from proofreaders;
```

Eine `UNION` innerhalb einer Unterabfrage:

```
select name, phone, hourly_rate from clowns
where hourly_rate < all
  (select hourly_rate from jugglers
     union
   select hourly_rate from acrobats)
order by hourly_rate;
```

6.1.9. ORDER BY

Wenn eine `SELECT`-Anweisung ausgeführt wird, wird die Ergebnismenge in keiner Weise sortiert. Es kommt oft vor, dass Zeilen chronologisch sortiert erscheinen, einfach weil sie in der gleichen Reihenfolge zurückgegeben werden, in der sie durch `INSERT`-Anweisungen zur Tabelle hinzugefügt wurden. Darauf sollten Sie sich nicht verlassen: Die Reihenfolge kann sich je nach Plan oder Aktualisierungen von Zeilen usw. ändern. Um eine explizite Sortierreihenfolge für die Mengenangabe anzugeben, wird eine `ORDER BY`-Klausel verwendet.

Syntax

```
SELECT ... FROM ...
...
ORDER BY <ordering-item> [, <ordering-item> ⋯]

<ordering-item> ::=
  {col-name | col-alias | col-position | <expression>}
  [COLLATE collation-name]
  [ASC[ENDING] | DESC[ENDING]]
  [NULLS {FIRST|LAST}]
```

Tabelle 75. Argumente für die `ORDER BY`*-Klausel*

Argument	Beschreibung
col-name	Vollständiger Spaltenname

Argument	Beschreibung
col-alias	Spaltenalias
col-position	Spaltenposition in der SELECT-Liste
expression	Beliebiger Ausdruck
collation-name	Collationsname (Sortierreihenfolge für Stringdatentypen)

Der ORDER BY besteht aus einer durch Kommas getrennten Liste der Spalten, nach denen der Ergebnisdatensatz sortiert werden soll. Die Sortierreihenfolge kann durch den Namen der Spalte angegeben werden — aber nur, wenn die Spalte zuvor nicht als Alias in der SELECT-Spaltenliste angegeben wurde. Der Alias muss verwendet werden, wenn er in der Auswahlliste verwendet wurde. Die ordinale Positionsnummer der Spalte in der SELECT-Spaltenliste, der der Spalte in der SELECT-Liste mit Hilfe des Schlüsselworts AS gegebene Alias oder die Nummer der Spalte in der SELECT-Liste kann uneingeschränkt verwendet werden.

Die drei Ausdrucksformen der Spalten für die Sortierreihenfolge können in derselben ORDER BY -Klausel gemischt werden. Beispielsweise kann eine Spalte in der Liste durch ihren Namen und eine andere Spalte durch ihre Nummer angegeben werden.

Wenn Sie nach Spaltenposition oder Alias sortieren, wird der dieser Position (Alias) entsprechende Ausdruck aus der SELECT-Liste kopiert. Dies gilt auch für Unterabfragen, daher wird die Unterabfrage mindestens zweimal ausgeführt.

Wenn Sie die Spaltenposition verwenden, um die Sortierreihenfolge für eine Abfrage des Stils SELECT * anzugeben, erweitert der Server das Sternchen auf die vollständige Spaltenliste, um die Spalten für die Sortierung zu bestimmen. Es wird jedoch als „schlechte Praxis“ angesehen, geordnete Sets auf diese Weise zu entwerfen.

Sortierrichtung

Das Schlüsselwort ASCENDING – normalerweise abgekürzt mit ASC – gibt eine Sortierrichtung von der niedrigsten zur höchsten an. ASCENDING ist die Standardsortierrichtung.

Das Schlüsselwort DESCENDING — normalerweise abgekürzt mit DESC — gibt eine Sortierrichtung von der höchsten zur niedrigsten an.

Die Angabe einer aufsteigenden Reihenfolge für eine Spalte und einer absteigenden Reihenfolge für eine andere ist zulässig.

Sortierreihenfolge

Das Schlüsselwort COLLATE gibt die Sortierreihenfolge für eine Zeichenfolgenspalte an, wenn Sie eine andere Sortierung als die normale für diese Spalte benötigen. Die normale Sortierreihenfolge ist entweder die Standardreihenfolge für den Datenbankzeichensatz oder die explizit in der Spaltendefinition festgelegte.

NULLS-Position

Das Schlüsselwort `NULLS` definiert, wo NULL in der zugeordneten Spalte in der Sortierreihenfolge liegt: `NULLS FIRST` platziert die Zeilen mit der `NULL`-Spalte *über* Zeilen geordnet nach dem Wert dieser Spalte; `NULLS LAST` platziert diese Zeilen *nach* den geordneten Zeilen.

`NULLS FIRST` ist die Vorgabe.

Sortierung von `UNION`s

Die diskreten Abfragen, die zu einer `UNION` beitragen, können keine `ORDER BY`-Klausel annehmen. Die einzige Möglichkeit besteht darin, die gesamte Ausgabe zu sortieren, indem eine `ORDER BY`-Klausel am Ende der Gesamtabfrage verwendet wird.

Die einfachste — und in manchen Fällen die einzige — Methode zum Angeben der Sortierreihenfolge ist die Ordinalspaltenposition. Es ist jedoch auch zulässig, die Spaltennamen oder Aliase aus der ersten beitragenden Abfrage *nur* zu verwenden.

Für dieses globale Set stehen die Direktiven `ASC`/`DESC` und/oder `NULLS` zur Verfügung.

Wenn eine diskrete Sortierung innerhalb der beitragenden Menge erforderlich ist, kann die Verwendung abgeleiteter Tabellen oder allgemeiner Tabellenausdrücke für diese Mengen eine Lösung sein.

Beispiele für `ORDER BY`

Sortieren der Ergebnismenge in aufsteigender Reihenfolge, Sortierung nach den Spalten `RDB$CHARACTER_SET_ID` und `RDB$COLLATION_ID` der Tabelle `RDB$COLLATIONS`:

```
SELECT
  RDB$CHARACTER_SET_ID AS CHARSET_ID,
  RDB$COLLATION_ID AS COLL_ID,
  RDB$COLLATION_NAME AS NAME
FROM RDB$COLLATIONS
ORDER BY RDB$CHARACTER_SET_ID, RDB$COLLATION_ID;
```

Das gleiche, aber nach den Spaltenaliasen sortiert:

```
SELECT
  RDB$CHARACTER_SET_ID AS CHARSET_ID,
  RDB$COLLATION_ID AS COLL_ID,
  RDB$COLLATION_NAME AS NAME
FROM RDB$COLLATIONS
ORDER BY CHARSET_ID, COLL_ID;
```

Sortieren der Ausgabedaten nach den Spaltenpositionsnummern:

```
SELECT
```

```
  RDB$CHARACTER_SET_ID AS CHARSET_ID,
  RDB$COLLATION_ID AS COLL_ID,
  RDB$COLLATION_NAME AS NAME
FROM RDB$COLLATIONS
ORDER BY 1, 2;
```

Sortieren einer `SELECT *`-Abfrage nach Positionsnummern — möglich, aber *böse* und nicht empfohlen:

```
SELECT *
FROM RDB$COLLATIONS
ORDER BY 3, 2;
```

Sortieren nach der zweiten Spalte in der `BOOKS`-Tabelle oder — wenn `BOOKS` nur eine Spalte hat — der `FILMS.DIRECTOR`-Spalte:

```
SELECT
    BOOKS.*,
    FILMS.DIRECTOR
FROM BOOKS, FILMS
ORDER BY 2;
```

Sortieren in absteigender Reihenfolge nach den Werten der Spalte `PROCESS_TIME`, wobei `NULL`s am Anfang der Menge stehen:

```
SELECT *
FROM MSG
ORDER BY PROCESS_TIME DESC NULLS FIRST;
```

Sortieren der Menge, die durch eine `UNION` von zwei Abfragen erhalten wurde. Die Ergebnisse werden in absteigender Reihenfolge nach den Werten in der zweiten Spalte sortiert, mit `NULL`s am Ende der Menge; und in aufsteigender Reihenfolge für die Werte der ersten Spalte mit `NULL`s am Anfang.

```
SELECT
  DOC_NUMBER, DOC_DATE
FROM PAYORDER
UNION ALL
SELECT
  DOC_NUMBER, DOC_DATE
FROM BUDGORDER
ORDER BY 2 DESC NULLS LAST, 1 ASC NULLS FIRST;
```

6.1.10. ROWS

Verwendet für

Abrufen eines Zeilenabschnitts aus einer geordneten Menge

Verfügbar in

DSQL, PSQL

Syntax

```
SELECT <columns> FROM ...
  [WHERE ...]
  [ORDER BY ...]
  ROWS m [TO n]
```

Tabelle 76. Argumente für die ROWS-Klausel

Argument	Beschreibung
m, n	Beliebige Integer-Ausdrücke

ROWS ist keine Standard-Syntax

ROWS ist eine Firebird-spezifische Klausel. Verwenden Sie nach Möglichkeit die SQL-Standardsyntax OFFSET, FETCH.

Begrenzt die Anzahl der Zeilen, die von der SELECT-Anweisung zurückgegeben werden, auf eine angegebene Anzahl oder einen bestimmten Bereich.

Die ROWS-Klausel erfüllt auch die gleiche Aufgabe wie die FIRST- und SKIP-Klauseln, ist jedoch nicht SQL-kompatibel. Im Gegensatz zu FIRST und SKIP sowie OFFSET und FETCH akzeptieren die ROWS- und TO-Klauseln jede Art von Integer-Ausdruck als Argumente ohne Klammern. Natürlich können für verschachtelte Auswertungen innerhalb des Ausdrucks immer noch Klammern benötigt werden, und eine Unterabfrage muss immer in Klammern eingeschlossen werden.

- Die Nummerierung der Zeilen im Zwischensatz – der Gesamtsatz, der auf der Festplatte zwischengespeichert wird, bevor der „slice“ extrahiert wird – beginnt bei 1.
- OFFSET/FETCH, FIRST/SKIP und ROWS können alle ohne die ORDER BY-Klausel verwendet werden, obwohl dies selten sinnvoll ist – außer vielleicht, wenn Sie dies tun möchten Schauen Sie sich die Tabellendaten kurz an und kümmern Sie sich nicht darum, dass die Zeilen in einer nicht deterministischen Reihenfolge vorliegen. Zu diesem Zweck würde eine Abfrage wie “SELECT * FROM TABLE1 ROWS 20” die ersten 20 Zeilen zurückgeben statt einer ganzen Tabelle, die ziemlich groß sein könnte.

Der Aufruf von ROWS m ruft die ersten *m* Datensätze aus der angegebenen Menge ab.

Merkmale der Verwendung von `ROWS m` ohne eine `TO`-Klausel:

- Wenn *m* größer als die Gesamtzahl der Datensätze im Zwischendatensatz ist, wird der gesamte Satz zurückgegeben
- Wenn *m* = 0, wird eine leere Menge zurückgegeben
- Wenn *m* < 0, schlägt der Aufruf der SELECT-Anweisung mit einem Fehler fehl

Der Aufruf von `ROWS m TO n` ruft die Zeilen aus der Menge ab, beginnend bei Zeile *m* und endend nach Zeile *n* — die Menge ist inklusiv.

Merkmale der Verwendung von `ROWS m` mit einer `TO`-Klausel:

- Wenn *m* größer als die Gesamtzahl der Zeilen in der Zwischenmenge ist und *n* >= *m*, wird eine leere Menge zurückgegeben
- Wenn *m* nicht größer als *n* und *n* größer als die Gesamtzahl der Zeilen in der Zwischenmenge ist, wird die Ergebnismenge auf Zeilen beginnend mit *m* bis zum Ende der Menge begrenzt
- Wenn *m* < 1 und *n* < 1 ist, schlägt der Aufruf der `SELECT`-Anweisung mit einem Fehler fehl
- Wenn *n* = *m* - 1, wird eine leere Menge zurückgegeben
- Wenn *n* < *m* - 1, schlägt der Aufruf der SELECT-Anweisung mit einem Fehler fehl

Verwenden einer `TO`-Klausel ohne eine `ROWS`-Klausel:

Während `ROWS` die `FIRST`- und `SKIP`-Syntax ersetzt, gibt es eine Situation, in der die `ROWS`-Syntax nicht das gleiche Verhalten bietet: Die Angabe von `SKIP n` allein gibt den gesamten Zwischensatz zurück, ohne das erste *n* Reihen. Die Syntax von `ROWS ⋯ TO` benötigt dazu ein wenig Hilfe.

Bei der `ROWS`-Syntax benötigen Sie eine `ROWS`-Klausel *in Verbindung mit* der `TO`-Klausel und machen das zweite Argument (*n*) bewusst größer als die Größe des Zwischendatensatzes. Dies wird erreicht, indem ein Ausdruck für *n* erstellt wird, der eine Unterabfrage verwendet, um die Anzahl der Zeilen im Zwischensatz abzurufen und 1 hinzufügt.

Ersetzen von `FIRST/SKIP` und `OFFSET/FETCH`

Die `ROWS`-Klausel kann anstelle der SQL-Standard-`OFFSET/FETCH`- oder Nicht-Standard-`FIRST/SKIP` -Klauseln verwendet werden, außer wenn nur `OFFSET` oder `SKIP` verwendet wird, dass ist, wenn die gesamte Ergebnismenge zurückgegeben wird, außer dass die angegebene Anzahl von Zeilen vom Anfang übersprungen wird.

Um dieses Verhalten mit `ROWS` zu implementieren, müssen Sie die `TO`-Klausel mit einem Wert angeben, der größer als die Größe der zurückgegebenen Ergebnismenge ist.

Mischen von `ROWS` und `FIRST/SKIP` oder `OFFSET/FETCH`

Die `ROWS`-Syntax kann nicht mit `FIRST/SKIP` oder `OFFSET/FETCH` im selben `SELECT`-Ausdruck gemischt werden. Die Verwendung der unterschiedlichen Syntaxen in verschiedenen Unterabfragen in derselben Anweisung ist zulässig.

ROWS-Syntax in UNION-Abfragen

Wenn ROWS in einer UNION-Abfrage verwendet wird, wird die ROWS-Direktive auf die vereinigte Menge angewendet und muss nach der letzten SELECT-Anweisung platziert werden.

Wenn die Notwendigkeit besteht, die von einer oder mehreren SELECT-Anweisungen innerhalb von UNION zurückgegebenen Teilmengen zu begrenzen, gibt es mehrere Optionen:

1. Verwenden Sie die FIRST/SKIP-Syntax in diesen SELECT-Anweisungen — beachten Sie, dass eine Sortierklausel (ORDER BY) nicht lokal auf die diskreten Abfragen angewendet werden kann, sondern nur auf die kombinierte Ausgabe.
2. Konvertieren Sie die Abfragen in abgeleitete Tabellen mit ihren eigenen ROWS-Klauseln.

Beispiele für ROWS

Die folgenden Beispiele schreiben die Beispiele um, die im Abschnitt über FIRST und SKIP verwendet wurden, früher in diesem Kapitel.

Rufen Sie die ersten zehn Namen aus der Ausgabe einer sortierten Abfrage in der Tabelle 'PEOPLE' ab:

```
SELECT id, name
FROM People
ORDER BY name ASC
ROWS 1 TO 10;
```

oder sein Äquivalent

```
SELECT id, name
FROM People
ORDER BY name ASC
ROWS 10;
```

Alle Datensätze aus der Tabelle PEOPLE zurückgeben mit Ausnahme der ersten 10 Namen:

```
SELECT id, name
FROM People
ORDER BY name ASC
ROWS 11 TO (SELECT COUNT(*) FROM People);
```

Und diese Abfrage gibt die letzten 10 Datensätze zurück (achten Sie auf die Klammern):

```
SELECT id, name
FROM People
ORDER BY name ASC
ROWS (SELECT COUNT(*) - 9 FROM People)
```

```
TO (SELECT COUNT(*) FROM People);
```

Dieser gibt die Zeilen 81-100 aus der Tabelle PEOPLE zurück:

```
SELECT id, name
FROM People
ORDER BY name ASC
ROWS 81 TO 100;
```

ROWS kann auch mit den Anweisungen UPDATE und DELETE verwendet werden.

Siehe auch

FIRST, SKIP, OFFSET, FETCH

6.1.11. OFFSET, FETCH

Verwendet für

Abrufen eines Zeilenabschnitts aus einer geordneten Menge

Verfügbar in

DSQL, PSQL

Syntax

```
SELECT <columns> FROM ...
  [WHERE ...]
  [ORDER BY ...]
  [OFFSET <m> {ROW | ROWS}]
  [FETCH {FIRST | NEXT} [ <n> ] { ROW | ROWS } ONLY]

<m>, <n>  ::=
    <integer-literal>
  | <query-parameter>
```

Tabelle 77. Argumente für die OFFSET- und FETCH-Klausel

Argument	Beschreibung
integer-literal	Ganzzahlliteral
query-parameter	Platzhalter für Abfrageparameter. ? in DSQL und :paramname in PSQL

Die Klauseln OFFSET und FETCH sind ein SQL:2008-kompatibles Äquivalent für FIRST/SKIP und eine Alternative für ROWS. Die OFFSET-Klausel gibt die Anzahl der zu überspringenden Zeilen an. Die FETCH -Klausel gibt die Anzahl der abzurufenden Zeilen an.

Wenn <*n*> in der FETCH-Klausel weggelassen wird (zB FETCH FIRST ROW ONLY), wird eine Zeile geholt.

Die Wahl zwischen ROW oder ROWS, oder FIRST oder NEXT in den Klauseln ist nur aus ästhetischen

Gründen (zB um die Abfrage lesbarer oder grammatikalisch korrekt zu machen). Technisch gesehen gibt es keinen Unterschied zwischen `OFFSET 10 ROW` oder `OFFSET 10 ROWS`, oder `FETCH NEXT 10 ROWS ONLY` oder `FETCH FIRST 10 ROWS ONLY`.

Wie bei `SKIP` und `FIRST` können `OFFSET`- und `FETCH`-Klauseln unabhängig voneinander sowohl in Top-Level- als auch in verschachtelten Abfrageausdrücken angewendet werden.

1. Firebird unterstützt den im SQL-Standard definierten Prozentsatz `FETCH` nicht.
2. Firebird unterstützt nicht das im SQL-Standard definierte `FETCH … WITH TIES`.
3. Die Klauseln `FIRST`/`SKIP` und `ROWS` sind nicht standardmäßige Alternativen.
4. Die Klauseln `OFFSET` und/oder `FETCH` können nicht mit `ROWS` oder `FIRST`/`SKIP` im gleichen Abfrageausdruck kombiniert werden.
5. Ausdrücke, Spaltenreferenzen usw. sind in keiner der Klauseln zulässig.
6. Im Gegensatz zur `ROWS`-Klausel stehen `OFFSET` und `FETCH` nur bei `SELECT` -Anweisungen zur Verfügung.

Beispiele für `OFFSET` und `FETCH`

Alle Zeilen außer den ersten 10 zurückgeben, sortiert nach Spalte COL1

```
SELECT *
FROM T1
ORDER BY COL1
OFFSET 10 ROWS
```

Geben Sie die ersten 10 Zeilen zurück, sortiert nach Spalte COL1

```
SELECT *
FROM T1
ORDER BY COL1
FETCH FIRST 10 ROWS ONLY
```

Verwenden von `OFFSET`- und `FETCH`-Klauseln in einer abgeleiteten Tabelle und in der äußeren Abfrage

```
SELECT *
FROM (
  SELECT *
  FROM T1
  ORDER BY COL1 DESC
  OFFSET 1 ROW
  FETCH NEXT 10 ROWS ONLY
) a
ORDER BY a.COL1
FETCH FIRST ROW ONLY
```

Die folgenden Beispiele schreiben die `FIRST`/`SKIP`-Beispiele und `ROWS`-Beispiele weiter oben in diesem

Kapitel.

Rufen Sie die ersten zehn Namen aus der Ausgabe einer sortierten Abfrage in der Tabelle "PEOPLE" ab:

```
SELECT id, name
FROM People
ORDER BY name ASC
FETCH NEXT 10 ROWS ONLY;
```

Alle Datensätze aus der Tabelle `PEOPLE` zurückgeben mit Ausnahme der ersten 10 Namen:

```
SELECT id, name
FROM People
ORDER BY name ASC
OFFSET 10 ROWS;
```

Und diese Abfrage gibt die letzten 10 Datensätze zurück. Im Gegensatz zu `FIRST`/`SKIP` und `ROWS` können wir keine Ausdrücke (einschließlich Unterabfragen) verwenden. Um die letzten 10 Zeilen abzurufen, kehren Sie die Sortierung zu den ersten (letzten) 10 Zeilen um und sortieren Sie dann in der richtigen Reihenfolge.

```
SELECT id, name
FROM (
  SELECT id, name
  FROM People
  ORDER BY name DESC
  FETCH FIRST 10 ROWS ONLY
) a
ORDER BY name ASC;
```

Dieser gibt die Zeilen 81-100 aus der Tabelle `PEOPLE` zurück:

```
SELECT id, name
FROM People
ORDER BY name ASC
OFFSET 80 ROWS
FETCH NEXT 20 ROWS;
```

Siehe auch

`FIRST`, `SKIP`, `ROWS`

6.1.12. FOR UPDATE [OF]

Syntax

```
SELECT ... FROM single_table
  [WHERE ...]
  [FOR UPDATE [OF <column_list>]]
```

`FOR UPDATE` tut nicht das, was der Name vermuten lässt. Der einzige Effekt besteht derzeit darin, den Prefetch-Puffer zu deaktivieren.

Es wird sich wahrscheinlich in Zukunft ändern: Der Plan ist, mit `FOR UPDATE` markierte Cursor zu validieren, wenn sie wirklich aktualisierbar sind, und positionierte Aktualisierungen und Löschungen für als nicht aktualisierbar bewertete Cursor abzulehnen.

Die Unterklausel `OF` tut überhaupt nichts.

6.1.13. `WITH LOCK`

Verwendet für

Begrenzte pessimistische Sperrung

Verfügbar in

DSQL, PSQL

Syntax

```
SELECT ... FROM single_table
  [WHERE ...]
  [FOR UPDATE [OF <column_list>]]
  WITH LOCK
```

`WITH LOCK` bietet eine begrenzte explizite pessimistische Sperrfunktion für die vorsichtige Verwendung unter Bedingungen, in denen das betroffene Rowset:

a. extrem klein (idealerweise Singleton), *und*

b. präzise gesteuert durch den Anwendungscode.

Dies ist nur für Experten!

Die Notwendigkeit einer pessimistischen Sperre in Firebird ist in der Tat sehr selten und sollte gut verstanden werden, bevor die Verwendung dieser Erweiterung in Betracht gezogen wird.

Es ist wichtig, die Auswirkungen der Transaktionsisolation und anderer Transaktionsattribute zu verstehen, bevor Sie versuchen, explizite Sperren in Ihrer Anwendung zu implementieren.

Wenn die Klausel `WITH LOCK` erfolgreich ist, sichert sie eine Sperre für die ausgewählten Zeilen und

verhindert, dass andere Transaktionen Schreibzugriff auf eine dieser Zeilen oder ihre abhängigen Zeilen erhalten, bis Ihre Transaktion beendet ist.

`WITH LOCK` kann nur mit einer SELECT-Anweisung der obersten Ebene für eine einzelne Tabelle verwendet werden. Es ist nicht verfügbar:

- in einer Unterabfragespezifikation
- für verbundene Sets
- mit dem `DISTINCT`-Operator, einer `GROUP BY`-Klausel oder einer anderen Aggregationsoperation
- mit einer Sicht
- mit der Ausgabe einer wählbaren Stored Procedure
- mit externem Tisch
- mit einer `UNION`-Abfrage

Da die Engine wiederum berücksichtigt, dass jeder Datensatz unter eine explizite Sperranweisung fällt, gibt sie entweder die aktuellste festgeschriebene Datensatzversion zurück, unabhängig vom Datenbankstatus, als die Anweisung übergeben wurde, oder eine Ausnahme.

Das Warteverhalten und die Konfliktmeldung hängen von den im TPB-Block angegebenen Transaktionsparametern ab:

Tabelle 78. How TPB settings affect explicit locking

TPB-Modus	Verhalten
isc_tpb_consistency	Explizite Sperren werden durch implizite oder explizite Sperren auf Tabellenebene außer Kraft gesetzt und ignoriert.
isc_tpb_concurrency + isc_tpb_nowait	Wenn ein Datensatz von einer Transaktion geändert wird, die seit dem Versuch der Transaktion, eine explizite Sperre zu starten, festgeschrieben wurde oder eine aktive Transaktion eine Änderung dieses Datensatzes durchgeführt hat, wird sofort eine Aktualisierungskonfliktausnahme ausgelöst.
isc_tpb_concurrency + isc_tpb_wait	Wenn der Datensatz von einer Transaktion geändert wird, die seit dem Versuch der Transaktion, eine explizite Sperre zu starten, festgeschrieben wurde, wird sofort eine Aktualisierungskonfliktausnahme ausgelöst. Wenn eine aktive Transaktion das Eigentum an diesem Datensatz hält (über eine explizite Sperre oder eine normale optimistische Schreibsperre), wartet die Transaktion, die die explizite Sperre versucht, auf das Ergebnis der blockierenden Transaktion und versucht, wenn sie beendet ist, die Sperre für die noch einmal aufnehmen. Das bedeutet, dass, wenn die blockierende Transaktion eine geänderte Version dieses Datensatzes festgeschrieben hat, eine Aktualisierungskonfliktausnahme ausgelöst wird.

TPB-Modus	Verhalten
isc_tpb_read_committed + isc_tpb_nowait	Wenn es eine aktive Transaktion gibt, die das Eigentum an diesem Datensatz hält (über explizites Sperren oder normale Aktualisierung), wird sofort eine Aktualisierungskonfliktausnahme ausgelöst.
isc_tpb_read_committed + isc_tpb_wait	Wenn eine aktive Transaktion das Eigentum an diesem Datensatz hält (über eine explizite Sperre oder eine normale optimistische Schreibsperre), wartet die Transaktion, die die explizite Sperre versucht, auf das Ergebnis der Sperrung der Transaktion und versucht, wenn sie beendet ist, die Sperre für die wieder aufnehmen. Ausnahmen bei Aktualisierungskonflikten können in diesem TPB-Modus niemals durch eine explizite Sperranweisung ausgelöst werden.

Verwendung mit einer FOR UPDATE-Klausel

Wenn die Unterklausel FOR UPDATE vor der Unterklausel WITH LOCK steht, werden gepufferte Abrufe unterdrückt. Somit wird die Sperre nacheinander auf jede Zeile angewendet, sobald sie abgerufen wird. Es wird dann möglich, dass eine Sperre, die auf Anforderung erfolgreich erschien, trotzdem *nachträglich fehlschlägt*, wenn versucht wird, eine Zeile abzurufen, die zwischenzeitlich durch eine andere Transaktion gesperrt wurde.

Alternativ kann es in Ihren Zugriffskomponenten möglich sein, die Größe des Fetch-Puffers auf 1 zu setzen. Auf diese Weise können Sie die aktuell gesperrte Zeile verarbeiten, bevor die nächste abgerufen und gesperrt wird, oder Fehler behandeln, ohne Ihre Transaktion rückgängig zu machen.

OF <column_list>

Diese optionale Unterklausel tut überhaupt nichts.

Siehe auch
FOR UPDATE [OF]

Wie die Engine mit WITH LOCK umgeht

Wenn eine UPDATE-Anweisung versucht, auf einen Datensatz zuzugreifen, der durch eine andere Transaktion gesperrt ist, löst sie je nach TPB-Modus entweder eine Aktualisierungskonfliktausnahme aus oder wartet auf den Abschluss der Sperrtransaktion. Das Engine-Verhalten ist hier dasselbe, als ob dieser Datensatz bereits durch die Sperrtransaktion geändert worden wäre.

Bei Konflikten mit pessimistischen Sperren werden keine speziellen gdscodes zurückgegeben.

Die Engine garantiert, dass alle von einer expliziten Sperranweisung zurückgegebenen Datensätze tatsächlich gesperrt sind und *ob* die in der WHERE-Klausel angegebenen Suchbedingungen erfüllen, solange die Suchbedingungen nicht von anderen Tabellen abhängen, über Joins, Unterabfragen usw. Es garantiert auch, dass Zeilen, die die Suchbedingungen nicht erfüllen, nicht durch die Anweisung gesperrt werden. Es kann *nicht* garantieren, dass es keine Zeilen gibt, die zwar die

Suchbedingungen erfüllen, aber nicht gesperrt sind.

Diese Situation kann eintreten, wenn andere, parallele Transaktionen ihre Änderungen während der Ausführung der Sperranweisung festschreiben.

Die Engine sperrt Zeilen zum Abrufzeitpunkt. Dies hat wichtige Konsequenzen, wenn Sie mehrere Zeilen gleichzeitig sperren. Viele Zugriffsmethoden für Firebird-Datenbanken holen standardmäßig die Ausgabe in Paketen von einigen hundert Zeilen (“buffered fetches”). Die meisten Datenzugriffskomponenten können Ihnen die im zuletzt abgerufenen Paket enthaltenen Zeilen nicht liefern, wenn ein Fehler aufgetreten ist.

Fallstricke bei der Verwendung von `WITH LOCK`

- Das Zurücksetzen eines impliziten oder expliziten Sicherungspunkts gibt Datensatzsperren frei, die unter diesem Sicherungspunkt vorgenommen wurden, benachrichtigt jedoch keine wartenden Transaktionen. Anwendungen sollten nicht von diesem Verhalten abhängen, da es in Zukunft geändert werden kann.
- Während explizite Sperren verwendet werden können, um ungewöhnliche Aktualisierungskonfliktfehler zu verhindern und/oder zu behandeln, wird die Anzahl von Deadlock-Fehlern zunehmen, wenn Sie Ihre Sperrstrategie nicht sorgfältig entwerfen und sie rigoros kontrollieren.
- Die meisten Anwendungen benötigen überhaupt keine expliziten Sperren. Die Hauptzwecke von expliziten Sperren sind:

  1. um eine teure Behandlung von Updatekonfliktfehlern in stark belasteten Anwendungen zu vermeiden, und
  2. um die Integrität von Objekten aufrechtzuerhalten, die einer relationalen Datenbank in einer Clusterumgebung zugeordnet sind.

  Wenn Ihre Verwendung der expliziten Sperrung nicht in eine dieser beiden Kategorien fällt, ist dies der falsche Weg, die Aufgabe in Firebird zu erledigen.

- Explizites Sperren ist eine erweiterte Funktion; missbrauche es nicht! Während Lösungen für diese Art von Problemen für Websites mit Tausenden von gleichzeitigen Autoren oder für ERP/CRM-Systeme, die in großen Unternehmen betrieben werden, sehr wichtig sein können, müssen die meisten Anwendungsprogramme unter solchen Bedingungen nicht funktionieren.

Beispiele für die explizite Sperrung

i. Einfach:

```
SELECT * FROM DOCUMENT WHERE ID=? WITH LOCK;
```

ii. Mehrere Zeilen, Verarbeitung nacheinander mit SQL-Cursor:

```
SELECT * FROM DOCUMENT WHERE PARENT_ID=?
```

```
  FOR UPDATE WITH LOCK;
```

6.1.14. INTO

Verwendet für

SELECT-Ausgabe an Variablen übergeben

Verfügbar in

PSQL

Syntax

In PSQL wird die INTO-Klausel ganz am Ende der SELECT-Anweisung platziert.

```
SELECT [...] <column-list>
FROM ...
[...]
[INTO <variable-list>]

<variable-list> ::= [:]psqlvar [, [:]psqlvar ...]
```

Der Doppelpunkt-Präfix vor lokalen Variablennamen in PSQL ist in der INTO -Klausel optional.

In PSQL-Code (Trigger, Stored Procedures und ausführbare Blöcke) können die Ergebnisse einer SELECT-Anweisung zeilenweise in lokale Variablen geladen werden. Dies ist oft die einzige Möglichkeit, überhaupt etwas mit den zurückgegebenen Werten zu tun, es sei denn, es wird ein expliziter oder impliziter Cursorname angegeben. Anzahl, Reihenfolge und Typen der Variablen müssen mit den Spalten in der Ausgabezeile übereinstimmen.

Eine “plain” SELECT-Anweisung kann in PSQL nur verwendet werden, wenn sie höchstens eine Zeile zurückgibt, d.h. wenn es sich um eine *singleton* select handelt. Für mehrzeilige Selects bietet PSQL das Schleifenkonstrukt FOR SELECT, das später im PSQL-Kapitel besprochen wird. PSQL unterstützt auch die DECLARE CURSOR-Anweisung, die einen benannten Cursor an eine SELECT-Anweisung bindet. Der Cursor kann dann verwendet werden, um die Ergebnismenge zu durchlaufen.

Beispiele

1. Auswahl einiger aggregierter Werte und Übergabe an die zuvor deklarierten Variablen min_amt, avg_amt und max_amt:

```
select min(amount), avg(cast(amount as float)), max(amount)
  from orders
  where artno = 372218
  into min_amt, avg_amt, max_amt;
```

Der CAST dient dazu, den Durchschnitt zu einer reellen Zahl zu machen;

> andernfalls, da `Betrag` vermutlich ein ganzzahliges Feld ist, würden SQL-Regeln es auf die nächste niedrigere ganze Zahl kürzen.

2. Ein PSQL-Trigger, der zwei Werte als 'BLOB'-Feld abruft (unter Verwendung der 'LIST()'-Funktion) und ihm 'INTO' ein drittes Feld zuweist:

```
select list(name, ', ')
  from persons p
  where p.id in (new.father, new.mother)
  into new.parentnames;
```

6.1.15. Common Table Expressions ("`WITH ⋯ AS ⋯ SELECT`")

Verfügbar in

DSQL, PSQL

Syntax

```
<cte-construct> ::=
  <cte-defs>
  <main-query>

<cte-defs> ::= WITH [RECURSIVE] <cte> [, <cte> ...]

<cte> ::= name [(<column-list>)] AS (<cte-stmt>)

<column-list> ::= column-alias [, column-alias ...]
```

Tabelle 79. Argumente für Common Table Expressions

Argument	Beschreibung
cte-stmt	Jede `SELECT`-Anweisung, einschließlich `UNION`
main-query	Die `SELECT`-Hauptanweisung, die sich auf die in der Präambel definierten CTEs beziehen kann
name	Alias für einen Tabellenausdruck
column-alias	Alias für eine Spalte in einem Tabellenausdruck

Ein allgemeiner Tabellenausdruck oder *CTE* kann als virtuelle Tabelle oder Ansicht beschrieben werden, die in einer Präambel einer Hauptabfrage definiert ist und nach der Ausführung der Hauptabfrage den Gültigkeitsbereich verlässt. Die Hauptabfrage kann auf alle *CTE*s verweisen, die in der Präambel definiert sind, als wären es reguläre Tabellen oder Ansichten. *CTE*s können rekursiv, d.h. selbstreferenzierend, aber nicht verschachtelt sein.

CTE-Hinweise

- Eine *CTE*-Definition kann jede zulässige `SELECT`-Anweisung enthalten, solange sie keine eigene "`WITH⋯`"-Präambel hat (keine Verschachtelung).

- *CTE*s, die für dieselbe Hauptabfrage definiert sind, können aufeinander verweisen, aber es sollte darauf geachtet werden, Schleifen zu vermeiden.
- *CTE*s kann von überall in der Hauptabfrage referenziert werden.
- Jeder *CTE* kann in der Hauptabfrage mehrfach referenziert werden, ggf. mit unterschiedlichen Aliasnamen.
- In Klammern eingeschlossen können *CTE*-Konstrukte als Unterabfragen in `SELECT`-Anweisungen, aber auch in `UPDATE`s, `MERGE`s usw. verwendet werden.
- In PSQL werden *CTE*s auch in `FOR`-Schleifenheadern unterstützt:

```
for
  with my_rivers as (select * from rivers where owner = 'me')
    select name, length from my_rivers into :rname, :rlen
do
begin
  ..
end
```

Wenn in Firebird 3.0.2 und früher ein *CTE* deklariert wird, muss es später verwendet werden: andernfalls erhalten Sie eine Fehlermeldung wie diese: “CTE "AAA" is not used in query”.

Diese Einschränkung wurde in Firebird 3.0.3 entfernt.

Beispiele

```
with dept_year_budget as (
  select fiscal_year,
         dept_no,
         sum(projected_budget) as budget
  from proj_dept_budget
  group by fiscal_year, dept_no
)
select d.dept_no,
       d.department,
       dyb_2008.budget as budget_08,
       dyb_2009.budget as budget_09
from department d
     left join dept_year_budget dyb_2008
       on d.dept_no = dyb_2008.dept_no
       and dyb_2008.fiscal_year = 2008
     left join dept_year_budget dyb_2009
       on d.dept_no = dyb_2009.dept_no
       and dyb_2009.fiscal_year = 2009
where exists (
  select * from proj_dept_budget b
  where d.dept_no = b.dept_no
```

```
);
```

Rekursive CTEs

Ein rekursiver (selbstreferenzierender) *CTE* ist eine `UNION`, die mindestens ein nicht-rekursives Element namens *anchor* haben muss. Das/die nicht-rekursive(n) Element(e) muss/müssen vor dem/den rekursiven Element(en) platziert werden. Rekursive Elemente sind miteinander und mit ihrem nicht-rekursiven Nachbarn durch `UNION ALL`-Operatoren verknüpft. Die Vereinigungen zwischen nicht-rekursiven Mitgliedern können von jedem Typ sein.

Rekursive *CTE*s erfordern, dass das Schlüsselwort `RECURSIVE` direkt nach `WITH` vorhanden ist. Jedes rekursive Unionsmitglied darf nur einmal auf sich selbst verweisen, und zwar in einer `FROM`-Klausel.

Ein großer Vorteil rekursiver *CTE*s besteht darin, dass sie weit weniger Speicher und CPU-Zyklen benötigen als eine entsprechende rekursive gespeicherte Prozedur.

Ausführungsmuster

Das Ausführungsmuster eines rekursiven *CTE* sieht wie folgt aus:

- Die Engine beginnt mit der Ausführung von einem nicht-rekursiven Member.
- Für jede ausgewertete Zeile beginnt es, jedes rekursive Element nacheinander auszuführen, wobei die aktuellen Werte aus der äußeren Zeile als Parameter verwendet werden.
- Wenn die aktuell ausgeführte Instanz eines rekursiven Members keine Zeilen erzeugt, führt die Ausführung eine Schleife zurück und ruft die nächste Zeile aus der äußeren Ergebnismenge ab.

Beispiel für rekursive *CTE*s

```
WITH RECURSIVE DEPT_YEAR_BUDGET AS (
  SELECT
      FISCAL_YEAR,
      DEPT_NO,
      SUM(PROJECTED_BUDGET) BUDGET
  FROM PROJ_DEPT_BUDGET
  GROUP BY FISCAL_YEAR, DEPT_NO
),
DEPT_TREE AS (
  SELECT
      DEPT_NO,
      HEAD_DEPT,
      DEPARTMENT,
      CAST('' AS VARCHAR(255)) AS INDENT
  FROM DEPARTMENT
  WHERE HEAD_DEPT IS NULL
  UNION ALL
  SELECT
      D.DEPT_NO,
      D.HEAD_DEPT,
      D.DEPARTMENT,
```

```
      H.INDENT || ' '
  FROM DEPARTMENT D
    JOIN DEPT_TREE H ON H.HEAD_DEPT = D.DEPT_NO
)
SELECT
    D.DEPT_NO,
    D.INDENT || D.DEPARTMENT DEPARTMENT,
    DYB_2008.BUDGET AS BUDGET_08,
    DYB_2009.BUDGET AS BUDGET_09
FROM DEPT_TREE D
    LEFT JOIN DEPT_YEAR_BUDGET DYB_2008 ON
      (D.DEPT_NO = DYB_2008.DEPT_NO) AND
      (DYB_2008.FISCAL_YEAR = 2008)
    LEFT JOIN DEPT_YEAR_BUDGET DYB_2009 ON
      (D.DEPT_NO = DYB_2009.DEPT_NO) AND
      (DYB_2009.FISCAL_YEAR = 2009);
```

Das nächste Beispiel gibt den Stammbaum eines Pferdes zurück. Der Hauptunterschied besteht darin, dass die Rekursion in zwei Zweigen des Stammbaums gleichzeitig auftritt.

```
WITH RECURSIVE PEDIGREE (
  CODE_HORSE,
  CODE_FATHER,
  CODE_MOTHER,
  NAME,
  MARK,
  DEPTH)
AS (SELECT
      HORSE.CODE_HORSE,
      HORSE.CODE_FATHER,
      HORSE.CODE_MOTHER,
      HORSE.NAME,
      CAST('' AS VARCHAR(80)),
      0
    FROM
      HORSE
    WHERE
      HORSE.CODE_HORSE = :CODE_HORSE
    UNION ALL
    SELECT
      HORSE.CODE_HORSE,
      HORSE.CODE_FATHER,
      HORSE.CODE_MOTHER,
      HORSE.NAME,
      'F' || PEDIGREE.MARK,
      PEDIGREE.DEPTH + 1
    FROM
      HORSE
      JOIN PEDIGREE
        ON HORSE.CODE_HORSE = PEDIGREE.CODE_FATHER
```

```
    WHERE
      PEDIGREE.DEPTH < :MAX_DEPTH
    UNION ALL
    SELECT
      HORSE.CODE_HORSE,
      HORSE.CODE_FATHER,
      HORSE.CODE_MOTHER,
      HORSE.NAME,
      'M' || PEDIGREE.MARK,
      PEDIGREE.DEPTH + 1
    FROM
      HORSE
      JOIN PEDIGREE
        ON HORSE.CODE_HORSE = PEDIGREE.CODE_MOTHER
    WHERE
      PEDIGREE.DEPTH < :MAX_DEPTH
)
SELECT
  CODE_HORSE,
  NAME,
  MARK,
  DEPTH
FROM
  PEDIGREE
```

Hinweise zu rekursiven CTEs

- Aggregate (`DISTINCT`, `GROUP BY`, `HAVING`) und Aggregatfunktionen (`SUM`, `COUNT`, `MAX` usw.) sind in rekursiven Unionselementen nicht erlaubt.
- Eine rekursive Referenz kann nicht an einem Outer Join teilnehmen.
- Die maximale Rekursionstiefe beträgt 1024.

6.2. INSERT

Verwendet für

Einfügen von Datenzeilen in eine Tabelle

Verfügbar in

DSQL, ESQL, PSQL

Syntax

```
INSERT INTO target
  {DEFAULT VALUES | [(<column_list>)] <value_source>}
  [RETURNING <returning_list> [INTO <variables>]]

<column_list> ::= colname [, colname ...]

<value_source> ::= VALUES (<value_list>) | <select_stmt>
```

```
<value_list> ::= <value> [, <value> ...]

<returning_list> ::=
  <ret_value> [[AS] ret_alias] [, <ret_value> [[AS] ret_alias] ...]

<ret_value> ::= { colname | target.colname | <value> }

<variables> ::= [:]varname [, [:]varname ...]
```

Tabelle 80. Arguments for the `INSERT`*-Anweisungsparameter*

Argument	Beschreibung
target	Der Name der Tabelle oder Ansicht, zu der eine neue Zeile oder ein Zeilenstapel hinzugefügt werden soll
colname	Spalte in der Tabelle oder Ansicht
value	Ein Ausdruck, dessen Wert zum Einfügen in die Tabelle oder zum Zurückgeben verwendet wird
ret_value	Der in der `RETURNING`-Klausel zurückzugebende Ausdruck
varname	Name einer lokalen PSQL-Variablen

Die `INSERT`-Anweisung wird verwendet, um einer Tabelle oder einer oder mehreren Tabellen, die einer Ansicht zugrunde liegen, Zeilen hinzuzufügen:

- Wenn die Spaltenwerte in einer `VALUES`-Klausel übergeben werden, wird genau eine Zeile eingefügt
- Die Werte können stattdessen durch einen `SELECT`-Ausdruck bereitgestellt werden, in diesem Fall können null bis viele Zeilen eingefügt werden
- Bei der `DEFAULT VALUES`-Klausel werden überhaupt keine Werte angegeben und genau eine Zeile eingefügt.

Einschränkungen

- Spalten, die an die Kontextvariablen `NEW.column_name` in Triggern zurückgegeben werden, sollten keinen Doppelpunkt (“:”) vor ihrem Namen haben
- In der Spaltenliste darf keine Spalte mehr als einmal vorkommen.

ALERT : `BEFORE INSERT`**-Triggers**

Achten Sie unabhängig von der zum Einfügen von Zeilen verwendeten Methode auf alle Spalten in der Zieltabelle oder -ansicht, die von `BEFORE INSERT`-Triggern gefüllt werden, wie z. B. Primärschlüssel und Suchspalten, bei denen die Groß-/Kleinschreibung nicht beachtet wird. Diese Spalten sollten sowohl aus der *column_list* als auch aus der `VALUES`-Liste ausgeschlossen werden, wenn die Trigger den `NEW.column_name` wie gewünscht auf `NULL` testen.

6.2.1. INSERT ⋯ VALUES

Die VALUES-Liste muss für jede Spalte in der Spaltenliste einen Wert in der gleichen Reihenfolge und vom richtigen Typ liefern. Die Spaltenliste muss nicht jede Spalte im Ziel angeben, aber wenn die Spaltenliste nicht vorhanden ist, benötigt die Engine einen Wert für jede Spalte in der Tabelle oder Ansicht (ohne berechnete Spalten).

Einführungssyntax bietet eine Möglichkeit, den Zeichensatz eines Werts zu identifizieren, der eine Zeichenfolgenkonstante (Literal) ist. Die Introducer-Syntax funktioniert nur mit Literal-Strings: Sie kann nicht auf String-Variablen, Parameter, Spaltenreferenzen oder Werte angewendet werden, die Ausdrücke sind.

Beispiele

```
INSERT INTO cars (make, model, year)
VALUES ('Ford', 'T', 1908);

INSERT INTO cars
VALUES ('Ford', 'T', 1908, 'USA', 850);

-- notice the '_' prefix (introducer syntax)
INSERT INTO People
VALUES (_ISO8859_1 'Hans-Jörg Schäfer');
```

6.2.2. INSERT ⋯ SELECT

Bei dieser Einfügemethode müssen die Ausgabespalten der SELECT-Anweisung für jede Zielspalte in der Spaltenliste einen Wert in der gleichen Reihenfolge und vom richtigen Typ liefern.

Literale Werte, Kontextvariablen oder Ausdrücke kompatiblen Typs können für jede Spalte in der Quellzeile ersetzt werden. In diesem Fall werden eine Quellspaltenliste und eine entsprechende VALUES-Liste benötigt.

Wenn die Spaltenliste fehlt – wie bei der Verwendung von SELECT * für den Quellausdruck – muss die *column_list* die Namen jeder Spalte in der Zieltabelle oder Sicht enthalten (berechnete Spalten ausgeschlossen).

Beispiele

```
INSERT INTO cars (make, model, year)
  SELECT make, model, year
  FROM new_cars;

INSERT INTO cars
  SELECT * FROM new_cars;

INSERT INTO Members (number, name)
  SELECT number, name FROM NewMembers
```

```
    WHERE Accepted = 1
UNION ALL
  SELECT number, name FROM SuspendedMembers
    WHERE Vindicated = 1

INSERT INTO numbers(num)
  WITH RECURSIVE r(n) as (
    SELECT 1 FROM rdb$database
    UNION ALL
    SELECT n+1 FROM r WHERE n < 100
  )
SELECT n FROM r
```

Natürlich müssen die Spaltennamen in der Quelltabelle nicht mit denen in der Zieltabelle übereinstimmen. Jede Art von SELECT-Anweisung ist zulässig, solange ihre Ausgabespalten in Anzahl, Reihenfolge und Typ genau mit den Einfügespalten übereinstimmen. Typen müssen nicht exakt gleich sein, aber sie müssen zuweisungskompatibel sein.

Bei der Verwendung von und INSERT ⋯ SELECT mit einer RETURNING-Klausel muss das SELECT höchstens eine Zeile produzieren, da RETURNING derzeit nur für Anweisungen funktioniert, die höchstens eine Zeile betreffen.

Dieses Verhalten kann sich in zukünftigen Firebird-Versionen ändern.

6.2.3. INSERT ⋯ DEFAULT VALUES

Die DEFAULT VALUES-Klausel ermöglicht das Einfügen eines Datensatzes, ohne irgendwelche Werte bereitzustellen, entweder direkt oder aus einer SELECT-Anweisung. Dies ist nur möglich, wenn jede NOT NULL- oder CHECK-Spalte in der Tabelle entweder einen gültigen Standardwert deklariert hat oder einen solchen Wert von einem BEFORE INSERT-Trigger erhält. Darüber hinaus dürfen Trigger, die erforderliche Feldwerte bereitstellen, nicht vom Vorhandensein von Eingabewerten abhängen.

Beispiel

```
INSERT INTO journal
  DEFAULT VALUES
RETURNING entry_id;
```

6.2.4. Die RETURNING-Klausel

Eine INSERT-Anweisung, die *höchstens eine Zeile* hinzufügt, kann optional eine RETURNING-Klausel enthalten, um Werte aus der eingefügten Zeile zurückzugeben. Die Klausel muss, falls vorhanden, nicht alle Einfügespalten enthalten und kann auch andere Spalten oder Ausdrücke enthalten. Die zurückgegebenen Werte spiegeln alle Änderungen wider, die möglicherweise an BEFORE INSERT -Triggern vorgenommen wurden.

Die optionale Unterklausel INTO ist nur in PSQL gültig.

Mehrfache `INSERT`s

In DSQL gibt eine Anweisung mit `RETURNING` immer nur eine Zeile zurück. Wenn die `RETURNING`-Klausel angegeben ist und mehr als eine Zeile durch die `INSERT` -Anweisung eingefügt wird, schlägt die Anweisung fehl und es wird eine Fehlermeldung zurückgegeben. Dieses Verhalten kann sich in zukünftigen Firebird-Versionen ändern.

Beispiele

```
INSERT INTO Scholars (
  firstname,
  lastname,
  address,
  phone,
  email)
VALUES (
  'Henry',
  'Higgins',
  '27A Wimpole Street',
  '3231212',
  NULL)
RETURNING lastname, fullname, id;

INSERT INTO Dumbbells (firstname, lastname, iq)
  SELECT fname, lname, iq
FROM Friends
  ORDER BY iq ROWS 1
  RETURNING id, firstname, iq
INTO :id, :fname, :iq;
```

Anmerkungen

- `RETURNING` wird für `VALUES`- und `DEFAULT VALUES`-Inserts und Singleton-`SELECT`-Inserts unterstützt.
- In DSQL gibt eine Anweisung mit einer `RETURNING`-Klausel *immer* genau eine Zeile zurück. Wenn tatsächlich kein Datensatz eingefügt wurde, sind die Felder in dieser Zeile alle `NULL`. Dieses Verhalten kann sich in einer späteren Version von Firebird ändern. Wenn in PSQL keine Zeile eingefügt wurde, wird nichts zurückgegeben und die Zielvariablen behalten ihre vorhandenen Werte bei.

6.2.5. Einfügen in 'BLOB'-Spalten

Das Einfügen in 'BLOB'-Spalten ist nur unter folgenden Umständen möglich:

1. Die Client-Anwendung hat spezielle Vorkehrungen für solche Einfügungen getroffen, indem sie die Firebird-API verwendet. In diesem Fall ist der *modus operandi* anwendungsspezifisch und liegt außerhalb des Rahmens dieses Handbuchs.
2. Der eingefügte Wert ist ein Zeichenfolgenliteral von nicht mehr als 65.533 Byte (64 KB - 3).

Ein Grenzwert in Zeichen wird zur Laufzeit für Zeichenfolgen berechnet, die sich in Mehrbytezeichensätzen befinden, um ein Überschreiten des Bytegrenzwertes zu vermeiden. Für einen UTF8-String (max. 4 Byte/Zeichen) liegt die Laufzeitbegrenzung beispielsweise bei (floor(65533/4)) = 16383 Zeichen.

3. Sie verwenden das Formular "`INSERT ⋯ SELECT`" und eine oder mehrere Spalten in der Ergebnismenge sind `BLOB`s.

6.3. UPDATE

Verwendet für

Zeilen in Tabellen und Ansichten ändern

Verfügbar in

DSQL, ESQL, PSQL

Syntax

```
UPDATE target [[AS] alias]
  SET col = <value> [, col = <value> ...]
  [WHERE {<search-conditions> | CURRENT OF cursorname}]
  [PLAN <plan_items>]
  [ORDER BY <sort_items>]
  [ROWS m [TO n]]
  [RETURNING <returning_list> [INTO <variables>]]

<returning_list> ::=
  <ret_value> [[AS] ret_alias] [, <ret_value> [[AS] ret_alias] ...]

<ret_value> ::=
    colname
  | table_or_alias.colname
  | NEW.colname
  | OLD.colname
  | <value>

<variables> ::= [:]varname [, [:]varname ...]
```

Tabelle 81. Argumente für die UPDATE-Anweisungsparameter

Argument	Beschreibung
target	Der Name der Tabelle oder Ansicht, in der die Datensätze aktualisiert werden
alias	Alias für die Tabelle oder Ansicht
col	Name oder Alias einer Spalte in der Tabelle oder Ansicht

Argument	Beschreibung
value	Ausdruck für den neuen Wert für eine Spalte, die in der Tabelle oder Ansicht durch die Anweisung aktualisiert werden soll, oder einen zurückzugebenden Wert
search-conditions	Eine Suchbedingung, die die Menge der zu aktualisierenden Zeilen einschränkt
cursorname	Der Name des Cursors, über den die zu aktualisierende(n) Zeile(n) positioniert werden
plan_items	Klauseln im Abfrageplan
sort_items	Spalten, die in einer `ORDER BY`-Klausel aufgeführt sind
m, n	Integer-Ausdrücke zum Begrenzen der Anzahl der zu aktualisierenden Zeilen
ret_value	Ein in der `RETURNING`-Klausel zurückzugebender Wert
varname	Name einer lokalen PSQL-Variablen

Die `UPDATE`-Anweisung ändert Werte in einer Tabelle oder in einer oder mehreren Tabellen, die einer Ansicht zugrunde liegen. Die betroffenen Spalten werden in der `SET`-Klausel angegeben. Die betroffenen Zeilen können durch die Klauseln `WHERE` und `ROWS` eingeschränkt werden. Wenn weder 'WHERE' noch 'ROWS' vorhanden sind, werden alle Datensätze in der Tabelle aktualisiert.

6.3.1. Alias verwenden

Wenn Sie einer Tabelle oder Sicht einen Alias zuweisen, *muss* der Alias bei der Angabe von Spalten und auch in allen Spaltenreferenzen in anderen Klauseln verwendet werden.

Beispiel

Korrekte Verwendung

```
update Fruit set soort = 'pisang' where ...

update Fruit set Fruit.soort = 'pisang' where ...

update Fruit F set soort = 'pisang' where ...

update Fruit F set F.soort = 'pisang' where ...
```

Nicht möglich:

```
update Fruit F set Fruit.soort = 'pisang' where ...
```

6.3.2. Die SET-Klausel

In der SET-Klausel werden die Zuweisungsphrasen, die die Spalten mit den zu setzenden Werten enthalten, durch Kommas getrennt. In einer Zuweisungsphrase befinden sich links die Spaltennamen und rechts die Werte oder Ausdrücke, die die Zuweisungswerte enthalten. Eine Spalte darf nur einmal in der SET-Klausel enthalten sein.

In Ausdrücken auf der rechten Seite kann ein Spaltenname verwendet werden. In diesen Werten auf der rechten Seite wird immer der alte Wert der Spalte verwendet, auch wenn der Spalte bereits früher in der SET-Klausel ein neuer Wert zugewiesen wurde.

Hier ist ein Beispiel

Daten in der TSET-Tabelle:

```
A B
---
1 0
2 0
```

Die Anweisung:

```
UPDATE tset SET a = 5, b = a;
```

ändert die Werte in:

```
A B
---
5 1
5 2
```

Beachten Sie, dass die alten Werte (1 und 2) verwendet werden, um die Spalte b zu aktualisieren, auch nachdem der Spalte ein neuer Wert zugewiesen wurde (5).

Es war nicht immer so. Vor Version 2.5 erhielten Spalten ihre neuen Werte sofort bei der Zuweisung. Es war ein nicht standardmäßiges Verhalten, das in Version 2.5 behoben wurde.

Um die Kompatibilität mit Legacy-Code zu gewährleisten, enthält die Konfigurationsdatei `firebird.conf` den Parameter `OldSetClauseSemantics`, der auf True (1) gesetzt werden kann, um das alte, schlechte Verhalten wiederherzustellen. Es handelt sich um eine vorübergehende Maßnahme – der Parameter wird in Zukunft entfernt.

6.3.3. Die WHERE-Klausel

Die WHERE-Klausel legt die Bedingungen fest, die die Menge der Datensätze für ein *searched update*

begrenzen.

Wenn in PSQL ein benannter Cursor zum Aktualisieren einer Menge verwendet wird, ist die Aktion mit der `WHERE CURRENT OF`-Klausel auf die Zeile beschränkt, in der sich der Cursor gerade befindet. Dies ist ein *positioniertes Update*.

Die Klausel `WHERE CURRENT OF` ist nur in PSQL verfügbar, da es in DSQL keine Anweisung zum Erstellen und Bearbeiten eines expliziten Cursors gibt. Gesuchte Updates sind natürlich auch in PSQL verfügbar.

Beispiele

```
UPDATE People
  SET firstname = 'Boris'
  WHERE lastname = 'Johnson';

UPDATE employee e
  SET salary = salary * 1.05
  WHERE EXISTS(
         SELECT *
           FROM employee_project ep
           WHERE e.emp_no = ep.emp_no);

UPDATE addresses
  SET city = 'Saint Petersburg', citycode = 'PET'
  WHERE city = 'Leningrad'

UPDATE employees
  SET salary = 2.5 * salary
  WHERE title = 'CEO'
```

Für String-Literale, bei denen der Parser Hilfe benötigt, um den Zeichensatz der Daten zu interpretieren, kann die Introducer-Syntax verwendet werden. Dem Zeichenfolgenliteral geht der Zeichensatzname voran, dem ein Unterstrich vorangestellt ist:

```
-- beachten Sie das '_'-Präfix

UPDATE People
SET name = _ISO8859_1 'Hans-Jörg Schäfer'
WHERE id = 53662;
```

6.3.4. Die Klauseln `ORDER BY` und `ROWS`

Die Klauseln `ORDER BY` und `ROWS` sind nur sinnvoll, wenn sie zusammen verwendet werden. Sie können jedoch separat verwendet werden.

Wenn `ROWS` ein Argument hat, *m*, werden die zu aktualisierenden Zeilen auf die ersten *m* Zeilen beschränkt.

Hinweise

- Wenn *m* > die Anzahl der verarbeiteten Zeilen ist, wird der gesamte Zeilensatz aktualisiert
- Wenn *m* = 0, werden keine Zeilen aktualisiert
- Wenn *m* < 0, tritt ein Fehler auf und das Update schlägt fehl

Wenn zwei Argumente verwendet werden, *m* und *n*, begrenzt `ROWS` die Zeilen, die aktualisiert werden, auf Zeilen von *m* bis einschließlich *n*. Beide Argumente sind ganze Zahlen und beginnen bei 1.

Hinweise

- Wenn *m* > die Anzahl der verarbeiteten Zeilen ist, werden keine Zeilen aktualisiert
- Wenn *n* > die Anzahl der Zeilen, werden Zeilen von *m* bis zum Ende des Satzes aktualisiert
- Wenn *m* < 1 oder *n* < 1 ist, tritt ein Fehler auf und das Update schlägt fehl
- Wenn *n* = *m* - 1, werden keine Zeilen aktualisiert
- Wenn *n* < *m* -1, tritt ein Fehler auf und das Update schlägt fehl

ROWS-Beispiel

```
UPDATE employees
SET salary = salary + 50
ORDER BY salary ASC
ROWS 20;
```

6.3.5. Die `RETURNING`-Klausel

Eine `UPDATE`-Anweisung, die *höchstens eine Zeile* umfasst, kann `RETURNING` enthalten, um einige Werte aus der aktualisierten Zeile zurückzugeben. "RETURNING" kann Daten aus einer beliebigen Spalte der Zeile enthalten, nicht unbedingt aus den Spalten, die gerade aktualisiert werden. Es kann Literale oder Ausdrücke enthalten, die nicht mit Spalten verknüpft sind, wenn dies erforderlich ist.

Wenn das `RETURNING`-Set Daten aus der aktuellen Zeile enthält, melden die zurückgegebenen Werte Änderungen, die in den `BEFORE UPDATE`-Triggern vorgenommen wurden, aber nicht die in `AFTER UPDATE`-Triggern.

Als Spaltennamen können die Kontextvariablen `OLD.fieldname` und `NEW.fieldname` verwendet werden. Wenn `OLD.` oder `NEW.` nicht angegeben wird, sind die zurückgegebenen Spaltenwerte die `NEW.`-Werte.

In DSQL gibt eine Anweisung mit `RETURNING` immer eine einzelne Zeile zurück. Versuche, ein `UPDATE ⋯ RETURNING ⋯` auszuführen, das mehrere Zeilen betrifft, führen zu dem Fehler “multiple rows in singleton select”. Wenn die Anweisung keine Datensätze aktualisiert, enthalten die zurückgegebenen Werte `NULL`. Dieses Verhalten kann sich in zukünftigen Firebird-Versionen ändern.

Die INTO-Unterklausel

In PSQL kann die INTO-Klausel verwendet werden, um die Rückgabewerte an lokale Variablen zu übergeben. Es ist in DSQL nicht verfügbar. Wenn keine Datensätze aktualisiert werden, wird nichts zurückgegeben und die in RETURNING angegebenen Variablen behalten ihre vorherigen Werte.

RETURNING-Beispiel (DSQL)

```
UPDATE Scholars
SET firstname = 'Hugh', lastname = 'Pickering'
WHERE firstname = 'Henry' and lastname = 'Higgins'
RETURNING id, old.lastname, new.lastname;
```

6.3.6. 'BLOB'-Spalten aktualisieren

Das Aktualisieren einer BLOB-Spalte ersetzt immer den gesamten Inhalt. Sogar die BLOB ID, das "handle", das direkt in der Spalte gespeichert wird, wird geändert. BLOBs können aktualisiert werden, wenn:

1. Die Client-Anwendung hat für diesen Vorgang spezielle Vorkehrungen getroffen, indem sie die Firebird-API verwendet. In diesem Fall ist der *modus operandi* anwendungsspezifisch und liegt außerhalb des Rahmens dieses Handbuchs.
2. Der neue Wert ist ein Zeichenfolgenliteral von nicht mehr als 65.533 Byte (64 KB - 3).

Ein Grenzwert in Zeichen wird zur Laufzeit für Zeichenfolgen berechnet, die sich in Mehrbytezeichensätzen befinden, um ein Überschreiten des Bytegrenzwertes zu vermeiden. Für einen UTF8-String (max. 4 Byte/Zeichen) liegt die Laufzeitbegrenzung beispielsweise bei (floor(65533/4)) = 16383 Zeichen.

3. Die Quelle ist selbst eine 'BLOB'-Spalte oder allgemeiner ein Ausdruck, der ein 'BLOB' zurückgibt.
4. Sie verwenden die Anweisung INSERT CURSOR (nur ESQL).

6.4. UPDATE OR INSERT

Verwendet für

Aktualisieren eines bestehenden Datensatzes in einer Tabelle oder, falls er nicht existiert, einfügen

Verfügbar in

DSQL, PSQL

Syntax

```
UPDATE OR INSERT INTO
  target [(<column_list>)]
  VALUES (<value_list>)
```

```
  [MATCHING (<column_list>)]
  [RETURNING <values> [INTO <variables>]]

<column_list> ::= colname  [, colname ...]

<value_list> ::= <value> [, <value> ...]

<returning_list> ::= <ret_value> [, <ret_value> ...]

<ret_value> ::=
    colname
  | NEW.colname
  | OLD.colname
  | <value>

<variables> ::= [:]varname [, [:]varname ...]
```

Tabelle 82. Argumente für den `UPDATE OR INSERT`*-Anweisungsparameter*

Argument	Beschreibung
target	Der Name der Tabelle oder Ansicht, in der der/die Datensatz(e) aktualisiert oder ein neuer Datensatz eingefügt werden soll
colname	Name einer Spalte in der Tabelle oder Ansicht
value	Ein Ausdruck, dessen Wert zum Einfügen oder Aktualisieren der Tabelle oder zum Zurückgeben eines Werts verwendet werden soll
ret_value	Ein in der RETURNING-Klausel zurückgegebener Ausdruck
varname	Variablenname – nur PSQL

`UPDATE OR INSERT` fügt einen neuen Datensatz ein oder aktualisiert einen oder mehrere bestehende Datensätze. Die durchgeführte Aktion hängt von den Werten ab, die für die Spalten in der `MATCHING` -Klausel (oder, falls letztere fehlt, im Primärschlüssel) bereitgestellt werden. Wenn Datensätze gefunden werden, die diesen Werten entsprechen, werden sie aktualisiert. Wenn nicht, wird ein neuer Datensatz eingefügt. Eine Übereinstimmung zählt nur, wenn alle Werte in den `MATCHING`- oder Primärschlüsselspalten gleich sind. Der Abgleich erfolgt mit dem Operator `IS NOT DISTINCT`, sodass ein `NULL` mit einem anderen übereinstimmt.

Einschränkungen

- Wenn die Tabelle keinen Primärschlüssel hat, ist die `MATCHING`-Klausel obligatorisch.
- In der `MATCHING`-Liste sowie in der Update/Insert-Spaltenliste darf jeder Spaltenname nur einmal vorkommen.
- Die Unterklausel “`INTO <variables>`” ist nur in PSQL verfügbar.
- Bei Rückgabe von Werten in die Kontextvariable `NEW` darf diesem Namen kein Doppelpunkt vorangestellt werden (“`:`”).

6.4.1. Die RETURNING-Klausel

Die optionale RETURNING-Klausel, falls vorhanden, muss nicht alle in der Anweisung erwähnten Spalten enthalten und kann auch andere Spalten oder Ausdrücke enthalten. Die zurückgegebenen Werte spiegeln alle Änderungen wider, die möglicherweise in BEFORE-Triggern vorgenommen wurden, aber nicht in AFTER-Triggern. OLD.fieldname und NEW.fieldname können beide in der Liste der zurückzugebenden Spalten verwendet werden; für Feldnamen, denen keiner von diesen vorangeht, wird der neue Wert zurückgegeben.

In DSQL gibt eine Anweisung mit einer RETURNING-Klausel *immer* genau eine Zeile zurück. Wenn eine RETURNING-Klausel vorhanden ist und mehr als ein übereinstimmender Datensatz gefunden wird, wird ein Fehler "multiple rows in singleton select" ausgegeben. Dieses Verhalten kann sich in einer späteren Version von Firebird ändern.

Die optionale Unterklausel INTO ist nur in PSQL gültig.

6.4.2. Beispiel für UPDATE OR INSERT

Ändern von Daten in einer Tabelle mit UPDATE OR INSERT in einem PSQL-Modul. Der Rückgabewert wird an eine lokale Variable übergeben, deren Doppelpunkt-Präfix optional ist.

```
UPDATE OR INSERT INTO Cows (Name, Number, Location)
  VALUES ('Suzy Creamcheese', 3278823, 'Green Pastures')
  MATCHING (Number)
  RETURNING rec_id into :id;
```

6.5. DELETE

Verwendet für

Zeilen aus einer Tabelle oder Ansicht löschen

Verfügbar in

DSQL, ESQL, PSQL

Syntax

```
DELETE
  FROM target [[AS] alias]
  [WHERE {<search-conditions> | CURRENT OF cursorname}]
  [PLAN <plan_items>]
  [ORDER BY <sort_items>]
  [ROWS m [TO n]]
  [RETURNING <returning_list> [INTO <variables>]]

<returning_list> ::=
  <ret_value> [[AS] ret_alias] [, <ret_value> [[AS] ret_alias] ...]

<ret_value> ::=
```

```
  { colname | target_or_alias.colname | <value> }

<variables> ::=
  [:]varname [, [:]varname ...]
```

Tabelle 83. Argumente der DELETE*-Anweisungsparameter*

Argument	Beschreibung
target	Der Name der Tabelle oder Ansicht, aus der die Datensätze gelöscht werden sollen
alias	Alias für die Zieltabelle oder -ansicht
search-conditions	Suchbedingung, die den Satz von Zeilen einschränkt, die gelöscht werden sollen
cursorname	Der Name des Cursors, in dem der aktuelle Datensatz zum Löschen positioniert ist
plan_items	Abfrageplanklausel
sort_items	ORDER BY-Klausel
m, n	Integer-Ausdrücke zum Begrenzen der Anzahl der zu löschenden Zeilen
ret_value	Ein in der RETURNING-Klausel zurückzugebender Ausdruck
value	Ein Ausdruck, dessen Wert für die Rückgabe verwendet wird
varname	Name einer PSQL-Variablen

DELETE entfernt Zeilen aus einer Datenbanktabelle oder aus einer oder mehreren Tabellen, die einer Ansicht zugrunde liegen. WHERE- und ROWS-Klauseln können die Anzahl der gelöschten Zeilen begrenzen. Wenn weder WHERE noch ROWS vorhanden sind, entfernt DELETE alle Zeilen in der Relation.

6.5.1. Aliases

Wenn für die Zieltabelle oder -sicht ein Alias angegeben wird, muss dieser verwendet werden, um alle Feldnamenreferenzen in der DELETE-Anweisung zu qualifizieren.

Beispiele

Unterstützte Nutzung:

```
delete from Cities where name starting 'Alex';

delete from Cities where Cities.name starting 'Alex';

delete from Cities C where name starting 'Alex';

delete from Cities C where C.name starting 'Alex';
```

Nicht möglich:

```
delete from Cities C where Cities.name starting 'Alex';
```

6.5.2. WHERE

Die WHERE-Klausel legt die Bedingungen fest, die die Menge der Datensätze für ein *searched delete* begrenzen.

Wenn in PSQL ein benannter Cursor zum Löschen einer Menge verwendet wird, ist die Aktion mit der Klausel WHERE CURRENT OF auf die Zeile beschränkt, in der sich der Cursor gerade befindet. Dies ist ein *positioniertes Löschen*.

Die Klausel WHERE CURRENT OF ist nur in PSQL und ESQL verfügbar, da es in DSQL keine Anweisung zum Erstellen und Bearbeiten eines expliziten Cursors gibt. Gesuchte Löschvorgänge sind natürlich auch in PSQL verfügbar.

Beispiele

```
DELETE FROM People
  WHERE firstname <> 'Boris' AND lastname <> 'Johnson';

DELETE FROM employee e
  WHERE NOT EXISTS(
    SELECT *
    FROM employee_project ep
     WHERE e.emp_no = ep.emp_no);

DELETE FROM Cities
  WHERE CURRENT OF Cur_Cities;  -- ESQL and PSQL only
```

6.5.3. PLAN

Eine PLAN-Klausel ermöglicht es dem Benutzer, die Operation manuell zu optimieren.

Beispiel

```
DELETE FROM Submissions
  WHERE date_entered < '1-Jan-2002'
  PLAN (Submissions INDEX ix_subm_date);
```

6.5.4. ORDER BY und ROWS

Die ORDER BY-Klausel ordnet die Menge, bevor das eigentliche Löschen stattfindet. Es macht nur in Kombination mit ROWS Sinn, ist aber auch ohne gültig.

Die ROWS-Klausel begrenzt die Anzahl der zu löschenden Zeilen. Für die Argumente *m* und *n* können ganzzahlige Literale oder beliebige ganzzahlige Ausdrücke verwendet werden.

Wenn `ROWS` ein Argument hat, *m*, werden die zu löschenden Zeilen auf die ersten *m* Zeilen beschränkt.

Hinweise

- Wenn *m* > die Anzahl der verarbeiteten Zeilen ist, wird der gesamte Satz von Zeilen gelöscht
- Bei *m* = 0 werden keine Zeilen gelöscht
- Wenn *m* < 0, tritt ein Fehler auf und das Löschen schlägt fehl

Wenn zwei Argumente verwendet werden, *m* und *n*, begrenzt `ROWS` die zu löschenden Zeilen auf Zeilen von *m* bis einschließlich *n*. Beide Argumente sind ganze Zahlen und beginnen bei 1.

Hinweise

- Wenn *m* > die Anzahl der verarbeiteten Zeilen ist, werden keine Zeilen gelöscht
- Wenn *m* > 0 und <= die Anzahl der Zeilen im Set und *n* außerhalb dieser Werte liegt, werden Zeilen von *m* bis zum Ende des Sets gelöscht
- Wenn *m* < 1 oder *n* < 1 ist, tritt ein Fehler auf und das Löschen schlägt fehl
- Wenn *n* = *m* - 1, werden keine Zeilen gelöscht
- Wenn *n* < *m* -1, tritt ein Fehler auf und das Löschen schlägt fehl

Beispiele

Löschen des ältesten Kaufs:

```
DELETE FROM Purchases
  ORDER BY date ROWS 1;
```

Löschen des/der höchsten Custno(s):

```
DELETE FROM Sales
  ORDER BY custno DESC ROWS 1 to 10;
```

Löschen aller Verkäufe, `ORDER BY`-Klausel sinnlos:

```
DELETE FROM Sales
  ORDER BY custno DESC;
```

Löschen eines Datensatzes am Ende beginnend, also ab Z...:

```
DELETE FROM popgroups
  ORDER BY name DESC ROWS 1;
```

Löschen der fünf ältesten Gruppen:

```
DELETE FROM popgroups
  ORDER BY formed ROWS 5;
```

Da keine Sortierung (`ORDER BY`) angegeben ist, werden 8 gefundene Datensätze, beginnend mit dem fünften, gelöscht:

```
DELETE FROM popgroups
  ROWS 5 TO 12;
```

6.5.5. RETURNING

Eine `DELETE`-Anweisung, die *höchstens eine Zeile* entfernt, kann optional eine `RETURNING`-Klausel enthalten, um Werte aus der gelöschten Zeile zurückzugeben. Die Klausel, falls vorhanden, muss nicht alle Spalten der Relation enthalten und kann auch andere Spalten oder Ausdrücke enthalten.

- In DSQL gibt eine Anweisung mit `RETURNING` immer ein Singleton zurück, niemals ein Set mit mehreren Zeilen. Wenn eine `RETURNING`-Klausel vorhanden ist und mehr als ein übereinstimmender Datensatz gefunden wird, wird ein Fehler “multiple rows in singleton select” ausgegeben. Wenn keine Datensätze gelöscht werden, enthalten die zurückgegebenen Spalten `NULL`. Dieses Verhalten kann sich in zukünftigen Firebird-Versionen ändern
- Die `INTO`-Klausel ist nur in PSQL verfügbar
  - Wenn die Zeile nicht gelöscht wird, wird nichts zurückgegeben und die Zielvariablen behalten ihre Werte

Beispiele

```
DELETE FROM Scholars
  WHERE firstname = 'Henry' and lastname = 'Higgins'
  RETURNING lastname, fullname, id;

DELETE FROM Dumbbells
  ORDER BY iq DESC
  ROWS 1
  RETURNING lastname, iq into :lname, :iq;
```

6.6. MERGE

Verwendet für

Zusammenführen von Daten aus einem Quellsatz in eine Zielrelation

Verfügbar in

DSQL, PSQL

Syntax

```
MERGE INTO target [[AS] target_alias]
  USING <source> [[AS] source_alias]
  ON <join_condition>
  <merge_when> [<merge_when> ...]
  [RETURNING <returning_list> [INTO <variables>]]

<merge_when> ::=
    <merge_when_matched>
  | <merge_when_not_matched>

<merge_when_matched> ::=
  WHEN MATCHED [ AND <condition> ] THEN
  { UPDATE SET <assignment-list>
  | DELETE }

<merge_when_not_matched> ::=
  WHEN NOT MATCHED [ AND <condition> ] THEN
  INSERT [( <column_list> )] VALUES ( <value_list> )

<source> ::= tablename | (<select_stmt>)

<assignment_list ::=
  colname = <value> [, <colname> = <value> ...]]

<column_list> ::= colname [, colname ...]

<value_list> ::= <value> [, <value> ...]

<returning_list> ::=
  <ret_value> [[AS] ret_alias] [, <ret_value> [[AS] ret_alias] ...]

<ret_value> ::=
    colname
  | table_or_alias.colname
  | NEW.colname
  | OLD.colname
  | <value>

<variables> ::=
  [:]varname [, [:]varname ...]
```

Tabelle 84. Argumente für die MERGE*-Anweisungsparameter*

Argument	Beschreibung
target	Name der Zielbeziehung (Tabelle oder aktualisierbare Sicht)
source	Datenquelle. Dies kann eine Tabelle, eine Ansicht, eine gespeicherte Prozedur oder eine abgeleitete Tabelle sein

Argument	Beschreibung
target_alias	Alias für die Zielbeziehung (Tabelle oder aktualisierbare Ansicht)
source_alias	Alias für die Quellrelation oder Menge
join_conditions	Die (ON) Bedingung(en) zum Abgleichen der Quelldatensätze mit denen im Ziel
condition	Zusätzliche Testbedingung in der Klausel WHEN MATCHED oder WHEN NOT MATCHED
tablename	Tabellen- oder Ansichtsname
select_stmt	Select-Anweisung der abgeleiteten Tabelle
colname	Name einer Spalte in der Zielrelation
value	Der einer Spalte in der Zieltabelle zugewiesene Wert. Dieser Ausdruck kann ein Literalwert, eine PSQL-Variable, eine Spalte aus der Quelle oder eine kompatible Kontextvariable sein
ret_value	Der in der RETURNING-Klausel zurückzugebende Ausdruck Kann ein Spaltenverweis auf Quelle oder Ziel oder ein Spaltenverweis des NEW- oder OLD-Kontexts des Ziels oder ein Wert sein.
ret_alias	Alias für den Wertausdruck in der RETURNING-Klausel
varname	Name einer lokalen PSQL-Variablen

Die 'MERGE'-Anweisung führt Datensätze aus der Quelle in eine Zieltabelle oder eine aktualisierbare Sicht zusammen. Die Quelle kann eine Tabelle, ein View oder “alles, was mit SELECT abfragen” können. Jeder Quelldatensatz wird verwendet, um einen oder mehrere Zieldatensätze zu aktualisieren, einen neuen Datensatz in die Zieltabelle einzufügen, einen Datensatz aus der Zieltabelle zu löschen oder nichts zu tun.

Welche Aktion ausgeführt wird, hängt von der angegebenen Join-Bedingung, der/den WHEN -Klausel(n) und der - optionalen - Bedingung in der WHEN-Klausel ab. Die Join-Bedingung und die Bedingung im WHEN enthalten normalerweise einen Vergleich von Feldern in den Quell- und Zielbeziehungen.

Mehrere WHEN MATCHED- und WHEN NOT MATCHED-Klauseln sind zulässig. Für jede Zeile in der Quelle werden die WHEN-Klauseln in der Reihenfolge überprüft, in der sie in der Anweisung angegeben sind. Wenn die Bedingung in der WHEN-Klausel nicht als wahr ausgewertet wird, wird die Klausel übersprungen und die nächste Klausel wird geprüft. Dies wird getan, bis die Bedingung für eine WHEN-Klausel wahr ist oder eine WHEN-Klausel ohne Bedingung zutrifft oder es keine WHEN-Klauseln mehr gibt. Wenn eine übereinstimmende Klausel gefunden wird, wird die mit der Klausel verknüpfte Aktion ausgeführt. Für jede Zeile in der Quelle wird höchstens eine Aktion ausgeführt.

Mindestens eine WHEN-Klausel muss vorhanden sein.

WHEN NOT MATCHED wird aus der Quellsicht ausgewertet, dh der in USING angegebenen Tabelle oder Menge. Es muss so funktionieren, denn wenn der Quelldatensatz nicht mit einem Zieldatensatz übereinstimmt, wird INSERT ausgeführt. Wenn es einen Zieldatensatz gibt, der nicht mit einem Quelldatensatz

übereinstimmt, wird natürlich nichts unternommen.

Derzeit gibt die Variable ROW_COUNT den Wert 1 zurück, auch wenn mehr als ein Datensatz geändert oder eingefügt wird. Einzelheiten und Fortschritte finden Sie unter Tracker-Ticket CORE-4400.

ALERT : Eine weitere Unregelmäßigkeit!

Wenn die WHEN MATCHED-Klausel vorhanden ist und mehrere Datensätze mit einem einzigen Datensatz in der Zieltabelle übereinstimmen, wird ein UPDATE für diesen einen Zieldatensatz für jeden der übereinstimmenden Quelldatensätze ausgeführt, wobei jede nachfolgende Aktualisierung den vorherigen überschreibt . Dieses Verhalten entspricht nicht dem SQL:2003-Standard, der erfordert, dass diese Situation eine Ausnahme (einen Fehler) auslöst.

Dies wurde in Firebird 4 behoben und führt stattdessen zu einem Fehler. Siehe auch CORE-2274

6.6.1. Die RETURNING-Klausel

Eine MERGE-Anweisung, die höchstens eine Zeile betrifft, kann eine RETURNING-Klausel enthalten, um hinzugefügte, geänderte oder entfernte Werte zurückzugeben. Wenn eine RETURNING-Klausel vorhanden ist und mehr als ein übereinstimmender Datensatz gefunden wird, wird ein Fehler “multiple rows in singleton select” ausgegeben. Die RETURNING-Klausel kann beliebige Spalten aus der Zieltabelle (oder aktualisierbaren View) sowie andere Spalten (zB aus der Quelle) und Ausdrücke enthalten.

Die optionale Unterklausel INTO ist nur in PSQL gültig.

Die Einschränkung, dass RETURNING nur mit einer Anweisung verwendet werden kann, die höchstens eine Zeile betrifft, könnte in einer zukünftigen Version entfernt werden.

Spaltennamen können durch das Präfix "OLD" oder "NEW" qualifiziert werden, um genau zu definieren, welcher Wert zurückgegeben werden soll: vor oder nach der Änderung. Die zurückgegebenen Werte enthalten die Änderungen, die von BEFORE-Triggern vorgenommen wurden.

Für die Aktion UPDATE oder INSERT verhalten sich unqualifizierte Spaltennamen oder solche, die durch den Zieltabellennamen oder Alias qualifiziert sind, als ob sie durch NEW qualifiziert wären, während sie für die DELETE Aktion wie durch OLD qualifiziert wären.

Das folgende Beispiel modifiziert das vorherige Beispiel, um eine Zeile zu betreffen, und fügt eine RETURNING-Klausel hinzu, um die alte und neue Warenmenge sowie die Differenz zwischen diesen Werten zurückzugeben.

Verwendung von MERGE *mit einer* RETURNING*-Klausel*

```
MERGE INTO PRODUCT_INVENTORY AS TARGET
USING (
```

```
  SELECT
    SL.ID_PRODUCT,
    SUM(SL.QUANTITY)
  FROM SALES_ORDER_LINE SL
  JOIN SALES_ORDER S ON S.ID = SL.ID_SALES_ORDER
  WHERE S.BYDATE = CURRENT_DATE
  AND SL.ID_PRODUCT =: ID_PRODUCT
  GROUP BY 1
) AS SRC (ID_PRODUCT, QUANTITY)
ON TARGET.ID_PRODUCT = SRC.ID_PRODUCT
WHEN MATCHED AND TARGET.QUANTITY - SRC.QUANTITY <= 0 THEN
  DELETE
WHEN MATCHED THEN
  UPDATE SET
    TARGET.QUANTITY = TARGET.QUANTITY - SRC.QUANTITY,
    TARGET.BYDATE = CURRENT_DATE
RETURNING OLD.QUANTITY, NEW.QUANTITY, SRC.QUANTITY
INTO : OLD_QUANTITY, :NEW_QUANTITY, :DIFF_QUANTITY
```

6.6.2. Beispiele für `MERGE`

1. Aktualisieren Sie Bücher, wenn vorhanden, oder fügen Sie einen neuen Datensatz hinzu, wenn Sie abwesend sind

```
MERGE INTO books b
  USING purchases p
  ON p.title = b.title and p.type = 'bk'
  WHEN MATCHED THEN
    UPDATE SET b.desc = b.desc || '; ' || p.desc
  WHEN NOT MATCHED THEN
    INSERT (title, desc, bought) values (p.title, p.desc, p.bought);
```

2. Verwenden einer abgeleiteten Tabelle

```
MERGE INTO customers c
  USING (SELECT * from customers_delta WHERE id > 10) cd
  ON (c.id = cd.id)
  WHEN MATCHED THEN
    UPDATE SET name = cd.name
  WHEN NOT MATCHED THEN
    INSERT (id, name) values (cd.id, cd.name);
```

3. Zusammen mit einem rekursiven CTE

```
MERGE INTO numbers
  USING (
    WITH RECURSIVE r(n) AS (
```

```
      SELECT 1 FROM rdb$database
      UNION ALL
      SELECT n+1 FROM r WHERE n < 200
    )
    SELECT n FROM r
  ) t
  ON numbers.num = t.n
  WHEN NOT MATCHED THEN
    INSERT(num) VALUES(t.n);
```

4. Verwenden der `DELETE`-Klausel

```
MERGE INTO SALARY_HISTORY
USING (
  SELECT EMP_NO
  FROM EMPLOYEE
  WHERE DEPT_NO = 120) EMP
ON SALARY_HISTORY.EMP_NO = EMP.EMP_NO
WHEN MATCHED THEN DELETE
```

5. Im folgenden Beispiel wird die Tabelle "PRODUCT_INVENTORY" täglich basierend auf den in der Tabelle "SALES_ORDER_LINE" verarbeiteten Bestellungen aktualisiert. Wenn der Lagerbestand des Produkts auf null oder darunter sinken würde, wird die Zeile für dieses Produkt aus der Tabelle `PRODUCT_INVENTORY` entfernt.

```
MERGE INTO PRODUCT_INVENTORY AS TARGET
USING (
  SELECT
    SL.ID_PRODUCT,
    SUM (SL.QUANTITY)
  FROM SALES_ORDER_LINE SL
  JOIN SALES_ORDER S ON S.ID = SL.ID_SALES_ORDER
  WHERE S.BYDATE = CURRENT_DATE
  GROUP BY 1
) AS SRC (ID_PRODUCT, QUANTITY)
ON TARGET.ID_PRODUCT = SRC.ID_PRODUCT
WHEN MATCHED AND TARGET.QUANTITY - SRC.QUANTITY <= 0 THEN
  DELETE
WHEN MATCHED THEN
  UPDATE SET
    TARGET.QUANTITY = TARGET.QUANTITY - SRC.QUANTITY,
    TARGET.BYDATE = CURRENT_DATE
```

Siehe auch

`SELECT`, `INSERT`, `UPDATE`, `UPDATE OR INSERT`, `DELETE`

6.7. EXECUTE PROCEDURE

Verwendet für

Ausführen einer gespeicherten Prozedur

Verfügbar in

DSQL, ESQL, PSQL

Syntax

```
EXECUTE PROCEDURE procname
   [{ <inparam-list | ( <inparam-list> ) }]
   [RETURNING_VALUES { <outvar-list> | ( <outvar-list ) }]

<inparam-list> ::=
  <inparam> [, <inparam> ...]

<outvar-list> ::=
  <outvar> [, <outvar> ...]

<outvar> ::= [:]varname
```

Tabelle 85. Arguments for the `EXECUTE PROCEDURE`*-Anweisungsparameter*

Argument	Beschreibung
procname	Name der gespeicherten Prozedur
inparam	Ein Ausdruck, der den deklarierten Datentyp eines Eingabeparameters auswertet
varname	Eine PSQL-Variable, um den Rückgabewert zu erhalten

Führt eine *ausführbare gespeicherte Prozedur* aus, nimmt eine Liste mit einem oder mehreren Eingabeparametern, falls diese für die Prozedur definiert sind, und gibt einen einzeiligen Satz von Ausgabewerten zurück, wenn sie für die Prozedur definiert sind.

6.7.1. “Executable” Stored Procedure

Die `EXECUTE PROCEDURE`-Anweisung wird am häufigsten verwendet, um den Stil gespeicherter Prozeduren aufzurufen, die geschrieben werden, um auf der Serverseite eine Aufgabe zur Datenänderung auszuführen – solche, die keine `SUSPEND`-Anweisungen in ihrem Code enthalten. Sie können so konzipiert sein, dass sie eine Ergebnismenge, die nur aus einer Zeile besteht, die normalerweise über einen Satz von `RETURNING_VALUES()`-Variablen an eine andere gespeicherte Prozedur übergeben wird, die sie aufruft, zurückgeben. Clientschnittstellen verfügen normalerweise über einen API-Wrapper, der die Ausgabewerte in einen Einzelzeilenpuffer abrufen kann, wenn `EXECUTE PROCEDURE` in DSQL aufgerufen wird.

Das Aufrufen des anderen Stils von Stored Procedures - einer “selectable” - ist mit `EXECUTE PROCEDURE` möglich, aber es gibt nur die erste Zeile eines Ausgabesatzes zurück, der mit ziemlicher Sicherheit mehrzeilig ist. Auswählbare gespeicherte Prozeduren sind so konzipiert, dass sie durch

eine SELECT-Anweisung aufgerufen werden und eine Ausgabe erzeugen, die sich wie eine virtuelle Tabelle verhält.

- In PSQL und DSQL können Eingabeparameter jeder Ausdruck sein, der in den erwarteten Typ aufgelöst wird.
- Obwohl nach dem Namen der gespeicherten Prozedur keine Klammern erforderlich sind, um die Eingabeparameter einzuschließen, wird ihre Verwendung aus Gründen der guten Verwaltung empfohlen.
- Wenn in einer Prozedur Ausgabeparameter definiert wurden, kann die `RETURNING_VALUES'-Klausel in PSQL verwendet werden, um sie in eine Liste zuvor deklarierter Variablen abzurufen, die in Reihenfolge, Datentyp und Anzahl mit den definierten Ausgabeparametern übereinstimmt.
- Die Liste der RETURNING_VALUES kann optional in Klammern eingeschlossen werden und ihre Verwendung wird empfohlen.
- Wenn DSQL-Anwendungen EXECUTE PROCEDURE unter Verwendung der Firebird-API oder einer Form von Wrapper dafür aufrufen, wird ein Puffer zum Empfangen der Ausgabezeile vorbereitet und die RETURNING_VALUES-Klausel wird nicht verwendet.

6.7.2. Beispiele für EXECUTE PROCEDURE

1. In PSQL mit optionalen Doppelpunkten und ohne optionale Klammern:

```
EXECUTE PROCEDURE MakeFullName
  :FirstName, :MiddleName, :LastName
  RETURNING_VALUES :FullName;
```

2. In Firebirds Befehlszeilen-Dienstprogramm *isql*, mit Literalparametern und optionalen Klammern:

```
EXECUTE PROCEDURE MakeFullName ('J', 'Edgar', 'Hoover');
```

In DSQL (zB in *isql*) wird RETURNING_VALUES nicht verwendet. Eventuelle Ausgabewerte werden von der Anwendung erfasst und automatisch angezeigt.

3. Ein PSQL-Beispiel mit Ausdrucksparametern und optionalen Klammern:

```
EXECUTE PROCEDURE MakeFullName
  ('Mr./Mrs. ' || FirstName, MiddleName, upper(LastName))
  RETURNING_VALUES (FullName);
```

6.8. EXECUTE BLOCK

Verwendet für

Erstellen eines "anonymen" Blocks von PSQL-Code in DSQL zur sofortigen Ausführung

Verfügbar in

DSQL

Syntax

```
EXECUTE BLOCK [(<inparams>)]
  [RETURNS (<outparams>)]
  <psql-module-body>

<inparams> ::=  <param_decl> = ? [, <inparams> ]

<outparams> ::=  <param_decl> [, <outparams>]

<param_decl> ::=
  paramname <domain_or_non_array_type> [NOT NULL] [COLLATE collation]

<domain_or_non_array_type> ::=
  !! Siehe auch Skalardatentypen-Syntax !!

<psql-module-body> ::=
  !! Siehe auch Syntax für Modul-Bodys !!
```

Tabelle 86. Argumente für die `EXECUTE BLOCK`*-Anweisungsparameter*

Argument	Beschreibung
param_decl	Name und Beschreibung eines Eingabe- oder Ausgabeparameters
paramname	Der Name eines Eingangs- oder Ausgangsparameters des Verfahrensblocks, bis zu 31 Zeichen lang. Der Name muss unter Ein- und Ausgabeparametern und lokalen Variablen im Block eindeutig sein
collation	Sortierreihenfolge

Führt einen Block von PSQL-Code wie eine gespeicherte Prozedur aus, optional mit Eingabe- und Ausgabeparametern und Variablendeklarationen. Dies ermöglicht dem Benutzer, PSQL "on-the-fly" in einem DSQL-Kontext auszuführen.

6.8.1. Beispiele

1. In diesem Beispiel werden die Zahlen 0 bis 127 und die entsprechenden ASCII-Zeichen in die Tabelle `ASCIITABLE` eingefügt:

```
EXECUTE BLOCK
AS
declare i INT = 0;
```

```
BEGIN
  WHILE (i < 128) DO
  BEGIN
    INSERT INTO AsciiTable VALUES (:i, ascii_char(:i));
    i = i + 1;
  END
END
```

2. Das nächste Beispiel berechnet das geometrische Mittel zweier Zahlen und gibt es an den Benutzer zurück:

```
EXECUTE BLOCK (x DOUBLE PRECISION = ?, y DOUBLE PRECISION = ?)
RETURNS (gmean DOUBLE PRECISION)
AS
BEGIN
  gmean = SQRT(x*y);
  SUSPEND;
END
```

   Da dieser Block Eingangsparameter hat, muss er zuerst vorbereitet werden. Anschließend können die Parameter eingestellt und der Block ausgeführt werden. Es hängt von der Client-Software ab, wie dies zu tun ist und ob es überhaupt möglich ist – siehe die Hinweise unten.

3. Unser letztes Beispiel nimmt zwei ganzzahlige Werte an, `kleinste` und `größte`. Für alle Zahlen im Bereich `kleinste`...`größte` gibt der Block die Zahl selbst, ihr Quadrat, ihren Kubus und ihre vierte Potenz aus.

```
EXECUTE BLOCK (smallest INT = ?, largest INT = ?)
RETURNS (number INT, square BIGINT, cube BIGINT, fourth BIGINT)
AS
BEGIN
  number = smallest;
  WHILE (number <= largest) DO
  BEGIN
    square = number * number;
    cube   = number * square;
    fourth = number * cube;
    SUSPEND;
    number = number + 1;
  END
END
```

   Auch hier hängt es von der Client-Software ab, ob und wie Sie die Parameterwerte einstellen können.

6.8.2. Eingabe- und Ausgabeparameter

Die Ausführung eines Blocks ohne Eingabeparameter sollte mit jedem Firebird-Client möglich sein, der es dem Benutzer erlaubt, eigene DSQL-Anweisungen einzugeben. Wenn es Eingabeparameter gibt, wird es schwieriger: Diese Parameter müssen ihre Werte erhalten, nachdem die Anweisung vorbereitet wurde, aber bevor sie ausgeführt wird. Dies erfordert besondere Vorkehrungen, die nicht jede Client-Anwendung bietet. (Firebirds eigenes *isql* zum Beispiel nicht.)

Der Server akzeptiert nur Fragezeichen ("?") als Platzhalter für die Eingabewerte, nicht "`:a`", "`:MyParam`" etc., oder wörtliche Werte. Client-Software unterstützt jedoch möglicherweise das Formular "`:xxx`" und wird es vorverarbeiten, bevor es an den Server gesendet wird.

Wenn der Block Ausgangsparameter hat, *muss* Sie `SUSPEND` verwenden, sonst wird nichts zurückgegeben.

Die Ausgabe wird immer in Form einer Ergebnismenge zurückgegeben, genau wie bei einer `SELECT` -Anweisung. Sie können `RETURNING_VALUES` nicht verwenden oder den Block `INTO` einige Variablen ausführen, selbst wenn es nur eine Ergebniszeile gibt.

PSQL-Links

Weitere Informationen zum Schreiben von PSQL finden Sie in Kapitel *Procedural SQL (PSQL)-Anweisungen*.

6.8.3. Statement-Terminatoren

Einige Editoren für SQL-Anweisungen – insbesondere das Dienstprogramm *isql*, das mit Firebird geliefert wird, und möglicherweise einige Editoren von Drittanbietern – verwenden eine interne Konvention, die erfordert, dass alle Anweisungen mit einem Semikolon abgeschlossen werden. Dies führt beim Codieren in diesen Umgebungen zu einem Konflikt mit der PSQL-Syntax. Wenn Sie dieses Problem und seine Lösung nicht kennen, lesen Sie bitte die Details im PSQL-Kapitel im Abschnitt Terminator in *isql* umschalten.

Kapitel 7. Prozedurale SQL-Anweisungen (PSQL)

Prozedurales SQL (PSQL) ist eine prozedurale Erweiterung von SQL. Diese Sprachuntermenge wird zum Schreiben von gespeicherten Prozeduren, Triggern und PSQL-Blöcken verwendet.

PSQL bietet alle grundlegenden Konstrukte traditioneller strukturierter Programmiersprachen und enthält auch DML-Anweisungen (`SELECT`, `INSERT`, `UPDATE`, `DELETE` usw.), in einigen Fällen mit geringfügigen Änderungen der Syntax.

7.1. Elemente der PSQL

Eine prozedurale Erweiterung kann Deklarationen von lokalen Variablen und Cursorn, Zuweisungen, bedingten Anweisungen, Schleifen, Anweisungen zum Abrufen von benutzerdefinierten Ausnahmen, Fehlerbehandlung und Senden von Nachrichten (Ereignissen) an Clientanwendungen enthalten. Trigger haben Zugriff auf spezielle Kontextvariablen, zwei Arrays, die die `NEW`-Werte für alle Spalten während der Einfüge- und Aktualisierungsaktivität bzw. die `OLD` -Werte während der Aktualisierungs- und Löscharbeiten speichern.

Anweisungen, die Metadaten ändern (DDL), sind in PSQL nicht verfügbar.

7.1.1. DML-Anweisungen mit Parametern

Wenn DML-Anweisungen (`SELECT`, `INSERT`, `UPDATE`, `DELETE` usw.) im Rumpf des Moduls (Prozedur, Trigger oder Block) Parameter verwenden, können nur benannte Parameter verwendet werden und sie müssen “existieren” bevor die Anweisung diese verwenden kann. Sie können verfügbar gemacht werden, indem sie entweder als Ein- oder Ausgabeparameter im Header des Moduls oder als lokale Variablen in `DECLARE [VARIABLE]`-Anweisungen im unteren Headerbereich deklariert werden.

Wenn eine DML-Anweisung mit Parametern im PSQL-Code enthalten ist, muss dem Parameternamen in den meisten Situationen ein Doppelpunkt (‘:’) vorangestellt werden. Der Doppelpunkt ist in PSQL-spezifischer Anweisungssyntax wie Zuweisungen und Bedingungen optional. Das Doppelpunktpräfix für Parameter ist nicht erforderlich, wenn gespeicherte Prozeduren von einem anderen PSQL-Modul oder in DSQL aufgerufen werden.

7.1.2. Transaktionen

Gespeicherte Prozeduren werden im Kontext der Transaktion ausgeführt, in der sie aufgerufen werden. Trigger werden als ein intrinsischer Teil der Operation der DML-Anweisung ausgeführt: ihre Ausführung befindet sich also innerhalb des gleichen Transaktionskontextes wie die Anweisung selbst. Einzelne Transaktionen werden für Datenbank-Trigger gestartet.

Anweisungen, die Transaktionen starten und beenden, sind in PSQL nicht verfügbar, aber es ist möglich, eine Anweisung oder einen Anweisungsblock in einer autonomen Transaktion auszuführen.

7.1.3. Module Structure

PSQL-Codemodule bestehen aus einem Header und einem Body. Die DDL-Anweisungen zum Definieren dieser sind *komplexe Anweisungen*; das heißt, sie sind Bestandteile einer einzigen Anweisung, die Blöcke von mehreren Anweisungen umfasst. Diese Anweisungen beginnen mit einem Verb (`CREATE`, `ALTER`, `DROP`, `RECREATE`, `CREATE OR ALTER`) und enden mit die letzten `END`-Anweisung des Bodys.

Der Modul-Header

Der Header liefert den Modulnamen und definiert eventuelle Ein- und Ausgabeparameter oder – bei Funktionen – den Rückgabetyp. Gespeicherte Prozeduren und PSQL-Blöcke können Eingabe- und Ausgabeparameter haben. Funktionen können Eingabeparameter haben und müssen einen skalaren Rückgabetyp haben. Trigger haben weder Eingabe- noch Ausgabeparameter.

Der Header eines Triggers gibt das Datenbankereignis (insert, update oder delete oder eine Kombination) und die Betriebsphase (`BEFORE` oder `AFTER` this event) an, die zum "Auslösen" des Triggers führt.

Der Modul-Body

Der Modulrumpf ist entweder ein PSQL-Modulrumpf oder ein externer Modulrumpf.

Syntax eines Modulkörpers

```
<module-body> ::=
  <psql-module-body> | <external-module-body>

<psql-module-body> ::=
  AS
    [<declarations>]
  BEGIN
    [<PSQL_statements>]
  END

<external-module-body> ::=
  EXTERNAL [NAME <extname>] ENGINE engine
  [AS '<extbody>']

<declarations> ::= <declare-item> [<declare-item ...]

<declare-item> ::=
    <declare-var>
  | <declare-cursor>
  | <declare-subfunc>
  | <declare-subproc>

<extname> ::=
  '<module-name>!<routine-name>[!<misc-info>]'
```

Tabelle 87. Modulkörperparameter

Parameter	Beschreibung
declarations	Abschnitt zum Deklarieren lokaler Variablen, benannter Cursor und Unterprogramme
PSQL_statements	Prozedurale SQL-Anweisungen. Einige PSQL-Anweisungen sind möglicherweise nicht in allen PSQL-Typen gültig. Zum Beispiel ist `RETURN <Wert>;` nur in Funktionen gültig.
declare_var	Lokale Variablendeklaration
declare_cursor	Benannte Cursor-Deklaration
declare-subfunc	Unterfunktionsdeklaration
declare-subproc	Unterverfahrenserklärung
extname	String, der die externe Prozedur identifiziert
engine	String, der die UDR-Engine identifiziert
extbody	Externer Verfahrenskörper. Ein Zeichenfolgenliteral, das von UDRs für verschiedene Zwecke verwendet werden kann.
module-name	Der Name des Moduls, das die Prozedur enthält
routine-name	Der interne Name der Prozedur innerhalb des externen Moduls
misc-info	Optionaler String, der an die Prozedur im externen Modul übergeben wird

Der PSQL-Modul-Body

Der PSQL-Hauptteil beginnt mit einem optionalen Abschnitt, der Variablen und Subroutinen deklariert, gefolgt von einem Block von Anweisungen, die in einer logischen Reihenfolge wie ein Programm ausgeführt werden. Ein Block von Anweisungen – oder zusammengesetzte Anweisungen – wird von den Schlüsselwörtern `BEGIN` und `END` eingeschlossen und als einzelne Codeeinheit ausgeführt. Der Hauptblock `BEGIN···END` kann eine beliebige Anzahl anderer `BEGIN···END` Blöcke enthalten, sowohl eingebettet als auch sequentiell. Blöcke können bis zu einer maximalen Tiefe von 512 Blöcken verschachtelt werden. Alle Anweisungen außer `BEGIN` und `END` werden durch Semikolons (``;'') abgeschlossen. Kein anderes Zeichen ist als Abschlusszeichen für PSQL-Anweisungen gültig.

In der Firebird 2.5 Language Reference wurde die Deklaration lokaler Variablen und Cursor als Teil des Modulheaders betrachtet. Mit der Einführung von UDR (externe Routinen) in Firebird 3.0 betrachten wir diesen Deklarationsabschnitt nun als Teil des — PSQL — Modulrumpfs.

Umschalten des Terminators in *isql*

Hier werden wir ein wenig abschweifen, um zu erklären, wie man das Terminatorzeichen im Dienstprogramm *isql* umschaltet, um es zu ermöglichen, PSQL-Module in dieser Umgebung zu definieren, ohne mit *isql* selbst in Konflikt zu geraten, da *isql* dasselbe Zeichen, Semikolon

(‘;’), als eigenen Anweisungsabschluss verwendet.

isql-Befehl SET TERM

Verwendet für

Ändern des Terminatorzeichens, um Konflikte mit dem Terminatorzeichen in PSQL-Anweisungen zu vermeiden

Verfügbar in

nur in ISQL

Syntax

```
SET TERM new_terminator old_terminator
```

Tabelle 88. `SET TERM`*-Parameter*

Argument	Beschreibung
new_terminator	Neuer Terminator
old_terminator	Alter Terminator

Wenn Sie Ihre Trigger und gespeicherten Prozeduren in *isql* schreiben – entweder in der interaktiven Schnittstelle oder in Skripten – ist die Ausführung einer `SET TERM`-Anweisung erforderlich, um das normale *isql*-Anweisungs-Terminator vom Semikolon in ein anderes Zeichen oder eine andere kurze Zeichenfolge umzuschalten vermeiden Sie Konflikte mit dem nicht änderbaren Semikolon-Terminator in PSQL. Der Wechsel zu einem alternativen Terminator muss erfolgen, bevor Sie mit der Definition von PSQL-Objekten oder dem Ausführen Ihrer Skripts beginnen.

Das alternative Abschlusszeichen kann eine beliebige Zeichenfolge sein, mit Ausnahme eines Leerzeichens, eines Apostrophs oder der aktuellen Abschlusszeichen. Bei allen verwendeten Buchstaben muss die Groß-/Kleinschreibung beachtet werden.

Beispiel

Das Standard-Semikolon in ‘^’ (Caret) ändern und es verwenden, um eine Stored-Procedure-Definition zu übergeben: Zeichen als alternatives Abschlusszeichen:

```
SET TERM ^;

CREATE OR ALTER PROCEDURE SHIP_ORDER (
  PO_NUM CHAR(8))
AS
BEGIN
  /* Stored procedure body */
END^

/* Other stored procedures and triggers */
```

```
SET TERM ;^

/* Other DDL statements */
```

Der externe Modul-Body

Der externe Modulrumpf gibt die UDR-Engine an, die zum Ausführen des externen Moduls verwendet wird, und gibt optional den Namen der aufzurufenden UDR-Routine (<*extname*>) und/oder einen String (<*extbody*>) mit UDR-spezifischer Semantik an .

Die Konfiguration externer Module und UDR-Engines wird in dieser Sprachreferenz nicht weiter behandelt. Weitere Informationen finden Sie in der Dokumentation einer bestimmten UDR-Engine.

7.2. Gespeicherte Prozeduren

Eine gespeicherte Prozedur ist ein Programm, das in den Datenbankmetadaten zur Ausführung auf dem Server gespeichert ist. Eine gespeicherte Prozedur kann durch gespeicherte Prozeduren (einschließlich sich selbst), Trigger und Clientanwendungen aufgerufen werden. Eine Prozedur, die sich selbst aufruft, heißt *rekursiv*.

7.2.1. Vorteile von gespeicherten Prozeduren

Gespeicherte Prozeduren besitzen die folgenden Vorteile:

Modularität	Anwendungen, die mit der Datenbank arbeiten, können die gleiche gespeicherte Prozedur verwenden, wodurch die Größe des Anwendungscodes reduziert wird und eine Codeduplizierung vermieden wird.
Vereinfachte Anwendungsunterstützung	Wenn eine gespeicherte Prozedur geändert wird, werden Änderungen sofort allen Host-Anwendungen angezeigt, ohne dass sie bei unveränderten Parametern neu kompiliert werden müssen.
Verbesserte Leistung	Da gespeicherte Prozeduren auf einem Server statt auf dem Client ausgeführt werden, wird der Netzwerkverkehr reduziert, wodurch die Leistung verbessert wird.

7.2.2. Varianten der gespeicherten Prozeduren

Firebird untertützt zwei Arten der gespeicherten Prozeduren: *executable* (ausführbar) *selectable* (abfragbar).

Ausführbare Prozeduren

Ausführbare Prozeduren ändern normalerweise Daten in einer Datenbank. Sie können Eingabeparameter empfangen und einen einzigen Satz von Ausgabeparametern (`RETURNS`) zurückgeben. Sie werden mit der Anweisung `EXECUTE PROCEDURE` aufgerufen. Siehe auch ein Beispiel für eine ausführbare gespeicherte Prozedur am Ende des Abschnitts `CREATE PROCEDURE` von Kapitel 5.

Abfragbare Prozeduren

Abfragbare bzw. auswählbare gespeicherte Prozeduren rufen normalerweise Daten aus einer Datenbank ab und geben eine beliebige Anzahl von Zeilen an den Aufrufer zurück. Der Aufrufer erhält die Ausgabe Zeile für Zeile aus einem Zeilenpuffer, der von der Datenbank-Engine darauf vorbereitet wird.

Auswählbare Prozeduren können nützlich sein, um komplexe Datensätze zu erhalten, die mit regulären DSQL-SELECT-Abfragen oft unmöglich oder zu schwierig oder zu langsam abzurufen sind. Typischerweise durchläuft diese Art der Prozedur einen Schleifenprozess zum Extrahieren von Daten, möglicherweise transformiert er sie, bevor die Ausgabevariablen (Parameter) bei jeder Iteration der Schleife mit neuen Daten gefüllt werden. Eine `SUSPEND`-Anweisung am Ende der Iteration füllt den Puffer und wartet darauf, dass der Aufrufer die Zeile abruft. Die Ausführung der nächsten Iteration der Schleife beginnt, wenn der Puffer gelöscht wurde.

Auswählbare Prozeduren können Eingabeparameter haben, und die Ausgabemenge wird durch die `RETURNS`-Klausel im Header angegeben.

Eine auswählbare gespeicherte Prozedur wird mit einer `SELECT`-Anweisung aufgerufen. Siehe ein Beispiel für eine auswählbare gespeicherte Prozedur am Ende von `CREATE PROCEDURE` section von Kapitel 5.

7.2.3. Erstellen einer gespeicherten Prozedur

Die Syntax zum Erstellen ausführbarer gespeicherter Prozeduren und abfragbarer gespeicherter Prozeduren ist exakt gleich. Der Unterschied liegt in der Logik des Programmcodes.

Informationen zum Erstellen gespeicherter Prozeduren finden Sie unter `CREATE PROCEDURE` in Kapitel *Datendefinitionsanweisungen (DDL)*.

7.2.4. Anpassen einer gespeicherten Prozedur

Eine vorhandene gespeicherte Prozedur kann geändert werden, um die Sätze von Ein- und Ausgabeparametern und alles im Prozedurhauptteil zu ändern.

7.2.5. Löschen einer gespeicherte Prozedur

Die Anweisung `DROP PROCEDURE` wird verwendet um gespeicherte Prozeduren zu löschen.

7.3. Gespeicherte Funktionen

Eine gespeicherte Funktion ist ausführbarer Code, der in den Datenbankmetadaten zur Ausführung auf dem Server gespeichert ist. Eine gespeicherte Funktion kann von anderen gespeicherten Funktionen (einschließlich sich selbst), Prozeduren, Triggern und Clientanwendungen aufgerufen werden. Eine Funktion, die sich selbst aufruft, wird als *rekursiv* bezeichnet.

Im Gegensatz zu gespeicherten Prozeduren geben gespeicherte Funktionen immer einen Skalarwert zurück. Um einen Wert aus einer gespeicherten Funktion zurückzugeben, verwenden Sie die `RETURN`-Anweisung, die die Funktion sofort beendet.

7.3.1. Erstellen einer gespeicherten Funktion

Informationen zum Erstellen gespeicherter Funktionen finden Sie unter `CREATE FUNCTION` im Kapitel *Anweisungen zur Datendefinition (DDL)*.

7.3.2. Ändern einer gespeicherten Funktion

Informationen zum Ändern gespeicherter Funktionen finden Sie unter `ALTER FUNCTION`, `CREATE OR ALTER FUNCTION`, `RECREATE FUNCTION`, im Kapitel *Datendefinitions-(DDL-)Anweisungen*.

7.3.3. Löschen einer gespeicherten Funktion

Informationen zum Löschen gespeicherter Prozeduren finden Sie unter `DROP FUNCTION` in Kapitel *Anweisungen zur Datendefinition (DDL)*.

7.4. PSQL-Blöcke

Ein in sich geschlossener, unbenannte ("'anonymous'") Block von PSQL-Code kann dynamisch in DSQL ausgeführt werden, unter Verwendung der `EXECUTE BLOCK`-Syntax. Der Header eines anonymen PSQL-Blocks kann optional Eingabe- und Ausgabeparameter enthalten. Der Hauptteil kann lokale Variablen, Cursor-Deklarationen und lokale Routinen enthalten, gefolgt von einem Block von PSQL-Anweisungen.

Ein anonymer PSQL-Block wird im Gegensatz zu gespeicherten Prozeduren und Triggern nicht als Objekt definiert und gespeichert. Es wird zur Laufzeit ausgeführt und kann nicht auf sich selbst verweisen.

Genau wie gespeicherte Prozeduren können anonyme PSQL-Blöcke verwendet werden, um Daten zu verarbeiten und Daten aus der Datenbank abzurufen.

Syntax (unvollständig)

```
EXECUTE BLOCK
  [(<inparam> = ? [, <inparam> = ? ...])]
  [RETURNS (<outparam> [, <outparam> ...])]
  <psql-module-body>

<psql-module-body> ::=
```

```
!! Siehe Syntax des Modul-Bodys !!
```

Tabelle 89. PSQL Block Parameters

Argument	Beschreibung
inparam	Beschreibung der Eingabeparameter
outparam	Beschreibung der Ausgangsparameter
declarations	Ein Abschnitt zum Deklarieren lokaler Variablen und benannter Cursor
PSQL statements	PSQL- und DML-Anweisungen

Siehe auch

Siehe auch `EXECUTE BLOCK` für weitere Details.

7.5. Pakete

Ein Paket ist eine Gruppe von gespeicherten Prozeduren und Funktionen, die als einzelnes Datenbankobjekt definiert sind.

Firebird-Pakete bestehen aus zwei Teilen: einem Header (`PACKAGE`-Schlüsselwort) und einem Body (`PACKAGE BODY`-Schlüsselwort). Diese Trennung ist Delphi-Modulen sehr ähnlich, der Header entspricht dem Schnittstellenteil und der Rumpf entspricht dem Implementierungsteil.

7.5.1. Vorteile von Paketen

Der Begriff “Paketieren” der Codekomponenten einer Datenbankoperation hat mehrere Vorteile:

Modularisierung

Blöcke von voneinander abhängigem Code werden in logische Module gruppiert, wie es in anderen Programmiersprachen der Fall ist.

In der Programmierung ist bekannt, dass es eine gute Sache ist, Code auf verschiedene Weise zu gruppieren, beispielsweise in Namespaces, Units oder Klassen. Dies ist mit standardmäßigen gespeicherten Prozeduren und Funktionen in der Datenbank nicht möglich. Obwohl sie in verschiedene Skriptdateien gruppiert werden können, bleiben zwei Probleme:

a. Die Gruppierung wird nicht in den Datenbankmetadaten dargestellt.
b. Skriptierte Routinen nehmen alle an einem flachen Namensraum teil und können von jedem aufgerufen werden (wir beziehen uns hier nicht auf Sicherheitsberechtigungen).

Einfachere Verfolgung von Abhängigkeiten

Pakete erleichtern das Nachverfolgen von Abhängigkeiten zwischen einer Sammlung verwandter Routinen sowie zwischen dieser Sammlung und anderen gepackten und nicht gepackten Routinen.

Immer wenn eine gepackte Routine feststellt, dass sie ein bestimmtes Datenbankobjekt verwendet, wird eine Abhängigkeit von diesem Objekt in den Systemtabellen von Firebird registriert. Um das Objekt anschließend zu löschen oder möglicherweise zu ändern, müssen Sie

zuerst die davon abhängigen Elemente entfernen. Da die Abhängigkeit von anderen Objekten nur für den Paketkörper existiert und nicht für den Paketkörper, kann dieser Paketkörper leicht entfernt werden, selbst wenn ein anderes Objekt von diesem Paket abhängt. Wenn der Körper gelöscht wird, bleibt der Header erhalten, sodass Sie seinen Körper neu erstellen können, sobald die Änderungen in Bezug auf das entfernte Objekt abgeschlossen sind.

Berechtigungsverwaltung vereinfachen

Da Firebird Routinen mit den Anrufer-Privilegien ausführt, ist es auch notwendig, jeder Routine die Ressourcennutzung zu gewähren, wenn diese Ressourcen für den Anrufer nicht direkt zugänglich wären. Die Verwendung jeder Routine muss Benutzern und/oder Rollen gewährt werden.

Gepackte Routinen haben keine individuellen Privilegien. Die Privilegien gelten für das Paket als Ganzes. Den Paketen gewährte Privilegien gelten für alle Paketrumpfroutinen, einschließlich der privaten, werden jedoch für den Paketheader gespeichert. Ein `EXECUTE`-Privileg für ein Paket, das einem Benutzer (oder einem anderen Objekt) gewährt wird, gewährt diesem Benutzer das Privileg, alle im Paket-Header definierten Routinen auszuführen.

Zum Beispiel

```
GRANT SELECT ON TABLE secret TO PACKAGE pk_secret;
GRANT EXECUTE ON PACKAGE pk_secret TO ROLE role_secret;
```

Private Bereiche

Gespeicherte Prozeduren und Funktionen können privat sein; das heißt, sie werden nur für die interne Verwendung innerhalb des definierenden Pakets verfügbar gemacht.

Alle Programmiersprachen haben den Begriff des Routineumfangs, der ohne irgendeine Form der Gruppierung nicht möglich ist. Firebird-Pakete funktionieren in dieser Hinsicht auch wie Delphi-Einheiten. Wenn eine Routine nicht im Paketheader (Schnittstelle) deklariert und im Rumpf implementiert ist (Implementierung), wird sie zu einer privaten Routine. Eine private Routine kann nur innerhalb ihres Pakets aufgerufen werden.

7.5.2. Erstellen eines Pakets

Informationen zum Erstellen von Paketen finden Sie unter `CREATE PACKAGE`, `CREATE PACKAGE BODY`

7.5.3. Ändern eines Pakets

Informationen zum Ändern vorhandener Paketköpfe oder -körper, siehe auch `ALTER PACKAGE`, `CREATE OR ALTER PACKAGE`, `RECREATE PACKAGE`, `ALTER PACKAGE BODY`, `RECREATE PACKAGE BODY`

7.5.4. Löschen eines Pakets

Informationen zum Löschen eines Pakets finden Sie unter `DROP PACKAGE`, `DROP PACKAGE BODY`

7.6. Trigger

Ein Trigger ist eine andere Form von ausführbarem Code, der in den Metadaten der Datenbank zur Ausführung durch den Server gespeichert wird. Ein Trigger kann nicht direkt aufgerufen werden. Er wird automatisch aufgerufen (“gefeuert”), wenn Datenänderungsereignisse mit einer bestimmten Tabelle oder Sicht (View) auftreten.

Ein Trigger gilt für genau eine Tabelle oder Sicht und nur eine *Phase* in einem Ereignis (vor (`BEFORE`) oder nach (`AFTER`) dem Ereignis). Ein einzelner Trigger kann nur dann ausgelöst werden, wenn ein bestimmtes Datenänderungsereignis auftritt (`INSERT` / `UPDATE` / `DELETE`) oder wenn es auf mehr als eines dieser Ereignisse angewendet werden soll.

Ein DML-Trigger wird im Kontext der Transaktion ausgeführt, in der die datenändernde DML-Anweisung ausgeführt wird. Bei Triggern, die auf Datenbankereignisse reagieren, ist die Regel unterschiedlich: Für einige von ihnen wird eine Standardtransaktion gestartet.

7.6.1. Reihenfolge der Ausführung

Für jede Phase-Ereignis-Kombination kann mehr als ein Trigger definiert werden. Die Reihenfolge, in der sie ausgeführt werden (bekannt als “firing order”, kann explizit mit dem optionalen Argument `POSITION` in der Triggerdefinition angegeben werden.) Sie haben 32.767 Nummern zur Auswahl. Die niedrigsten Positionsnummern feuern zuerst.

Wenn eine Klausel `POSITION` weggelassen wird oder mehrere übereinstimmende Ereignisphasen-Trigger die gleiche Positionsnummer haben, werden die Trigger in alphabetischer Reihenfolge ausgelöst.

7.6.2. DML-Trigger

DML-Trigger sind solche, die ausgelöst werden, wenn eine DML-Operation den Datenstatus ändert: Zeilen in Tabellen ändern, neue Zeilen einfügen oder Zeilen löschen. Sie können sowohl für Tabellen als auch für Ansichten definiert werden.

Trigger-Optionen

Für die Ereignis-Phasen-Kombination für Tabellen und Ansichten stehen sechs Basisoptionen zur Verfügung:

Bevor eine neue Zeile eingefügt wird	`BEFORE INSERT`
Nachdem eine neue Zeile eingefügt wurde	`AFTER INSERT`
Bevor eine Zeile aktualisiert wird	`BEFORE UPDATE`
Nachdem eine Zeile aktualisiert wurde	`AFTER UPDATE`
Bevor eine Zeile gelöscht wird	`BEFORE DELETE`
Nachdem eine Zeile gelöscht wurde	`AFTER DELETE`

Diese Basisformulare dienen zum Erstellen von Einzelphasen- / Einzelereignisauslösern. Firebird unterstützt auch Formulare zum Erstellen von Auslösern für eine Phase und mehrere Ereignisse,

z.B. BEFORE INSERT OR UPDATE OR DELETE, oder AFTER UPDATE OR DELETE: Die Kombinationen unterliegen Ihrer Wahl.

"Multiphasen-"-Trigger, wie BEFORE OR AFTER..., sind nicht möglich.

Die booleschen Kontextvariablen INSERTING, UPDATING und DELETING können im Hauptteil eines Triggers, um die Art des Ereignisses zu bestimmen, das den Trigger ausgelöst hat.

Kontextvariablen OLD und NEW

Für DML-Trigger bietet die Firebird-Engine Zugriff auf Sätze von 'OLD'- und 'NEW'-Kontextvariablen. Jeder ist ein Array der Werte der gesamten Zeile: einer für die Werte, wie sie vor dem Datenänderungsereignis sind (die 'BEFORE'-Phase) und einer für die Werte, wie sie nach dem Ereignis (die 'AFTER'-Phase) sein werden). Sie werden in Anweisungen in der Form NEW.column_name bzw. OLD.column_name referenziert. Der *column_name* kann eine beliebige Spalte in der Tabellendefinition sein, nicht nur die, die aktualisiert werden.

Die Variablen NEW und OLD unterliegen einigen Regeln:

- In allen Triggern ist der 'OLD'-Wert schreibgeschützt
- In BEFORE UPDATE- und BEFORE INSERT-Code ist der NEW-Wert lesen/schreiben, es sei denn, es handelt sich um eine COMPUTED BY-Spalte
- In INSERT-Triggern sind Verweise auf die OLD-Variablen ungültig und lösen eine Ausnahme aus
- In DELETE-Triggern sind Verweise auf die NEW-Variablen ungültig und lösen eine Ausnahme aus
- In allen 'AFTER'-Triggercodes sind die 'NEW'-Variablen schreibgeschützt

7.6.3. Datenbank-Trigger

Ein mit einer Datenbank oder einem Transaktionsereignis verknüpfter Trigger kann für die folgenden Ereignisse definiert werden:

Verbindung mit einer Datenbank herstellen	ON CONNECT	Bevor der Trigger ausgeführt wird, wird automatisch eine Standardtransaktion gestartet
Trennen von einer Datenbank	ON DISCONNECT	Bevor der Trigger ausgeführt wird, wird automatisch eine Standardtransaktion gestartet
Wenn eine Transaktion gestartet wird	ON TRANSACTION START	Der Trigger wird im aktuellen Transaktionskontext ausgeführt
Wenn eine Transaktion übergeben wird	ON TRANSACTION COMMIT	Der Trigger wird im aktuellen Transaktionskontext ausgeführt
Wenn eine Transaktion abgebrochen wird	ON TRANSACTION ROLLBACK	Der Trigger wird im aktuellen Transaktionskontext ausgeführt

7.6.4. DDL-Trigger

DDL löst bei bestimmten Metadatenänderungsereignissen in einer bestimmten Phase die Auslösung

aus. BEFORE-Trigger werden vor Änderungen an Systemtabellen ausgeführt. AFTER-Trigger werden nach Änderungen in Systemtabellen ausgeführt.

DDL-Trigger sind eine spezielle Art von Datenbank-Triggern, daher gelten die meisten Regeln und Semantiken von Datenbank-Triggern auch für DDL-Trigger.

Semantik

1. BEFORE-Trigger werden vor Änderungen an den Systemtabellen ausgelöst. 'AFTER'-Trigger werden nach Änderungen der Systemtabelle ausgelöst.

Wichtige Regel

Der Ereignistyp [BEFORE | AFTER] eines DDL-Triggers kann nicht geändert werden.

2. Wenn eine DDL-Anweisung einen Trigger auslöst, der eine Ausnahme auslöst (BEFORE oder AFTER, absichtlich oder unabsichtlich), wird die Anweisung nicht festgeschrieben. Das heißt, Ausnahmen können verwendet werden, um sicherzustellen, dass ein DDL-Vorgang fehlschlägt, wenn die Bedingungen nicht genau wie beabsichtigt sind.
3. DDL-Trigger-Aktionen werden nur ausgeführt, wenn die Transaktion, in der der betroffene DDL-Befehl ausgeführt wird, *commiting* ist. Übersehen Sie nie die Tatsache, dass in einem AFTER -Trigger genau das möglich ist, was nach einem DDL-Befehl ohne Autocommit möglich ist. Sie können beispielsweise keine Tabelle erstellen und diese dann im Trigger verwenden.
4. Bei "CREATE OR ALTER"-Anweisungen wird je nach vorheriger Existenz des Objekts einmalig ein Trigger beim CREATE-Ereignis oder beim ALTER-Ereignis ausgelöst. Bei RECREATE-Anweisungen wird ein Trigger für das DROP-Ereignis ausgelöst, wenn das Objekt existiert, und für das CREATE -Ereignis.
5. ALTER- und DROP-Ereignisse werden im Allgemeinen nicht ausgelöst, wenn der Objektname nicht existiert. Ausnahme siehe Punkt 6.
6. Die Ausnahme von Regel 5 ist, dass BEFORE ALTER/DROP USER das Feuer auslöst, auch wenn der Benutzername nicht existiert. Dies liegt daran, dass diese Befehle darunter DML in der Sicherheitsdatenbank ausführen und die Überprüfung nicht durchgeführt wird, bevor der Befehl darauf ausgeführt wird. Dies ist bei eingebetteten Benutzern wahrscheinlich anders, schreiben Sie also keinen Code, der davon abhängt.
7. Wenn eine Ausnahme ausgelöst wird, nachdem der DDL-Befehl seine Ausführung gestartet hat und bevor 'AFTER'-Trigger ausgelöst werden, werden 'AFTER'-Trigger nicht ausgelöst.
8. Verpackte Prozeduren und Trigger lösen einzelne {CREATE | ÄNDERN | DROP} {VERFAHREN | FUNCTION} auslöst.

Der DDL_TRIGGER-Kontext-Namespace

Wenn ein DDL-Trigger ausgeführt wird, steht der Namespace DDL_TRIGGER für die Verwendung mit RDB$GET_CONTEXT zur Verfügung. Dieser Namespace enthält Informationen zum aktuell ausgelösten Trigger.

Siehe auch Der DDL_TRIGGER-Namespace im Abschnitt RDB$GET_CONTEXT im Kapitel *Eingebaute*

Skalarfunktionen.

7.6.5. Trigger erstellen

Informationen zum Erstellen von Triggern finden Sie unter `CREATE TRIGGER`, `CREATE OR ALTER TRIGGER`, `RECREATE TRIGGER` im Kapitel *Datendefinitions-(DDL-)Anweisungen*.

7.6.6. Trigger ändern

Informationen zum Ändern von Triggern finden Sie unter `ALTER TRIGGER`, `CREATE OR ALTER TRIGGER`, `RECREATE TRIGGER` im Kapitel *Anweisungen zur Datendefinition (DDL)*.

7.6.7. Trigger löschen

Informationen zum Löschen von Triggern finden Sie unter `DROP TRIGGER` im Kapitel *Anweisungen zur Datendefinition (DDL)*.

7.7. Schreiben des Body-Codes

In diesem Abschnitt werden die prozeduralen SQL-Sprachkonstrukte und -Anweisungen näher betrachtet, die zum Codieren des Rumpfs einer gespeicherten Prozedur, eines Triggers oder eines anonymen PSQL-Blocks verfügbar sind.

Doppelpunktmarkierung (':')

Das Doppelpunkt-Markierungspräfix (':') wird in PSQL verwendet, um eine Referenz auf eine Variable in einer DML-Anweisung zu markieren. Der Doppelpunkt-Marker ist vor Variablennamen in anderem PSQL-Code nicht erforderlich.

Seit Firebird 3.0 kann der Doppelpunkt-Präfix auch für die Kontexte `NEW` und `OLD` sowie für Cursor-Variablen verwendet werden.

7.7.1. Zuweisungs-Statements

Verwendet für

Zuweisen eines Werts zu einer Variablen

Verfügbar in

PSQL

Syntax

```
varname = <value_expr>;
```

Tabelle 90. Zuweisungs-Statement-Parameter

Argument	Beschreibung
varname	Name eines Parameters oder einer lokalen Variablen
value_expr	Ein Ausdruck, eine Konstante oder eine Variable, dessen Wert in den gleichen Datentyp wie *varname*

PSQL verwendet das Äquivalenzsymbol ('=') als Zuweisungsoperator. Die Zuweisungsanweisung weist der Variablen links vom Operator den rechten SQL-Ausdruckswert zu. Der Ausdruck kann ein beliebiger gültiger SQL-Ausdruck sein: Er kann Literale, interne Variablennamen, Arithmetik-, logische und Zeichenfolgenoperationen, Aufrufe von internen Funktionen oder externe Funktionen (UDFs) enthalten.

Beispiel mit Zuweisungsanweisungen

```
CREATE PROCEDURE MYPROC (
  a INTEGER,
  b INTEGER,
  name VARCHAR (30)
)
RETURNS (
  c INTEGER,
  str VARCHAR(100))
AS
BEGIN
  -- assigning a constant
  c = 0;
  str = '';
  SUSPEND;
  -- assigning expression values
  c = a + b;
  str = name || CAST(b AS VARCHAR(10));
  SUSPEND;
  -- assigning expression value
  -- built by a query
  c = (SELECT 1 FROM rdb$database);
  -- assigning a value from a context variable
  str = CURRENT_USER;
  SUSPEND;
END
```

Siehe auch

DECLARE VARIABLE

7.7.2. DECLARE VARIABLE

Verwendet für

Eine lokale Variable deklarieren

Verfügbar in

PSQL

Syntax

```
DECLARE [VARIABLE] varname
  <domain_or_non_array_type> [NOT NULL] [COLLATE collation]
  [{DEFAULT | = } <initvalue>];

<domain_or_non_array_type> ::=
  !! Siehe auch Skalardatentypen !!

<initvalue> ::= <literal> | <context_var>
```

Tabelle 91. `DECLARE VARIABLE`-*Anweisungsparameter*

Argument	Beschreibung
varname	Name der lokalen Variablen
collation	Sortierreihenfolge
initvalue	Anfangswert für diese Variable
literal	Literal eines Typs, der mit dem Typ der lokalen Variablen kompatibel ist
context_var	Jede Kontextvariable, deren Typ mit dem Typ der lokalen Variablen kompatibel ist

Die Anweisung `DECLARE [VARIABLE]` wird verwendet, um eine lokale Variable zu deklarieren. Das Schlüsselwort `VARIABLE` kann weggelassen werden. Für jede lokale Variable ist eine `DECLARE [VARIABLE]`-Anweisung erforderlich. Es können beliebig viele `DECLARE [VARIABLE]`-Anweisungen in beliebiger Reihenfolge eingefügt werden. Der Name einer lokalen Variablen muss unter den Namen der für das Modul deklarierten lokalen Variablen und Ein- und Ausgabeparameter eindeutig sein.

Ein Sonderfall von `DECLARE [VARIABLE]` — das Deklarieren von Cursorn — wird separat in `DECLARE .. CURSOR` . behandelt

Datentyp für Variablen

Eine lokale Variable kann einen beliebigen SQL-Typ außer einem Array sein.

- Als Typ kann ein Domainname angegeben werden; die Variable erbt alle ihre Attribute.
- Wenn stattdessen die Klausel `TYPE OF domain` verwendet wird, erbt die Variable nur den Datentyp der Domäne und gegebenenfalls deren Zeichensatz- und Kollatierungsattribute. Alle Standardwerte oder Einschränkungen wie `NOT NULL` oder `CHECK` Einschränkungen werden nicht vererbt.
- Wenn die Option `TYPE OF COLUMN relation.column>` verwendet wird, um aus einer Spalte in einer Tabelle oder Ansicht zu „borgen“, erbt die Variable nur den Datentyp der Spalte und gegebenenfalls den Zeichensatz und die Kollatierung Attribute. Alle anderen Attribute werden ignoriert.

NICHT NULL-Beschränkung

Für lokale Variablen können Sie die Einschränkung NOT NULL angeben, wodurch NULL-Werte für die Variable nicht zugelassen werden. Wenn als Datentyp eine Domäne angegeben wurde und die Domäne bereits die Einschränkung NOT NULL hat, ist die Deklaration unnötig. Für andere Formen, einschließlich der Verwendung einer Domäne, die null zulässt, kann die Einschränkung NOT NULL bei Bedarf eingefügt werden.

CHARACTER SET- und COLLATE-Klauseln

Sofern nicht anders angegeben, sind der Zeichensatz und die Kollatierungssequenz einer String-Variablen die Datenbank-Standardwerte. Eine CHARACTER SET-Klausel kann bei Bedarf eingefügt werden, um Zeichenfolgendaten zu verarbeiten, die in einem anderen Zeichensatz vorliegen. Eine gültige Kollatierungssequenz (COLLATE-Klausel) kann auch mit oder ohne Zeichensatz-Klausel eingeschlossen werden.

Initialisieren einer Variablen

Lokale Variablen sind NULL, wenn die Ausführung des Moduls beginnt. Sie können initialisiert werden, so dass ein Start- oder Standardwert verfügbar ist, wenn sie zum ersten Mal referenziert werden. Es kann die Form DEFAULT <initvalue> verwendet werden oder nur der Zuweisungsoperator '=': = <initvalue>. Der Wert kann ein beliebiges typkompatibles Literal oder eine Kontextvariable sein, einschließlich NULL.

Stellen Sie sicher, dass Sie diese Klausel für alle Variablen verwenden, die eine NOT NULL-Beschränkung haben und für die sonst kein Standardwert verfügbar ist.

Beispiele für verschiedene Möglichkeiten, lokale Variablen zu deklarieren

```
CREATE OR ALTER PROCEDURE SOME_PROC
AS
  -- Deklaration einer Variablen vom Typ INT
   DECLARE I INT;
   -- Eine Variable vom Typ INT deklarieren, die NULL nicht zulässt
   DECLARE VARIABLE J INT NOT NULL;
   -- Deklarieren einer Variablen vom Typ INT mit dem Standardwert 0
   DECLARE VARIABLE K INT DEFAULT 0;
   -- Deklarieren einer Variablen vom Typ INT mit dem Standardwert 1
   DECLARE VARIABLE L INT = 1;
   -- Deklarieren einer Variablen basierend auf der COUNTRYNAME-Domain
   DECLARE FARM_COUNTRY COUNTRYNAME;
   -- Deklarieren einer Variablen des Typs gleich der Domäne COUNTRYNAME
   DECLARE FROM_COUNTRY TYPE OF COUNTRYNAME;
   -- Deklarieren einer Variablen mit dem Typ der Spalte CAPITAL in der Tabelle
COUNTRY
   DECLARE CAPITAL TYPE OF COLUMN COUNTRY.CAPITAL;
BEGIN
  /* PSQL-Anweisungen */
END
```

Siehe auch

Datentypen und Unterdatentypen, Benutzerdefinierte Datentypen – Domains, `CREATE DOMAIN`

7.7.3. DECLARE .. CURSOR

Verwendet für

Deklarieren eines benannten Cursors

Verfügbar in

PSQL

Syntax

```
DECLARE [VARIABLE] cursor_name
  [[NO] SCROLL] CURSOR
  FOR (<select>);
```

Tabelle 92. `DECLARE … CURSOR`-Anweisungsparameter

Argument	Beschreibung
cursorname	Name des Cursors
select	`SELECT`-Anweisunge

Die `DECLARE … CURSOR … FOR`-Anweisung bindet einen benannten Cursor an die Ergebnismenge, die in der `SELECT`-Anweisung erhalten wurde, die in der `FOR`-Klausel angegeben ist. Im Body-Code kann der Cursor geöffnet, zum zeilenweisen Durchlaufen der Ergebnismenge verwendet und geschlossen werden. Während der Cursor geöffnet ist, kann der Code positionierte Aktualisierungen und Löschungen durchführen, indem das `WHERE CURRENT OF` in der `UPDATE`- oder `DELETE`-Anweisung verwendet wird.

Syntaktisch ist die `DECLARE … CURSOR`-Anweisung ein Sonderfall von `DECLARE VARIABLE`.

Vorwärts- und scrollbare Cursor

Der Cursor kann nur vorwärts (unidirektional) oder scrollbar sein. Die optionale Klausel `SCROLL` macht den Cursor scrollbar, die `NO SCROLL` Klausel nur vorwärts. Standardmäßig sind Cursor nur vorwärts.

Nur-Vorwärts-Cursor können sich – wie der Name schon sagt – im Datensatz nur vorwärts bewegen. Vorwärtscursor unterstützen nur die Anweisung `FETCH [NEXT FROM]`, andere Befehle geben einen Fehler aus. Scrollbare Cursor ermöglichen es Ihnen, sich im Datensatz nicht nur vorwärts, sondern auch rückwärts zu bewegen, sowie *N* Positionen relativ zur aktuellen Position.

Scrollbare Cursor werden als temporäres Dataset materialisiert und verbrauchen daher zusätzlichen Speicher oder Festplattenspeicher. Verwenden Sie sie also nur, wenn Sie sie wirklich brauchen.

Cursor-Idiosynkrasien

- Die optionale `FOR UPDATE`-Klausel kann in die `SELECT`-Anweisung aufgenommen werden, ihr Fehlen verhindert jedoch nicht die erfolgreiche Ausführung eines positionierten Updates oder Deletes
- Es sollte darauf geachtet werden, dass die Namen deklarierter Cursor nicht mit Namen kollidieren, die später in Anweisungen für `AS CURSOR`-Klauseln verwendet werden
- Wenn der Cursor nur zum Durchlaufen der Ergebnismenge benötigt wird, ist es fast immer einfacher und weniger fehleranfällig, eine `FOR SELECT`-Anweisung mit der `AS CURSOR`-Klausel zu verwenden. Deklarierte Cursor müssen explizit geöffnet, zum Abrufen von Daten verwendet und geschlossen werden. Die Kontextvariable `ROW_COUNT` muss nach jedem Fetch überprüft werden und wenn ihr Wert null ist, muss die Schleife beendet werden. Eine `FOR SELECT` -Anweisung macht dies automatisch.

  Dennoch bieten deklarierte Cursor ein hohes Maß an Kontrolle über sequentielle Ereignisse und ermöglichen die parallele Verwaltung mehrerer Cursor.

- Die `SELECT`-Anweisung kann Parameter enthalten. Zum Beispiel:

```
SELECT NAME || :SFX FROM NAMES WHERE NUMBER = :NUM
```

  Jeder Parameter muss zuvor als PSQL-Variable deklariert worden sein, auch wenn sie als Ein- und Ausgabeparameter stammen. Beim Öffnen des Cursors wird dem Parameter der aktuelle Wert der Variablen zugewiesen.

Instabile Variablen und Cursors

Wenn sich der Wert der PSQL-Variablen, die in der `SELECT`-Anweisung des Cursors verwendet wird, während der Ausführung der Schleife ändert, kann ihr neuer Wert - aber nicht immer - beim Auswählen der nächsten Zeilen verwendet werden. Es ist besser, solche Situationen zu vermeiden. Wenn Sie dieses Verhalten wirklich benötigen, sollten Sie Ihren Code gründlich testen und sicherstellen, dass Sie verstehen, wie sich Änderungen an der Variablen auf die Abfrageergebnisse auswirken.

Beachten Sie insbesondere, dass das Verhalten vom Abfrageplan abhängen kann, insbesondere von den verwendeten Indizes. Derzeit gibt es keine strengen Regeln für dieses Verhalten, und dies kann sich in zukünftigen Versionen von Firebird ändern.

Beispiele mit benannten Cursors

1. Deklarieren eines benannten Cursors im Trigger.

```
CREATE OR ALTER TRIGGER TBU_STOCK
  BEFORE UPDATE ON STOCK
AS
  DECLARE C_COUNTRY CURSOR FOR (
```

```
    SELECT
      COUNTRY,
      CAPITAL
    FROM COUNTRY
  );
BEGIN
  /* PSQL statements */
END
```

2. Einen scrollbaren Cursor deklarieren

```
EXECUTE BLOCK
  RETURNS (
    N INT,
    RNAME CHAR(31))
AS
  - Declaring a scrollable cursor
  DECLARE C SCROLL CURSOR FOR (
    SELECT
      ROW_NUMBER() OVER (ORDER BY RDB$RELATION_NAME) AS N,
      RDB$RELATION_NAME
    FROM RDB$RELATIONS
    ORDER BY RDB$RELATION_NAME);
BEGIN
  / * PSQL-Anweisungen * /
END
```

3. Eine Sammlung von Skripten zum Erstellen von Ansichten mit einem PSQL-Block unter Verwendung von benannten Cursorn.

```
EXECUTE BLOCK
RETURNS (
  SCRIPT BLOB SUB_TYPE TEXT)
AS
  DECLARE VARIABLE FIELDS VARCHAR(8191);
  DECLARE VARIABLE FIELD_NAME TYPE OF RDB$FIELD_NAME;
  DECLARE VARIABLE RELATION RDB$RELATION_NAME;
  DECLARE VARIABLE SOURCE TYPE OF COLUMN RDB$RELATIONS.RDB$VIEW_SOURCE;
  DECLARE VARIABLE CUR_R CURSOR FOR (
    SELECT
      RDB$RELATION_NAME,
      RDB$VIEW_SOURCE
    FROM
      RDB$RELATIONS
    WHERE
      RDB$VIEW_SOURCE IS NOT NULL);
  -- Declaring a named cursor where
  -- a local variable is used
  DECLARE CUR_F CURSOR FOR (
```

```
    SELECT
      RDB$FIELD_NAME
    FROM
      RDB$RELATION_FIELDS
    WHERE
      -- It is important that the variable must be declared earlier
      RDB$RELATION_NAME = :RELATION);
BEGIN
  OPEN CUR_R;
  WHILE (1 = 1) DO
  BEGIN
    FETCH CUR_R
    INTO :RELATION, :SOURCE;
    IF (ROW_COUNT = 0) THEN
      LEAVE;

    FIELDS = NULL;
    -- The CUR_F cursor will use the value
    -- of the RELATION variable initiated above
    OPEN CUR_F;
    WHILE (1 = 1) DO
    BEGIN
      FETCH CUR_F
      INTO :FIELD_NAME;
      IF (ROW_COUNT = 0) THEN
        LEAVE;
      IF (FIELDS IS NULL) THEN
        FIELDS = TRIM(FIELD_NAME);
      ELSE
        FIELDS = FIELDS || ', ' || TRIM(FIELD_NAME);
    END
    CLOSE CUR_F;

    SCRIPT = 'CREATE VIEW ' || RELATION;

    IF (FIELDS IS NOT NULL) THEN
      SCRIPT = SCRIPT || ' (' || FIELDS || ')';

    SCRIPT = SCRIPT || ' AS ' || ASCII_CHAR(13);
    SCRIPT = SCRIPT || SOURCE;

    SUSPEND;
  END
  CLOSE CUR_R;
END
```

Siehe auch

OPEN, FETCH, CLOSE

7.7.4. DECLARE FUNCTION

Verwendet für

Deklaration einer lokalen Variablen

Verfügbar in

PSQL

Syntax

```
DECLARE FUNCTION subfuncname [ ( [ <in_params> ] ) ]
  RETURNS <domain_or_non_array_type> [COLLATE collation]
  [DETERMINISTIC]
  <psql-module-body>

<in_params> ::=
  !! Siehe CREATE FUNCTION-Syntax !!

<domain_or_non_array_type> ::=
  !! Siehe Syntax für skalare Datentypen !!

<psql-module-body> ::=
  !! Siehe Syntax des Modulbodys !!
```

Tabelle 93. DECLARE FUNCTION*-Anweisungsparameter*

Argument	Beschreibung
subfuncname	Unterfunktionsname
collation	Kollationsname

Die Anweisung DECLARE FUNCTION deklariert eine Unterfunktion. Eine Unterfunktion ist nur für das PSQL-Modul sichtbar, das die Unterfunktion definiert hat.

Unterfunktionen haben eine Reihe von Einschränkungen:

- Eine Unterfunktion kann nicht in eine andere Unterroutine eingebettet werden. Unterroutinen werden nur in PSQL-Modulen der obersten Ebene unterstützt (gespeicherte Prozeduren, gespeicherte Funktionen, Trigger und anonyme PSQL-Blöcke). Diese Einschränkung wird durch die Syntax nicht erzwungen, aber Versuche, verschachtelte Unterfunktionen zu erstellen, führen zu einem Fehler "*feature is not supported*" mit der Detailmeldung "*nested sub function*".
- Derzeit hat die Unterfunktion keinen direkten Zugriff, um Variablen, Cursor und andere Routinen (einschließlich sich selbst) von ihrem Elternmodul aus zu verwenden. Dies kann sich in einer zukünftigen Firebird-Version ändern.
  - Aufgrund dieser Einschränkung kann sich eine Unterfunktion nicht selbst rekursiv aufrufen; Versuche, es aufzurufen, führen zu Fehler "*Function unknown:* ***subfuncname***".

Wenn Sie eine Unterfunktion mit demselben Namen wie eine gespeicherte Funktion deklarieren, wird diese gespeicherte Funktion aus Ihrem Modul

ausgeblendet. Es ist nicht möglich, diese gespeicherte Funktion aufzurufen.

Im Gegensatz zu `DECLARE [VARIABLE]` wird eine `DECLARE FUNCTION` nicht mit einem Semikolon abgeschlossen. Das `END` seines Hauptblocks `BEGIN ⋯ END` wird als sein Abschlusszeichen betrachtet.

Beispiele für Unterfunktionen

Unterfunktion innerhalb einer gespeicherten Funktion

```
CREATE OR ALTER FUNCTION FUNC1 (n1 INTEGER, n2 INTEGER)
  RETURNS INTEGER
AS
- Subfunction
  DECLARE FUNCTION SUBFUNC (n1 INTEGER, n2 INTEGER)
    RETURNS INTEGER
  AS
  BEGIN
    RETURN n1 + n2;
  END
BEGIN
  RETURN SUBFUNC (n1, n2);
END
```

Siehe auch

`DECLARE PROCEDURE`, `CREATE FUNCTION`

7.7.5. DECLARE PROCEDURE

Verwendet für

Deklaration eines Unterverfahrens

Verfügbar in

PSQL

Syntax

```
DECLARE subprocname [ ( [ <in_params> ] ) ]
  [RETURNS (<out_params>)]
  <psq-module-body>

<in_params> ::=
  !! Siehe auch CREATE PROCEDURE-Syntax !!

<domain_or_non_array_type> ::=
  !! Siehe auch Syntax für skalare Datentypen !!

<psql-module-body> ::=
```

```
  !! Siehe auch Syntax des Modul-Bodys !!
```

Tabelle 94. DECLARE PROCEDURE-*Anweisungsparameter*

Argument	Beschreibung
subprocname	Name des Unterverfahrens
collation	Kollationsname

Die Anweisung DECLARE PROCEDURE deklariert eine Unterprozedur. Eine Unterprozedur ist nur für das PSQL-Modul sichtbar, das die Unterprozedur definiert hat.

Unterverfahren haben eine Reihe von Einschränkungen:

- Eine Unterprozedur kann nicht in eine andere Unterroutine geschachtelt werden. Unterroutinen werden nur in PSQL-Modulen der obersten Ebene unterstützt (gespeicherte Prozeduren, gespeicherte Funktionen, Trigger und anonyme PSQL-Blöcke). Diese Einschränkung wird durch die Syntax nicht erzwungen, aber Versuche, verschachtelte Unterprozeduren zu erstellen, führen zu einem Fehler "*feature is not supported*" mit der Detailmeldung "*nested sub procedure*".
- Derzeit hat die Unterprozedur keinen direkten Zugriff, um Variablen, Cursor und andere Routinen (einschließlich sich selbst) von ihrem Elternmodul aus zu verwenden. Dies kann sich in einer zukünftigen Firebird-Version ändern.
  - Aufgrund dieser Einschränkung kann sich eine Unterprozedur nicht selbst rekursiv aufrufen; Versuche, es aufzurufen, führen zum Fehler "*Function unknown:* ***subprocname***".

Wenn Sie eine Unterprozedur mit demselben Namen wie eine gespeicherte Prozedur, Tabelle oder Ansicht deklarieren, wird diese gespeicherte Prozedur, Tabelle oder Ansicht von Ihrem Modul ausgeblendet. Es ist nicht möglich, diese gespeicherte Prozedur, Tabelle oder Ansicht aufzurufen.

Im Gegensatz zu DECLARE [VARIABLE] wird ein DECLARE PROCEDURE nicht mit einem Semikolon abgeschlossen. Das END seines Hauptblocks BEGIN … END wird als sein Abschlusszeichen betrachtet.

Beispiele für Unterprozeduren

Unterprogramme in EXECUTE BLOCK

```
EXECUTE BLOCK
  RETURNS (name VARCHAR(31))
AS
-- Unterprozedur, die eine Liste von Tabellen zurückgibt
  DECLARE PROCEDURE get_tables
    RETURNS (table_name VARCHAR(31))
  AS
  BEGIN
    FOR SELECT RDB$RELATION_NAME
```

```
      FROM RDB$RELATIONS
      WHERE RDB$VIEW_BLR IS NULL
      INTO table_name
    DO SUSPEND;
  END
-- Unterprozedur, die eine Liste von Ansichten zurückgibt
  DECLARE PROCEDURE get_views
    RETURNS (view_name VARCHAR(31))
  AS
  BEGIN
    FOR SELECT RDB$RELATION_NAME
      FROM RDB$RELATIONS
      WHERE RDB$VIEW_BLR IS NOT NULL
      INTO view_name
    DO SUSPEND;
  END
BEGIN
  FOR SELECT table_name
    FROM get_tables
    UNION ALL
    SELECT view_name
    FROM get_views
    INTO name
  DO SUSPEND;
END
```

Siehe auch

`DECLARE FUNCTION`, `CREATE PROCEDURE`

7.7.6. BEGIN ⋯ END

Verwendet für

Einen Block von Anweisungen abgrenzen

Verfügbar in

PSQL

Syntax

```
<block> ::=
  BEGIN
    [<compound_statement> ...]
  END

<compound_statement> ::= {<block> | <statement>}
```

Das Konstrukt `BEGIN ⋯ END` ist eine zweiteilige Anweisung, die einen Block von Anweisungen umschließt, die als eine Codeeinheit ausgeführt werden. Jeder Block beginnt mit der Halbanweisung "BEGIN" und endet mit der anderen Halbanweisung "END". Blöcke können mit

einer maximalen Tiefe von 512 verschachtelten Blöcken verschachtelt werden. Ein Block kann leer sein, sodass sie als Stubs fungieren können, ohne dass Dummy-Anweisungen geschrieben werden müssen.

Die Anweisungen BEGIN und END haben keine Zeilenabschlusszeichen (Semikolon). Beim Definieren oder Ändern eines PSQL-Moduls im Dienstprogramm *isql* erfordert diese Anwendung jedoch, dass der letzten `END`-Anweisung ein eigenes Abschlusszeichen folgt, das zuvor mit `SET TERM` in eine andere Zeichenfolge als umgestellt wurde ein Semikolon. Dieser Terminator ist nicht Teil der PSQL-Syntax.

Die letzte oder äußerste `END`-Anweisung in einem Trigger beendet den Trigger. Was die letzte `END` -Anweisung in einer Stored Procedure macht, hängt vom Prozedurtyp ab:

- In einer auswählbaren Prozedur gibt die letzte `END`-Anweisung die Kontrolle an den Aufrufer zurück und gibt SQLCODE 100 zurück, was angibt, dass keine weiteren Zeilen zum Abrufen vorhanden sind
- In einer ausführbaren Prozedur gibt die letzte `END`-Anweisung die Kontrolle an den Aufrufer zurück, zusammen mit den aktuellen Werten aller definierten Ausgabeparameter.

`BEGIN ⋯ END`-Beispiele

Eine Beispielprozedur aus der Datenbank `employee.fdb`*, die die einfache Verwendung von* `BEGIN⋯END` *-Blöcken zeigt:*

```
SET TERM ^;
CREATE OR ALTER PROCEDURE DEPT_BUDGET (
  DNO CHAR(3))
RETURNS (
  TOT DECIMAL(12,2))
AS
  DECLARE VARIABLE SUMB DECIMAL(12,2);
  DECLARE VARIABLE RDNO CHAR(3);
  DECLARE VARIABLE CNT  INTEGER;
BEGIN
  TOT = 0;

  SELECT BUDGET
  FROM DEPARTMENT
  WHERE DEPT_NO = :DNO
  INTO :TOT;

  SELECT COUNT(BUDGET)
  FROM DEPARTMENT
  WHERE HEAD_DEPT = :DNO
  INTO :CNT;

  IF (CNT = 0) THEN
    SUSPEND;

  FOR SELECT DEPT_NO
```

```
    FROM DEPARTMENT
    WHERE HEAD_DEPT = :DNO
    INTO :RDNO
  DO
  BEGIN
    EXECUTE PROCEDURE DEPT_BUDGET(:RDNO)
      RETURNING_VALUES :SUMB;
    TOT = TOT + SUMB;
  END

  SUSPEND;
END^
SET TERM ;^
```

Siehe auch

EXIT, SET TERM

7.7.7. IF ⋯ THEN ⋯ ELSE

Verwendet für

Bedingte Verzweigung

Verfügbar in

PSQL

Syntax

```
IF (<condition>)
  THEN <compound_statement>
  [ELSE <compound_statement>]
```

Tabelle 95. IF ⋯ THEN ⋯ ELSE *Parameters*

Argument	Beschreibung
condition	Eine logische Bedingung, die TRUE, FALSE oder UNKNOWN zurückgibt
compound_statement	Eine einzelne Anweisung oder zwei oder mehr Anweisungen, die in BEGIN ⋯ END . eingeschlossen sind

Die bedingte Sprunganweisung IF ⋯ THEN wird verwendet, um den Ausführungsprozess in einem PSQL-Modul zu verzweigen. Die Bedingung ist immer in Klammern eingeschlossen. Wenn es den Wert TRUE zurückgibt, verzweigt die Ausführung in die Anweisung oder den Anweisungsblock nach dem Schlüsselwort THEN. Wenn eine ELSE vorhanden ist und die Bedingung FALSE oder UNKNOWN zurückgibt, verzweigt die Ausführung in die Anweisung oder den Anweisungsblock danach.

Verzweigungen mit mehreren Unterverzweigungen

PSQL bietet keine fortgeschritteneren Multi-Branch-Sprünge wie CASE oder SWITCH. Es ist jedoch möglich, IF ⋯ THEN ⋯ ELSE-Anweisungen zu verketten, siehe den Beispielabschnitt unten. Alternativ steht die CASE-Anweisung von DSQL in PSQL zur Verfügung und kann zumindest einige Anwendungsfälle nach Art eines Schalters erfüllen:

```
CASE <test_expr>
  WHEN <expr> THEN <result>
  [WHEN <expr> THEN <result> ...]
  [ELSE <defaultresult>]
END

CASE
  WHEN <bool_expr> THEN <result>
  [WHEN <bool_expr> THEN <result> ...]
  [ELSE <defaultresult>]
END
```

Beispiel in PSQL

```
...
C = CASE
      WHEN A=2 THEN 1
      WHEN A=1 THEN 3
      ELSE 0
    END;
...
```

IF-Beispiele

1. Ein Beispiel mit der IF-Anweisung. Angenommen, die Variablen FIRST, LINE2 und LAST wurden früher deklariert.

```
...
IF (FIRST IS NOT NULL) THEN
  LINE2 = FIRST || ' ' || LAST;
ELSE
  LINE2 = LAST;
...
```

2. Da IF ⋯ THEN ⋯ ELSE eine Anweisung ist, ist es möglich, sie miteinander zu verketten. Angenommen, die Variablen INT_VALUE und STRING_VALUE wurden früher deklariert.

```
IF (INT_VALUE = 1) THEN
```

```
  STRING_VALUE = 'one';
ELSE IF (INT_VALUE = 2) THEN
  STRING_VALUE = 'two';
ELSE IF (INT_VALUE = 3) THEN
  STRING_VALUE = 'three';
ELSE
  STRING_VALUE = 'too much';
```

Dieses spezielle Beispiel kann durch ein Einfaches CASE oder die Funktion DECODE ersetzt werden.

Siehe auch

WHILE ⋯ DO, CASE

7.7.8. WHILE ⋯ DO

Verwendet für

Schleifenkonstrukte

Verfügbar in

PSQL

Syntax

```
[label:]
WHILE <condition> DO
  <compound_statement>
```

Tabelle 96. WHILE ⋯ DO Parameters

Argument	Beschreibung
label	Optionales Label für LEAVE und CONTINUE. Befolgt die Regeln für Bezeichner.
condition	Eine logische Bedingung, die TRUE, FALSE oder UNKNOWN zurückgibt
compound_statement	Zwei oder mehr Anweisungen, die in BEGIN ⋯ END verpackt sind

Eine WHILE-Anweisung implementiert das Schleifenkonstrukt in PSQL. Die Anweisung oder der Anweisungsblock wird ausgeführt, bis die Bedingung TRUE zurückgibt. Schleifen können beliebig tief verschachtelt werden.

WHILE ⋯ DO-Beispiele

Eine Prozedur, die die Summe der Zahlen von 1 bis I berechnet, zeigt, wie das Schleifenkonstrukt verwendet wird.

```
CREATE PROCEDURE SUM_INT (I INTEGER)
RETURNS (S INTEGER)
AS
```

```
BEGIN
  s = 0;
  WHILE (i > 0) DO
  BEGIN
    s = s + i;
    i = i - 1;
  END
END
```

Ausführen der Prozedur in *isql*:

```
EXECUTE PROCEDURE SUM_INT(4);
```

Das Ergebnis ist:

```
S
==========
10
```

Siehe auch

`IF ⋯ THEN ⋯ ELSE`, `LEAVE`, `EXIT`, `FOR SELECT`, `FOR EXECUTE STATEMENT`

7.7.9. BREAK

Verwendet für

Verlassen einer Schleife

Verfügbar in

PSQL

Syntax

```
[label:]
<loop_stmt>
BEGIN
  ...
  BREAK;
  ...
END

<loop_stmt> ::=
    FOR <select_stmt> INTO <var_list> DO
  | FOR EXECUTE STATEMENT ... INTO <var_list> DO
  | WHILE (<condition>)} DO
```

Tabelle 97. BREAK-Anweisungsparameter

Argument	Beschreibung
label	Label
select_stmt	SELECT-Anweisungen
condition	Eine logische Bedingung, die TRUE, FALSE oder UNKNOWN zurückgibt

Die BREAK-Anweisung beendet sofort die innere Schleife einer WHILE- oder FOR-Schleife. Der Code wird ab der ersten Anweisung nach dem beendeten Schleifenblock weiter ausgeführt.

BREAK ähnelt LEAVE, unterstützt jedoch kein Label.

Siehe auch

LEAVE

7.7.10. LEAVE

Verwendet für

Eine Schleife beenden

Verfügbar in

PSQL

Syntax

```
[label:]
<loop_stmt>
BEGIN
  ...
  LEAVE [label];
  ...
END

<loop_stmt> ::=
    FOR <select_stmt> INTO <var_list> DO
  | FOR EXECUTE STATEMENT ... INTO <var_list> DO
  | WHILE (<condition>)} DO
```

Tabelle 98. LEAVE-Anweisungsparameter

Argument	Beschreibung
label	Label
select_stmt	SELECT-Statement
condition	Eine logische Bedingung, die TRUE, FALSE oder UNKNOWN zurückgibt

Die LEAVE-Anweisung beendet sofort die innere Schleife einer WHILE- oder FOR-Schleife. Mit dem optionalen Parameter *label* kann LEAVE auch eine äußere Schleife verlassen, also die Schleife, die mit *label* gekennzeichnet ist. Der Code wird ab der ersten Anweisung nach dem beendeten Schleifenblock weiter ausgeführt.

LEAVE-Beispiele

1. Eine Schleife verlassen, wenn ein Fehler beim Einfügen in die NUMBERS-Tabelle auftritt. Der Code wird ab der Zeile `C = 0` weiter ausgeführt.

```
...
WHILE (B < 10) DO
BEGIN
  INSERT INTO NUMBERS(B)
  VALUES (:B);
  B = B + 1;
  WHEN ANY DO
  BEGIN
    EXECUTE PROCEDURE LOG_ERROR (
      CURRENT_TIMESTAMP,
      'ERROR IN B LOOP');
    LEAVE;
  END
END
C = 0;
...
```

2. Ein Beispiel für die Verwendung von Labels in der LEAVE-Anweisung. `LEAVE LOOPA` beendet die äußere Schleife und `LEAVE LOOPB` beendet die innere Schleife. Beachten Sie, dass die einfache Anweisung LEAVE ausreichen würde, um die innere Schleife zu beenden.

```
...
STMT1 = 'SELECT NAME FROM FARMS';
LOOPA:
FOR EXECUTE STATEMENT :STMT1
INTO :FARM DO
BEGIN
  STMT2 = 'SELECT NAME ' || 'FROM ANIMALS WHERE FARM = ''';
  LOOPB:
  FOR EXECUTE STATEMENT :STMT2 || :FARM || ''''
  INTO :ANIMAL DO
  BEGIN
    IF (ANIMAL = 'FLUFFY') THEN
      LEAVE LOOPB;
    ELSE IF (ANIMAL = FARM) THEN
      LEAVE LOOPA;
    ELSE
      SUSPEND;
  END
END
...
```

Siehe auch

BREAK, CONTINUE, EXIT

7.7.11. CONTINUE

Verwendet für

Weiter mit der nächsten Iteration einer Schleife

Verfügbar in

PSQL

Syntax

```
[label:]
<loop_stmt>
BEGIN
  ...
  CONTINUE [label];
  ...
END

<loop_stmt> ::=
    FOR <select_stmt> INTO <var_list> DO
  | FOR EXECUTE STATEMENT ... INTO <var_list> DO
  | WHILE (<condition>)} DO
```

Tabelle 99. CONTINUE-Anweisungsparameter

Argument	Beschreibung
label	Label
select_stmt	SELECT-Anweisung
condition	Eine logische Bedingung, die TRUE, FALSE oder UNKNOWN zurückgibt

Die CONTINUE-Anweisung überspringt den Rest des aktuellen Blocks einer Schleife und startet die nächste Iteration der aktuellen WHILE- oder FOR-Schleife. Mit dem optionalen Parameter *label* kann CONTINUE auch die nächste Iteration einer äußeren Schleife starten, dh der Schleife, die mit *label* gekennzeichnet ist.

CONTINUE-Beispiele

Verwenden der CONTINUE-Anweisung

```
FOR SELECT A, D
  FROM ATABLE INTO achar, ddate
DO
BEGIN
  IF (ddate < current_date - 30) THEN
    CONTINUE;
  ELSE
  BEGIN
```

```
    /* mach was */
  END
END
```

Siehe auch

BREAK, LEAVE, EXIT

7.7.12. EXIT

Verwendet für

Beenden der Modulausführung

Verfügbar in

PSQL

Syntax

```
EXIT;
```

Die Anweisung EXIT bewirkt, dass die Ausführung der Prozedur oder des Triggers von jedem Punkt des Codes zur endgültigen END-Anweisung springt, wodurch das Programm beendet wird.

Calling EXIT in a function will result in the function returning NULL.

EXIT-Beispiele

Verwendung der EXIT*-Anweisung in einer wählbaren Prozedur*

```
CREATE PROCEDURE GEN_100
  RETURNS (I INTEGER)
AS
BEGIN
  I = 1;
  WHILE (1=1) DO
  BEGIN
    SUSPEND;
    IF (I=100) THEN
      EXIT;
    I = I + 1;
  END
END
```

Siehe auch

BREAK, LEAVE, CONTINUE, SUSPEND

7.7.13. SUSPEND

Verwendet für

Übergeben der Ausgabe an den Puffer und Aussetzen der Ausführung, während darauf gewartet wird, dass der Aufrufer sie abruft

Verfügbar in

PSQL

Syntax

```
SUSPEND;
```

Die Anweisung "SUSPEND" wird in einer auswählbaren gespeicherten Prozedur verwendet, um die Werte von Ausgabeparametern an einen Puffer zu übergeben und die Ausführung anzuhalten. Die Ausführung bleibt ausgesetzt, bis die aufrufende Anwendung den Inhalt des Puffers abruft. Die Ausführung wird von der Anweisung direkt nach der SUSPEND-Anweisung fortgesetzt. In der Praxis ist dies wahrscheinlich eine neue Iteration eines Schleifenprozesses.

Wichtige Notizen

1. Die SUSPEND-Anweisung kann nur in gespeicherten Prozeduren oder Unterprozeduren vorkommen
2. Das Vorhandensein des Schlüsselworts SUSPEND definiert eine gespeicherte Prozedur als auswählbare Prozedur
3. Anwendungen, die Schnittstellen verwenden, die die API umschließen, führen die Abrufe aus auswählbaren Prozeduren transparent durch.
4. Wenn eine auswählbare Prozedur mit EXECUTE PROCEDURE ausgeführt wird, verhält sie sich wie eine ausführbare Prozedur. Wenn eine 'SUSPEND'-Anweisung in einer solchen Stored Procedure ausgeführt wird, ist dies dasselbe wie die Ausführung der 'EXIT'-Anweisung, was zur sofortigen Beendigung der Prozedur führt.
5. SUSPEND“unterbricht” die Atomarität des Blocks, in dem es sich befindet. Wenn in einer wählbaren Prozedur ein Fehler auftritt, werden Anweisungen, die nach der letzten SUSPEND-Anweisung ausgeführt werden, zurückgesetzt. Anweisungen, die vor der letzten SUSPEND-Anweisung ausgeführt wurden, werden nicht zurückgesetzt, es sei denn, die Transaktion wird zurückgesetzt.

SUSPEND-Beispiele

Verwenden der SUSPEND-Anweisung in einer wählbaren Prozedur

```
CREATE PROCEDURE GEN_100
  RETURNS (I INTEGER)
AS
BEGIN
  I = 1;
  WHILE (1=1) DO
  BEGIN
    SUSPEND;
    IF (I=100) THEN
```

```
      EXIT;
    I = I + 1;
  END
END
```

Siehe auch

EXIT

7.7.14. EXECUTE STATEMENT

Verwendet für

Ausführen von dynamisch erstellten SQL-Anweisungen

Verfügbar in

PSQL

Syntax

```
<execute_statement> ::= EXECUTE STATEMENT <argument>
  [<option> ...]
  [INTO <variables>];

<argument> ::= <paramless_stmt>
             | (<paramless_stmt>)
             | (<stmt_with_params>) (<param_values>)

<param_values> ::= <named_values> | <positional_values>

<named_values> ::= paramname := <value_expr>
   [, paramname := <value_expr> ...]

<positional_values> ::= <value_expr> [, <value_expr> ...]

<option> ::=
    WITH {AUTONOMOUS | COMMON} TRANSACTION
  | WITH CALLER PRIVILEGES
  | AS USER user
  | PASSWORD password
  | ROLE role
  | ON EXTERNAL [DATA SOURCE] <connection_string>

<connection_string> ::=
  !! Siehe auch <filespec> im Abschnitt CREATE DATABASE-Syntax !!

<variables> ::= [:]varname [, [:]varname ...]
```

Tabelle 100. EXECUTE STATEMENT-*Anweisungsparameter*

Argument	Beschreibung
paramless_stmt	Literale Zeichenfolge oder Variable, die eine nicht parametrisierte SQL-Abfrage enthält
stmt_with_params	Literale Zeichenfolge oder Variable, die eine parametrisierte SQL-Abfrage enthält
paramname	Name des SQL-Abfrageparameters
value_expr	SQL-Ausdruck, der in einen Wert aufgelöst wird
user	Nutzername. Dies kann eine Zeichenfolge, `CURRENT_USER` oder eine Zeichenfolgenvariable sein
password	Passwort. Es kann eine Zeichenfolge oder eine Zeichenfolgevariable sein
role	Rolle. Dies kann eine Zeichenfolge, `CURRENT_ROLE` oder eine Zeichenfolgenvariable sein
connection_string	Verbindungszeichenfolge. Es kann eine Zeichenfolge oder eine Zeichenfolgevariable sein
varname	Variable

Die Anweisung `EXECUTE STATEMENT` verwendet einen Zeichenfolgenparameter und führt ihn wie eine DSQL-Anweisung aus. Wenn die Anweisung Daten zurückgibt, kann sie über eine `INTO` -Klausel an lokale Variablen übergeben werden.

`EXECUTE STATEMENT` kann nur eine einzelne Datenzeile erzeugen. Anweisungen, die mehrere Datenzeilen erzeugen, müssen mit `FOR EXECUTE STATEMENT` ausgeführt werden.

Parametrisierte Anweisungen

Sie können die Parameter — entweder benannt oder positional — in der DSQL-Anweisungsfolge verwenden. Jedem Parameter muss ein Wert zugewiesen werden.

Spezielle Regeln für parametrisierte Anweisungen

1. Benannte und Positionsparameter können nicht in einer Abfrage gemischt werden
2. Wenn die Anweisung Parameter hat, müssen diese beim Aufruf von `EXECUTE STATEMENT` in Klammern eingeschlossen werden, egal ob sie direkt als Strings, als Variablennamen oder als Ausdrücke kommen
3. Jedem benannten Parameter muss ein Doppelpunkt (‘:’) in der Anweisungszeichenfolge selbst vorangestellt werden, jedoch nicht, wenn dem Parameter ein Wert zugewiesen wird
4. Positionsparametern müssen ihre Werte in der Reihenfolge zugewiesen werden, in der sie im Abfragetext erscheinen
5. Der Zuweisungsoperator für Parameter ist der spezielle Operator “`:=`”, ähnlich dem Zuweisungsoperator in Pascal
6. Jeder benannte Parameter kann in der Anweisung mehrmals verwendet werden, sein Wert darf jedoch nur einmal zugewiesen werden

7. Bei Positionsparametern muss die Anzahl der zugewiesenen Werte genau mit der Anzahl der Parameterplatzhalter (Fragezeichen) in der Anweisung übereinstimmen
8. Ein benannter Parameter im Anweisungstext kann nur ein regulärer Bezeichner sein (er darf kein Bezeichner in Anführungszeichen sein)

Beispiele für `EXECUTE STATEMENT` mit Parametern

Mit benannten Parametern:

```
...
DECLARE license_num VARCHAR(15);
DECLARE connect_string VARCHAR (100);
DECLARE stmt VARCHAR (100) =
  'SELECT license
   FROM cars
   WHERE driver = :driver AND location = :loc';
BEGIN
  ...
  SELECT connstr
  FROM databases
  WHERE cust_id = :id
  INTO connect_string;
  ...
  FOR
    SELECT id
    FROM drivers
    INTO current_driver
  DO
  BEGIN
    FOR
      SELECT location
      FROM driver_locations
      WHERE driver_id = :current_driver
      INTO current_location
    DO
    BEGIN
      ...
      EXECUTE STATEMENT (stmt)
        (driver := current_driver,
         loc := current_location)
      ON EXTERNAL connect_string
      INTO license_num;
      ...
```

Derselbe Code mit Positionsparametern:

```
DECLARE license_num VARCHAR (15);
DECLARE connect_string VARCHAR (100);
DECLARE stmt VARCHAR (100) =
```

```
  'SELECT license
   FROM cars
   WHERE driver = ? AND location = ?';
BEGIN
  ...
  SELECT connstr
  FROM databases
  WHERE cust_id = :id
  into connect_string;
  ...
  FOR
    SELECT id
    FROM drivers
    INTO current_driver
  DO
  BEGIN
    FOR
      SELECT location
      FROM driver_locations
      WHERE driver_id = :current_driver
      INTO current_location
    DO
    BEGIN
      ...
      EXECUTE STATEMENT (stmt)
        (current_driver, current_location)
      ON EXTERNAL connect_string
      INTO license_num;
      ...
```

WITH {AUTONOMOUS | COMMON} TRANSACTION

Standardmäßig wird die ausgeführte SQL-Anweisung innerhalb der aktuellen Transaktion ausgeführt. Die Verwendung von `WITH AUTONOMOUS TRANSACTION` bewirkt, dass eine separate Transaktion mit den gleichen Parametern wie die aktuelle Transaktion gestartet wird. Diese separate Transaktion wird festgeschrieben, wenn die Anweisung fehlerfrei ausgeführt und ansonsten zurückgesetzt wurde.

Die Klausel `WITH COMMON TRANSACTION` verwendet nach Möglichkeit die aktuelle Transaktion; Dies ist das Standardverhalten. Wenn die Anweisung in einer separaten Verbindung ausgeführt werden muss, wird eine bereits gestartete Transaktion innerhalb dieser Verbindung verwendet, sofern verfügbar. Andernfalls wird eine neue Transaktion mit denselben Parametern wie die aktuelle Transaktion gestartet. Alle neuen Transaktionen, die unter dem Regime “`COMMON`” gestartet wurden, werden mit der aktuellen Transaktion festgeschrieben oder zurückgesetzt.

WITH CALLER PRIVILEGES

Standardmäßig wird die SQL-Anweisung mit den Berechtigungen des aktuellen Benutzers ausgeführt. Die Angabe von `WITH CALLER PRIVILEGES` fügt dazu die Privilegien der aufrufenden Prozedur oder des Triggers hinzu, so als ob die Anweisung direkt von der Routine ausgeführt

würde. WITH CALLER PRIVILEGES hat keine Auswirkung, wenn die Klausel ON EXTERNAL ebenfalls vorhanden ist.

ON EXTERNAL [DATA SOURCE]

Mit ON EXTERNAL [DATA SOURCE] wird die SQL-Anweisung in einer separaten Verbindung zu derselben oder einer anderen Datenbank ausgeführt, möglicherweise sogar auf einem anderen Server. Wenn die Verbindungszeichenfolge NULL oder "''" (leere Zeichenfolge) ist, wird die gesamte Klausel ON EXTERNAL [DATA SOURCE] als abwesend betrachtet und die Anweisung wird für die aktuelle Datenbank ausgeführt.

Verbindungspooling

- Externe Verbindungen, die durch Anweisungen WITH COMMON TRANSACTION (der Standardwert) hergestellt werden, bleiben geöffnet, bis die aktuelle Transaktion beendet wird. Sie können durch nachfolgende Aufrufe an EXECUTE STATEMENT wiederverwendet werden, aber nur, wenn die Verbindungszeichenfolge genau gleich ist, einschließlich case
- Externe Verbindungen, die durch Anweisungen WITH AUTONOMOUS TRANSACTION hergestellt werden, werden geschlossen, sobald die Anweisung ausgeführt wurde
- Beachten Sie, dass Statements unter WITH AUTONOMOUS TRANSACTION-Verbindungen, die zuvor von Anweisungen unter WITH COMMON TRANSACTION geöffnet wurden, wiederverwendet werden. Wenn dies geschieht, bleibt die wiederverwendete Verbindung nach der Ausführung der Anweisung offen. (Dies geschieht, da es mindestens eine nicht-abgeschlossene Transaktion gibt!)

Transaktionspooling

- Wenn WITH COMMON TRANSACTION aktiviert ist, werden Transaktionen so oft wie möglich wiederverwendet. Sie werden zusammen mit der aktuellen Transaktion festgeschrieben oder zurückgesetzt
- Wenn WITH AUTONOMOUS TRANSACTION angegeben ist, wird immer eine neue Transaktion für die Anweisung gestartet. Diese Transaktion wird unmittelbar nach der Ausführung der Anweisung festgeschrieben oder zurückgesetzt

Ausnahmebehandlung

Ausnahmebehandlung: Wenn ON EXTERNAL verwendet wird, erfolgt die zusätzliche Verbindung immer über einen sogenannten externen Provider, auch wenn die Verbindung zur aktuellen Datenbank besteht. Eine der Folgen ist, dass Ausnahmen nicht auf die übliche Art und Weise abgefangen werden können. Jede von der Anweisung verursachte Ausnahme wird entweder in einen eds_connection- oder einen eds_statement-Fehler enden. Um sie in Ihrem PSQL-Code abzufangen, müssen Sie WHEN GDSCODE eds_connection, WHEN GDSCODE eds_statement oder WHEN ANY verwenden.

Ohne ON EXTERNAL werden Ausnahmen auf die übliche Weise abgefangen, selbst wenn eine zusätzliche Verbindung zur aktuellen Datenbank hergestellt wird.

Verschiedene Hinweise

- Der für die externe Verbindung verwendete Zeichensatz ist der gleiche wie für die aktuelle Verbindung
- Zweiphasen-Commits werden nicht unterstützt

AS USER, PASSWORD and ROLE

Die optionalen Klauseln `AS USER`, `PASSWORD` und `ROLE` erlauben die Angabe unter welchem Benutzer und unter welcher Rolle das SQL-Statement ausgeführt wird. Die Methode der Benutzeranmeldung und die Existenz einer separaten offenen Verbindung hängt von dem Vorhandensein und den Werten der Klauseln `ON EXTERNAL [DATA SOURCE]`, `AS USER`, `PASSWORD` und `ROLE` ab:

- Wenn `ON EXTERNAL` verwendet wird, wird immer eine neue Verbindung aufgebaut und:
  - Wenn mindestens eines von `AS USER`, `PASSWORD` und `ROLE` vorhanden ist, wird die native Authentifizierung mit den angegebenen Parameterwerten versucht (lokal oder remote abhängig von der Verbindungszeichenfolge). Für fehlende Parameter werden keine Standardwerte verwendet
  - Wenn alle drei nicht vorhanden sind und die Verbindungszeichenfolge keinen Hostnamen enthält, wird die neue Verbindung auf dem lokalen Host mit demselben Benutzer und derselben Rolle wie die aktuelle Verbindung hergestellt. Der Begriff "lokal" bedeutet hier “auf der gleichen Maschine wie der Server”. Dies ist nicht unbedingt der Standort des Clients
  - Wenn alle drei nicht vorhanden sind und die Verbindungszeichenfolge einen Hostnamen enthält, wird eine vertrauenswürdige Authentifizierung auf dem Remote-Host versucht (aus der Perspektive des Servers wiederum "Remote"). Wenn dies erfolgreich ist, gibt das Remote-Betriebssystem den Benutzernamen an (normalerweise das Betriebssystemkonto, unter dem der Firebird-Prozess ausgeführt wird).
- Fehlt `ON EXTERNAL`:
  - Wenn mindestens eines von `AS USER`, `PASSWORD` und `ROLE` vorhanden ist, wird eine neue Verbindung zur aktuellen Datenbank mit den angegebenen Parameterwerten geöffnet. Für fehlende Parameter werden keine Standardwerte verwendet
  - Wenn alle drei nicht vorhanden sind, wird die Anweisung innerhalb der aktuellen Verbindung ausgeführt

Wenn ein Parameterwert `NULL` oder “`''`” (leere Zeichenfolge) ist, wird der gesamte Parameter als abwesend betrachtet. Darüber hinaus gilt `AS USER` als abwesend, wenn der Wert gleich `CURRENT_USER` und `ROLE` wenn es identisch mit `CURRENT_ROLE` ist.

Vorsicht mit EXECUTE STATEMENT

1. Es gibt keine Möglichkeit, die Syntax der enthaltenen Anweisung zu überprüfen
2. Es gibt keine Abhängigkeitsprüfungen, um festzustellen, ob Tabellen oder Spalten gelöscht wurden
3. Obwohl die Leistung in Schleifen in Firebird 2.5 erheblich verbessert wurde, ist die Ausführung

immer noch erheblich langsamer als wenn dieselben Anweisungen direkt gestartet werden

4. Rückgabewerte werden streng auf den Datentyp überprüft, um unvorhersehbare Ausnahmen für das Typcasting zu vermeiden. Beispielsweise würde die Zeichenfolge `'1234'` in eine Ganzzahl, 1234, konvertiert, aber `'abc'` würde einen Konvertierungsfehler ergeben

Alles in allem sollte diese Funktion sehr vorsichtig verwendet werden und Sie sollten immer die Vorbehalte berücksichtigen. Wenn Sie das gleiche Ergebnis mit PSQL und / oder DSQL erzielen können, ist dies fast immer vorzuziehen.

Siehe auch

`FOR EXECUTE STATEMENT`

7.7.15. FOR SELECT

Verwendet für

Zeilenweises Durchlaufen einer abgefragten Ergebnismenge

Verfügbar in

PSQL

Syntax

```
[label:]
FOR <select_stmt> [AS CURSOR cursor_name]
  DO <compound_statement>
```

Tabelle 101. `FOR SELECT`-*Anweisungsparameter*

Argument	Beschreibung
label	Optionales Label für `LEAVE` und `CONTINUE`. Befolgt die Regeln für Bezeichner.
select_stmt	`SELECT`-Anweisung
cursor_name	Cursorname. Er muss unter den Cursornamen im PSQL-Modul (gespeicherte Prozedur, gespeicherte Funktion, Trigger oder PSQL-Block) eindeutig sein.
compound_statement	Eine einzelne Anweisung oder ein in `BEGIN`⋯`END` eingeschlossener Anweisungsblock, der die gesamte Verarbeitung für diese `FOR`-Schleife durchführt

Die `FOR SELECT`-Anweisung

- ruft jede Zeile nacheinander aus der Ergebnismenge ab und führt die Anweisung oder den Anweisungsblock für jede Zeile aus. Bei jeder Iteration der Schleife werden die Feldwerte der aktuellen Zeile in vorab deklarierte Variablen kopiert.

  Das Einschließen der `AS CURSOR`-Klausel ermöglicht das Ausführen von positionierten Löschungen und Aktualisierungen – siehe Hinweise unten

- kann andere FOR SELECT-Anweisungen einbetten
- kann benannte Parameter enthalten, die zuvor in der DECLARE VARIABLE-Anweisung deklariert werden müssen oder als Eingabe- oder Ausgabeparameter der Prozedur existieren
- erfordert eine INTO-Klausel am Ende der SELECT ⋯ FROM ⋯-Spezifikation. Bei jeder Iteration der Schleife werden die Feldwerte der aktuellen Zeile in die in der INTO-Klausel angegebene Variablenliste kopiert. Die Schleife wiederholt sich, bis alle Zeilen abgerufen wurden, danach wird sie beendet
- kann mit einer BREAK-, LEAVE- oder EXIT-Anweisung beendet werden, bevor alle Zeilen abgerufen wurden

Der undeklarierte Cursor

Die optionale AS CURSOR-Klausel zeigt die Menge in der FOR SELECT-Struktur als nicht deklarierten, benannten Cursor, der mit der WHERE CURRENT OF-Klausel innerhalb der Anweisung oder des Blocks nach dem DO-Befehl bearbeitet werden kann, in der richtigen Reihenfolge um die aktuelle Zeile zu löschen oder zu aktualisieren, bevor die Ausführung in die nächste Zeile übergeht. Darüber hinaus ist es möglich, den Cursornamen als Datensatzvariable zu verwenden (ähnlich wie OLD und NEW in Triggern), um auf die Spalten der Ergebnismenge zuzugreifen (z. B. *cursor_name.columnname*).

Regeln für Cursor-Variablen

- Beim Zugriff auf eine Cursorvariable in einer DML-Anweisung kann der Doppelpunkt-Präfix vor dem Cursornamen (d. h. `:cursor_name.columnname`) zur Disambiguierung hinzugefügt werden, ähnlich wie bei Variablen.

  Die Cursorvariable kann ohne Doppelpunkt-Präfix referenziert werden, aber in diesem Fall kann der Name je nach Umfang der Kontexte in der Anweisung statt in den Cursor in den Anweisungskontext aufgelöst werden (z. B. Sie wählen aus einer Tabelle mit demselben Namen als Cursor).

- Cursorvariablen sind schreibgeschützt
- In einer FOR SELECT-Anweisung ohne AS CURSOR-Klausel müssen Sie die INTO-Klausel verwenden. Wenn eine AS CURSOR-Klausel angegeben wird, ist die INTO-Klausel erlaubt, aber optional; Sie können stattdessen mit dem Cursor auf die Felder zugreifen.
- Das Lesen aus einer Cursor-Variablen gibt die aktuellen Feldwerte zurück. Das bedeutet, dass eine UPDATE-Anweisung (mit einer WHERE CURRENT OF-Klausel) nicht nur die Tabelle, sondern auch die Felder in der Cursor-Variablen für nachfolgende Lesevorgänge aktualisiert. Die Ausführung einer DELETE-Anweisung (mit einer WHERE CURRENT OF-Klausel) setzt alle Felder in der Cursor-Variablen für nachfolgende Lesevorgänge auf NULL

Weitere zu berücksichtigende Punkte in Bezug auf nicht deklarierte Cursor:

1. Die Anweisungen OPEN, FETCH und CLOSE können nicht auf einen Cursor angewendet werden, der von der AS CURSOR-Klausel angezeigt wird.
2. Das Argument *cursor_name*, das einer AS CURSOR-Klausel zugeordnet ist, darf nicht mit Namen kollidieren, die durch DECLARE VARIABLE- oder DECLARE CURSOR-Anweisungen oben im Modulrumpf erstellt wurden, noch mit anderen Cursorn, die von einer AS CURSOR-Klausel auftauchen

3. Die optionale FOR UPDATE-Klausel in der SELECT-Anweisung ist für ein positioniertes Update nicht erforderlich

Beispiele mit FOR SELECT

1. Eine einfache Schleife durch die Abfrageergebnisse:

```
CREATE PROCEDURE SHOWNUMS
RETURNS (
  AA INTEGER,
  BB INTEGER,
  SM INTEGER,
  DF INTEGER)
AS
BEGIN
  FOR SELECT DISTINCT A, B
      FROM NUMBERS
    ORDER BY A, B
    INTO AA, BB
  DO
  BEGIN
    SM = AA + BB;
    DF = AA - BB;
    SUSPEND;
  END
END
```

2. Verschachtelte FOR SELECT-Schleife:

```
CREATE PROCEDURE RELFIELDS
RETURNS (
  RELATION CHAR(32),
  POS INTEGER,
  FIELD CHAR(32))
AS
BEGIN
  FOR SELECT RDB$RELATION_NAME
      FROM RDB$RELATIONS
      ORDER BY 1
      INTO :RELATION
  DO
  BEGIN
    FOR SELECT
          RDB$FIELD_POSITION + 1,
          RDB$FIELD_NAME
        FROM RDB$RELATION_FIELDS
        WHERE
          RDB$RELATION_NAME = :RELATION
        ORDER BY RDB$FIELD_POSITION
```

```
        INTO :POS, :FIELD
    DO
    BEGIN
      IF (POS = 2) THEN
        RELATION = ' "';

      SUSPEND;
    END
  END
END
```

3. Verwenden Sie die `AS CURSOR`-Klausel, um einen Cursor für das positionierte Löschen eines Datensatzes zu verwenden:

```
CREATE PROCEDURE DELTOWN (
  TOWNTODELETE VARCHAR(24))
RETURNS (
  TOWN VARCHAR(24),
  POP INTEGER)
AS
BEGIN
  FOR SELECT TOWN, POP
      FROM TOWNS
      INTO :TOWN, :POP AS CURSOR TCUR
  DO
  BEGIN
    IF (:TOWN = :TOWNTODELETE) THEN
      -- Positional delete
      DELETE FROM TOWNS
      WHERE CURRENT OF TCUR;
    ELSE
      SUSPEND;
  END
END
```

4. Verwenden eines implizit deklarierten Cursors als Cursorvariable

```
EXECUTE BLOCK
 RETURNS (o CHAR(31))
AS
BEGIN
  FOR SELECT rdb$relation_name AS name
    FROM rdb$relations AS CURSOR c
  DO
  BEGIN
    o = c.name;
    SUSPEND;
  END
```

```
END
```

5. Cursorvariablen in Abfragen eindeutig machen

```
EXECUTE BLOCK
  RETURNS (o1 CHAR(31), o2 CHAR(31))
AS
BEGIN
  FOR SELECT rdb$relation_name
    FROM rdb$relations
    WHERE
      rdb$relation_name = 'RDB$RELATIONS' AS CURSOR c
  DO
  BEGIN
    FOR SELECT
        -- with a prefix resolves as a cursor
        :c.rdb$relation_name x1,
        -- no prefix as an alias for the rdb$relations table
        c.rdb$relation_name x2
      FROM rdb$relations c
      WHERE
        rdb$relation_name = 'RDB$DATABASE' AS CURSOR d
    DO
    BEGIN
      o1 = d.x1;
      o2 = d.x2;
      SUSPEND;
    END
  END
END
```

Siehe auch

`DECLARE .. CURSOR`, `BREAK`, `LEAVE`, `CONTINUE`, `EXIT`, `SELECT`, `UPDATE`, `DELETE`

7.7.16. FOR EXECUTE STATEMENT

Verwendet für

Ausführen von dynamisch erstellten SQL-Anweisungen, die einen Zeilensatz zurückgeben

Verfügbar in

PSQL

Syntax

```
[label:]
FOR <execute_statement> DO <compound_statement>
```

Tabelle 102. `FOR EXECUTE STATEMENT`-Anweisungsparameter

Argument	Beschreibung
label	Optionales Label für LEAVE und CONTINUE. Befolgt die Regeln für Bezeichner.
execute_stmt	Eine EXECUTE STATEMENT-Anweisung
compound_statement	Eine einzelne Anweisung oder ein in BEGIN…END eingeschlossener Anweisungsblock, der die gesamte Verarbeitung für diese FOR-Schleife durchführt

Die Anweisung FOR EXECUTE STATEMENT wird analog zu FOR SELECT verwendet, um die Ergebnismenge einer dynamisch ausgeführten Abfrage zu durchlaufen, die mehrere Zeilen zurückgibt.

FOR EXECUTE STATEMENT-Beispiele

Ausführen einer dynamisch konstruierten SELECT*-Abfrage, die einen Datensatz zurückgibt*

```
CREATE PROCEDURE DynamicSampleThree (
   Q_FIELD_NAME VARCHAR(100),
   Q_TABLE_NAME VARCHAR(100)
) RETURNS(
  LINE VARCHAR(32000)
)
AS
  DECLARE VARIABLE P_ONE_LINE VARCHAR(100);
BEGIN
  LINE = '';
  FOR
    EXECUTE STATEMENT
      'SELECT T1.' || :Q_FIELD_NAME ||
      ' FROM ' || :Q_TABLE_NAME || ' T1 '
    INTO :P_ONE_LINE
  DO
    IF (:P_ONE_LINE IS NOT NULL) THEN
      LINE = :LINE || :P_ONE_LINE || ' ';
  SUSPEND;
END
```

Siehe auch

EXECUTE STATEMENT, BREAK, LEAVE, CONTINUE

7.7.17. OPEN

Verwendet für

Öffnen eines deklarierten Cursors

Verfügbar in

PSQL

Syntax

```
OPEN cursor_name;
```

Tabelle 103. OPEN-Anweisungsparameter

Argument	Beschreibung
cursorname	Name des Cursors. Ein Cursor mit diesem Namen muss zuvor mit einer DECLARE CURSOR-Anweisung deklariert werden

Eine OPEN-Anweisung öffnet einen zuvor deklarierten Cursor, führt seine deklarierte SELECT -Anweisung aus und macht den ersten Datensatz der Ergebnisdatei zum Abruf bereit. OPEN kann nur auf Cursor angewendet werden, die zuvor in einer DECLARE .. CURSOR-Anweisung deklariert wurden.

Wenn die für den Cursor deklarierte Anweisung SELECT über Parameter verfügt, müssen sie als lokale Variablen deklariert sein oder als Ein- oder Ausgabeparameter vor dem Deklarieren des Cursors vorhanden sein. Wenn der Cursor geöffnet wird, wird dem Parameter der aktuelle Wert der Variablen zugewiesen.

OPEN-Beispiele

1. Mit der OPEN-Anweisung:

```
SET TERM ^;

CREATE OR ALTER PROCEDURE GET_RELATIONS_NAMES
RETURNS (
  RNAME CHAR(31)
)
AS
  DECLARE C CURSOR FOR (
    SELECT RDB$RELATION_NAME
    FROM RDB$RELATIONS);
BEGIN
  OPEN C;
  WHILE (1 = 1) DO
  BEGIN
    FETCH C INTO :RNAME;
    IF (ROW_COUNT = 0) THEN
      LEAVE;
    SUSPEND;
  END
  CLOSE C;
END^

SET TERM ;^
```

2. Eine Sammlung von Skripts zum Erstellen von Ansichten mit einem PSQL-Block mit benannten Cursorn:

```
EXECUTE BLOCK
RETURNS (
  SCRIPT BLOB SUB_TYPE TEXT)
AS
  DECLARE VARIABLE FIELDS VARCHAR(8191);
  DECLARE VARIABLE FIELD_NAME TYPE OF RDB$FIELD_NAME;
  DECLARE VARIABLE RELATION RDB$RELATION_NAME;
  DECLARE VARIABLE SOURCE TYPE OF COLUMN RDB$RELATIONS.RDB$VIEW_SOURCE;
  -- named cursor
  DECLARE VARIABLE CUR_R CURSOR FOR (
    SELECT
      RDB$RELATION_NAME,
      RDB$VIEW_SOURCE
    FROM
      RDB$RELATIONS
    WHERE
      RDB$VIEW_SOURCE IS NOT NULL);
  -- named cursor with local variable
  DECLARE CUR_F CURSOR FOR (
    SELECT
      RDB$FIELD_NAME
    FROM
      RDB$RELATION_FIELDS
    WHERE
      -- Wichtig! Die Variable muss vorher deklariert werden
      RDB$RELATION_NAME = :RELATION);
BEGIN
  OPEN CUR_R;
  WHILE (1 = 1) DO
  BEGIN
    FETCH CUR_R
      INTO :RELATION, :SOURCE;
    IF (ROW_COUNT = 0) THEN
      LEAVE;

    FIELDS = NULL;
    -- Der CUR_F-Cursor verwendet den Variablenwert
    -- von RELATION, der oben initialisiert wurde
    OPEN CUR_F;
    WHILE (1 = 1) DO
    BEGIN
      FETCH CUR_F
        INTO :FIELD_NAME;
      IF (ROW_COUNT = 0) THEN
        LEAVE;
      IF (FIELDS IS NULL) THEN
        FIELDS = TRIM(FIELD_NAME);
```

```
      ELSE
        FIELDS = FIELDS || ', ' || TRIM(FIELD_NAME);
    END
    CLOSE CUR_F;

    SCRIPT = 'CREATE VIEW ' || RELATION;

    IF (FIELDS IS NOT NULL) THEN
      SCRIPT = SCRIPT || ' (' || FIELDS || ')';

    SCRIPT = SCRIPT || ' AS ' || ASCII_CHAR(13);
    SCRIPT = SCRIPT || SOURCE;

    SUSPEND;
  END
  CLOSE CUR_R;
END
```

Siehe auch

DECLARE .. CURSOR, FETCH, CLOSE

7.7.18. FETCH

Verwendet für

Abrufen aufeinanderfolgender Datensätze aus einem Datensatz, der mit einem Cursor abgerufen wurde

Verfügbar in

PSQL

Syntax

```
FETCH [<fetch_scroll> FROM] cursor_name
  [INTO [:]varname [, [:]varname ...]];

<fetch_scroll> ::=
    NEXT | PRIOR | FIRST | LAST
  | RELATIVE n
  | ABSOLUTE n
```

Tabelle 104. FETCH*-Anweisungsparameter*

Argument	Beschreibung
cursor_name	Cursorname. Ein Cursor mit diesem Namen muss zuvor mit einem DECLARE … CURSOR-Statement deklariert und mit einem OPEN-Statement geöffnet werden.
varname	Variablenname
n	Ganzzahliger Ausdruck für die Anzahl der Zeilen

Die FETCH-Anweisung holt die erste und die nachfolgenden Zeilen aus der Ergebnismenge des Cursors und weist die Spaltenwerte PSQL-Variablen zu. Die Anweisung FETCH kann nur mit einem Cursor verwendet werden, der mit der Anweisung DECLARE .. CURSOR deklariert wurde.

Mit dem optionalen *fetch_scroll*-Teil der FETCH-Anweisung können Sie angeben, in welche Richtung und wie viele Zeilen die Cursorposition vorrücken soll. Die NEXT-Klausel kann für scrollbare und vorwärts gerichtete Cursor verwendet werden. Andere Klauseln werden nur für scrollbare Cursor unterstützt.

Die Scroll-Optionen

NEXT

bewegt den Cursor eine Zeile nach vorne; das ist die Standardeinstellung

PRIOR

Bewegt den Cursor einen Datensatz zurück

FIRST

bewegt den Cursor zum ersten Datensatz.

LAST

bewegt den Cursor zum letzten Datensatz

RELATIVE n

verschiebt den Cursor *n* Zeilen von der aktuellen Position; positive Zahlen bewegen sich vorwärts, negative Zahlen bewegen sich rückwärts; die Verwendung von null (0) bewegt den Cursor nicht, und ROW_COUNT wird auf null gesetzt, da keine neue Zeile abgerufen wurde.

Bug: Abrufen der ersten Zeile mit RELATIVE

In Firebird 3.0.7 und früher ist es nicht möglich, die erste Zeile mit FETCH RELATIVE 1 direkt nach dem Öffnen des Cursors abzurufen. Verwenden Sie als Workaround FETCH (oder FETCH NEXT), um die erste Zeile abzurufen.

Dies wird in Firebird 3.0.8 behoben, siehe CORE-6486

ABSOLUTE n

bewegt den Cursor in die angegebene Zeile; *n* ist ein ganzzahliger Ausdruck, wobei 1 die erste Zeile angibt. Bei negativen Werten wird die absolute Position vom Ende der Ergebnismenge genommen, also gibt '-1' die letzte Zeile an, '-2' die vorletzte Zeile usw. Ein Wert von Null (0) wird vor der ersten Zeile positioniert.

Bug: Positionierung über die Grenzen des Cursors hinaus

In Firebird 3.0.7 und früheren Versionen ist es mit ABSOLUTE und RELATIVE möglich, den Cursor über die Grenzen der Ergebnismenge hinaus zu positionieren — statt direkt vor der ersten Zeile oder unmittelbar nach der letzten Zeile. Nachfolgende Aufrufe von "FETCH RELATIVE" erfordern dann einen Offset, der groß genug ist, um innerhalb der Grenzen zurückzugehen, anstatt nur "1" oder "-1", um in die erste oder letzte Zeile zu gelangen.

> Dies wird in Firebird 3.0.8 behoben, siehe CORE-6487

Die optionale INTO-Klausel ruft Daten aus der aktuellen Zeile des Cursors ab und lädt sie in PSQL-Variablen. Wenn der Abruf über die Grenzen der Ergebnismenge hinaus verschoben wird, werden die Variablen auf NULL gesetzt.

Es ist auch möglich, den Cursornamen als Variable eines Zeilentyps zu verwenden (ähnlich wie OLD und NEW in Triggern), was den Zugriff auf die Spalten der Ergebnismenge ermöglicht (z. B. *cursor_name.columnname*).

Regeln für Cursor-Variablen

- Beim Zugriff auf eine Cursorvariable in einer DML-Anweisung kann der Doppelpunkt-Präfix vor dem Cursornamen (d. h. :cursor_name.columnname) zum Eindeutigmachen hinzugefügt werden, ähnlich wie bei Variablen.

  Die Cursorvariable kann ohne Doppelpunkt-Präfix referenziert werden, aber in diesem Fall kann der Name je nach Umfang der Kontexte in der Anweisung statt in den Cursor in den Anweisungskontext aufgelöst werden (z. B. Sie wählen aus einer Tabelle mit demselben Namen als Cursor).

- Cursorvariablen sind schreibgeschützt
- In einer FOR SELECT-Anweisung ohne AS CURSOR-Klausel müssen Sie die INTO-Klausel verwenden. Wenn eine AS CURSOR-Klausel angegeben wird, ist die INTO-Klausel erlaubt, aber optional; Sie können stattdessen mit dem Cursor auf die Felder zugreifen.
- Das Lesen aus einer Cursor-Variablen gibt die aktuellen Feldwerte zurück. Das bedeutet, dass eine UPDATE-Anweisung (mit einer WHERE CURRENT OF-Klausel) nicht nur die Tabelle, sondern auch die Felder in der Cursor-Variablen für nachfolgende Lesevorgänge aktualisiert. Die Ausführung einer DELETE-Anweisung (mit einer WHERE CURRENT OF-Klausel) setzt alle Felder in der Cursor-Variablen für nachfolgende Lesevorgänge auf NULL
- Wenn der Cursor nicht auf einer Zeile positioniert ist — er ist vor der ersten oder nach der letzten Zeile positioniert — führt der Versuch, aus der Cursor-Variablen zu lesen, zu einem Fehler "*Cursor **cursor_name** is not position in a valid aufzeichnen*"

Um zu überprüfen, ob alle Zeilen der Ergebnismenge geholt wurden, gibt die Kontextvariable ROW_COUNT die Anzahl der von der Anweisung geholten Zeilen zurück. Wenn ein Datensatz abgerufen wurde, ist ROW_COUNT eins (1), ansonsten null (0).

FETCH-Beispiele

1. Verwenden der FETCH-Anweisung:

```
CREATE OR ALTER PROCEDURE GET_RELATIONS_NAMES
  RETURNS (RNAME CHAR(31))
AS
  DECLARE C CURSOR FOR (
    SELECT RDB$RELATION_NAME
    FROM RDB$RELATIONS);
BEGIN
```

```
  OPEN C;
  WHILE (1 = 1) DO
  BEGIN
    FETCH C INTO RNAME;
    IF (ROW_COUNT = 0) THEN
      LEAVE;
    SUSPEND;
  END
  CLOSE C;
END
```

2. Verwenden der FETCH-Anweisung mit verschachtelten Cursorn:

```
EXECUTE BLOCK
  RETURNS (SCRIPT BLOB SUB_TYPE TEXT)
AS
  DECLARE VARIABLE FIELDS VARCHAR (8191);
  DECLARE VARIABLE FIELD_NAME TYPE OF RDB$FIELD_NAME;
  DECLARE VARIABLE RELATION RDB$RELATION_NAME;
  DECLARE VARIABLE SRC TYPE OF COLUMN RDB$RELATIONS.RDB$VIEW_SOURCE;
  -- Named cursor declaration
  DECLARE VARIABLE CUR_R CURSOR FOR (
    SELECT
      RDB$RELATION_NAME,
      RDB$VIEW_SOURCE
    FROM RDB$RELATIONS
    WHERE RDB$VIEW_SOURCE IS NOT NULL);
  -- Declaring a named cursor in which
  -- a local variable is used
  DECLARE CUR_F CURSOR FOR (
    SELECT RDB$FIELD_NAME
    FROM RDB$RELATION_FIELDS
    WHERE
    -- It is important that the variable must be declared earlier
      RDB$RELATION_NAME =: RELATION);
BEGIN
  OPEN CUR_R;
  WHILE (1 = 1) DO
  BEGIN
    FETCH CUR_R INTO RELATION, SRC;
    IF (ROW_COUNT = 0) THEN
      LEAVE;
    FIELDS = NULL;
    -- Cursor CUR_F will use the value
    -- the RELATION variable initialized above
    OPEN CUR_F;
    WHILE (1 = 1) DO
    BEGIN
      FETCH CUR_F INTO FIELD_NAME;
      IF (ROW_COUNT = 0) THEN
```

```
        LEAVE;
      IF (FIELDS IS NULL) THEN
        FIELDS = TRIM (FIELD_NAME);
      ELSE
        FIELDS = FIELDS || ',' || TRIM(FIELD_NAME);
    END
    CLOSE CUR_F;
    SCRIPT = 'CREATE VIEW' || RELATION;
    IF (FIELDS IS NOT NULL) THEN
      SCRIPT = SCRIPT || '(' || FIELDS || ')' ;
    SCRIPT = SCRIPT || 'AS' || ASCII_CHAR (13);
    SCRIPT = SCRIPT || SRC;
    SUSPEND;
  END
  CLOSE CUR_R;
EN
```

3. Ein Beispiel für die Verwendung der `FETCH`-Anweisung mit einem scrollbaren Cursor

```
EXECUTE BLOCK
  RETURNS (N INT, RNAME CHAR (31))
AS
  DECLARE C SCROLL CURSOR FOR (
    SELECT
      ROW_NUMBER() OVER (ORDER BY RDB$RELATION_NAME) AS N,
      RDB$RELATION_NAME
    FROM RDB$RELATIONS
    ORDER BY RDB$RELATION_NAME);
BEGIN
  OPEN C;
  -- move to the first record (N = 1)
  FETCH FIRST FROM C;
  RNAME = C.RDB$RELATION_NAME;
  N = C.N;
  SUSPEND;
  -- move 1 record forward (N = 2)
  FETCH NEXT FROM C;
  RNAME = C.RDB$RELATION_NAME;
  N = C.N;
  SUSPEND;
  -- move to the fifth record (N = 5)
  FETCH ABSOLUTE 5 FROM C;
  RNAME = C.RDB$RELATION_NAME;
  N = C.N;
  SUSPEND;
  -- move 1 record backward (N = 4)
  FETCH PRIOR FROM C;
  RNAME = C.RDB$RELATION_NAME;
  N = C.N;
  SUSPEND;
```

```
  -- move 3 records forward (N = 7)
  FETCH RELATIVE 3 FROM C;
  RNAME = C.RDB$RELATION_NAME;
  N = C.N;
  SUSPEND;
  -- move back 5 records (N = 2)
  FETCH RELATIVE -5 FROM C;
  RNAME = C.RDB$RELATION_NAME;
  N = C.N;
  SUSPEND;
  -- move to the first record (N = 1)
  FETCH FIRST FROM C;
  RNAME = C.RDB$RELATION_NAME;
  N = C.N;
  SUSPEND;
  -- move to the last entry
  FETCH LAST FROM C;
  RNAME = C.RDB$RELATION_NAME;
  N = C.N;
  SUSPEND;
  CLOSE C;
END
```

Siehe auch

`DECLARE .. CURSOR`, `OPEN`, `CLOSE`

7.7.19. CLOSE

Verwendet für

Einen deklarierten Cursor schließen

Verfügbar in

PSQL

Syntax

```
CLOSE cursor_name;
```

Tabelle 105. CLOSE-Anweisungsparameter

Argument	Beschreibung
cursor_name	Cursorname. Ein Cursor mit diesem Namen muss zuvor mit einem `DECLARE ⋯ CURSOR`-Statement deklariert und mit einem `OPEN`-Statement geöffnet werden

Eine `CLOSE`-Anweisung schließt einen geöffneten Cursor. Alle noch geöffneten Cursor werden automatisch geschlossen, nachdem der Modulcode die Ausführung abgeschlossen hat. Nur ein Cursor, der mit `DECLARE .. CURSOR` deklariert wurde, kann mit einer `CLOSE`-Anweisung geschlossen werden.

CLOSE-Beispiele

See FETCH-Beispiele

Siehe auch

DECLARE .. CURSOR, OPEN, FETCH

7.7.20. IN AUTONOMOUS TRANSACTION

Verwendet für

Ausführen einer Anweisung oder eines Anweisungsblocks in einer autonomen Transaktion

Verfügbar in

PSQL

Syntax

```
IN AUTONOMOUS TRANSACTION DO <compound_statement>
```

Tabelle 106. IN AUTONOMOUS TRANSACTION-*Anweisungsparameter*

Argument	Beschreibung
compound_statement	Ein Statement oder ein Block von Statements

Eine Anweisung IN AUTONOMOUS TRANSACTION ermöglicht die Ausführung einer Anweisung oder eines Anweisungsblocks in einer autonomen Transaktion. Code, der in einer autonomen Transaktion ausgeführt wird, wird unmittelbar nach seiner erfolgreichen Ausführung unabhängig vom Status seiner übergeordneten Transaktion festgeschrieben. Dies kann erforderlich sein, wenn bestimmte Vorgänge nicht zurückgesetzt werden sollen, auch wenn in der übergeordneten Transaktion ein Fehler auftritt.

Eine autonome Transaktion hat dieselbe Isolationsstufe wie ihre übergeordnete Transaktion. Jede Ausnahme, die im Block des autonomen Transaktionscodes ausgelöst wird, führt dazu, dass die autonome Transaktion zurückgesetzt wird und alle vorgenommenen Änderungen storniert werden. Wenn der Code erfolgreich ausgeführt wird, wird die autonome Transaktion festgeschrieben.

IN AUTONOMOUS TRANSACTION-Beispiele

Verwendung einer autonomen Transaktion in einem Trigger für das Datenbankereignis ON CONNECT, um alle Verbindungsversuche, einschließlich der fehlgeschlagenen, zu protokollieren:

```
CREATE TRIGGER TR_CONNECT ON CONNECT
AS
BEGIN
  -- Logging all attempts to connect to the database
  IN AUTONOMOUS TRANSACTION DO
    INSERT INTO LOG(MSG)
    VALUES ('USER ' || CURRENT_USER || ' CONNECTS.');
```

```
  IF (EXISTS(SELECT *
             FROM BLOCKED_USERS
             WHERE USERNAME = CURRENT_USER)) THEN
  BEGIN
    -- Logging that the attempt to connect
    -- to the database failed and sending
    -- a message about the event
    IN AUTONOMOUS TRANSACTION DO
    BEGIN
      INSERT INTO LOG(MSG)
      VALUES ('USER ' || CURRENT_USER || ' REFUSED.');
      POST_EVENT 'CONNECTION ATTEMPT BY BLOCKED USER!';
    END
    -- now calling an exception
    EXCEPTION EX_BADUSER;
  END
END
```

Siehe auch

Transaktionssteuerung

7.7.21. POST_EVENT

Verwendet für

Benachrichtigung von Listening-Clients über Datenbankereignisse in einem Modul

Verfügbar in

PSQL

Syntax

```
POST_EVENT event_name;
```

Tabelle 107. POST_EVENT*-Anweisungsparameter*

Argument	Beschreibung
event_name	Ereignisname (Nachricht) ist auf 127 Byte beschränkt

Die Anweisung POST_EVENT benachrichtigt den Ereignismanager über das Ereignis, das es in einer Ereignistabelle speichert. Wenn die Transaktion festgeschrieben ist, benachrichtigt der Ereignismanager Anwendungen, die ihr Interesse an dem Ereignis signalisieren.

Der Ereignisname kann eine Art Code oder eine kurze Nachricht sein: Die Auswahl ist offen, da es sich nur um eine Zeichenfolge mit bis zu 127 Bytes handelt.

Der Inhalt des Strings kann ein Stringliteral, eine Variable oder ein beliebiger gültiger SQL-Ausdruck sein, der in einen String aufgelöst wird.

POST_EVENT-Beispiele

Benachrichtigen der Listening-Anwendungen über das Einfügen eines Datensatzes in die SALES -Tabelle:

```
CREATE TRIGGER POST_NEW_ORDER FOR SALES
ACTIVE AFTER INSERT POSITION 0
AS
BEGIN
  POST_EVENT 'new_order';
END
```

7.7.22. RETURN

Verwendet für

Einen Wert aus einer gespeicherten Funktion zurückgeben

Verfügbar in

PSQL

Syntax

```
RETURN value;
```

Tabelle 108. RETURN*-Anweisungsparameter*

Argument	Beschreibung
value	Ausdruck mit dem zurückzugebenden Wert; Kann jeder Ausdruckstyp sein, der mit dem Rückgabetyp der Funktion kompatibel ist

Die RETURN-Anweisung beendet die Ausführung einer Funktion und gibt den Wert des Ausdrucks *value* zurück.

RETURN kann nur in PSQL-Funktionen (gespeicherte und lokale Funktionen) verwendet werden.

RETURN-Beispiele

Siehe auch CREATE FUNCTION-Beispiele

7.8. Abfangen und Behandeln von Fehlern

Firebird verfügt über ein nützliches Lexikon von PSQL-Anweisungen und -Ressourcen zum Abfangen von Fehlern in Modulen und deren Behandlung. Firebird verwendet integrierte Ausnahmen, die bei Fehlern ausgelöst werden, die beim Arbeiten mit DML- und DDL-Anweisungen auftreten.

Im PSQL-Code werden Ausnahmen mit der WHEN-Anweisung behandelt. Das Behandeln einer Ausnahme im Code beinhaltet entweder das Beheben des Problems vor Ort oder das Überwinden

des Problems. Bei beiden Lösungen kann die Ausführung fortgesetzt werden, ohne dass eine Ausnahmenachricht an den Client zurückgegeben wird.

Eine Ausnahme führt dazu, dass die Ausführung im aktuellen Block beendet wird. Anstatt die Ausführung an die `END`-Anweisung zu übergeben, bewegt sich die Prozedur ausgehend von dem Block, in dem die Ausnahme abgefangen wurde, durch Ebenen verschachtelter Blöcke nach außen und sucht nach dem Code des Handlers, der diese Ausnahme „kennt“. Es stoppt die Suche, wenn es die erste `WHEN`-Anweisung findet, die diese Ausnahme behandeln kann.

7.8.1. Systemausnahmen

Eine Ausnahme ist eine Nachricht, die generiert wird, wenn ein Fehler auftritt.

Alle von Firebird behandelten Ausnahmen haben vordefinierte numerische Werte für Kontextvariablen (Symbole) und damit verbundene Textnachrichten. Fehlermeldungen werden standardmäßig in Englisch ausgegeben. Es sind lokalisierte Firebird-Builds verfügbar, bei denen Fehlermeldungen in andere Sprachen übersetzt werden.

Vollständige Auflistungen der Systemausnahmen finden Sie in *Anhang B: Ausnahmecodes und Meldungen*:

- SQLSTATE-Fehlercodes und -beschreibungen
- "GDSCODE-Fehlercodes, SQLCODEs und Beschreibungen"

7.8.2. Benutzerdefinierte Ausnahmen

Benutzerdefinierte Ausnahmen können in der Datenbank als persistente Objekte deklariert und im PSQL-Code aufgerufen werden, um bestimmte Fehler zu signalisieren; B. um bestimmte Geschäftsregeln durchzusetzen. Eine benutzerdefinierte Ausnahme besteht aus einem Bezeichner und einer Standardnachricht von 1021 Byte. Weitere Informationen finden Sie unter `CREATE EXCEPTION`.

7.8.3. `EXCEPTION`

Verwendet für

Eine benutzerdefinierte Ausnahme auslösen oder eine Ausnahme erneut auslösen

Verfügbar in

PSQL

Syntax

```
EXCEPTION [
    exception_name
    [ custom_message
    | USING (<value_list>)]
  ]

<value_list> ::= <val> [, <val> ...]
```

Tabelle 109. `EXCEPTION`-*Anweisungsparameter*

Argument	Beschreibung
exception_name	Name der Ausnahme
custom_message	Alternativer Nachrichtentext, der an die Aufruferschnittstelle zurückgegeben wird, wenn eine Ausnahme ausgelöst wird. Die maximale Länge der Textnachricht beträgt 1.021 Byte
val	Wertausdruck, der Parameter-Slots im Ausnahmenachrichtentext ersetzt

Eine Anweisung `EXCEPTION` löst die benutzerdefinierte Ausnahme mit dem angegebenen Namen aus. Ein alternativer Nachrichtentext von bis zu 1.021 Byte kann optional den Standardnachrichtentext der Ausnahme überschreiben.

Die Standardausnahmenachricht kann Slots für Parameter enthalten, die beim Auslösen einer Ausnahme gefüllt werden können. Um Parameterwerte an eine Ausnahme zu übergeben, verwenden Sie die `USING`-Klausel. Betrachten wir in der Reihenfolge von links nach rechts, dass jeder Parameter, der in der Anweisung zum Auslösen von Ausnahmen als “the *N*th” übergeben wird, mit *N* beginnend bei 1:

- Wenn der *N*te Parameter nicht übergeben wird, wird sein Slot nicht ersetzt
- Wird ein `NULL` Parameter übergeben, wird der Slot durch den String “*** `null` ***” ersetzt
- Werden mehr Parameter übergeben, als in der Ausnahmemeldung definiert sind, werden die überzähligen ignoriert
- Die maximale Anzahl von Parametern beträgt 9
- Die maximale Nachrichtenlänge einschließlich Parameterwerten beträgt 1053 Byte

Der Statusvektor wird durch diese Codekombination `isc_except, <Exception number>, isc_formatted_exception, <formatted Exception message>, <Exception parameters>` generiert.

Da ein neuer Fehlercode (`isc_formatted_exception`) verwendet wird, muss der Client Version 3.0 sein oder zumindest die `firebird.msg` von Version 3.0 verwenden, um den Statusvektor in einen String zu übersetzen.

Wenn die *message* eine Parameter-Slot-Nummer enthält, die größer als 9 ist, werden die zweite und die nachfolgenden Ziffern als Literaltext behandelt. Zum Beispiel wird `@10` als Slot 1 interpretiert, gefolgt von einem Literal ``0’.

Als Beispiel:

```
CREATE EXCEPTION ex1
  'something wrong in @ 1 @ 2 @ 3 @ 4 @ 5 @ 6 @ 7 @ 8 @ 9 @ 10 @ 11';
SET TERM ^;
EXECUTE BLOCK AS
BEGIN
  EXCEPTION ex1 USING ( 'a' , 'b' , 'c' , 'd' , 'e' , 'f' , 'g' , 'h' ,
```

```
'i' );
END^
```

Dies erzeugt die folgende Ausgabe

```
Statement failed, SQLSTATE = HY000
exception 1
-EX1
-something wrong in abcdefghi a0 a1
```

Ausnahmen können in einer `WHEN ⋯ DO`-Anweisung behandelt werden. Wenn eine Ausnahme in einem Modul nicht behandelt wird, werden die Auswirkungen der in diesem Modul ausgeführten Aktionen aufgehoben und das aufrufende Programm empfängt die Ausnahme (entweder den Standardtext oder den benutzerdefinierten Text).

Innerhalb des Ausnahmebehandlungsblocks — und nur darin — kann die abgefangene Ausnahme erneut ausgelöst werden, indem die `EXCEPTION`-Anweisung ohne Parameter ausgeführt wird. Wenn er sich außerhalb des Blocks befindet, hat der erneut ausgelöste `EXCEPTION`-Aufruf keine Wirkung.

Custom exceptions are stored in the system table `RDB$EXCEPTIONS`.

`EXCEPTION`-Beispiele

1. Auslösen einer Ausnahme bei einer Bedingung in der gespeicherten Prozedur `SHIP_ORDER`:

```
CREATE OR ALTER PROCEDURE SHIP_ORDER (
  PO_NUM CHAR(8))
AS
  DECLARE VARIABLE ord_stat  CHAR(7);
  DECLARE VARIABLE hold_stat CHAR(1);
  DECLARE VARIABLE cust_no   INTEGER;
  DECLARE VARIABLE any_po    CHAR(8);
BEGIN
  SELECT
    s.order_status,
    c.on_hold,
    c.cust_no
  FROM
    sales s, customer c
  WHERE
    po_number = :po_num AND
    s.cust_no = c.cust_no
  INTO :ord_stat,
       :hold_stat,
       :cust_no;

  IF (ord_stat = 'shipped') THEN
    EXCEPTION order_already_shipped;
```

```
  /* Other statements */
END
```

2. Eine Ausnahme bei einer Bedingung auslösen und die ursprüngliche Nachricht durch eine alternative Nachricht ersetzen:

```
CREATE OR ALTER PROCEDURE SHIP_ORDER (
  PO_NUM CHAR(8))
AS
  DECLARE VARIABLE ord_stat  CHAR(7);
  DECLARE VARIABLE hold_stat CHAR(1);
  DECLARE VARIABLE cust_no   INTEGER;
  DECLARE VARIABLE any_po    CHAR(8);
BEGIN
  SELECT
    s.order_status,
    c.on_hold,
    c.cust_no
  FROM
    sales s, customer c
  WHERE
    po_number = :po_num AND
    s.cust_no = c.cust_no
  INTO :ord_stat,
       :hold_stat,
       :cust_no;

  IF (ord_stat = 'shipped') THEN
    EXCEPTION order_already_shipped
      'Order status is "' || ord_stat || '"';
  /* Other statements */
END
```

3. Verwenden einer parametrisierten Ausnahme:

```
CREATE EXCEPTION EX_BAD_SP_NAME
  'Name of procedures must start with' '@ 1' ':' '@ 2' '' ;
...
CREATE TRIGGER TRG_SP_CREATE BEFORE CREATE PROCEDURE
AS
  DECLARE SP_NAME VARCHAR(255);
BEGIN
  SP_NAME = RDB$GET_CONTEXT ('DDL_TRIGGER' , 'OBJECT_NAME');
  IF (SP_NAME NOT STARTING 'SP_') THEN
    EXCEPTION EX_BAD_SP_NAME USING ('SP_', SP_NAME);
END
```

4. Logging an error and re-throwing it in the `WHEN` block:

```
CREATE PROCEDURE ADD_COUNTRY (
  ACountryName COUNTRYNAME,
  ACurrency VARCHAR(10))
AS
BEGIN
  INSERT INTO country (country,
                       currency)
  VALUES (:ACountryName,
          :ACurrency);
  WHEN ANY DO
  BEGIN
    -- write an error in log
    IN AUTONOMOUS TRANSACTION DO
      INSERT INTO ERROR_LOG (PSQL_MODULE,
                             GDS_CODE,
                             SQL_CODE,
                             SQL_STATE)
      VALUES ('ADD_COUNTRY',
              GDSCODE,
              SQLCODE,
              SQLSTATE);
    -- Re-throw exception
    EXCEPTION;
  END
END
```

Siehe auch

`CREATE EXCEPTION`, `WHEN … DO`

7.8.4. WHEN … DO

Verwendet für

Eine Ausnahme abfangen und den Fehler behandeln

Verfügbar in

PSQL

Syntax

```
WHEN {<error> [, <error> ...] | ANY}
DO <compound_statement>

<error> ::=
  { EXCEPTION exception_name
  | SQLCODE number
  | GDSCODE errcode
  | SQLSTATE sqlstate_code }
```

Tabelle 110. `WHEN … DO`*-Anweisungsparameter*

Argument	Beschreibung
exception_name	Name der Ausnahme
number	SQLCODE-Fehlercode
errcode	Symbolischer GDSCODE-Fehlername
sqlstate_code	String-Literal mit dem SQLSTATE-Fehlercode
compound_statement	Eine einzelne Anweisung oder ein Block von Anweisungen

Die Anweisung `WHEN … DO` wird verwendet, um Fehler und benutzerdefinierte Ausnahmen zu behandeln. Die Anweisung erfasst alle Fehler und benutzerdefinierten Ausnahmen, die nach dem Schlüsselwort `WHEN` aufgeführt sind. Wenn `WHEN` das Schlüsselwort `ANY` folgt, fängt die Anweisung jeden Fehler oder jede benutzerdefinierte Ausnahme ab, auch wenn sie bereits in einer `WHEN` -Anweisung weiter oben im Block behandelt wurden.

Der `WHEN … DO`-Block muss sich am Ende eines Anweisungsblocks befinden, vor der Anweisung `END` des Blocks.

Auf das Schlüsselwort `DO` folgt eine Anweisung oder ein Anweisungsblock innerhalb eines `BEGIN … END`-Wrappers, der die Ausnahme behandelt. Die Kontextvariablen `SQLCODE`, `GDSCODE` und `SQLSTATE` stehen im Kontext dieser Anweisung oder dieses Blocks zur Verfügung. Die Anweisung `EXCEPTION` ohne Parameter kann auch in diesem Kontext verwendet werden, um den Fehler oder die Ausnahme erneut zu werfen.

Bezüglich `GDSCODE`

Das Argument für die Klausel `WHEN GDSCODE` ist der symbolische Name, der der intern definierten Ausnahme zugeordnet ist, z.B. `grant_obj_notfound` für den GDS-Fehler 335544551.

In einer Anweisung oder einem Anweisungsblock der `DO`-Klausel wird eine `GDSCODE` -Kontextvariable verfügbar, die den numerischen Code enthält. Dieser numerische Code ist erforderlich, wenn Sie eine `GDSCODE`-Ausnahme mit einem gezielten Fehler vergleichen möchten. Um ihn mit einem bestimmten Fehler zu vergleichen, müssen Sie einen numerischen Wert verwenden, zum Beispiel `335544551` für `grant_obj_notfound`.

Ähnliche Kontextvariablen sind für `SQLCODE` und `SQLSTATE` verfügbar.

Die Anweisung oder der Block `WHEN … DO` wird nur ausgeführt, wenn eines der von seinen Bedingungen betroffenen Ereignisse zur Laufzeit eintritt. Wenn die Anweisung `WHEN … DO` ausgeführt wird, wird die Ausführung auch dann fortgesetzt, als ob kein Fehler aufgetreten wäre: Der Fehler oder die benutzerdefinierte Ausnahme beendet weder die Operationen des Triggers oder der gespeicherten Prozedur noch setzt sie diese zurück.

Wenn jedoch die `WHEN … DO`-Anweisung oder der Block nichts zur Behandlung oder Behebung des Fehlers tut, wird die DML-Anweisung (`SELECT`, `INSERT`, `UPDATE`, `DELETE`, `MERGE`), die den Fehler verursacht hat, error wird zurückgesetzt und keine der Anweisungen darunter im selben Anweisungsblock wird ausgeführt.

1. Wenn der Fehler nicht durch eine der DML-Anweisungen (`SELECT`, `INSERT`, `UPDATE`, `DELETE`, `MERGE`) verursacht wird, wird der gesamte Anweisungsblock zurückgesetzt, nicht nur der, der den Fehler verursacht hat ein Fehler. Alle Operationen in der `WHEN … DO`-Anweisung werden ebenfalls zurückgesetzt. Die gleiche Einschränkung gilt für die Anweisung `EXECUTE PROCEDURE`. Lesen Sie eine interessante Diskussion des Phänomens im Firebird Tracker-Ticket CORE-4483.
2. In auswählbaren gespeicherten Prozeduren bleiben Ausgabezeilen, die bereits in früheren Iterationen einer `FOR SELECT … DO … SUSPEND'-Schleife an den Client übergeben wurden, für den Client verfügbar, wenn anschließend beim Abrufen von Zeilen eine Ausnahme ausgelöst wird.

Anwendungsbereiche einer `WHEN … DO`-Anweisung

Eine Anweisung `WHEN … DO` fängt Fehler und Ausnahmen im aktuellen Anweisungsblock ab. Es fängt auch ähnliche Ausnahmen in verschachtelten Blöcken ab, wenn diese Ausnahmen nicht in ihnen behandelt wurden.

Alle Änderungen, die vor der Anweisung vorgenommen wurden, die den Fehler verursacht hat, sind für eine `WHEN … DO`-Anweisung sichtbar. Wenn Sie jedoch versuchen, sie in einer autonomen Transaktion zu protokollieren, sind diese Änderungen nicht verfügbar, da die Transaktion, bei der die Änderungen stattfanden, zu dem Zeitpunkt, zu dem die autonome Transaktion gestartet wird, nicht festgeschrieben ist. Das untere Beispiel 4 zeigt dieses Verhalten.

Bei der Behandlung von Ausnahmen ist es manchmal wünschenswert, die Ausnahme zu behandeln, indem eine Protokollnachricht geschrieben wird, um den Fehler zu markieren und die Ausführung über den fehlerhaften Datensatz hinaus fortsetzen zu lassen. Logs können in reguläre Tabellen geschrieben werden, aber dabei gibt es ein Problem: Die Log-Records werden “verschwinden”, wenn ein nicht behandelter Fehler dazu führt, dass das Modul nicht mehr ausgeführt wird und ein Rollback durchgeführt wird. Die Verwendung von external tables kann hier sinnvoll sein, da die Daten, die in diese geschrieben werden, transaktionsunabhängig sind. Die verknüpfte externe Datei ist weiterhin vorhanden, unabhängig davon, ob der Gesamtprozess erfolgreich ist oder nicht.

Beispiele für `WHEN…DO`

1. Ersetzen des Standardfehlers durch einen benutzerdefinierten Fehler:

```
CREATE EXCEPTION COUNTRY_EXIST '';
SET TERM ^;
CREATE PROCEDURE ADD_COUNTRY (
  ACountryName COUNTRYNAME,
  ACurrency VARCHAR(10) )
AS
BEGIN
  INSERT INTO country (country, currency)
    VALUES (:ACountryName, :ACurrency);
```

```
  WHEN SQLCODE -803 DO
    EXCEPTION COUNTRY_EXIST 'Country already exists!';
END^
SET TERM ^;
```

2. Einen Fehler protokollieren und erneut in den `WHEN`-Block werfen:

```
CREATE PROCEDURE ADD_COUNTRY (
  ACountryName COUNTRYNAME,
  ACurrency VARCHAR(10) )
AS
BEGIN
  INSERT INTO country (country,
                       currency)
  VALUES (:ACountryName,
          :ACurrency);
  WHEN ANY DO
  BEGIN
    -- write an error in log
    IN AUTONOMOUS TRANSACTION DO
      INSERT INTO ERROR_LOG (PSQL_MODULE,
                             GDS_CODE,
                             SQL_CODE,
                             SQL_STATE)
      VALUES ('ADD_COUNTRY',
              GDSCODE,
              SQLCODE,
              SQLSTATE);
    -- Re-throw exception
    EXCEPTION;
  END
END
```

3. Behandeln mehrerer Fehler in einem `WHEN`-Block

```
...
WHEN GDSCODE GRANT_OBJ_NOTFOUND,
     GDSCODE GRANT_FLD_NOTFOUND,
     GDSCODE GRANT_NOPRIV,
     GDSCODE GRANT_NOPRIV_ON_BASE
DO
BEGIN
  EXECUTE PROCEDURE LOG_GRANT_ERROR(GDSCODE);
  EXIT;
END
...
```

4. Abfangen von Fehlern mit dem SQLSTATE-Code

```
EXECUTE BLOCK
AS
  DECLARE VARIABLE I INT;
BEGIN
  BEGIN
    I = 1/0;
    WHEN SQLSTATE '22003' DO
      EXCEPTION E_CUSTOM_EXCEPTION
        'Numeric value out of range.';
    WHEN SQLSTATE '22012' DO
      EXCEPTION E_CUSTOM_EXCEPTION
        'Division by zero.';
    WHEN SQLSTATE '23000' DO
      EXCEPTION E_CUSTOM_EXCEPTION
       'Integrity constraint violation.';
  END
END
```

Siehe auch

EXCEPTION, CREATE EXCEPTION, SQLCODE- und GDSCODE-Fehlercodes und Meldungstexte und SQLSTATE-Codes und Nachrichtentexte, GDSCODE, SQLCODE, SQLSTATE

Kapitel 8. Eingebaute Skalarfunktionen

Upgrader: BITTE LESEN!

Eine große Anzahl von Funktionen, die in früheren Versionen von Firebird als externe Funktionen (UDFs) implementiert wurden, wurden schrittweise als interne (eingebaute) Funktionen neu implementiert. Wenn eine externe Funktion mit dem gleichen Namen wie eine integrierte Funktion in Ihrer Datenbank deklariert ist, bleibt sie dort und überschreibt alle internen Funktionen desselben Namens.

Um die internen Funktionen verfügbar sind, müssen Sie entweder ein `DROP` der UDF durchführen oder mittels `ALTER EXTERNAL FUNCTION` den Namen der UDF ändern.

8.1. Kontextfunktionen

8.1.1. RDB$GET_CONTEXT()

Verfügbar in

DSQL, PSQL * Als deklariertes UDF sollte es in ESQL verfügbar sein

Ergebnistyp

`VARCHAR(255)`

Syntax

```
RDB$GET_CONTEXT ('<namespace>', <varname>)

<namespace> ::= SYSTEM | USER_SESSION | USER_TRANSACTION | DDL_TRIGGER
<varname>   ::= Eine Zeichenfolge in Anführungszeichen von max. 80 Zeichen
```

Tabelle 111. `RDB$GET_CONTEXT`-*Funktionsparameter*

Parameter	Beschreibung
namespace	Namespace
varname	Variablenname. Groß-/Kleinschreibung beachten. Die maximale Länge beträgt 80 Zeichen

Retrieves the value of a context variable from one of the namespaces `SYSTEM`, `USER_SESSION` and `USER_TRANSACTION`.

Die Namensräume

Die Namensräume `USER_SESSION` und `USER_TRANSACTION` sind zunächst leer. Der Benutzer kann mit `RDB$SET_CONTEXT()` Variablen darin erstellen und setzen und mit `RDB$GET_CONTEXT()` abrufen. Der Namespace `SYSTEM` ist schreibgeschützt. Der Namespace `DDL_TRIGGER` ist nur in DDL-Triggern gültig und schreibgeschützt. Es enthält eine Reihe vordefinierter Variablen (siehe unten).

Rückgabewerte und Fehlerverhalten

Existiert die abgefragte Variable im angegebenen Namespace, wird ihr Wert als String von max. 255 Zeichen. Existiert der Namespace nicht oder versucht man auf eine nicht vorhandene Variable im `SYSTEM`-Namespace zuzugreifen, wird ein Fehler ausgegeben. Wenn Sie eine nicht vorhandene Variable in einem der anderen Namespaces anfordern, wird `NULL` zurückgegeben. Sowohl Namespace- als auch Variablennamen müssen in einfachen Anführungszeichen angegeben werden, wobei die Groß-/Kleinschreibung beachtet werden muss, nicht `NULL`-Strings.

The `SYSTEM` Namespace

Kontextvariablen im SYSTEM-Namensraum

`CLIENT_ADDRESS`

Für TCPv4 ist dies die IP-Adresse. Für XNET die lokale Prozess-ID. Für alle anderen Protokolle ist diese Variable `NULL`.

`CURRENT_ROLE`

Wie die globale Variable `CURRENT_ROLE`.

`CURRENT_USER`

Same als globale Variable `CURRENT_USER`.

`DB_NAME`

Entweder der vollständige Pfad zur Datenbank oder – falls eine Verbindung über den Pfad nicht erlaubt ist – ihr Alias.

`ENGINE_VERSION`

Die Firebird-Engine (Server)-Version.

`ISOLATION_LEVEL`

Die Isolationsstufe der aktuellen Transaktion: `'READ COMMITTED'`, `'SNAPSHOT'` oder `'CONSISTENCY'`.

`NETZWERK_PROTOKOLL`

Das für die Verbindung verwendete Protokoll: `'TCPv4'`, `'WNET'`, `'XNET'` oder `NULL`.

`SESSION_ID`

Entspricht der globalen Variable `CURRENT_CONNECTION`.

`TRANSACTION_ID`

Wie die globale Variable `CURRENT_TRANSACTION`.

`WIRE_COMPRESSED`

Komprimierungsstatus der aktuellen Verbindung. Wenn die Verbindung komprimiert ist, wird `TRUE` zurückgegeben; wenn es nicht komprimiert ist, wird `FALSE` zurückgegeben. Gibt `NULL` zurück, wenn die Verbindung eingebettet ist.

Eingeführt in Firebird 3.0.4.

`WIRE_ENCRYPTED`

Verschlüsselungsstatus der aktuellen Verbindung. Wenn die Verbindung verschlüsselt ist, wird `TRUE` zurückgegeben; wenn es nicht verschlüsselt ist, wird `FALSE` zurückgegeben. Gibt `NULL` zurück, wenn die Verbindung eingebettet ist.

Eingeführt in Firebird 3.0.4.

Der `DDL_TRIGGER`-Namespace

Der Namespace `DDL_TRIGGER` ist nur gültig, wenn ein DDL-Trigger ausgeführt wird. Seine Verwendung ist auch in gespeicherten Prozeduren und Funktionen gültig, die von DDL-Triggern aufgerufen werden.

Der Kontext `DDL_TRIGGER` funktioniert wie ein Stack. Bevor ein DDL-Trigger ausgelöst wird, werden die Werte relativ zum ausgeführten Befehl auf diesen Stack gelegt. Nach Abschluss des Triggers werden die Werte ausgegeben. Wenn also im Fall von kaskadierten DDL-Anweisungen ein Benutzer-DDL-Befehl einen DDL-Trigger auslöst und dieser Trigger einen anderen DDL-Befehl mit `EXECUTE STATEMENT` ausführt, sind die Werte des `DDL_TRIGGER`-Namensraums diejenigen relativ zu dem Befehl, der den letzten ausgelöst hat DDL-Trigger in der Aufrufliste.

Kontextvariablen im DDL_TRIGGER-Namespace

`EVENT_TYPE`

Ereignistyp (`CREATE`, `ALTER`, `DROP`)

`OBJECT_TYPE`

Objekttyp (`TABLE`, `VIEW`, etc)

`DDL_EVENT`

Ereignisname (`<ddl event item>`), wobei `<ddl_event_item> EVENT_TYPE || . ist ' ' || OBJECT_TYPE`

`OBJECT_NAME`

Name des Metadatenobjekts

`OLD_OBJECT_NAME`

zum Nachverfolgen der Umbenennung einer Domain (siehe Hinweis)

`NEW_OBJECT_NAME`

zum Nachverfolgen der Umbenennung einer Domain (siehe Hinweis)

`SQL_TEXT`

SQL-Anweisungstext

`ALTER DOMAIN old-name TO new-name` setzt `OLD_OBJECT_NAME` und `NEW_OBJECT_NAME` sowohl in den `BEFORE`- als auch `AFTER`-Triggern. Für diesen Befehl hat `OBJECT_NAME` den alten Objektnamen in `BEFORE` Triggern und den neuen Objektnamen in `AFTER` Triggern.

Beispiele

```
select rdb$get_context('SYSTEM', 'DB_NAME') from rdb$database

New.UserAddr = rdb$get_context('SYSTEM', 'CLIENT_ADDRESS');

insert into MyTable (TestField)
  values (rdb$get_context('USER_SESSION', 'MyVar'))
```

Siehe auch

RDB$SET_CONTEXT()

8.1.2. RDB$SET_CONTEXT()

Verfügbar in

DSQL, PSQL * Als deklariertes UDF sollte es in ESQL verfügbar sein

Ergebnistyp

INTEGER

Syntax

```
RDB$SET_CONTEXT ('<namespace>', <varname>, <value> | NULL)

<namespace> ::= USER_SESSION | USER_TRANSACTION
<varname>   ::= Eine Zeichenfolge in Anführungszeichen von max. 80 Zeichen
<value>     ::= Ein Wert beliebiger Art, solange er umsetzbar ist
                zu einem VARCHAR(255)
```

Tabelle 112. RDB$SET_CONTEXT-*Funktionsparameter*

Parameter	Beschreibung
namespace	Namespace
varname	Variablenname. Groß-/Kleinschreibung beachten. Die maximale Länge beträgt 80 Zeichen
value	Daten eines beliebigen Typs, sofern sie in VARCHAR(255) umgewandelt werden können

Erstellt, setzt oder hebt eine Variable in einem der vom Benutzer beschreibbaren Namensräume USER_SESSION und USER_TRANSACTION auf.

Die Namensräume

Die Namensräume USER_SESSION und USER_TRANSACTION sind zunächst leer. Der Benutzer kann mit RDB$SET_CONTEXT() Variablen darin erstellen und setzen und mit RDB$GET_CONTEXT() abrufen. Der Kontext USER_SESSION ist an die aktuelle Verbindung gebunden. Variablen in USER_TRANSACTION existieren nur in der Transaktion, in der sie gesetzt wurden. Wenn die Transaktion endet, werden der Kontext und alle darin definierten Variablen zerstört.

Rückgabewerte und Fehlerverhalten

Die Funktion gibt 1 zurück, wenn die Variable bereits vor dem Aufruf existierte und `0` wenn dies nicht der Fall war. Um eine Variable aus einem Kontext zu entfernen, setzen Sie sie auf `NULL`. Wenn der angegebene Namespace nicht existiert, wird ein Fehler ausgegeben. Sowohl Namensraum- als auch Variablennamen müssen in einfachen Anführungszeichen eingegeben werden, wobei die Groß-/Kleinschreibung beachtet werden muss, nicht `NULL`-Zeichenfolgen.

- Die maximale Anzahl von Variablen in einem einzelnen Kontext beträgt 1000.
- Alle `USER_TRANSACTION`-Variablen überleben ein `ROLLBACK RETAIN` (siehe `ROLLBACK` -Optionen) oder `ROLLBACK TO SAVEPOINT` unverändert, egal zu welchem Zeitpunkt der Transaktion sie gesetzt wurden.
- Aufgrund seiner UDF-ähnlichen Natur kann `RDB$SET_CONTEXT` — nur in PSQL — wie eine void-Funktion aufgerufen werden, ohne das Ergebnis zuzuweisen, wie im zweiten Beispiel oben. Reguläre interne Funktionen erlauben diese Art der Nutzung nicht.

Beispiele

```
select rdb$set_context('USER_SESSION', 'MyVar', 493) from rdb$database

rdb$set_context('USER_SESSION', 'RecordsFound', RecCounter);

select rdb$set_context('USER_TRANSACTION', 'Savepoints', 'Yes')
  from rdb$database
```

Siehe auch

`RDB$GET_CONTEXT()`

8.2. Mathematische Funktionen

8.2.1. ABS()

Verfügbar in

DSQL, PSQL

Möglicher Namenskonflikt

YES → Details lesen

Ergebnistyp

Numerisch

Syntax

```
ABS (number)
```

Tabelle 113. ABS-Funktionsparameter

Parameter	Beschreibung
number	Ein Ausdruck eines numerischen Typs

Gibt den absoluten Wert des Arguments zurück.

8.2.2. ACOS()

Verfügbar in

DSQL, PSQL

Möglicher Namenskonflikt

YES → Details lesen

Ergebnistyp

`DOUBLE PRECISION`

Syntax

```
ACOS (number)
```

Tabelle 114. ACOS-Funktionsparameter

Parameter	Beschreibung
number	Ein Ausdruck eines numerischen Typs im Bereich [-1, 1]

Gibt den Arkuskosinus des Arguments zurück.

- Das Ergebnis ist ein Winkel im Bereich [0, pi].

Siehe auch

`COS()`, `ASIN()`, `ATAN()`

8.2.3. ACOSH()

Verfügbar in

DSQL, PSQL

Ergebnistyp

`DOUBLE PRECISION`

Syntax

```
ACOSH (number)
```

Tabelle 115. ACOSH-Funktionsparameter

Parameter	Beschreibung
number	Jeder Nicht-`NULL`-Wert im Bereich [1, INF].

Gibt den inversen hyperbolischen Kosinus des Arguments zurück.

- Das Ergebnis liegt im Bereich [0, INF].

Siehe auch

`COSH()`, `ASINH()`, `ATANH()`

8.2.4. ASIN()

Verfügbar in

DSQL, PSQL

Möglicher Namenskonflikt

YES → Details lesen

Ergebnistyp

`DOUBLE PRECISION`

Syntax

```
ASIN (number)
```

Tabelle 116. ASIN-Funktionsparameter

Parameter	Beschreibung
number	Ein Ausdruck eines numerischen Typs im Bereich [-1, 1]

Gibt den Arkussinus des Arguments zurück.

- Das Ergebnis ist ein Winkel im Bereich [-pi/2, pi/2].

Siehe auch

`SIN()`, `ACOS()`, `ATAN()`

8.2.5. ASINH()

Verfügbar in

DSQL, PSQL

Ergebnistyp

`DOUBLE PRECISION`

Syntax

```
ASINH (number)
```

Tabelle 117. ASINH-Funktionsparameter

Parameter	Beschreibung
number	Jeder Nicht-NULL-Wert im Bereich [-INF, INF].

Gibt den inversen hyperbolischen Sinus des Arguments zurück.

- Das Ergebnis liegt im Bereich [-INF, INF].

Siehe auch

SINH(), ACOSH(), ATANH()

8.2.6. ATAN()

Verfügbar in

DSQL, PSQL

Möglicher Namenskonflikt

YES → Details lesen

Ergebnistyp

DOUBLE PRECISION

Syntax

```
ATAN (number)
```

Tabelle 118. ATAN-Funktionsparameter

Parameter	Beschreibung
number	Ein Ausdruck eines numerischen Typs

Die Funktion ATAN gibt den Arkustangens des Arguments zurück. Das Ergebnis ist ein Winkel im Bereich <-pi/2, pi/2>.

Siehe auch

ATAN2(), TAN(), ACOS(), ASIN()

8.2.7. ATAN2()

Verfügbar in

DSQL, PSQL

Möglicher Namenskonflikt

YES → Details lesen

Ergebnistyp

DOUBLE PRECISION

Syntax

```
ATAN2 (y, x)
```

Tabelle 119. ATAN2-Funktionsparameter

Parameter	Beschreibung
y	Ein Ausdruck eines numerischen Typs
x	Ein Ausdruck eines numerischen Typs

Gibt den Winkel zurück, dessen Sinus-zu-Cosinus-*Verhältnis* durch die beiden Argumente gegeben ist und dessen Sinus- und Kosinus-*Vorzeichen* den Vorzeichen der Argumente entsprechen. Dies ermöglicht Ergebnisse über den gesamten Kreis, einschließlich der Winkel -pi/2 und pi/2.

- Das Ergebnis ist ein Winkel im Bereich [-pi, pi].
- Wenn *x* negativ ist, ist das Ergebnis pi, wenn *y* 0 ist, und -pi, wenn *y* -0 ist.
- Wenn sowohl *y* als auch *x* 0 sind, ist das Ergebnis bedeutungslos. Ab Firebird 3.0 wird ein Fehler ausgegeben, wenn beide Argumente 0 sind. Bei v.2.5.4 ist es in niedrigeren Versionen immer noch nicht behoben. Weitere Informationen finden Sie unter Tracker-Ticket CORE-3201.

- Eine vollständig äquivalente Beschreibung dieser Funktion ist die folgende: `ATAN2(y, x)` ist der Winkel zwischen der positiven X-Achse und der Linie vom Ursprung zum Punkt *(x, y)*. Damit ist auch klar, dass `ATAN2(0, 0)` undefiniert ist.
- Wenn *x* größer als 0 ist, ist `ATAN2(y, x)` dasselbe wie `ATAN(y/x)`.
- Wenn sowohl Sinus als auch Kosinus des Winkels bereits bekannt sind, gibt `ATAN2(sin, cos)` den Winkel an.

8.2.8. ATANH()

Verfügbar in

DSQL, PSQL

Ergebnistyp

`DOUBLE PRECISION`

Syntax

```
ATANH (number)
```

Tabelle 120. ATANH-Funktionsparameter

Parameter	Beschreibung
number	Jeder Nicht-`NULL`-Wert im Bereich <-1, 1>.

Gibt den inversen hyperbolischen Tangens des Arguments zurück.

- Das Ergebnis ist eine Zahl im Bereich [-INF, INF].

Siehe auch

`TANH()`, `ACOSH()`, `ASINH()`

8.2.9. `CEIL()`, `CEILING()`

Verfügbar in

DSQL, PSQL

Möglicher Namenskonflikt

YES → Details lesen (Betrifft nur `CEILING`)

Ergebnistyp

`BIGINT` für exakte numerische *Zahl* oder `DOUBLE PRECISION` für Gleitkomma-*Zahl*

Syntax

```
CEIL[ING] (number)
```

Tabelle 121. `CEIL[ING]`-Funktionsparameter

Parameter	Beschreibung
number	Ein Ausdruck eines numerischen Typs

Gibt die kleinste ganze Zahl zurück, die größer oder gleich dem Argument ist.

Siehe auch

`FLOOR()`, `ROUND()`, `TRUNC()`

8.2.10. `COS()`

Verfügbar in

DSQL, PSQL

Möglicher Namenskonflikt

YES → Details lesen

Ergebnistyp

`DOUBLE PRECISION`

Syntax

```
COS (angle)
```

Tabelle 122. `COS`-Funktionsparameter

Parameter	Beschreibung
angle	Ein Winkel im Bogenmaß

Gibt den Kosinus eines Winkels zurück. Das Argument muss im Bogenmaß angegeben werden.

- Jedes Ergebnis, das nicht `NULL` ist, liegt — offensichtlich — im Bereich [-1, 1].

Siehe auch

`ACOS()`, `COT()`, `SIN()`, `TAN()`

8.2.11. `COSH()`

Verfügbar in

DSQL, PSQL

Möglicher Namenskonflikt

YES → Details lesen

Ergebnistyp

`DOUBLE PRECISION`

Syntax

```
COSH (number)
```

Tabelle 123. `COSH`-Funktionsparameter

Parameter	Beschreibung
number	Eine Zahl eines numerischen Typs

Gibt den hyperbolischen Kosinus des Arguments zurück.

- Jedes Ergebnis, das nicht `NULL` ist, liegt im Bereich [1, INF].

Siehe auch

`ACOSH()`, `SINH()`, `TANH()`

8.2.12. `COT()`

Verfügbar in

DSQL, PSQL

Möglicher Namenskonflikt

YES → Details lesen

Ergebnistyp

`DOUBLE PRECISION`

Syntax

```
COT (angle)
```

Tabelle 124. COT-Funktionsparameter

Parameter	Beschreibung
angle	Ein Winkel im Bogenmaß

Gibt den Kotangens eines Winkels zurück. Das Argument muss im Bogenmaß angegeben werden.

Siehe auch

COS(), SIN(), TAN()

8.2.13. EXP()

Verfügbar in

DSQL, PSQL

Ergebnistyp

DOUBLE PRECISION

Syntax

```
EXP (number)
```

Tabelle 125. EXP-Funktionsparameter

Parameter	Beschreibung
number	Eine Zahl eines numerischen Typs

Gibt die natürliche Exponentialfunktion zurück, e^{number}

Siehe auch

LN()

8.2.14. FLOOR()

Verfügbar in

DSQL, PSQL

Möglicher Namenskonflikt

YES → Details lesen

Ergebnistyp

BIGINT für genaue numerische *number*, oder DOUBLE PRECISION für fließkommagenaue *number*

Syntax

```
FLOOR (number)
```

Tabelle 126. FLOOR-Funktionsparameter

Parameter	Beschreibung
number	Ein Ausdruck eines numerischen Typs

Gibt die größte ganze Zahl zurück, die kleiner oder gleich dem Argument ist.

Siehe auch

CEIL(), CEILING(), ROUND(), TRUNC()

8.2.15. LN()

Verfügbar in

DSQL, PSQL

Möglicher Namenskonflikt

YES → Details lesen

Ergebnistyp

DOUBLE PRECISION

Syntax

```
LN (number)
```

Tabelle 127. LN-Funktionsparameter

Parameter	Beschreibung
number	Ein Ausdruck eines numerischen Typs

Gibt den natürlichen Logarithmus des Arguments zurück.

- Ein Fehler wird ausgegeben, wenn das Argument negativ oder 0 ist.

Siehe auch

EXP(), LOG(), LOG10()

8.2.16. LOG()

Verfügbar in

DSQL, PSQL

Möglicher Namenskonflikt

YES → Details lesen

Ergebnistyp
`DOUBLE PRECISION`

Syntax

```
LOG (x, y)
```

Tabelle 128. `LOG`-Funktionsparameter

Parameter	Beschreibung
x	Basis. Ein Ausdruck eines numerischen Typs
y	Ein Ausdruck eines numerischen Typs

Gibt den *x*-basierten Logarithmus von *y* zurück.

- Wenn eines der Argumente 0 oder kleiner ist, wird ein Fehler ausgegeben. (Vor 2.5 würde dies `NaN`, `+/-INF` oder 0 ergeben, abhängig von den genauen Werten der Argumente.)
- Wenn beide Argumente 1 sind, wird `NaN` zurückgegeben.
- Wenn *x* = 1 und *y* < 1 ist, wird `-INF` zurückgegeben.
- Wenn *x* = 1 und *y* > 1 ist, wird `INF` zurückgegeben.

Siehe auch
`POWER()`, `LN()`, `LOG10()`

8.2.17. `LOG10()`

Verfügbar in
DSQL, PSQL

Möglicher Namenskonflikt
YES → Details lesen

Ergebnistyp
`DOUBLE PRECISION`

Syntax

```
LOG10 (number)
```

Tabelle 129. `LOG10`-Funktionsparameter

Parameter	Beschreibung
number	Ein Ausdruck eines numerischen Typs

Gibt den 10-basierten Logarithmus des Arguments zurück.

- Ein Fehler wird ausgegeben, wenn das Argument negativ oder 0 ist. (In Versionen vor 2.5

würden solche Werte zu `NaN` bzw. `-INF` führen.)

Siehe auch

`POWER()`, `LN()`, `LOG()`

8.2.18. `MOD()`

Verfügbar in

DSQL, PSQL

Möglicher Namenskonflikt

YES → Details lesen

Ergebnistyp

`SMALLINT`, `INTEGER` oder `BIGINT` je nach Typ von *a*. Wenn *a* ein Gleitkommatyp ist, ist das Ergebnis ein `BIGINT`.

Syntax

```
MOD (a, b)
```

Tabelle 130. `MOD`-Funktionsparameter

Parameter	Beschreibung
a	Ein Ausdruck eines numerischen Typs
b	Ein Ausdruck eines numerischen Typs

Gibt den Rest einer ganzzahligen Division zurück.

- Nicht ganzzahlige Argumente werden vor der Division gerundet. “`mod(7.5, 2.5)`” ergibt also 2 (“`mod(8, 3)`”), nicht 0.

8.2.19. `PI()`

Verfügbar in

DSQL, PSQL

Möglicher Namenskonflikt

YES → Details lesen

Ergebnistyp
`DOUBLE PRECISION`

Syntax

```
PI ()
```

Gibt eine Annäherung an den Wert von *pi* zurück.

8.2.20. POWER()

Verfügbar in

DSQL, PSQL

Möglicher Namenskonflikt

YES → Details lesen

Ergebnistyp

`DOUBLE PRECISION`

Syntax

```
POWER (x, y)
```

Tabelle 131. POWER-Funktionsparameter

Parameter	Beschreibung
x	Ein Ausdruck eines numerischen Typs
y	Ein Ausdruck eines numerischen Typs

Gibt *x* hoch *y* (x^y) zurück.

Siehe auch

`EXP()`, `LOG()`, `LOG10()`, `SQRT()`

8.2.21. RAND()

Verfügbar in

DSQL, PSQL

Möglicher Namenskonflikt

YES → Details lesen

Ergebnistyp

`DOUBLE PRECISION`

Syntax

```
RAND ()
```

Gibt eine Zufallszahl zwischen 0 und 1 zurück.

8.2.22. ROUND()

Verfügbar in

DSQL, PSQL

Möglicher Namenskonflikt

YES → Details lesen

Ergebnistyp

`INTEGER`, (skaliert) `BIGINT` oder `DOUBLE PRECISION`

Syntax

```
ROUND (number [, scale])
```

Tabelle 132. `ROUND`-Funktionsparameter

Parameter	Description
number	Ein Ausdruck eines numerischen Typs
scale	Eine ganze Zahl, die die Anzahl der Nachkommastellen angibt, auf die gerundet werden soll, z. B.: • 2 zum Runden auf das nächste Vielfache von 0,01 • 1 zum Runden auf das nächste Vielfache von 0,1 • 0 zum Runden auf die nächste ganze Zahl • -1 zum Runden auf das nächste Vielfache von 10 • -2 zum Runden auf das nächste Vielfache von 100

Rundet eine Zahl auf die nächste ganze Zahl. Wenn der Bruchteil genau '0,5' ist, wird bei positiven Zahlen nach oben und bei negativen Zahlen nach unten gerundet. Mit dem optionalen Argument *scale* kann die Zahl auf Zehnerpotenzen (Zehner, Hunderter, Zehntel, Hundertstel usw.) statt auf ganze Zahlen gerundet werden.

Wenn Sie an das Verhalten der externen Funktion `ROUND` gewöhnt sind, beachten Sie bitte, dass die Funktion *internal* von Null immer auf Hälften rundet, d.h. bei negativen Zahlen nach unten.

`ROUND`-Beispiele

Wenn das Argument *scale* vorhanden ist, hat das Ergebnis normalerweise die gleiche Skalierung wie das erste Argument:

```
ROUND(123.654, 1) -- Ergebnis 123.700 (not 123.7)
ROUND(8341.7, -3) -- Ergebnis 8000.0 (not 8000)
ROUND(45.1212, 0) -- Ergebnis 45.0000 (not 45)
```

Andernfalls ist die Ergebnisskalierung 0:

```
ROUND(45.1212) -- Ergebnis 45
```

Siehe auch

`CEIL()`, `CEILING()`, `FLOOR()`, `TRUNC()`

8.2.23. `SIGN()`

Verfügbar in

DSQL, PSQL

Möglicher Namenskonflikt

YES → Details lesen

Ergebnistyp

`SMALLINT`

Syntax

```
SIGN (number)
```

Tabelle 133. `SIGN`-Funktionsparameter

Parameter	Beschreibung
number	Ein Ausdruck eines numerischen Typs

Gibt das Vorzeichen des Arguments zurück: -1, 0 oder 1.

8.2.24. `SIN()`

Verfügbar in

DSQL, PSQL

Möglicher Namenskonflikt

YES → Details lesen

Ergebnistyp

`DOUBLE PRECISION`

Syntax

```
SIN (angle)
```

Tabelle 134. `SIN`-Funktionsparameter

Parameter	Beschreibung
angle	Ein Winkel im Bogenmaß

Gibt den Sinus eines Winkels zurück. Das Argument muss im Bogenmaß angegeben werden.

- Jedes Ergebnis, das nicht `NULL` ist, liegt — offensichtlich — im Bereich [-1, 1].

Siehe auch

`ASIN()`, `COS()`, `COT()`, `TAN()`

8.2.25. SINH()

Verfügbar in

DSQL, PSQL

Möglicher Namenskonflikt

YES → Details lesen

Ergebnistyp

`DOUBLE PRECISION`

Syntax

```
SINH (number)
```

Tabelle 135. SINH-Funktionsparameter

Parameter	Beschreibung
number	Ein Ausdruck eines numerischen Typs

Gibt den hyperbolischen Sinus des Arguments zurück.

Siehe auch

`ASINH()`, `COSH()`, `TANH()`

8.2.26. SQRT()

Verfügbar in

DSQL, PSQL

Möglicher Namenskonflikt

YES → Details lesen

Ergebnistyp

`DOUBLE PRECISION`

Syntax

```
SQRT (number)
```

Tabelle 136. SQRT-Funktionsparameter

Parameter	Beschreibung
number	Ein Ausdruck eines numerischen Typs

Gibt die Quadratwurzel des Arguments zurück.

- Wenn *number* negativ ist, wird ein Fehler ausgegeben.

Siehe auch
`POWER()`

8.2.27. `TAN()`

Verfügbar in
DSQL, PSQL

Möglicher Namenskonflikt
YES → Details lesen

Ergebnistyp
`DOUBLE PRECISION`

Syntax

```
TAN (angle)
```

Tabelle 137. `TAN`-Funktionsparameter

Parameter	Beschreibung
angle	Ein Winkel im Bogenmaß

Gibt den Tangens eines Winkels zurück. Das Argument muss im Bogenmaß angegeben werden.

Siehe auch
`ATAN()`, `ATAN2()`, `COS()`, `COT()`, `SIN()`, `TAN()`

8.2.28. `TANH()`

Verfügbar in
DSQL, PSQL

Möglicher Namenskonflikt
YES → Details lesen

Ergebnistyp
`DOUBLE PRECISION`

Syntax

```
TANH (number)
```

Tabelle 138. `TANH`-Funktionsparameter

Parameter	Beschreibung
number	Ein Ausdruck eines numerischen Typs

Gibt den hyperbolischen Tangens des Arguments zurück.

- Aufgrund von Rundungen liegt jedes Ergebnis, das nicht `NULL` ist, im Bereich [-1, 1] (mathematisch ist es <-1, 1>).

Siehe auch

`ATANH()`, `COSH()`, `TANH()`

8.2.29. `TRUNC()`

Verfügbar in

DSQL, PSQL

Ergebnistyp

`INTEGER`, (scaled) `BIGINT` or `DOUBLE PRECISION`

Syntax

```
TRUNC (number [, scale])
```

Tabelle 139. `TRUNC`-Funktionsparameter

Parameter	Description
number	Ein Ausdruck eines numerischen Typs
scale	Eine ganze Zahl, die die Anzahl der Dezimalstellen angibt, auf die abgeschnitten werden soll, z. B.: • 2 zum Abschneiden auf das nächste Vielfache von 0,01 • 1 zum Abschneiden auf das nächste Vielfache von 0,1 • 0 zum Abschneiden auf die nächste ganze Zahl • -1 zum Abschneiden auf das nächste Vielfache von 10 • -2 zum Abschneiden auf das nächste Vielfache von 100

Gibt den ganzzahligen Teil einer Zahl zurück. Mit dem optionalen Argument *scale* kann die Zahl auf Zehnerpotenzen (Zehner, Hunderter, Zehntel, Hundertstel usw.) statt auf ganze Zahlen gekürzt werden.

- Wenn das Argument *scale* vorhanden ist, hat das Ergebnis normalerweise die gleiche Skala wie das erste Argument, z.
  - `TRUNC(789.2225, 2)` gibt 789.2200 (nicht 789.22) zurück
  - `TRUNC(345.4, -2)` gibt 300.0 (nicht 300) zurück
  - `TRUNC(-163.41, 0)` gibt -163.00 (nicht -163) zurück

- Andernfalls ist die Ergebnisskala 0:
  - `TRUNC(-163.41)` gibt -163 zurück

Wenn Sie an das Verhalten der externen Funktion `TRUNCATE` gewöhnt sind, beachten Sie bitte, dass die *interne* Funktion `TRUNC` immer gegen Null abschneidet, d.h. für negative Zahlen nach oben.

Siehe auch

`CEIL()`, `CEILING()`, `FLOOR()`, `ROUND()`

8.3. String-Funktionen

8.3.1. ASCII_CHAR()

Verfügbar in

DSQL, PSQL

Möglicher Namenskonflikt

YES → Details lesen

Ergebnistyp

`CHAR(1) CHARACTER SET NONE`

Syntax

```
ASCII_CHAR (code)
```

Tabelle 140. `ASCII_CHAR`-Funktionsparameter

Parameter	Beschreibung
code	Eine ganze Zahl im Bereich von 0 bis 255

Gibt das ASCII-Zeichen zurück, das der im Argument übergebenen Zahl entspricht.

- Wenn Sie das Verhalten des `ASCII_CHAR`-UDF gewohnt sind, das einen leeren String zurückgibt, wenn das Argument 0 ist, beachten Sie bitte, dass die interne Funktion hier korrekt ein Zeichen mit dem ASCII-Code 0 zurückgibt.

8.3.2. ASCII_VAL()

Verfügbar in

DSQL, PSQL

Möglicher Namenskonflikt

YES → Details lesen

Ergebnistyp

`SMALLINT`

Syntax

```
ASCII_VAL (ch)
```

Tabelle 141. `ASCII_VAL`-Funktionsparameter

Parameter	Beschreibung
ch	Ein String vom Datentyp `[VAR]CHAR` oder ein Text `BLOB` mit der maximalen Größe von 32.767 Bytes

Gibt den ASCII-Code des übergebenen Zeichens zurück.

- Wenn das Argument ein String mit mehr als einem Zeichen ist, wird der ASCII-Code des ersten Zeichens zurückgegeben.
- Wenn das Argument ein leerer String ist, wird 0 zurückgegeben.
- Wenn das Argument `NULL` ist, wird `NULL` zurückgegeben.
- Wenn das erste Zeichen der Argumentzeichenfolge aus mehreren Byte besteht, wird ein Fehler ausgegeben. (Ein Fehler in Firebird 2.1 - 2.1.3 und 2.5.0 führt zu einem Fehler, wenn *beliebiges* Zeichen in der Zeichenfolge aus mehreren Byte besteht. Dies ist in den Versionen 2.1.4 und 2.5.1 behoben.)

8.3.3. `BIT_LENGTH()`

Verfügbar in

DSQL, PSQL

Ergebnistyp

`INTEGER`

Syntax

```
BIT_LENGTH (string)
```

Tabelle 142. `BIT_LENGTH`-Funktionsparameter

Parameter	Beschreibung
string	Ein Ausdruck eines String-Typs

Gibt die Länge des Eingabestrings in Bits an. Bei Mehrbyte-Zeichensätzen kann dies kleiner sein als die Anzahl der Zeichen mal 8 mal die "formale" Anzahl von Bytes pro Zeichen wie in `RDB$CHARACTER_SETS` gefunden.

Bei Argumenten vom Typ `CHAR` berücksichtigt diese Funktion die gesamte formale Stringlänge (d.h. die deklarierte Länge eines Feldes oder einer Variablen). Wenn Sie die "logische" Bitlänge erhalten möchten, ohne die abschließenden

Leerzeichen zu zählen, rechts-TRIM das Argument vor der Übergabe an BIT_LENGTH.

BLOB-*Unterstützung*

Seit Firebird 2.1 unterstützt diese Funktion vollständig Text-BLOBs jeder Länge und jedes beliebigen Zeichensatzes.

BIT_LENGTH-Beispiele

```
select bit_length('Hello!') from rdb$database
-- Ergebnis 48

select bit_length(_iso8859_1 'Grüß di!') from rdb$database
-- Ergebnis 64: ü und ß belegen in ISO8859_1 jeweils ein Byte

select bit_length
  (cast (_iso8859_1 'Grüß di!' as varchar(24) character set utf8))
from rdb$database
-- Ergebnis 80: ü und ß belegen in ISO8859_1 jeweils ein Byte

select bit_length
  (cast (_iso8859_1 'Grüß di!' as char(24) character set utf8))
from rdb$database
-- Ergebnis 208: alle 24 CHAR-Positionen zählen, und zwei davon sind 16-Bit
```

Siehe auch

OCTET_LENGTH(), CHAR_LENGTH(), CHARACTER_LENGTH()

8.3.4. CHAR_LENGTH(), CHARACTER_LENGTH()

Verfügbar in

DSQL, PSQL

Ergebnistyp

INTEGER

Syntax

```
  CHAR_LENGTH (string)
| CHARACTER_LENGTH (string)
```

Tabelle 143. CHAR[ACTER]_LENGTH-*Funktionsparameter*

Parameter	Beschreibung
string	Ein Ausdruck eines String-Typs

Gibt die Länge des Eingabestrings in Zeichen an.

- Bei Argumenten vom Typ CHAR liefert diese Funktion die formale Stringlänge

(d.h. die deklarierte Länge eines Feldes oder einer Variablen). Wenn Sie die "logische" Länge erhalten möchten, ohne die abschließenden Leerzeichen zu zählen, rechts-`TRIM` das Argument vor der Übergabe an `CHAR[ACTER]_LENGTH`.

- **`BLOB`-Unterstützung**: Seit Firebird 2.1 unterstützt diese Funktion vollständig Text-`BLOB`s jeder Länge und jedes beliebigen Zeichensatzes.

`CHAR_LENGTH`-Beispiele

```
select char_length('Hello!') from rdb$database
-- Ergebnis 6

select char_length(_iso8859_1 'Grüß di!') from rdb$database
-- Ergebnis 8

select char_length
  (cast (_iso8859_1 'Grüß di!' as varchar(24) character set utf8))
from rdb$database
-- Ergebnis 8; dass ü und ß jeweils zwei Bytes belegen ist irrelevant

select char_length
  (cast (_iso8859_1 'Grüß di!' as char(24) character set utf8))
from rdb$database
-- Ergebnis 24: alle 24 CHAR-Positionen zählen
```

Siehe auch

`BIT_LENGTH()`, `OCTET_LENGTH()`

8.3.5. `HASH()`

Verfügbar in

DSQL, PSQL

Ergebnistyp

`BIGINT`

Syntax

```
HASH (string)
```

Tabelle 144. `HASH`-Funktionsparameter

Parameter	Beschreibung
string	Ein Ausdruck eines String-Typs

Gibt einen Hashwert für die Eingabezeichenfolge zurück. Diese Funktion unterstützt vollständig Text-`BLOB`s jeder Länge und jedes beliebigen Zeichensatzes.

8.3.6. LEFT()

Verfügbar in

DSQL, PSQL

Ergebnistyp

VARCHAR oder BLOB

Syntax

```
LEFT (string, length)
```

Tabelle 145. LEFT-Funktionsparameter

Parameter	Beschreibung
string	Ein Ausdruck eines String-Typs
length	Ganzzahliger Ausdruck. Definiert die Anzahl der zurückzugebenden Zeichen

Gibt den äußersten linken Teil der Argumentzeichenfolge zurück. Die Anzahl der Zeichen wird im zweiten Argument angegeben.

- Diese Funktion unterstützt vollständig Text-BLOBs jeder Länge, einschließlich solcher mit einem Multi-Byte-Zeichensatz.
- Wenn *string* ein BLOB ist, ist das Ergebnis ein BLOB. Andernfalls ist das Ergebnis ein VARCHAR(n) mit *n* der Länge des Eingabestrings.
- Wenn das Argument *length* die Stringlänge überschreitet, wird der Eingabestring unverändert zurückgegeben.
- Wenn das Argument *length* keine ganze Zahl ist, wird Banker-Rundung (auf gerade) angewendet, d. h. 0,5 wird zu 0, 1,5 wird zu 2, 2,5 wird zu 2, 3,5 wird zu 4 usw.

Siehe auch

RIGHT()

8.3.7. LOWER()

Verfügbar in

DSQL, ESQL, PSQL

Möglicher Namenskonflikt

YES → Lesen Sie die Details unten

Ergebnistyp

(VAR)CHAR or BLOB

Syntax

```
LOWER (string)
```

Tabelle 146. LOWER-FunktionsparameterS

Parameter	Beschreibung
string	Ein Ausdruck eines String-Typs

Gibt das Äquivalent der Eingabezeichenfolge in Kleinbuchstaben zurück. Das genaue Ergebnis hängt vom Zeichensatz ab. Bei ASCII oder NONE beispielsweise werden nur ASCII-Zeichen kleingeschrieben; mit OCTETS wird der gesamte String unverändert zurückgegeben. Seit Firebird 2.1 unterstützt diese Funktion auch Text-BLOBs beliebiger Länge und beliebigem Zeichensatz.

Namenskonflikt

Da LOWER ein reserviertes Wort ist, hat die interne Funktion Vorrang, auch wenn die externe Funktion mit diesem Namen ebenfalls deklariert wurde. Um die (minderwertige!) externe Funktion aufzurufen, verwenden Sie doppelte Anführungszeichen und die genaue Großschreibung, wie in "LOWER"(string).

LOWER-Beispiele

```
select Sheriff from Towns
  where lower(Name) = 'cooper''s valley'
```

Siehe auch

UPPER()

8.3.8. LPAD()

Verfügbar in

DSQL, PSQL

Möglicher Namenskonflikt

YES → Details lesen

Ergebnistyp

VARCHAR oder BLOB

Syntax

```
LPAD (str, endlen [, padstr])
```

Tabelle 147. LPAD-Funktionsparameter

Parameter	Beschreibung
str	Ein Ausdruck eines String-Typs
endlen	Länge der Ausgabezeichenfolge
padstr	Das Zeichen oder die Zeichenfolge, die verwendet werden soll, um die Quellzeichenfolge bis zur angegebenen Länge aufzufüllen. Standard ist Leerzeichen (“' '”)

Füllt eine Zeichenfolge mit der linken Maustaste mit Leerzeichen oder mit einer vom Benutzer angegebenen Zeichenfolge auf, bis eine bestimmte Länge erreicht ist.

- Diese Funktion unterstützt vollständig Text `BLOB`s jeder Länge und jedes beliebigen Zeichensatzes.
- Wenn *str* ein `BLOB` ist, ist das Ergebnis ein `BLOB`. Andernfalls ist das Ergebnis ein `VARCHAR(endlen)`.
- Wenn *padstr* angegeben ist und gleich `''` (leerer String) ist, findet kein Auffüllen statt.
- Wenn *endlen* kleiner als die aktuelle Stringlänge ist, wird der String auf *endlen* gekürzt, auch wenn *padstr* der leere String ist.

In Firebird 2.1-2.1.3 waren alle Nicht-`BLOB`-Ergebnisse vom Typ `VARCHAR(32765)`, was es ratsam machte, sie auf eine bescheidenere Größe umzuwandeln. Dies ist nicht mehr der Fall.

Bei Verwendung auf einem 'BLOB' muss diese Funktion möglicherweise das gesamte Objekt in den Speicher laden. Obwohl es versucht, den Speicherverbrauch zu begrenzen, kann dies die Leistung beeinträchtigen, wenn es um große `BLOB`s geht.

`LPAD`-Beispiele

```
lpad ('Hello', 12)              -- Ergebnis '       Hello'
lpad ('Hello', 12, '-')         -- Ergebnis '-------Hello'
lpad ('Hello', 12, '')          -- Ergebnis 'Hello'
lpad ('Hello', 12, 'abc')       -- Ergebnis 'abcabcaHello'
lpad ('Hello', 12, 'abcdefghij') -- Ergebnis 'abcdefgHello'
lpad ('Hello', 2)               -- Ergebnis 'He'
lpad ('Hello', 2, '-')          -- Ergebnis 'He'
lpad ('Hello', 2, '')           -- Ergebnis 'He'
```

Siehe auch

`RPAD()`

8.3.9. `OCTET_LENGTH()`

Verfügbar in

DSQL, PSQL

Ergebnistyp

INTEGER

Syntax

```
OCTET_LENGTH (string)
```

Tabelle 148. OCTET_LENGTH-*Funktionsparameter*

Parameter	Beschreibung
string	Ein Ausdruck eines String-Typs

Gibt die Länge des Eingabestrings in Bytes (Oktetts) an. Bei Mehrbyte-Zeichensätzen kann dies kleiner sein als die Anzahl der Zeichen mal der "formalen" Anzahl von Bytes pro Zeichen, wie in RDB$CHARACTER_SETS gefunden.

Bei Argumenten vom Typ CHAR berücksichtigt diese Funktion die gesamte formale Stringlänge (d.h. die deklarierte Länge eines Feldes oder einer Variablen). Wenn Sie die "logische" Bytelänge erhalten möchten, ohne die abschließenden Leerzeichen zu zählen, rechts-TRIM das Argument vor der Übergabe an OCTET_LENGTH.

BLOB-*Unterstützung*

Seit Firebird 2.1 unterstützt diese Funktion vollständig Text-BLOBs jeder Länge und jedes beliebigen Zeichensatzes.

OCTET_LENGTH-Beispiele

```
select octet_length('Hello!') from rdb$database
-- Ergebnis 6

select octet_length(_iso8859_1 'Grüß di!') from rdb$database
-- Ergebnis 8: ü und ß belegen in ISO8859_1 jeweils ein Byte

select octet_length
  (cast (_iso8859_1 'Grüß di!' as varchar(24) character set utf8))
from rdb$database
-- Ergebnis 10: ü und ß belegen in UTF8 jeweils zwei Byte

select octet_length
  (cast (_iso8859_1 'Grüß di!' as char(24) character set utf8))
from rdb$database
-- Ergebnis 26: alle 24 CHAR-Positionen zählen, und zwei davon sind 2-Byte
```

Siehe auch

BIT_LENGTH(), CHAR_LENGTH(), CHARACTER_LENGTH()

8.3.10. OVERLAY()

Verfügbar in

DSQL, PSQL

Ergebnistyp

VARCHAR oder BLOB

Syntax

```
OVERLAY (string PLACING replacement FROM pos [FOR length])
```

Tabelle 149. OVERLAY-Funktionsparameter

Parameter	Beschreibung
string	Die Zeichenfolge, in die die Ersetzung erfolgt
replacement	Ersetzende Zeichenkette
pos	Die Position, von der aus ersetzt wird (Ausgangsposition)
length	Die Anzahl der zu überschreibenden Zeichen

OVERLAY() überschreibt einen Teil eines Strings mit einem anderen String. Standardmäßig entspricht die Anzahl der aus der Hostzeichenfolge entfernten (überschriebenen) Zeichen der Länge der Ersetzungszeichenfolge. Mit dem optionalen vierten Argument kann eine andere Anzahl von Zeichen zum Entfernen angegeben werden.

- Diese Funktion unterstützt BLOBs beliebiger Länge.
- Wenn *string* oder *replacement* ein BLOB ist, ist das Ergebnis ein BLOB. Andernfalls ist das Ergebnis ein VARCHAR(n) mit *n* der Summe der Längen von *string* und *replacement*.
- Wie bei SQL-Stringfunktionen üblich, ist *pos* 1-basiert.
- Wenn *pos* hinter dem Ende von *string* steht, wird *replacement* direkt nach *string* platziert.
- Wenn die Anzahl der Zeichen von *pos* bis zum Ende von *string* kleiner ist als die Länge von *replacement* (oder als das *length*-Argument, falls vorhanden), wird *string* an *pos* abgeschnitten und *replacement* dahinter platziert.
- Eine "FOR 0"-Klausel bewirkt, dass *replacement* einfach in *string* eingefügt wird.
- Wenn ein Argument NULL ist, ist das Ergebnis NULL.
- Wenn *pos* oder *length* keine ganze Zahl ist, wird Banker-Rundung (auf-gerade) angewendet, d. h. 0,5 wird zu 0, 1,5 wird zu 2, 2,5 wird zu 2, 3,5 wird zu 4 usw.

Bei Verwendung auf einem 'BLOB' muss diese Funktion möglicherweise das gesamte Objekt in den Speicher laden. Dies kann die Leistung beeinträchtigen, wenn es um große BLOBs geht.

OVERLAY-Beispiele

```
overlay ('Goodbye' placing 'Hello' from 2)   -- Ergebnis 'GHelloe'
overlay ('Goodbye' placing 'Hello' from 5)   -- Ergebnis 'GoodHello'
overlay ('Goodbye' placing 'Hello' from 8)   -- Ergebnis 'GoodbyeHello'
overlay ('Goodbye' placing 'Hello' from 20)  -- Ergebnis 'GoodbyeHello'

overlay ('Goodbye' placing 'Hello' from 2 for 0) -- Ergebnis 'GHellooodbye'
overlay ('Goodbye' placing 'Hello' from 2 for 3) -- Ergebnis 'GHellobye'
overlay ('Goodbye' placing 'Hello' from 2 for 6) -- Ergebnis 'GHello'
overlay ('Goodbye' placing 'Hello' from 2 for 9) -- Ergebnis 'GHello'

overlay ('Goodbye' placing '' from 4)        -- Ergebnis 'Goodbye'
overlay ('Goodbye' placing '' from 4 for 3)  -- Ergebnis 'Gooe'
overlay ('Goodbye' placing '' from 4 for 20) -- Ergebnis 'Goo'

overlay ('' placing 'Hello' from 4)          -- Ergebnis 'Hello'
overlay ('' placing 'Hello' from 4 for 0)    -- Ergebnis 'Hello'
overlay ('' placing 'Hello' from 4 for 20)   -- Ergebnis 'Hello'
```

Siehe auch

REPLACE()

8.3.11. POSITION()

Verfügbar in

DSQL, PSQL

Ergebnistyp

INTEGER

Syntax

```
  POSITION (substr IN string)
| POSITION (substr, string [, startpos])
```

Tabelle 150. POSITION-*Funktionsparameter*

Parameter	Beschreibung
substr	Der Teilstring, dessen Position gesucht werden soll
string	Der zu suchende String
startpos	Die Position in *string*, an der die Suche beginnen soll

Gibt die (1-basierte) Position des ersten Vorkommens einer Teilzeichenfolge in einer Hostzeichenfolge zurück. Mit dem optionalen dritten Argument beginnt die Suche an einem bestimmten Offset, wobei alle Übereinstimmungen ignoriert werden, die früher in der Zeichenfolge auftreten können. Wenn keine Übereinstimmung gefunden wird, ist das Ergebnis 0.

- Das optionale dritte Argument wird nur in der zweiten Syntax (Komma-Syntax) unterstützt.
- Die leere Zeichenfolge wird als Teilzeichenfolge jeder Zeichenfolge betrachtet. Wenn also *substr* `''` (leerer String) ist und *string* nicht `NULL` ist, ist das Ergebnis:
  - 1 wenn *startpos* nicht angegeben ist;
  - *startpos* wenn *startpos* innerhalb von *string* liegt;
  - 0, wenn *startpos* hinter dem Ende von *string* liegt.

  Hinweis: Ein Fehler in Firebird 2.1 - 2.1.3 und 2.5.0 führt dazu, dass `POSITION` *immer* 1 zurückgibt, wenn *substr* der leere String ist. Dies ist in 2.1.4 und 2.5.1 behoben.
- Diese Funktion unterstützt vollständig Text-`BLOB`s jeder Größe und jedes Zeichensatzes.

Bei Verwendung auf einem 'BLOB' muss diese Funktion möglicherweise das gesamte Objekt in den Speicher laden. Dies kann die Leistung beeinträchtigen, wenn es um große `BLOB`s geht.

`POSITION`-Beispiele

```
position ('be' in 'To be or not to be')   -- Ergebnis 4
position ('be', 'To be or not to be')     -- Ergebnis 4
position ('be', 'To be or not to be', 4)  -- Ergebnis 4
position ('be', 'To be or not to be', 8)  -- Ergebnis 17
position ('be', 'To be or not to be', 18) -- Ergebnis 0
position ('be' in 'Alas, poor Yorick!')   -- Ergebnis 0
```

Siehe auch

`SUBSTRING()`

8.3.12. `REPLACE()`

Verfügbar in

DSQL, PSQL

Ergebnistyp

`VARCHAR` oder `BLOB`

Syntax

```
REPLACE (str, find, repl)
```

Tabelle 151. `REPLACE`*-Funktionsparameter*

Parameter	Beschreibung
str	Die Zeichenfolge, in der die Ersetzung erfolgen soll
find	Die Zeichenfolge, nach der gesucht werden soll
repl	Die Ersatzzeichenfolge

Ersetzt alle Vorkommen einer Teilzeichenfolge in einer Zeichenfolge.

- Diese Funktion unterstützt vollständig Text `BLOB`s jeder Länge und jedes beliebigen Zeichensatzes.
- Wenn ein Argument ein `BLOB` ist, ist das Ergebnis ein `BLOB`. Andernfalls ist das Ergebnis ein `VARCHAR(n)` mit *n*, das aus den Längen von *str*, *find* und *repl* so berechnet wird, dass auch die maximal mögliche Anzahl von Ersetzungen das Feld nicht überläuft.
- Wenn *find* der leere String ist, wird *str* unverändert zurückgegeben.
- Wenn *repl* der leere String ist, werden alle Vorkommen von *find* aus *str* gelöscht.
- Wenn ein Argument `NULL` ist, ist das Ergebnis immer `NULL`, auch wenn nichts ersetzt worden wäre.

Bei Verwendung auf einem 'BLOB' muss diese Funktion möglicherweise das gesamte Objekt in den Speicher laden. Dies kann die Leistung beeinträchtigen, wenn es um große `BLOB`s geht.

`REPLACE`-Beispiele

```
replace ('Billy Wilder',  'il', 'oog') -- Ergebnis 'Boogly Woogder'
replace ('Billy Wilder',  'il',    '') -- Ergebnis 'Bly Wder'
replace ('Billy Wilder',  null, 'oog') -- Ergebnis NULL
replace ('Billy Wilder',  'il',  null) -- Ergebnis NULL
replace ('Billy Wilder', 'xyz',  null) -- Ergebnis NULL (!)
replace ('Billy Wilder', 'xyz', 'abc') -- Ergebnis 'Billy Wilder'
replace ('Billy Wilder',    '', 'abc') -- Ergebnis 'Billy Wilder'
```

Siehe auch

`OVERLAY()`, `SUBSTRING()`, `POSITION()`, `CHAR_LENGTH()`, `CHARACTER_LENGTH()`

8.3.13. `REVERSE()`

Verfügbar in

DSQL, PSQL

Ergebnistyp

`VARCHAR`

Syntax

```
REVERSE (string)
```

Tabelle 152. REVERSE-Funktionsparameter

Parameter	Beschreibung
string	Ein Ausdruck eines String-Typs

Gibt eine Zeichenfolge rückwärts zurück.

REVERSE-Beispiele

```
reverse ('spoonful')            -- Ergebnis 'lufnoops'
reverse ('Was it a cat I saw?') -- Ergebnis '?was I tac a ti saW'
```

Diese Funktion ist sehr praktisch, wenn Sie nach String-Endungen gruppieren, suchen oder sortieren möchten, z.B. beim Umgang mit Domainnamen oder E-Mail-Adressen:

```
create index ix_people_email on people
  computed by (reverse(email));

select * from people
  where reverse(email) starting with reverse('.br');
```

8.3.14. RIGHT()

Verfügbar in

DSQL, PSQL

Möglicher Namenskonflikt

YES → Details lesen

Ergebnistyp

VARCHAR oder BLOB

Syntax

```
RIGHT (string, length)
```

Tabelle 153. RIGHT-Funktionsparameter

Parameter	Beschreibung
string	Ein Ausdruck eines String-Typs

Parameter	Beschreibung
length	Integer. Definiert die Anzahl der zurückzugebenden Zeichen

Gibt den ganz rechten Teil der Argumentzeichenfolge zurück. Die Anzahl der Zeichen wird im zweiten Argument angegeben.

- Diese Funktion unterstützt Text BLOB`s' beliebiger Länge, hat aber einen Fehler in den Versionen 2.1 - 2.1.3 und 2.5.0, der dazu führt, dass es bei Text BLOB``s fehlschlägt, die größer als 1024 Bytes sind, die ein Multi haben -Byte-Zeichensatz. Dies wurde in den Versionen 2.1.4 und 2.5.1 behoben.
- Wenn *string* ein BLOB ist, ist das Ergebnis ein BLOB. Andernfalls ist das Ergebnis ein VARCHAR(n) mit *n* der Länge des Eingabestrings.
- Wenn das Argument *length* die Stringlänge überschreitet, wird der Eingabestring unverändert zurückgegeben.
- Wenn das Argument *Länge* keine ganze Zahl ist, wird Banker-Rundung (auf-gerade) angewendet, d. h. 0,5 wird zu 0, 1,5 wird zu 2, 2,5 wird zu 2, 3,5 wird zu 4 usw.

Bei Verwendung auf einem 'BLOB' muss diese Funktion möglicherweise das gesamte Objekt in den Speicher laden. Dies kann die Leistung beeinträchtigen, wenn es um große BLOBs geht.

Siehe auch

LEFT(), SUBSTRING()

8.3.15. RPAD()

Verfügbar in

DSQL, PSQL

Möglicher Namenskonflikt

YES → Details lesen

Ergebnistyp

VARCHAR oder BLOB

Syntax

```
RPAD (str, endlen [, padstr])
```

Tabelle 154. RPAD-Funktionsparameter

Parameter	Beschreibung
str	Ein Ausdruck eines String-Typs
endlen	Länge der Ausgabezeichenfolge

Parameter	Beschreibung
endlen	Das Zeichen oder die Zeichenfolge, die verwendet werden soll, um die Quellzeichenfolge bis zur angegebenen Länge aufzufüllen. Standard ist Leerzeichen (' ')

Füllt eine Zeichenfolge mit der rechten Maustaste mit Leerzeichen oder mit einer vom Benutzer angegebenen Zeichenfolge auf, bis eine bestimmte Länge erreicht ist.

- Diese Funktion unterstützt vollständig Text `BLOB`s jeder Länge und jedes beliebigen Zeichensatzes.
- Wenn *str* ein `BLOB` ist, ist das Ergebnis ein `BLOB`. Andernfalls ist das Ergebnis ein `VARCHAR(endlen)`.
- Wenn *padstr* angegeben ist und gleich `''` (leerer String) ist, findet kein Auffüllen statt.
- Wenn *endlen* kleiner als die aktuelle Stringlänge ist, wird der String auf *endlen* gekürzt, auch wenn *padstr* der leere String ist.

In Firebird 2.1-2.1.3 waren alle Nicht-`BLOB`-Ergebnisse vom Typ `VARCHAR(32765)`, was es ratsam machte, sie auf eine bescheidenere Größe umzuwandeln. Dies ist nicht mehr der Fall.

Bei Verwendung auf einem 'BLOB' muss diese Funktion möglicherweise das gesamte Objekt in den Speicher laden. Obwohl es versucht, den Speicherverbrauch zu begrenzen, kann dies die Leistung beeinträchtigen, wenn es um große `BLOB`s geht.

RPAD-Beispiele

```
rpad ('Hello', 12)              -- Ergebnis 'Hello       '
rpad ('Hello', 12, '-')         -- Ergebnis 'Hello-------'
rpad ('Hello', 12, '')          -- Ergebnis 'Hello'
rpad ('Hello', 12, 'abc')       -- Ergebnis 'Helloabcabca'
rpad ('Hello', 12, 'abcdefghij') -- Ergebnis 'Helloabcdefg'
rpad ('Hello', 2)               -- Ergebnis 'He'
rpad ('Hello', 2, '-')          -- Ergebnis 'He'
rpad ('Hello', 2, '')           -- Ergebnis 'He'
```

Siehe auch

`LPAD()`

8.3.16. SUBSTRING()

Verfügbar in

DSQL, PSQL

Ergebnistyps

`VARCHAR` oder `BLOB`

Syntax

```
SUBSTRING ( <substring-args> )

<substring-args> ::=
    str FROM startpos [FOR length]
  | str SIMILAR <similar-pattern> ESCAPE <escape>

<similar-pattern> ::=
  <similar-pattern-R1>
  <escape> " <similar-pattern-R2> <escape> "
  <similar-pattern-R3>
```

Tabelle 155. SUBSTRING-Funktionsparameter

Parameter	Beschreibung
str	Ein Ausdruck eines String-Typs
startpos	Ganzzahliger Ausdruck, die Position, von der aus mit dem Abrufen der Teilzeichenfolge begonnen werden soll
length	Die Anzahl der abzurufenden Zeichen nach dem *startpos*
similar-pattern	Muster für reguläre SQL-Ausdrücke, um nach der Teilzeichenfolge zu suchen
escape	Escape-Zeichen

Gibt die Teilzeichenfolge einer Zeichenfolge beginnend an der angegebenen Position zurück, entweder bis zum Ende der Zeichenfolge oder mit einer bestimmten Länge, oder extrahiert eine Teilzeichenfolge mithilfe eines Musters für reguläre SQL-Ausdrücke.

Wenn ein Argument NULL ist, ist das Ergebnis auch NULL.

Bei Verwendung auf einem BLOB muss diese Funktion möglicherweise das gesamte Objekt in den Speicher laden. Obwohl es versucht, den Speicherverbrauch zu begrenzen, kann dies die Leistung beeinträchtigen, wenn es um große BLOBs geht.

Positionsbezogener SUBSTRING

In ihrer einfachen Positionsform (mit FROM) gibt diese Funktion den Teilstring ab der Zeichenposition *startpos* zurück (die erste Position ist 1). Ohne das Argument FOR gibt es alle verbleibenden Zeichen in der Zeichenfolge zurück. Bei FOR gibt es *length* Zeichen oder den Rest des Strings zurück, je nachdem welcher kürzer ist.

Die Funktion unterstützt vollständig binäre und Text BLOBs beliebiger Länge und mit jedem Zeichensatz. Wenn *str* ein BLOB ist, ist das Ergebnis auch ein BLOB. Bei jedem anderen Argumenttyp ist das Ergebnis ein VARCHAR.

Bei Nicht-BLOB-Argumenten entspricht die Breite des Ergebnisfelds immer der Länge von *str*, unabhängig von *startpos* und *length*. `substring('pinhead' from 4 for 2)` gibt also ein VARCHAR(7)

zurück, das den String `'he'` enthält.

Beispiele

```
insert into AbbrNames(AbbrName)
  select substring(LongName from 1 for 3) from LongNames
```

Regulärer Ausdruck `SUBSTRING`

In der Form des regulären Ausdrucks (mit `SIMILAR`) gibt die Funktion `SUBSTRING` einen Teil des Strings zurück, der einem Muster eines regulären SQL-Ausdrucks entspricht. Wenn keine Übereinstimmung gefunden wird, wird `NULL` zurückgegeben.

Das Muster "SIMILAR" wird aus drei Mustern für reguläre SQL-Ausdrücke gebildet, *R1*, *R2* und *R3*. Das gesamte Muster hat die Form `R1 || '<Escape>"' || R2 || '<Escape>"' || R3`, wobei *<escape>* das in der `ESCAPE`-Klausel definierte Escape-Zeichen ist. *R2* ist das Muster, das mit der zu extrahierenden Teilzeichenfolge übereinstimmt, und wird zwischen doppelten Anführungszeichen mit Escapezeichen eingeschlossen (`<escape>"`, zB “#"” mit Escape-Zeichen ‘ ##+’). *R1* entspricht dem Präfix des Strings und *R3* dem Suffix des Strings. Sowohl *R1* als auch *R3* sind optional (sie können leer sein), aber das Muster muss mit der gesamten Zeichenfolge übereinstimmen. Mit anderen Worten, es reicht nicht aus, ein Muster anzugeben, das nur die zu extrahierende Teilzeichenfolge findet.

Die mit Escapezeichen versehenen doppelten Anführungszeichen um *R2* können mit der Definition einer einzelnen Erfassungsgruppe in einer gängigeren Syntax für reguläre Ausdrücke wie PCRE verglichen werden. Das heißt, das vollständige Muster entspricht `R1(R2)R3`, das mit der gesamten Eingabezeichenfolge übereinstimmen muss, und die Erfassungsgruppe ist die zurückzugebende Teilzeichenfolge.

Wenn einer von *R1*, *R2* oder *R3* keine Zeichenfolge der Länge Null ist und nicht das Format eines regulären SQL-Ausdrucks hat, wird eine Ausnahme ausgelöst.

Das vollständige Format für reguläre SQL-Ausdrücke wird in Syntax: Reguläre SQL-Ausdrücke beschrieben

Beispiele

```
substring('abcabc' similar 'a#"bcab#"c' escape '#')  -- bcab
substring('abcabc' similar 'a#"%#"c' escape '#')     -- bcab
substring('abcabc' similar '_#"%#"_' escape '#')     -- bcab
substring('abcabc' similar '#"(abc)*#"' escape '#')  -- abcabc
substring('abcabc' similar '#"abc#"' escape '#')     -- <null>
```

Siehe auch

`POSITION()`, `LEFT()`, `RIGHT()`, `CHAR_LENGTH()`, `CHARACTER_LENGTH()`, `SIMILAR TO`

8.3.17. TRIM()

Verfügbar in

DSQL, PSQL

Ergebnistyp

VARCHAR or BLOB

Syntax

```
TRIM ([<adjust>] str)

<adjust> ::=  {[<where>] [what]} FROM

<where> ::=  BOTH | LEADING | TRAILING
```

Tabelle 156. TRIM-Funktionsparameter

Parameter	Beschreibung
str	Ein Ausdruck eines String-Typs
where	Die Position, aus der der Teilstring entfernt werden soll — BOTH | LEADING | TRAILING. BOTH ist die Standardeinstellung
what	Die Teilzeichenfolge, die am Anfang, am Ende oder auf beiden Seiten der Eingabezeichenfolge *str* entfernt werden soll (mehrmals bei mehreren Übereinstimmungen). Standardmäßig ist es Leerzeichen (' ')

Entfernt führende und/oder nachgestellte Leerzeichen (oder optional andere Zeichenfolgen) aus der Eingabezeichenfolge. Seit Firebird 2.1 unterstützt diese Funktion vollständig Text-BLOBs jeder Länge und jedes beliebigen Zeichensatzes.

- Wenn *str* ein BLOB ist, ist das Ergebnis ein BLOB. Andernfalls ist es ein VARCHAR(n) mit *n* der formalen Länge von *str*.
- Seit Firebird 3.0 wurde die maximale Größe von *what* — wenn ein `BLOB — auf 4 GB erhöht wurde, in früheren Versionen konnte der Wert von *what* 32.767 Byte nicht überschreiten.

Bei Verwendung auf einem 'BLOB' muss diese Funktion möglicherweise das gesamte Objekt in den Speicher laden. Dies kann die Leistung beeinträchtigen, wenn es um große BLOBs geht.

TRIM-Beispiele

```
select trim ('  Waste no space   ') from rdb$database
-- Ergebnis 'Waste no space'

select trim (leading from '  Waste no space   ') from rdb$database
```

```
-- Ergebnis 'Waste no space   '

select trim (leading '.' from '  Waste no space   ') from rdb$database
-- Ergebnis '  Waste no space   '

select trim (trailing '!' from 'Help!!!!') from rdb$database
-- Ergebnis 'Help'

select trim ('la' from 'lalala I love you Ella') from rdb$database
-- Ergebnis ' I love you El'

select trim ('la' from 'Lalala I love you Ella') from rdb$database
-- Ergebnis 'Lalala I love you El'
```

8.3.18. UPPER()

Verfügbar in

DSQL, ESQL, PSQL

Ergebnistyp

`(VAR)CHAR` oder `BLOB`

Syntax

```
UPPER (str)
```

Tabelle 157. `UPPER`-Funktionsparameter

Parameter	Beschreibung
str	Ein Ausdruck eines String-Typs

Gibt das Äquivalent der Eingabezeichenfolge in Großbuchstaben zurück. Das genaue Ergebnis hängt vom Zeichensatz ab. Bei `ASCII` oder `NONE` beispielsweise werden nur ASCII-Zeichen groß geschrieben; mit `OCTETS` wird der gesamte String unverändert zurückgegeben. Seit Firebird 2.1 unterstützt diese Funktion auch Text-`BLOB`s beliebiger Länge und beliebigem Zeichensatz.

`UPPER`-Beispiele

```
select upper(_iso8859_1 'Débâcle')
from rdb$database
-- Ergebnis 'DÉBÂCLE' (vor Firebird 2.0: 'DéBâCLE')

select upper(_iso8859_1 'Débâcle' collate fr_fr)
from rdb$database
-- Ergebnis 'DEBACLE', nach französischen Großbuchstabenregeln
```

Siehe auch

`LOWER()`

8.4. Datums- und Zeitfunktionen

8.4.1. DATEADD()

Verfügbar in

DSQL, PSQL

Ergebnistyp

DATE, TIME oder TIMESTAMP

Syntax

```
DATEADD (<args>)

<args> ::=
    <amount> <unit> TO <datetime>
  | <unit>, <amount>, <datetime>

<amount> ::= ein ganzzahliger Ausdruck (negativ zum Subtrahieren)
<unit> ::=
    YEAR | MONTH | WEEK | DAY
  | HOUR | MINUTE | SECOND | MILLISECOND
<datetime> ::= ein DATE-, TIME- oder TIMESTAMP-Ausdruck
```

Tabelle 158. DATEADD-*Funktionsparameter*

Parameter	Beschreibung
amount	Ein ganzzahliger Ausdruck vom Typ SMALLINT, INTEGER oder BIGINT. Für die Einheit MILLISECOND ist der Typ NUMERIC(18, 1). Ein negativer Wert wird abgezogen.
unit	Datum/Uhrzeit-Einheit
datetime	Ein Ausdruck vom Typ DATE, TIME oder TIMESTAMP

Addiert die angegebene Anzahl von Jahren, Monaten, Wochen, Tagen, Stunden, Minuten, Sekunden oder Millisekunden zu einem Datums-/Uhrzeitwert.

- Der Ergebnistyp wird durch das dritte Argument bestimmt.
- Mit den Argumenten TIMESTAMP und DATE können alle Einheiten verwendet werden.
- Bei TIME-Argumenten können nur HOUR, MINUTE, SECOND und MILLISECOND verwendet werden.

Beispiele of DATEADD

```
dateadd (28 day to current_date)
dateadd (-6 hour to current_time)
dateadd (month, 9, DateOfConception)
dateadd (-38 week to DateOfBirth)
dateadd (minute, 90, time 'now')
```

```
dateadd (? year to date '11-Sep-1973')
```

```
select
  cast(dateadd(-1 * extract(millisecond from ts) millisecond to ts) as varchar(30)) as
t,
  extract(millisecond from ts) as ms
from (
  select timestamp '2014-06-09 13:50:17.4971' as ts
  from rdb$database
) a
```

```
T                       MS
----------------------- ------
2014-06-09 13:50:17.0000  497.1
```

Siehe auch

`DATEDIFF()`, Operationen mit Datums- und Uhrzeitwerten

8.4.2. DATEDIFF()

Verfügbar in

DSQL, PSQL

Ergebnistyp

`BIGINT`, oder — seit Firebird 3.0.8 — `NUMERIC(18,1)` für `MILLISECOND`

Syntax

```
DATEDIFF (<args>)

<args> ::=
    <unit> FROM <moment1> TO <moment2>
  | <unit>, <moment1>, <moment2>

<unit> ::=
    YEAR | MONTH | WEEK | DAY
  | HOUR | MINUTE | SECOND | MILLISECOND
<momentN> ::= ein DATE-, TIME- oder TIMESTAMP-Ausdruck
```

Tabelle 159. `DATEDIFF`-*Funktionsparameter*

Parameter	Beschreibung
unit	Datum/Uhrzeit-Einheit
moment1	Ein Ausdruck vom Typ `DATE`, `TIME` oder `TIMESTAMP`
moment2	Ein Ausdruck vom Typ `DATE`, `TIME` oder `TIMESTAMP`

Gibt die Anzahl der Jahre, Monate, Wochen, Tage, Stunden, Minuten, Sekunden oder Millisekunden zurück, die zwischen zwei Datums-/Uhrzeitwerten verstrichen sind. (Die Einheit `WOCHE` ist neu in 2.5.)

- Die Argumente `DATE` und `TIMESTAMP` können kombiniert werden. Andere Mischungen sind nicht erlaubt.
- Mit den Argumenten `TIMESTAMP` und `DATE` können alle Einheiten verwendet werden. (Vor Firebird 2.5 waren Einheiten, die kleiner als `DAY` waren, für `DATE`s nicht zulässig.)
- Bei `TIME`-Argumenten können nur `HOUR`, `MINUTE`, `SECOND` und `MILLISECOND` verwendet werden.

Berechnung

- `DATEDIFF` betrachtet keine kleineren Einheiten als die im ersten Argument angegebene. Als Ergebnis,
  - `datediff (Jahr, Datum '1-Jan-2009', Datum '31-Dez-2009')` gibt 0 zurück, aber
  - `` `datediff (Jahr, Datum '31-Dez-2009', Datum '1-Jan-2010')' `` gibt 1 zurück
- Es betrachtet jedoch alle *größeren* Einheiten. So:
  - `` `datediff (Tag, Datum '26-Jun-1908', Datum '11-Sep-1973')' `` gibt 23818 zurück
- Ein negativer Ergebniswert bedeutet, dass *moment2* vor *moment1* liegt.

`DATEDIFF`-Beispiele

```
datediff (hour from current_timestamp to timestamp '12-Jun-2059 06:00')
datediff (minute from time '0:00' to current_time)
datediff (month, current_date, date '1-1-1900')
datediff (day from current_date to cast(? as date))
```

Siehe auch

`DATEADD()`, Operationen mit Datums- und Uhrzeitwerten

8.4.3. `EXTRACT()`

Verfügbar in

DSQL, ESQL, PSQL

Ergebnistyp

`SMALLINT` or `NUMERIC`

Syntax

```
EXTRACT (<part> FROM <datetime>)

<part> ::=
    YEAR | MONTH | WEEK
  | DAY | WEEKDAY | YEARDAY
  | HOUR | MINUTE | SECOND | MILLISECOND
```

```
<datetime> ::= ein DATE-, TIME- oder TIMESTAMP-Ausdruck
```

Tabelle 160. EXTRACT-*Funktionsparameter*

Parameter	Beschreibung
part	Datum/Uhrzeit-Einheit
datetime	Ein Ausdruck vom Typ DATE, TIME oder TIMESTAMP

Extrahiert ein Element aus einem DATE, TIME oder TIMESTAMP Ausdruck und gibt es zurück. Diese Funktion wurde bereits in InterBase 6 hinzugefügt, aber zu diesem Zeitpunkt nicht in der *Sprachreferenz* dokumentiert.

Zurückgegebene Datentypen und Bereiche

Die zurückgegebenen Datentypen und möglichen Bereiche sind in der folgenden Tabelle aufgeführt. Wenn Sie versuchen, einen Teil zu extrahieren, der nicht im Datum/Uhrzeit-Argument vorhanden ist (z. B. SECOND aus einem DATE oder YEAR aus einer TIME), tritt ein Fehler auf.

Tabelle 161. Arten und Bereiche von EXTRACT-*Ergebnissen*

Teil	Typ	Bereich	Anmerkung
YEAR	SMALLINT	1-9999	
MONTH	SMALLINT	1-12	
WEEK	SMALLINT	1-53	
DAY	SMALLINT	1-31	
WEEKDAY	SMALLINT	0-6	0 = Sonntag
YEARDAY	SMALLINT	0-365	0 = 1. Januar
HOUR	SMALLINT	0-23	
MINUTE	SMALLINT	0-59	
SECOND	NUMERIC(9,4)	0.0000-59.9999	beinhaltet Millisekunden als Bruch
MILLISECOND	NUMERIC(9,1)	0.0-999.9	fehlerhaft in 2.1, 2.1.1

MILLISECOND

Firebird 2.1 und höher unterstützen die Extraktion der Millisekunde aus einer TIME oder TIMESTAMP. Der zurückgegebene Datentyp ist NUMERIC(9,1).

Wenn Sie die Millisekunde aus CURRENT_TIME extrahieren, beachten Sie, dass diese Variable standardmäßig auf die Sekundengenauigkeit eingestellt ist, sodass das Ergebnis immer 0 ist. Extrahieren Sie aus CURRENT_TIME(3) oder CURRENT_TIMESTAMP, um eine Genauigkeit in Millisekunden zu erhalten.

WEEK

Firebird 2.1 und höher unterstützen die Extraktion der ISO-8601-Wochennummer aus einem

"DATE" oder "TIMESTAMP". ISO-8601-Wochen beginnen an einem Montag und haben immer die vollen sieben Tage. Woche 1 ist die erste Woche mit einem Großteil (mindestens 4) der Tage im neuen Jahr. Die ersten 1-3 Tage des Jahres können zur letzten Woche (52 oder 53) des Vorjahres gehören. Ebenso können die letzten 1-3 Tage eines Jahres zur ersten Woche des Folgejahres gehören.

Seien Sie vorsichtig, wenn Sie die Ergebnisse von `WOCHE` und `JAHR` kombinieren. Zum Beispiel liegt der 30. Dezember 2008 in Woche 1 von 2009, also gibt `extract(week from date '30 Dec 2008')` 1 zurück. Das Extrahieren von `YEAR` ergibt jedoch immer das Kalenderjahr, das 2008 ist. In diesem Fall sind "WOCHE" und "JAHR" uneins. Das gleiche passiert, wenn die ersten Januartage zur letzten Woche des Vorjahres gehören.

Bitte beachten Sie auch, dass `WEEKDAY` *nicht* ISO-8601-kompatibel ist: Es gibt 0 für Sonntag zurück, während ISO-8601 7 angibt.

Siehe auch

Datentypen für Datum und Uhrzeit

8.5. Typ-Casting-Funktionen

8.5.1. CAST()

Verfügbar in

DSQL, ESQL, PSQL

Ergebnistyp

Wie von *target_type* angegeben

Syntax

```
CAST (<expression> AS <target_type>)

<target_type> ::= <domain_or_non_array_type> | <array_datatype>

<domain_or_non_array_type> ::=
  !! Siehe auch Syntax für skalare Datentypen !!

<array_datatype> ::=
  !! Siehe auch Syntax der Array-Datentypen !!
```

Tabelle 162. CAST-Funktionsparameter

Parameter	Beschreibung
expression	SQL-Ausdruck
sql_datatype	SQL-Datentyp

CAST wandelt einen Ausdruck in den gewünschten Datentyp oder die gewünschte Domäne um. Wenn die Konvertierung nicht möglich ist, wird ein Fehler ausgegeben.

Casting BLOBs

Erfolgreiches Casting zu und von BLOBs ist seit Firebird 2.1 möglich.

Syntax für "Kurzschreibweise"

Alternative Syntax, die nur unterstützt wird, wenn ein Stringliteral in ein DATE, TIME oder TIMESTAMP umgewandelt wird:

```
datatype 'date/timestring'
```

Diese Syntax war bereits in InterBase verfügbar, wurde jedoch nie richtig dokumentiert. Im SQL-Standard heißt diese Funktion "datetime literals".

Die kurze Syntax wird sofort beim Parsen ausgewertet, wodurch der Wert unverändert bleibt, bis die Anweisung unvorbereitet ist. Für Datetime-Literale wie '12-Oct-2012' macht dies keinen Unterschied. Für die Pseudovariablen 'NOW', 'YESTERDAY', 'TODAY' und 'TOMORROW' ist dies möglicherweise nicht das, was Sie wollen. Wenn der Wert bei jedem Aufruf ausgewertet werden soll, verwenden Sie die vollständige CAST()-Syntax.

Firebird 4 verbietet die Verwendung von 'NOW'', 'YESTERDAY' und 'TOMORROW' in der Kurzform und erlaubt nur Literale, die einen festen Zeitpunkt definieren.

Zulässige Typumwandlungen

Die folgende Tabelle zeigt die mit CAST möglichen Typkonvertierungen.

Tabelle 163. Mögliche Type-Castings mit CAST

Von	Nach
Numerische Typen	Numerische Typen [VAR]CHAR BLOB
[VAR]CHAR BLOB	[VAR]CHAR BLOB Numerische Typen DATE TIME TIMESTAMP
DATE TIME	[VAR]CHAR BLOB TIMESTAMP

Von	Nach
TIMESTAMP	[VAR]CHAR BLOB DATE TIME

Denken Sie daran, dass manchmal Informationen verloren gehen, zum Beispiel wenn Sie einen TIMESTAMP in ein DATE umwandeln. Auch die Tatsache, dass Typen CAST-kompatibel sind, ist noch keine Garantie dafür, dass eine Konvertierung erfolgreich ist. "CAST(123456789 as SMALLINT)" führt definitiv zu einem Fehler, ebenso wie "CAST('Judgement Day' as DATE)".

Casting-Parameter

Seit Firebird 2.0 können Sie Anweisungsparameter in einen Datentyp umwandeln:

```
cast (? as integer)
```

Dies gibt Ihnen die Kontrolle über den Typ des Parameters, der von der Engine eingerichtet wird. Bitte beachten Sie, dass Sie bei Anweisungsparametern immer eine vollständige Syntaxumwandlung benötigen – Kurzformumwandlungen werden nicht unterstützt.

Casting in eine Domain oder deren Typ

Firebird 2.1 und höher unterstützen das Casting in eine Domäne oder deren Basistyp. Beim Casting in eine Domain müssen alle für die Domain deklarierten Constraints (NOT NULL und/oder CHECK) erfüllt sein, sonst schlägt das Casting fehl. Bitte beachten Sie, dass ein CHECK erfolgreich ist, wenn es als TRUE *oder* NULL ausgewertet wird! Also folgende Aussagen gegeben:

```
create domain quint as int check (value >= 5000);
select cast (2000 as quint) from rdb$database;     ①
select cast (8000 as quint) from rdb$database;     ②
select cast (null as quint) from rdb$database;     ③
```

nur die Besetzungsnummer *1* führt zu einem Fehler.

Wenn der Modifikator TYPE OF verwendet wird, wird der Ausdruck in den Basistyp der Domäne umgewandelt, wobei alle Einschränkungen ignoriert werden. Mit der oben definierten Domain 'quint' sind die folgenden beiden Casts äquivalent und werden beide erfolgreich sein:

```
select cast (2000 as type of quint) from rdb$database;
select cast (2000 as int) from rdb$database;
```

Wenn TYPE OF mit einem (VAR)CHAR-Typ verwendet wird, werden sein Zeichensatz und seine Kollatierung beibehalten:

```
create domain iso20 varchar(20) character set iso8859_1;
create domain dunl20 varchar(20) character set iso8859_1 collate du_nl;
create table zinnen (zin varchar(20));
commit;
insert into zinnen values ('Deze');
insert into zinnen values ('Die');
insert into zinnen values ('die');
insert into zinnen values ('deze');

select cast(zin as type of iso20) from zinnen order by 1;
-- Ergebnis Deze -> Die -> deze -> die

select cast(zin as type of dunl20) from zinnen order by 1;
-- Ergebnis deze -> Deze -> die -> Die
```

Wenn die Definition einer Domain geändert wird, können bestehende CASTs für diese Domain oder ihr Typ ungültig werden. Wenn diese CASTs in PSQL-Modulen vorkommen, kann ihre Ungültigkeit erkannt werden. Siehe Hinweis *Das RDB$VALID_BLR-Feld* in Anhang A.

Umwandeln in den Typ einer Spalte

In Firebird 2.5 und höher ist es möglich, Ausdrücke in den Typ einer vorhandenen Tabelle oder Ansichtsspalte umzuwandeln. Nur der Typ selbst wird verwendet; bei String-Typen umfasst dies den Zeichensatz, aber nicht die Kollatierung. Einschränkungen und Standardwerte der Quellspalte werden nicht angewendet.

```
create table ttt (
  s varchar(40) character set utf8 collate unicode_ci_ai
);
commit;

select cast ('Jag har många vänner' as type of column ttt.s)
from rdb$database;
```

Warnungen

Wenn die Definition einer Spalte geändert wird, können vorhandene CASTs für den Typ dieser Spalte ungültig werden. Wenn diese CASTs in PSQL-Modulen vorkommen, kann ihre Ungültigkeit erkannt werden. Siehe den Hinweis *Das RDB$VALID_BLR-Feld* in Anhang A.

Cast-Beispiele

Ein Vollsyntax Cast:

```
select cast ('12' || '-June-' || '1959' as date) from rdb$database
```

Eine Kurzschreibweise zur Umwandlung von String zu Datum:

```
update People set AgeCat = 'Old'
  where BirthDate < date '1-Jan-1943'
```

Beachten Sie, dass Sie sogar die Kurzform aus dem obigen Beispiel weglassen können, da die Engine aus dem Kontext (Vergleich mit einem DATE-Feld) versteht, wie die Zeichenfolge zu interpretieren ist:

```
update People set AgeCat = 'Old'
  where BirthDate < '1-Jan-1943'
```

Aber das ist nicht immer möglich. Der folgende Cast kann nicht weggelassen werden, sonst würde sich die Engine mit einer Ganzzahl wiederfinden, die von einer Zeichenfolge subtrahiert werden soll:

```
select date 'today' - 7 from rdb$database
```

8.6. Bitweise Funktionen

8.6.1. BIN_AND()

Verfügbar in

DSQL, PSQL

Möglicher Namenskonflikt

YES → Details lesen

Ergebnistyp

SMALLINT, INTEGER oder BIGINT

Das Ergebnis SMALLINT wird nur zurückgegeben, wenn alle Argumente explizit SMALLINTs oder NUMERIC(n, 0) mit *n* <= 4 sind; andernfalls geben kleine Ganzzahlen ein INTEGER-Ergebnis zurück.

Syntax

```
BIN_AND (number, number [, number ...])
```

Tabelle 164. BIN_AND*-Funktionsparameter*

Parameter	Beschreibung
number	Beliebige ganze Zahl (literal, smallint/integer/bigint, numerisch/dezimal mit Skalierung 0)

Gibt das Ergebnis der bitweisen *AND*-Operation für die Argumente zurück.

Siehe auch

`BIN_OR()`, `BIN_XOR()`

8.6.2. BIN_NOT()

Verfügbar in

DSQL, PSQL

Ergebnistyp

`SMALLINT`, `INTEGER` oder `BIGINT`

Das Ergebnis `SMALLINT` wird nur zurückgegeben, wenn alle Argumente explizit `SMALLINT`s oder `NUMERIC(n, 0)` mit *n* <= 4 sind; andernfalls geben kleine Ganzzahlen ein `INTEGER`-Ergebnis zurück.

Syntax

```
BIN_NOT (number)
```

Tabelle 165. `BIN_NOT`-Funktionsparameter

Parameter	Beschreibung
number	Beliebige ganze Zahl (literal, smallint/integer/bigint, numerisch/dezimal mit Skala 0)

Gibt das Ergebnis der bitweisen *NOT*-Operation für das Argument zurück, d.h. das *Einser-Komplement*.

Siehe auch

`BIN_OR()`, `BIN_XOR()` und weitere.

8.6.3. BIN_OR()

Verfügbar in

DSQL, PSQL

Möglicher Namenskonflikt

YES → Details lesen

Ergebnistyp

`SMALLINT`, `INTEGER` oder `BIGINT`

Das Ergebnis SMALLINT wird nur zurückgegeben, wenn alle Argumente explizit SMALLINTs oder NUMERIC(n, 0) mit *n* <= 4 sind; andernfalls geben kleine Ganzzahlen ein INTEGER-Ergebnis zurück.

Syntax

```
BIN_OR (number, number [, number ...])
```

Tabelle 166. BIN_OR*-Funktionsparameter*

Parameter	Beschreibung
number	Beliebige ganze Zahl (literal, smallint/integer/bigint, numerisch/dezimal mit Skalierung 0)

Gibt das Ergebnis der bitweisen *OR*-Operation für die Argumente zurück.

Siehe auch

BIN_AND(), BIN_XOR()

8.6.4. BIN_SHL()

Verfügbar in

DSQL, PSQL

Ergebnistyp

BIGINT

Syntax

```
BIN_SHL (number, shift)
```

Tabelle 167. BIN_SHL*-Funktionsparameter*

Parameter	Beschreibung
number	Eine Zahl eines ganzzahligen Typs
shift	Die Anzahl der Bits, um die der Zahlenwert verschoben wird

Gibt das erste Argument bitweise linksverschoben um das zweite Argument zurück, d. h. a << b oder $a \cdot 2^b$.

Siehe auch

BIN_SHR()

8.6.5. BIN_SHR()

Verfügbar in

DSQL, PSQL

Ergebnistyp

`BIGINT`

Syntax

```
BIN_SHR (number, shift)
```

Tabelle 168. `BIN_SHR`-Funktionsparameter

Parameter	Beschreibung
number	Eine Zahl eines ganzzahligen Typs
shift	Die Anzahl der Bits, um die der Zahlenwert verschoben wird

Gibt das erste Argument bitweise nach rechts verschoben um das zweite Argument zurück, d. h. `a >> b` oder $a/2^b$.

- Die ausgeführte Operation ist eine arithmetische Rechtsverschiebung (SAR), dh das Vorzeichen des ersten Operanden bleibt immer erhalten.

Siehe auch

`BIN_SHL()`

8.6.6. BIN_XOR()

Verfügbar in

DSQL, PSQL

Möglicher Namenskonflikt

YES → Details lesen

Ergebnistyp

`SMALLINT`, `INTEGER` oder `BIGINT`

Das Ergebnis `SMALLINT` wird nur zurückgegeben, wenn alle Argumente explizit `SMALLINT`s oder `NUMERIC(n, 0)` mit *n* <= 4 sind; andernfalls geben kleine Ganzzahlen ein `INTEGER`-Ergebnis zurück.

Syntax

```
BIN_XOR (number, number [, number ...])
```

Tabelle 169. `BIN_XOR`-Funktionsparameter

Parameter	Beschreibung
number	Beliebige ganze Zahl (literal, smallint/integer/bigint, numerisch/dezimal mit Skalierung 0)

Gibt das Ergebnis der bitweisen *XOR*-Operation für die Argumente zurück.

Siehe auch

BIN_AND(), BIN_OR()

8.7. UUID Functions

8.7.1. CHAR_TO_UUID()

Verfügbar in

DSQL, PSQL

Ergebnistyp

CHAR(16) CHARACTER SET OCTETS

Syntax

```
CHAR_TO_UUID (ascii_uuid)
```

Tabelle 170. CHAR_TO_UUID-Funktionsparameter

Parameter	Beschreibung
ascii_uuid	Eine 36-stellige Darstellung der UUID. '-' (Bindestrich) in den Positionen 9, 14, 19 und 24; gültige Hexadezimalziffern an beliebigen anderen Stellen, z.B. 'A0bF4E45-3029-2a44-D493-4998c9b439A3'

Konvertiert eine für Menschen lesbare 36-stellige UUID-Zeichenfolge in die entsprechende 16-Byte-UUID.

CHAR_TO_UUID-Beispiele

```
select char_to_uuid('A0bF4E45-3029-2a44-D493-4998c9b439A3') from rdb$database
-- Ergebnis A0BF4E4530292A44D4934998C9B439A3 (16-Byte String)

select char_to_uuid('A0bF4E45-3029-2A44-X493-4998c9b439A3') from rdb$database
-- Fehler: - Menschlich lesbares UUID-Argument für CHAR_TO_UUID
             muss eine Hex-Ziffer an Position 20 anstelle von "X (ASCII 88)" haben
```

Siehe auch

UUID_TO_CHAR(), GEN_UUID()

8.7.2. GEN_UUID()

Verfügbar in

DSQL, PSQL

Ergebnistyp

CHAR(16) CHARACTER SET OCTETS

Syntax

```
GEN_UUID ()
```

Gibt eine universell eindeutige ID als 16-Byte-Zeichenfolge zurück.

GEN_UUID-Beispiel

```
select gen_uuid() from rdb$database
-- Ergebnis e.g. 017347BFE212B2479C00FA4323B36320 (16-Byte String)
```

Siehe auch

UUID_TO_CHAR(), CHAR_TO_UUID()

8.7.3. UUID_TO_CHAR()

Verfügbar in

DSQL, PSQL

Ergebnistyp

CHAR(36)

Syntax

```
UUID_TO_CHAR (uuid)
```

Tabelle 171. UUID_TO_CHAR-*Funktionsparameter*

Parameter	Beschreibung
uuid	16-Byte UUID

Konvertiert eine 16-Byte-UUID in ihre 36-stellige, für Menschen lesbare ASCII-Darstellung.

UUID_TO_CHAR-Beispiele

```
select uuid_to_char(x'876C45F4569B320DBCB4735AC3509E5F') from rdb$database
-- Ergebnis '876C45F4-569B-320D-BCB4-735AC3509E5F'

select uuid_to_char(gen_uuid()) from rdb$database
-- Ergebnis e.g. '680D946B-45FF-DB4E-B103-BB5711529B86'

select uuid_to_char('Firebird swings!') from rdb$database
-- Ergebnis '46697265-6269-7264-2073-77696E677321'
```

Siehe auch

CHAR_TO_UUID(), GEN_UUID()

8.8. Funktionen für Sequenzen (Generatoren)

8.8.1. GEN_ID()

Verfügbar in

DSQL, ESQL, PSQL

Ergebnistyp

`BIGINT`

Syntax

```
GEN_ID (generator-name, step)
```

Tabelle 172. GEN_ID-Funktionsparameter

Parameter	Beschreibung
generator-name	Name eines vorhandenen Generators (Sequenz). Wenn er in doppelten Anführungszeichen mit einem Bezeichner definiert wurde, bei dem die Groß-/Kleinschreibung beachtet wird, muss er in derselben Form verwendet werden, es sei denn, der Name wird ausschließlich in Großbuchstaben geschrieben.
step	Ein ganzzahliger Ausdruck

Erhöht einen Generator oder eine Sequenz und gibt den neuen Wert zurück. Wenn Schritt gleich 0 ist, lässt die Funktion den Wert des Generators unverändert und gibt seinen aktuellen Wert zurück.

- Ab Firebird 2.0 wird die SQL-kompatible Syntax `NEXT VALUE FOR` bevorzugt, außer wenn eine andere Erhöhung als 1 benötigt wird.

Wenn der Wert des Schrittparameters kleiner als Null ist, wird der Wert des Generators verringert. Achtung! Bei solchen Manipulationen in der Datenbank sollten Sie äußerst vorsichtig sein, da sie die Datenintegrität beeinträchtigen könnten.

GEN_ID-Beispiel

```
new.rec_id = gen_id(gen_recnum, 1);
```

Siehe auch

`NEXT VALUE FOR`, `CREATE SEQUENCE (GENERATOR)`

8.9. Bedingte Funktionen

8.9.1. COALESCE()

Verfügbar in

DSQL, PSQL

Ergebnistyp

Abhängig von der Eingabe

Syntax

```
COALESCE (<exp1>, <exp2> [, <expN> ... ])
```

Tabelle 173. COALESCE-Funktionsparameter

Parameter	Beschreibung
exp1, exp2 ... expN	Eine Liste von Ausdrücken aller kompatiblen Typen

Die Funktion COALESCE nimmt zwei oder mehr Argumente und gibt den Wert des ersten Nicht-NULL -Arguments zurück. Wenn alle Argumente NULL ergeben, ist das Ergebnis NULL.

COALESCE-Beispiele

Dieses Beispiel wählt den Nickname aus der Persons-Tabelle. Wenn es NULL ist, geht es weiter zu FirstName. Ist auch dieser NULL, wird "'Mr./Mrs.'" verwendet. Schließlich fügt es den Familiennamen hinzu. Insgesamt wird versucht, aus den verfügbaren Daten einen möglichst informellen vollständigen Namen zusammenzustellen. Beachten Sie, dass dieses Schema nur funktioniert, wenn fehlende Spitznamen und Vornamen wirklich NULL sind: Wenn einer von ihnen stattdessen ein leerer String ist, wird COALESCE diesen glücklich an den Aufrufer zurückgeben.

```
select
  coalesce (Nickname, FirstName, 'Mr./Mrs.') || ' ' || LastName
    as FullName
from Persons
```

Siehe auch

IIF(), NULLIF(), CASE

8.9.2. DECODE()

Verfügbar in

DSQL, PSQL

Ergebnistyp

Abhängig von der Eingabe

Syntax

```
DECODE(<testexpr>,
```

```
  <expr1>, <result1>
  [<expr2>, <result2> ⋯]
  [, <defaultresult>])
```

Das äquivalente CASE-Konstrukt:

```
CASE <testexpr>
  WHEN <expr1> THEN <result1>
  [WHEN <expr2> THEN <result2> ⋯]
  [ELSE <defaultresult>]
END
```

Tabelle 174. DECODE-*Funktionsparameter*

Parameter	Beschreibung
testexpr	Ein Ausdruck eines beliebigen kompatiblen Typs, der mit den Ausdrücken expr1, expr2 ... exprN . verglichen wird
expr1, expr2, ... exprN	Ausdrücke beliebiger kompatibler Typen, mit denen der Ausdruck *testexpr* verglichen wird
result1, result2, ... resultN	Rückgabewerte jeglichen Typs
defaultresult	Der Ausdruck, der zurückgegeben werden soll, wenn keine der Bedingungen erfüllt ist

DECODE ist eine Abkürzung für das sogenannte "Einfaches CASE"-Konstrukt, in dem ein gegebener Ausdruck mit einer Anzahl von andere Ausdrücke, bis eine Übereinstimmung gefunden wird. Das Ergebnis wird durch den Wert bestimmt, der nach dem übereinstimmenden Ausdruck aufgeführt ist. Wenn keine Übereinstimmung gefunden wird, wird das Standardergebnis zurückgegeben, falls vorhanden. Andernfalls wird NULL zurückgegeben.

Der Abgleich erfolgt mit dem Operator '='. Wenn also *testexpr* NULL ist, wird es mit keinem der *expr*s übereinstimmen, nicht einmal mit denen, die NULL sind.

DECODE-Beispiele

```
select name,
  age,
  decode(upper(sex),
         'M', 'Male',
         'F', 'Female',
         'Unknown'),
  religion
from people
```

Siehe auch

CASE, Einfaches CASE

8.9.3. IIF()

Verfügbar in

DSQL, PSQL

Ergebnistyp

Abhängig von der Eingabe

Syntax

```
IIF (<condition>, ResultT, ResultF)
```

Tabelle 175. IIF-Funktionsparameter

Parameter	Beschreibung
condition	Ein wahrer|falscher Ausdruck
resultT	Der zurückgegebene Wert, wenn die Bedingung wahr ist
resultF	Der zurückgegebene Wert, wenn die Bedingung falsch ist

IIF benötigt drei Argumente. Wenn das erste Argument `true` ergibt, wird das zweite Argument zurückgegeben; andernfalls wird die dritte zurückgegeben.

IIF könnte in C-ähnlichen Sprachen mit dem ternären Operator "?:" verglichen werden.

IIF(<Cond>, Result1, Result2) ist eine Abkürzung für "CASE WHEN <Cond> THEN Result1 ELSE Result2 END".

IIF-Beispiele

```
select iif( sex = 'M', 'Sir', 'Madam' ) from Customers
```

Siehe auch

CASE, DECODE()

8.9.4. MAXVALUE()

Verfügbar in

DSQL, PSQL

Ergebnistyp

Variiert je nach Eingabe — das Ergebnis hat denselben Datentyp wie der erste Ausdruck in der Liste (*expr1*).

Syntax

```
MAXVALUE (<expr1> [, ... , <exprN> ])
```

Tabelle 176. MAXVALUE-Funktionsparameter

Parameter	Beschreibung
expr1 ... exprN	Liste der Ausdrücke kompatibler Typen

Gibt den Höchstwert aus einer Liste von numerischen, Zeichenfolgen- oder Datums-/Uhrzeitausdrücken zurück. Diese Funktion unterstützt vollständig Text-BLOBs jeder Länge und jedes beliebigen Zeichensatzes.

Wenn ein oder mehrere Ausdrücke in NULL aufgelöst werden, gibt MAXVALUE NULL zurück. Dieses Verhalten unterscheidet sich von der Aggregatfunktion MAX.

MAXVALUE-Beispiele

```
SELECT MAXVALUE(PRICE_1, PRICE_2) AS PRICE
  FROM PRICELIST
```

Siehe auch

MINVALUE()

8.9.5. MINVALUE()

Verfügbar in

DSQL, PSQL

Ergebnistyp

Variiert je nach Eingabe — das Ergebnis hat denselben Datentyp wie der erste Ausdruck in der Liste (*expr1*).

Syntax

```
MINVALUE (<expr1> [, ... , <exprN> ])
```

Tabelle 177. MINVALUE-Funktionsparameter

Parameter	Beschreibung
expr1 ... exprN	Liste der Ausdrücke kompatibler Typen

Gibt den Mindestwert aus einer Liste von numerischen, Zeichenfolgen- oder Datums-/Uhrzeitausdrücken zurück. Diese Funktion unterstützt vollständig Text-BLOBs jeder Länge und jedes beliebigen Zeichensatzes.

Wenn ein oder mehrere Ausdrücke in NULL aufgelöst werden, gibt MINVALUE NULL zurück. Dieses Verhalten unterscheidet sich von der Aggregatfunktion MIN.

MINVALUE-Beispiele

```
SELECT MINVALUE(PRICE_1, PRICE_2) AS PRICE
  FROM PRICELIST
```

Siehe auch

MAXVALUE()

8.9.6. NULLIF()

Verfügbar in

DSQL, PSQL

Ergebnistyp

Abhängig von der Eingabe

Syntax

```
NULLIF (<exp1>, <exp2>)
```

Tabelle 178. NULLIF-Funktionsparameter

Parameter	Beschreibung
exp1	Ein Ausdruck
exp2	Ein anderer Ausdruck eines Datentyps, der mit *exp1* kompatibel ist

NULLIF gibt den Wert des ersten Arguments zurück, es sei denn, es ist gleich dem zweiten. In diesem Fall wird NULL zurückgegeben.

NULLIF-Beispiel

```
select avg( nullif(Weight, -1) ) from FatPeople
```

Dadurch wird das durchschnittliche Gewicht der in FatPeople aufgelisteten Personen zurückgegeben, mit Ausnahme derer mit einem Gewicht von -1, da "AVG" "NULL"-Daten überspringt. Vermutlich bedeutet -1 in dieser Tabelle “Gewicht unbekannt”. Ein einfaches AVG(Weight) würde die -1 Gewichte enthalten, wodurch das Ergebnis verzerrt wird.

Siehe auch

COALESCE(), DECODE(), IIF(), CASE

Kapitel 9. Aggregatfunktionen

Aggregatfunktionen arbeiten mit Gruppen von Datensätzen und nicht mit einzelnen Datensätzen oder Variablen. Sie werden oft in Kombination mit einer GROUP BY-Klausel verwendet.

Die Aggregatfunktionen können mit der OVER()-Klausel auch als Fensterfunktionen verwendet werden. Weitere Informationen finden Sie unter Window (Analytical) Functions.

9.1. Allgemeine Aggregatfunktionen

9.1.1. AVG()

Verfügbar in

DSQL, ESQL, PSQL

Rückgabetyp

Ein numerischer Datentyp, der dem Datentyp des Arguments entspricht.

Syntax

```
AVG ([ALL | DISTINCT] <expr>)
```

Tabelle 179. AVG-Funktionsparameter

Parameter	Beschreibung
expr	Ausdruck. Sie kann eine Tabellenspalte, eine Konstante, eine Variable, einen Ausdruck, eine Nicht-Aggregatfunktion oder eine UDF enthalten, die einen numerischen Datentyp zurückgibt. Aggregatfunktionen sind als Ausdrücke nicht zulässig

AVG gibt den durchschnittlichen Argumentwert in der Gruppe zurück. NULL wird ignoriert.

- Parameter ALL (Standard) wendet die Aggregatfunktion auf alle Werte an.
- Der Parameter DISTINCT weist die AVG-Funktion an, nur eine Instanz jedes eindeutigen Werts zu berücksichtigen, egal wie oft dieser Wert auftritt.
- Wenn die Menge der abgerufenen Datensätze leer ist oder nur NULL enthält, ist das Ergebnis NULL.

AVG-Beispiele

```
SELECT
  dept_no,
  AVG(salary)
FROM employee
GROUP BY dept_no
```

Siehe auch

`SELECT`

9.1.2. `COUNT()`

Verfügbar in

DSQL, ESQL, PSQL

Rückgabetyp

`BIGINT`

Syntax

```
COUNT ([ALL | DISTINCT] <expr> | *)
```

Tabelle 180. `COUNT`-Funktionsparameter

Parameter	Beschreibung
expr	Ausdruck. Sie kann eine Tabellenspalte, eine Konstante, eine Variable, einen Ausdruck, eine Nicht-Aggregatfunktion oder eine UDF enthalten, die einen numerischen Datentyp zurückgibt. Aggregatfunktionen sind als Ausdrücke nicht zulässig

`COUNT` gibt die Anzahl der Nicht-Null-Werte in einer Gruppe zurück.

- `ALL` ist die Vorgabe: es zählt einfach alle Werte in der Menge, die nicht `NULL` sind.
- Wenn `DISTINCT` angegeben ist, werden Duplikate aus der gezählten Menge ausgeschlossen.
- Wenn `COUNT (*)` anstelle des Ausdrucks *expr* angegeben wird, werden alle Zeilen gezählt. `ZAHL (*)` —
  - akzeptiert keine Parameter
  - kann nicht mit dem Schlüsselwort `DISTINCT` verwendet werden
  - nimmt kein *expr*-Argument an, da sein Kontext per Definition spaltenunspezifisch ist
  - zählt jede Zeile separat und gibt die Anzahl der Zeilen in der angegebenen Tabelle oder Gruppe zurück, ohne doppelte Zeilen auszulassen
  - zählt Zeilen mit `NULL`
- Wenn die Ergebnismenge leer ist oder nur `NULL` in der/den angegebenen Spalte(n) enthält, ist der zurückgegebene Zähler null.

`COUNT`-Beispiele

```
SELECT
  dept_no,
  COUNT(*) AS cnt,
  COUNT(DISTINCT name) AS cnt_name
FROM employee
```

```
GROUP BY dept_no
```

Siehe auch

SELECT.

9.1.3. LIST()

Verfügbar in

DSQL, PSQL

Rückgabetyp

BLOB

Syntax

```
LIST ([ALL | DISTINCT] <expr> [, separator ])
```

Tabelle 181. LIST-Funktionsparameter

Parameter	Beschreibung
expr	Ausdruck. Es kann eine Tabellenspalte, eine Konstante, eine Variable, einen Ausdruck, eine Nicht-Aggregatfunktion oder eine UDF enthalten, die den String-Datentyp oder ein 'BLOB' zurückgibt. Felder des numerischen Typs und des Datums-/Uhrzeittyps werden in Zeichenfolgen umgewandelt. Aggregatfunktionen sind als Ausdrücke nicht zulässig.
separator	Optionales alternatives Trennzeichen, ein Zeichenfolgenausdruck. Komma ist das Standardtrennzeichen

LIST gibt einen String zurück, der aus den Nicht-NULL-Argumentwerten in der Gruppe besteht, getrennt entweder durch ein Komma oder durch ein vom Benutzer angegebenes Trennzeichen. Wenn keine Nicht-NULL-Werte vorhanden sind (dies schließt den Fall ein, in dem die Gruppe leer ist), wird NULL zurückgegeben.

- ALL (Standard) führt dazu, dass alle Nicht-NULL-Werte aufgelistet werden. Mit DISTINCT werden Duplikate entfernt, außer wenn *expr* ein BLOB ist.
- In Firebird 2.5 und höher kann das optionale Argument *separator* ein beliebiger String-Ausdruck sein. Dadurch ist es möglich, z.B. ascii_char(13) als Trennzeichen. (Diese Verbesserung wurde auch auf 2.1.4 zurückportiert.)
- Die Argumente *expr* und *separator* unterstützen BLOBs jeder Größe und jedes Zeichensatzes.
- Datum/Uhrzeit und numerische Argumente werden vor der Verkettung implizit in Zeichenfolgen umgewandelt.
- Das Ergebnis ist ein Text BLOB, außer wenn *expr* ein BLOB eines anderen Untertyps ist.
- Die Reihenfolge der Listenwerte ist undefiniert — die Reihenfolge, in der die Strings verkettet werden, wird durch die Lesereihenfolge aus dem Quellsatz bestimmt, die in Tabellen nicht allgemein definiert ist. Wenn die Sortierung wichtig ist, können die Quelldaten mithilfe einer

abgeleiteten Tabelle oder ähnlichem vorsortiert werden.

LIST-Beispiele

Abrufen der Liste, Sortierung undefiniert:

+

```
SELECT LIST (display_name, '; ') FROM GR_WORK;
```

1. Abrufen der Liste in alphabetischer Reihenfolge mithilfe einer abgeleiteten Tabelle:

```
SELECT LIST (display_name, '; ')
FROM (SELECT display_name
      FROM GR_WORK
      ORDER BY display_name);
```

Siehe auch

SELECT

9.1.4. MAX()

Verfügbar in

DSQL, ESQL, PSQL

Rückgabetyp

Gibt ein Ergebnis des gleichen Datentyps wie der Eingabeausdruck zurück.

Syntax

```
MAX ([ALL | DISTINCT] <expr>)
```

Tabelle 182. MAX-Funktionsparameter

Parameter	Beschreibung
expr	Ausdruck. Sie kann eine Tabellenspalte, eine Konstante, eine Variable, einen Ausdruck, eine Nicht-Aggregatfunktion oder eine UDF enthalten. Aggregatfunktionen sind als Ausdrücke nicht zulässig.

MAX gibt das maximale Nicht-NULL-Element in der Ergebnismenge zurück.

- Wenn die Gruppe leer ist oder nur NULLs enthält, ist das Ergebnis NULL.
- Wenn das Eingabeargument ein String ist, gibt die Funktion den Wert zurück, der zuletzt sortiert wird, wenn COLLATE verwendet wird.
- Diese Funktion unterstützt vollständig Text-BLOBs jeder Größe und jedes Zeichensatzes.

Der Parameter DISTINCT macht bei Verwendung mit MAX() keinen Sinn und wird

nur zur Einhaltung des Standards implementiert.

MAX-Beispiele

```
SELECT
  dept_no,
  MAX(salary)
FROM employee
GROUP BY dept_no
```

Siehe auch

MIN(), SELECT

9.1.5. MIN()

Verfügbar in

DSQL, ESQL, PSQL

Rückgabetyp

Gibt ein Ergebnis des gleichen Datentyps wie der Eingabeausdruck zurück.

Syntax

```
MIN ([ALL | DISTINCT] <expr>)
```

Tabelle 183. MIN-Funktionsparameter

Parameter	Beschreibung
expr	Ausdruck. Sie kann eine Tabellenspalte, eine Konstante, eine Variable, einen Ausdruck, eine Nicht-Aggregatfunktion oder eine UDF enthalten. Aggregatfunktionen sind als Ausdrücke nicht zulässig.

MIN gibt das minimale Nicht-NULL-Element in der Ergebnismenge zurück.

- Wenn die Gruppe leer ist oder nur NULLs enthält, ist das Ergebnis NULL.
- Wenn das Eingabeargument ein String ist, gibt die Funktion den Wert zurück, der zuerst sortiert wird, wenn COLLATE verwendet wird.
- Diese Funktion unterstützt vollständig Text-BLOBs jeder Größe und jedes Zeichensatzes.

Der Parameter DISTINCT macht bei Verwendung mit MIN() keinen Sinn und wird nur zur Einhaltung des Standards implementiert.

MIN-Beispiele

```
SELECT
  dept_no,
```

```
  MIN(salary)
FROM employee
GROUP BY dept_no
```

Siehe auch

MAX(), SELECT

9.1.6. SUM()

Verfügbar in

DSQL, ESQL, PSQL

Rückgabetyp

Gibt ein Ergebnis des gleichen Datentyps wie der Eingabeausdruck zurück.

Syntax

```
SUM ([ALL | DISTINCT] <expr>)
```

Tabelle 184. SUM-Funktionsparameter

Parameter	Beschreibung
expr	Numerischer Ausdruck. Sie kann eine Tabellenspalte, eine Konstante, eine Variable, einen Ausdruck, eine Nicht-Aggregatfunktion oder eine UDF enthalten. Aggregatfunktionen sind als Ausdrücke nicht zulässig.

SUM berechnet die Summe der Nicht-Null-Werte in der Gruppe und gibt sie zurück.

- Wenn die Gruppe leer ist oder nur NULLs enthält, ist das Ergebnis NULL.
- ALL ist die Standardoption — alle Werte in der Menge, die nicht NULL sind, werden verarbeitet. Bei Angabe von DISTINCT werden Duplikate aus dem Set entfernt und anschließend die SUM -Auswertung durchgeführt.

SUM-Beispiele

```
SELECT
  dept_no,
  SUM (salary),
FROM employee
GROUP BY dept_no
```

Siehe auch

SELECT

9.2. Statistische Aggregatfunktionen

9.2.1. CORR

Verfügbar in

DSQL, PSQL

Rückgabetyp

`DOUBLE PRECISION`

Syntax

```
CORR ( <expr1>, <expr2> )
```

Tabelle 185. CORR-Funktionsparameter

Parameter	Beschreibung
expr*N*	Numerischer Ausdruck. Sie kann eine Tabellenspalte, eine Konstante, eine Variable, einen Ausdruck, eine Nicht-Aggregatfunktion oder eine UDF enthalten. Aggregatfunktionen sind als Ausdrücke nicht zulässig.

Die Funktion `CORR` gibt den Korrelationskoeffizienten für ein Paar numerischer Ausdrücke zurück.

Die Funktion `CORR(<expr1>, <expr2>)` ist äquivalent zu

```
COVAR_POP(<expr1>, <expr2>) / (STDDEV_POP(<expr2>) * STDDEV_POP(<expr1>))
```

Dies wird auch als Korrelationskoeffizient nach Pearson bezeichnet.

Im statistischen Sinne ist Korrelation der Grad, mit dem ein Variablenpaar linear verbunden ist. Eine lineare Beziehung zwischen Variablen bedeutet, dass der Wert einer Variablen bis zu einem gewissen Grad den Wert der anderen vorhersagen kann. Der Korrelationskoeffizient stellt den Korrelationsgrad als Zahl von -1 (hohe inverse Korrelation) bis 1 (hohe Korrelation) dar. Ein Wert von 0 entspricht keiner Korrelation.

Wenn die Gruppe oder das Fenster leer ist oder nur `NULL`-Werte enthält, ist das Ergebnis `NULL`.

CORR-Beispiele

```
select
  corr(alength, aheight) AS c_corr
from measure
```

Siehe auch

`COVAR_POP`, `STDDEV_POP`

9.2.2. COVAR_POP

Verfügbar in

DSQL, PSQL

Rückgabetyp

`DOUBLE PRECISION`

Syntax

```
COVAR_POP ( <expr1>, <expr2> )
```

Tabelle 186. `COVAR_POP`-Funktionsparameter

Parameter	Beschreibung
expr*N*	Numerischer Ausdruck. Sie kann eine Tabellenspalte, eine Konstante, eine Variable, einen Ausdruck, eine Nicht-Aggregatfunktion oder eine UDF enthalten. Aggregatfunktionen sind als Ausdrücke nicht zulässig.

Die Funktion `COVAR_POP` gibt die Populationskovarianz für ein Paar numerischer Ausdrücke zurück.

Die Funktion `COVAR_POP(<expr1>, <expr2>)` ist äquivalent zu

```
(SUM(<expr1> * <expr2>) - SUM(<expr1>) * SUM(<expr2>) / COUNT(*)) / COUNT(*)
```

Wenn die Gruppe oder das Fenster leer ist oder nur `NULL`-Werte enthält, ist das Ergebnis `NULL`.

`COVAR_POP`-Beispiele

```
select
  covar_pop(alength, aheight) AS c_covar_pop
from measure
```

Siehe auch

`COVAR_SAMP`, `SUM()`, `COUNT()`

9.2.3. COVAR_SAMP

Verfügbar in

DSQL, PSQL

Rückgabetyp

`DOUBLE PRECISION`

Syntax

```
COVAR_SAMP ( <expr1>, <expr2> )
```

Tabelle 187. COVAR_SAMP-*Funktionsparameter*

Parameter	Beschreibung
expr*N*	Numerischer Ausdruck. Sie kann eine Tabellenspalte, eine Konstante, eine Variable, einen Ausdruck, eine Nicht-Aggregatfunktion oder eine UDF enthalten. Aggregatfunktionen sind als Ausdrücke nicht zulässig.

Die Funktion COVAR_SAMP gibt die Stichprobenkovarianz für ein Paar numerischer Ausdrücke zurück.

Die Funktion COVAR_SAMP(<expr1>, <expr2>) ist äquivalent zu

```
(SUM(<expr1> * <expr2>) - SUM(<expr1>) * SUM(<expr2>) / COUNT(*)) / (COUNT(*) - 1)
```

Wenn die Gruppe oder das Fenster leer ist, nur 1 Zeile enthält oder nur NULL-Werte enthält, ist das Ergebnis NULL.

COVAR_SAMP-Beispiele

```
select
  covar_samp(alength, aheight) AS c_covar_samp
from measure
```

Siehe auch

COVAR_POP, SUM(), COUNT()

9.2.4. STDDEV_POP

Verfügbar in

DSQL, PSQL

Rückgabetyp

DOUBLE PRECISION oder NUMERIC je nach Typ von *expr*

Syntax

```
STDDEV_POP ( <expr> )
```

Tabelle 188. STDDEV_POP-*Funktionsparameter*

Parameter	Beschreibung
expr	Numerischer Ausdruck. Sie kann eine Tabellenspalte, eine Konstante, eine Variable, einen Ausdruck, eine Nicht-Aggregatfunktion oder eine UDF enthalten. Aggregatfunktionen sind als Ausdrücke nicht zulässig.

Die Funktion STDDEV_POP gibt die Populationsstandardabweichung für eine Gruppe oder ein Fenster zurück. NULL-Werte werden übersprungen.

Die Funktion STDDEV_POP(<expr>) ist äquivalent zu

```
SQRT(VAR_POP(<expr>))
```

Wenn die Gruppe oder das Fenster leer ist oder nur NULL-Werte enthält, ist das Ergebnis NULL.

STDDEV_POP-Beispiele

```
select
  dept_no
  stddev_pop(salary)
from employee
group by dept_no
```

Siehe auch

STDDEV_SAMP, VAR_POP, SQRT

9.2.5. STDDEV_SAMP

Verfügbar in

DSQL, PSQL

Rückgabetyp

DOUBLE PRECISION oder NUMERIC je nach Typ von *expr*

Syntax

```
STDDEV_POP ( <expr> )
```

Tabelle 189. STDDEV_SAMP-*Funktionsparameter*

Parameter	Beschreibung
expr	Numerischer Ausdruck. Sie kann eine Tabellenspalte, eine Konstante, eine Variable, einen Ausdruck, eine Nicht-Aggregatfunktion oder eine UDF enthalten. Aggregatfunktionen sind als Ausdrücke nicht zulässig.

Die Funktion STDDEV_SAMP gibt die Standardabweichung der Stichprobe für eine Gruppe oder ein Fenster zurück. NULL-Werte werden übersprungen.

Die Funktion STDDEV_SAMP(<expr>) ist äquivalent zu

```
SQRT(VAR_SAMP(<expr>))
```

Wenn die Gruppe oder das Fenster leer ist, nur 1 Zeile enthält oder nur NULL-Werte enthält, ist das Ergebnis NULL.

STDDEV_SAMP-Beispiele

```
select
  dept_no
  stddev_samp(salary)
from employee
group by dept_no
```

Siehe auch

STDDEV_POP, VAR_SAMP, SQRT

9.2.6. VAR_POP

Verfügbar in

DSQL, PSQL

Rückgabetyp

DOUBLE PRECISION oder NUMERIC je nach Typ von *expr*

Syntax

```
VAR_POP ( <expr> )
```

Tabelle 190. VAR_POP-*Funktionsparameter*

Parameter	Beschreibung
expr	Numerischer Ausdruck. Sie kann eine Tabellenspalte, eine Konstante, eine Variable, einen Ausdruck, eine Nicht-Aggregatfunktion oder eine UDF enthalten. Aggregatfunktionen sind als Ausdrücke nicht zulässig.

Die Funktion VAR_POP gibt die Populationsvarianz für eine Gruppe oder ein Fenster zurück. NULL -Werte werden übersprungen.

Die Funktion VAR_POP(<expr>) ist äquivalent zu

```
(SUM(<expr> * <expr>) - SUM (<expr>) * SUM (<expr>) / COUNT(<expr>))
  / COUNT (<expr>)
```

Wenn die Gruppe oder das Fenster leer ist oder nur NULL-Werte enthält, ist das Ergebnis NULL.

VAR_POP-Beispiele

```
select
  dept_no
  var_pop(salary)
from employee
```

```
group by dept_no
```

Siehe auch

VAR_SAMP, SUM(), COUNT()

9.2.7. VAR_SAMP

Verfügbar in

DSQL, PSQL

Rückgabetyp

DOUBLE PRECISION oder NUMERIC je nach Typ von *expr*

Syntax

```
VAR_SAMP ( <expr> )
```

Tabelle 191. VAR_SAMP-*Funktionsparameter*

Parameter	Beschreibung
expr	Numerischer Ausdruck. Sie kann eine Tabellenspalte, eine Konstante, eine Variable, einen Ausdruck, eine Nicht-Aggregatfunktion oder eine UDF enthalten. Aggregatfunktionen sind als Ausdrücke nicht zulässig.

Die Funktion VAR_POP gibt die Stichprobenvarianz für eine Gruppe oder ein Fenster zurück. NULL -Werte werden übersprungen.

Die Funktion VAR_SAMP(<expr>) ist äquivalent zu

```
(SUM(<expr> * <expr>) - SUM(<expr>) * SUM (<expr>) / COUNT (<expr>))
  / (COUNT(<expr>) - 1)
```

Wenn die Gruppe oder das Fenster leer ist, nur 1 Zeile enthält oder nur NULL-Werte enthält, ist das Ergebnis NULL.

VAR_SAMP-Beispiele

```
select
  dept_no
  var_samp(salary)
from employee
group by dept_no
```

Siehe auch

VAR_POP, SUM(), COUNT()

9.3. Aggregatfunktionen der linearen Regression

Lineare Regressionsfunktionen sind nützlich für die Fortsetzung von Trendlinien. Die Trend- oder Regressionslinie ist normalerweise ein Muster, dem eine Reihe von Werten folgt. Die lineare Regression ist nützlich, um zukünftige Werte vorherzusagen. Um die Regressionsgerade fortzusetzen, müssen Sie die Steigung und den Schnittpunkt mit der y-Achse kennen. Zur Berechnung dieser Werte kann ein Satz linearer Funktionen verwendet werden.

In der Funktionssyntax wird *y* als *x*-abhängige Variable interpretiert.

Die Aggregatfunktionen der linearen Regression verwenden ein Argumentpaar, den abhängigen Variablenausdruck (*y*) und den unabhängigen Variablenausdruck (*x*), die beide numerische Wertausdrücke sind. Jede Zeile, in der eines der Argumente als `NULL` ausgewertet wird, wird aus den qualifizierenden Zeilen entfernt. Wenn keine qualifizierenden Zeilen vorhanden sind, ist das Ergebnis von `REGR_COUNT 0` (Null), und die anderen Aggregatfunktionen der linearen Regression ergeben `NULL`.

9.3.1. REGR_AVGX

Verfügbar in

DSQL, PSQL

Rückgabetyp

`DOUBLE PRECISION`

Syntax

```
REGR_AVGX ( <y>, <x> )
```

Tabelle 192. `REGR_AVGX`-Funktionsparameter

Parameter	Beschreibung
y	Abhängige Variable der Regressionsgerade. Sie kann eine Tabellenspalte, eine Konstante, eine Variable, einen Ausdruck, eine Nicht-Aggregatfunktion oder eine UDF enthalten. Aggregatfunktionen sind als Ausdrücke nicht zulässig.
x	Unabhängige Variable der Regressionsgerade. Sie kann eine Tabellenspalte, eine Konstante, eine Variable, einen Ausdruck, eine Nicht-Aggregatfunktion oder eine UDF enthalten. Aggregatfunktionen sind als Ausdrücke nicht zulässig.

Die Funktion `REGR_AVGX` berechnet den Durchschnitt der unabhängigen Variablen (*x*) der Regressionsgerade.

Die Funktion `REGR_AVGX(<y>, <x>)` ist äquivalent zu

```
SUM(<exprX>) / REGR_COUNT(<y>, <x>)
```

```
<exprX> :==
  CASE WHEN <x> IS NOT NULL AND <y> IS NOT NULL THEN <x> END
```

Siehe auch

REGR_AVGY, REGR_COUNT, SUM()

9.3.2. REGR_AVGY

Verfügbar in

DSQL, PSQL

Rückgabetyp

DOUBLE PRECISION

Syntax

```
REGR_AVGY ( <y>, <x> )
```

Tabelle 193. REGR_AVGY*-Funktionsparameter*

Parameter	Beschreibung
y	Abhängige Variable der Regressionsgerade. Sie kann eine Tabellenspalte, eine Konstante, eine Variable, einen Ausdruck, eine Nicht-Aggregatfunktion oder eine UDF enthalten. Aggregatfunktionen sind als Ausdrücke nicht zulässig.
x	Unabhängige Variable der Regressionsgerade. Sie kann eine Tabellenspalte, eine Konstante, eine Variable, einen Ausdruck, eine Nicht-Aggregatfunktion oder eine UDF enthalten. Aggregatfunktionen sind als Ausdrücke nicht zulässig.

Die Funktion REGR_AVGY berechnet den Durchschnitt der abhängigen Variablen (*y*) der Regressionsgerade.

Die Funktion REGR_AVGY(<y>, <x>) ist äquivalent zu

```
SUM(<exprY>) / REGR_COUNT(<y>, <x>)

<exprY> :==
  CASE WHEN <x> IS NOT NULL AND <y> IS NOT NULL THEN <y> END
```

Siehe auch

REGR_AVGX, REGR_COUNT, SUM()

9.3.3. REGR_COUNT

Verfügbar in

DSQL, PSQL

Rückgabetyp
`DOUBLE PRECISION`

Syntax

```
REGR_COUNT ( <y>, <x> )
```

Tabelle 194. `REGR_COUNT`-*Funktionsparameter*

Parameter	Beschreibung
y	Abhängige Variable der Regressionsgerade. Sie kann eine Tabellenspalte, eine Konstante, eine Variable, einen Ausdruck, eine Nicht-Aggregatfunktion oder eine UDF enthalten. Aggregatfunktionen sind als Ausdrücke nicht zulässig.
x	Unabhängige Variable der Regressionsgerade. Sie kann eine Tabellenspalte, eine Konstante, eine Variable, einen Ausdruck, eine Nicht-Aggregatfunktion oder eine UDF enthalten. Aggregatfunktionen sind als Ausdrücke nicht zulässig.

Die Funktion `REGR_COUNT` zählt die Anzahl der nicht leeren Paare der Regressionsgerade.

Die Funktion `REGR_COUNT(<y>, <x>)` ist äquivalent zu

```
SUM(<exprXY>) / REGR_COUNT(<y>, <x>)

<exprXY> :==
  CASE WHEN <x> IS NOT NULL AND <y> IS NOT NULL THEN 1 END
```

Siehe auch
`SUM()`

9.3.4. REGR_INTERCEPT

Verfügbar in
DSQL, PSQL

Rückgabetyp
`DOUBLE PRECISION`

Syntax

```
REGR_INTERCEPT ( <y>, <x> )
```

Tabelle 195. `REGR_INTERCEPT`-*Funktionsparameter*

Parameter	Beschreibung
y	Abhängige Variable der Regressionsgerade. Sie kann eine Tabellenspalte, eine Konstante, eine Variable, einen Ausdruck, eine Nicht-Aggregatfunktion oder eine UDF enthalten. Aggregatfunktionen sind als Ausdrücke nicht zulässig.
x	Unabhängige Variable der Regressionsgerade. Sie kann eine Tabellenspalte, eine Konstante, eine Variable, einen Ausdruck, eine Nicht-Aggregatfunktion oder eine UDF enthalten. Aggregatfunktionen sind als Ausdrücke nicht zulässig.

Die Funktion `REGR_INTERCEPT` berechnet den Schnittpunkt der Regressionsgerade mit der y-Achse.

Die Funktion `REGR_INTERCEPT(<y>, <x>)` ist äquivalent zu

```
REGR_AVGY(<y>, <x>) - REGR_SLOPE(<y>, <x>) * REGR_AVGX(<y>, <x>)
```

`REGR_INTERCEPT`-Beispiele

Prognose des Verkaufsvolumens

```
with recursive years (byyear) as (
  select 1991
  from rdb$database
  union all
  select byyear + 1
  from years
  where byyear < 2020
),
s as (
  select
    extract(year from order_date) as byyear,
    sum(total_value) as total_value
  from sales
  group by 1
),
regr as (
  select
    regr_intercept(total_value, byyear) as intercept,
    regr_slope(total_value, byyear) as slope
  from s
)
select
  years.byyear as byyear,
  intercept + (slope * years.byyear) as total_value
from years
cross join regr
```

```
BYYEAR TOTAL_VALUE
------ ------------
  1991    118377.35
  1992    414557.62
  1993    710737.89
  1994   1006918.16
  1995   1303098.43
  1996   1599278.69
  1997   1895458.96
  1998   2191639.23
  1999   2487819.50
  2000   2783999.77
...
```

Siehe auch

REGR_AVGX, REGR_AVGY, REGR_SLOPE

9.3.5. REGR_R2

Verfügbar in

DSQL, PSQL

Rückgabetyp

DOUBLE PRECISION

Syntax

```
REGR_R2 ( <y>, <x> )
```

Tabelle 196. REGR_R2*-Funktionsparameter*

Parameter	Beschreibung
y	Abhängige Variable der Regressionsgerade. Sie kann eine Tabellenspalte, eine Konstante, eine Variable, einen Ausdruck, eine Nicht-Aggregatfunktion oder eine UDF enthalten. Aggregatfunktionen sind als Ausdrücke nicht zulässig.
x	Unabhängige Variable der Regressionsgerade. Sie kann eine Tabellenspalte, eine Konstante, eine Variable, einen Ausdruck, eine Nicht-Aggregatfunktion oder eine UDF enthalten. Aggregatfunktionen sind als Ausdrücke nicht zulässig.

Die Funktion REGR_R2 berechnet das Bestimmtheitsmaß oder das R-Quadrat der Regressionsgerade.

Die Funktion REGR_R2(<y>, <x>) ist äquivalent zu

```
POWER(CORR(<y>, <x>), 2)
```

Siehe auch

CORR, POWER

9.3.6. REGR_SLOPE

Verfügbar in

DSQL, PSQL

Rückgabetyp

DOUBLE PRECISION

Syntax

```
REGR_SLOPE ( <y>, <x> )
```

Tabelle 197. REGR_SLOPE-Funktionsparameter

Parameter	Beschreibung
y	Abhängige Variable der Regressionsgerade. Sie kann eine Tabellenspalte, eine Konstante, eine Variable, einen Ausdruck, eine Nicht-Aggregatfunktion oder eine UDF enthalten. Aggregatfunktionen sind als Ausdrücke nicht zulässig.
x	Unabhängige Variable der Regressionsgerade. Sie kann eine Tabellenspalte, eine Konstante, eine Variable, einen Ausdruck, eine Nicht-Aggregatfunktion oder eine UDF enthalten. Aggregatfunktionen sind als Ausdrücke nicht zulässig.

Die Funktion REGR_SLOPE berechnet die Steigung der Regressionsgerade.

Die Funktion REGR_SLOPE(<y>, <x>) ist äquivalent zu

```
COVAR_POP(<y>, <x>) / VAR_POP(<exprX>)

<exprX> :==
  CASE WHEN <x> IS NOT NULL AND <y> IS NOT NULL THEN <x> END
```

Siehe auch

COVAR_POP, VAR_POP

9.3.7. REGR_SXX

Verfügbar in

DSQL, PSQL

Rückgabetyp

`DOUBLE PRECISION`

Syntax

```
REGR_SXX ( <y>, <x> )
```

Tabelle 198. `REGR_SXX`-Funktionsparameter

Parameter	Beschreibung
y	Abhängige Variable der Regressionsgerade. Sie kann eine Tabellenspalte, eine Konstante, eine Variable, einen Ausdruck, eine Nicht-Aggregatfunktion oder eine UDF enthalten. Aggregatfunktionen sind als Ausdrücke nicht zulässig.
x	Unabhängige Variable der Regressionsgerade. Sie kann eine Tabellenspalte, eine Konstante, eine Variable, einen Ausdruck, eine Nicht-Aggregatfunktion oder eine UDF enthalten. Aggregatfunktionen sind als Ausdrücke nicht zulässig.

Die Funktion `REGR_SXX` berechnet die Quadratsumme der unabhängigen Ausdrucksvariablen (*x*).

Die Funktion `REGR_SXX(<y>, <x>)` ist äquivalent zu

```
REGR_COUNT(<y>, <x>) * VAR_POP(<exprX>)

<exprX> :==
  CASE WHEN <x> IS NOT NULL AND <y> IS NOT NULL THEN <x> END
```

Siehe auch

`REGR_COUNT`, `VAR_POP`

9.3.8. REGR_SXY

Verfügbar in

DSQL, PSQL

Rückgabetyp

`DOUBLE PRECISION`

Syntax

```
REGR_SXY ( <y>, <x> )
```

Tabelle 199. `REGR_SXY`-Funktionsparameter

Parameter	Beschreibung
y	Abhängige Variable der Regressionsgerade. Sie kann eine Tabellenspalte, eine Konstante, eine Variable, einen Ausdruck, eine Nicht-Aggregatfunktion oder eine UDF enthalten. Aggregatfunktionen sind als Ausdrücke nicht zulässig.
x	Unabhängige Variable der Regressionsgerade. Sie kann eine Tabellenspalte, eine Konstante, eine Variable, einen Ausdruck, eine Nicht-Aggregatfunktion oder eine UDF enthalten. Aggregatfunktionen sind als Ausdrücke nicht zulässig.

Die Funktion REGR_SXY berechnet die Summe der Produkte des unabhängigen Variablenausdrucks (*x*) mal des abhängigen Variablenausdrucks (*y*).

Die Funktion REGR_SXY(<y>, <x>) ist äquivalent zu

```
REGR_COUNT(<y>, <x>) * COVAR_POP(<y>, <x>)
```

Siehe auch

COVAR_POP, REGR_COUNT

9.3.9. REGR_SYY

Verfügbar in

DSQL, PSQL

Rückgabetyp

DOUBLE PRECISION

Syntax

```
REGR_SYY ( <y>, <x> )
```

Tabelle 200. REGR_SYY*-Funktionsparameter*

Parameter	Beschreibung
y	Abhängige Variable der Regressionsgerade. Sie kann eine Tabellenspalte, eine Konstante, eine Variable, einen Ausdruck, eine Nicht-Aggregatfunktion oder eine UDF enthalten. Aggregatfunktionen sind als Ausdrücke nicht zulässig.
x	Unabhängige Variable der Regressionsgerade. Sie kann eine Tabellenspalte, eine Konstante, eine Variable, einen Ausdruck, eine Nicht-Aggregatfunktion oder eine UDF enthalten. Aggregatfunktionen sind als Ausdrücke nicht zulässig.

Die Funktion REGR_SYY berechnet die Quadratsumme der abhängigen Variablen (*y*).

Die Funktion `REGR_SYY(<y>, <x>)` ist äquivalent zu

```
REGR_COUNT(<y>, <x>) * VAR_POP(<exprY>)

<exprY> :==
  CASE WHEN <x> IS NOT NULL AND <y> IS NOT NULL THEN <y> END
```

Siehe auch

REGR_COUNT, VAR_POP

Kapitel 10. Window-Funktionen (analytisch)

Gemäß der SQL-Spezifikation sind Window-Funktionen (auch als analytische Funktionen bekannt) eine Art Aggregation, die jedoch nicht die Ergebnismenge einer Abfrage ""filtert". Die Zeilen mit aggregierten Daten werden mit dem Abfrageergebnissatz gemischt.

Die Window-Funktionen werden mit der `OVER`-Klausel verwendet. Sie dürfen nur in der `SELECT`-Liste oder der `ORDER BY`-Klausel einer Abfrage erscheinen.

Neben der `OVER`-Klausel können Firebird-Window-Funktionen partitioniert und geordnet werden.

Syntax

```
<window-function> ::=
  <window-function-name> ([<expr> [, <expr> ...]]) OVER <window-specification>

<window-function-name> ::=
    <aggregate-function>
  | <ranking-function>
  | <navigational-function>

<ranking-function> ::=
  RANK | DENSE_RANK | ROW_NUMBER

<navigational-function>
  LEAD | LAG | FIRST_VALUE | LAST_VALUE | NTH_VALUE

<window-specification> ::=
  ( [ <window-partition> ] [ <window-order> ] )

<window-partition> ::=
  [PARTITION BY <expr> [, <expr> ...]]

<window-order> ::=
  [ORDER BY
    <expr> [<direction>] [<nulls placement>]
    [, <expr> [<direction>] [<nulls placement>] ...]

<direction> ::= {ASC | DESC}

<nulls placement> ::= NULLS {FIRST | LAST}
```

Tabelle 201. Argumente für Window-Funktionen

Argument	Beschreibung
expr	Ausdruck. Kann eine Tabellenspalte, Konstante, Variable, Ausdruck, Skalar- oder Aggregatfunktion enthalten. Window-Funktionen sind als Ausdruck nicht erlaubt.
aggregate_function	Eine Aggregatfunktion, die als Window-Funktion verwendet wird.

10.1. Aggregatfunktionen als Window-Funktionen

Alle Aggregatfunktionen können als Window-Funktionen verwendet werden, indem die `OVER` -Klausel hinzugefügt wird.

Stellen Sie sich eine Tabelle `EMPLOYEE` mit den Spalten `ID`, `NAME` und `SALARY` vor und die Notwendigkeit, jedem Mitarbeiter sein jeweiliges Gehalt und den Prozentsatz seines Gehalts an der Gehaltsabrechnung anzuzeigen.

Eine normale Abfrage könnte dies wie folgt erreichen:

```
select
    id,
    department,
    salary,
    salary / (select sum(salary) from employee) portion
  from employee
  order by id;
```

Ergebnisse

```
id  department  salary  portion
--  ----------  ------  ----------
1   R & D        10.00      0.2040
2   SALES        12.00      0.2448
3   SALES         8.00      0.1632
4   R & D         9.00      0.1836
5   R & D        10.00      0.2040
```

Die Abfrage ist repetitiv und langwierig, insbesondere wenn es sich bei "EMPLOYEE" um eine komplexe Ansicht handelt.

Die gleiche Abfrage könnte mit einer Window-Funktion viel schneller und eleganter angegeben werden:

```
select
    id,
    department,
    salary,
    salary / sum(salary) OVER () portion
  from employee
  order by id;
```

Hier wird `sum(salary) over()` mit der Summe aller `SALARY` aus der Abfrage (der `EMPLOYEE`-Tabelle) berechnet.

10.2. Partitionierung

Wie Aggregatfunktionen, die allein oder in Bezug auf eine Gruppe arbeiten können, können Window-Funktionen auch auf einer Gruppe arbeiten, die als "Partition" bezeichnet wird.

Syntax

```
<window function>(...) OVER (PARTITION BY <expr> [, <expr> ...])
```

Die Aggregation über eine Gruppe kann mehr als eine Zeile erzeugen, daher wird das von einer Partition generierte Resultset mit der Hauptabfrage unter Verwendung derselben Ausdrucksliste wie die Partition verknüpft.

In Fortsetzung des Beispiels `EMPLOYEE` möchten wir, anstatt den Anteil des Gehalts jedes Mitarbeiters an der Gesamtzahl aller Mitarbeiter zu erhalten, den Anteil nur basierend auf den Mitarbeitern in derselben Abteilung erhalten:

```
select
    id,
    department,
    salary,
    salary / sum(salary) OVER (PARTITION BY department) portion
  from employee
  order by id;
```

Ergebnisse

```
id  department  salary  portion
--  ----------  ------  ----------
1   R & D        10.00      0.3448
2   SALES        12.00      0.6000
3   SALES         8.00      0.4000
4   R & D         9.00      0.3103
5   R & D        10.00      0.3448
```

10.3. Sortierung

Die Unterklausel `ORDER BY` kann mit oder ohne Partitionen verwendet werden. Die `ORDER BY`-Klausel innerhalb von `OVER` gibt die Reihenfolge an, in der die Window-Funktion Zeilen verarbeitet. Diese Reihenfolge muss nicht mit den Auftragszeilen übereinstimmen, die in der Ausgabe erscheinen.

Window-funktionen haben ein wichtiges Konzept: Für jede Zeile gibt es eine Reihe von Zeilen in ihrer Partition, die als *Window-Frames* bezeichnet wird. Standardmäßig besteht der Rahmen bei der Angabe von `ORDER BY` aus allen Zeilen vom Anfang der Partition bis zur aktuellen Zeile und Zeilen, die dem aktuellen `ORDER BY`-Ausdruck entsprechen. Ohne `ORDER BY` besteht der Standardrahmen aus allen Zeilen in der Partition.

Daher erzeugt die Klausel `ORDER BY` für Standardaggregationsfunktionen bei der Verarbeitung von Zeilen Teilaggregationsergebnisse.

Beispiel

```
select
    id,
    salary,
    sum(salary) over (order by salary) cumul_salary
  from employee
  order by salary;
```

Ergebnisse

```
id  salary  cumul_salary
--  ------  ------------
3     8.00          8.00
4     9.00         17.00
1    10.00         37.00
5    10.00         37.00
2    12.00         49.00
```

Dann gibt `cumul_salary` die partielle/akkumulierte (oder laufende) Aggregation (der `SUM`-Funktion) zurück. Es mag seltsam erscheinen, dass 37,00 für die IDs 1 und 5 wiederholt wird, aber so sollte es funktionieren. Die `ORDER BY`-Schlüssel werden zusammen gruppiert und die Aggregation wird einmal berechnet (aber die beiden summieren 10,00). Um dies zu vermeiden, können Sie das Feld "ID" am Ende der Klausel "ORDER BY" hinzufügen.

Es ist möglich, mehrere Windows mit unterschiedlichen Reihenfolgen und `ORDER BY`-Teilen wie `ASC` /`DESC` und `NULLS FIRST/LAST` zu verwenden.

Bei einer Partition funktioniert `ORDER BY` genauso, aber an jeder Partitionsgrenze wird die Aggregation zurückgesetzt.

Alle Aggregationsfunktionen können `ORDER BY` verwenden, außer `LIST()`.

10.4. Ranking-Funktionen

Die Rangordnungsfunktionen berechnen den Ordinalrang einer Zeile innerhalb der Window-Partition.

Diese Funktionen können mit oder ohne Partitionierung und Ordnung verwendet werden. Sie zu verwenden, ohne sie zu bestellen, macht jedoch fast nie Sinn.

Die Rangfolgefunktionen können verwendet werden, um verschiedene Arten von inkrementellen Zählern zu erstellen. Betrachten Sie `SUM(1) OVER (ORDER BY SALARY)` als Beispiel dafür, was sie tun können, jeder auf unterschiedliche Weise. Es folgt eine Beispielabfrage, die auch mit dem Verhalten von `SUM` verglichen wird.

```
select
    id,
    salary,
    dense_rank() over (order by salary),
    rank() over (order by salary),
    row_number() over (order by salary),
    sum(1) over (order by salary)
  from employee
  order by salary;
```

Ergebnisse

```
id  salary  dense_rank  rank  row_number  sum
--  ------  ----------  ----  ----------  ---
3     8.00           1     1           1    1
4     9.00           2     2           2    2
1    10.00           3     3           3    4
5    10.00           3     3           4    4
2    12.00           4     5           5    5
```

Der Unterschied zwischen "DENSE_RANK" und "RANK" besteht darin, dass nur in "RANK" eine Lücke in Bezug auf doppelte Zeilen (relativ zur Window-Reihenfolge) vorhanden ist. `DENSE_RANK` vergibt weiterhin fortlaufende Nummern nach dem doppelten Gehalt. Andererseits vergibt `ROW_NUMBER` immer fortlaufende Nummern, auch wenn es doppelte Werte gibt.

10.4.1. DENSE_RANK

Verfügbar in

DSQL, PSQL

Ergebnistyp

`BIGINT`

Syntax

```
DENSE_RANK () OVER <window-specification>
```

Gibt den Rang von Zeilen in einer Partition einer Ergebnismenge ohne Rangordnungslücken zurück. Zeilen mit den gleichen *window-order* Werten erhalten den gleichen Rang innerhalb der Partition *window-partition*, falls angegeben. Der dichte Rang einer Zeile ist gleich der Anzahl verschiedener Rangwerte in der Partition vor der aktuellen Zeile plus eins.

DENSE_RANK-Beispiele

```
select
  id,
  salary,
```

```
  dense_rank() over (order by salary)
from employee
order by salary;
```

Ergebnis

```
id salary dense_rank
- ------ ----------
3   8.00          1
4   9.00          2
1  10.00          3
5  10.00          3
2  12.00          4
```

10.4.2. RANK

Verfügbar in

DSQL, PSQL

Ergebnistyp

`BIGINT`

Syntax

```
RANK () OVER <window-specification>
```

Gibt den Rang jeder Zeile in einer Partition der Ergebnismenge zurück. Zeilen mit den gleichen Werten von *window-order* erhalten den gleichen Rang wie in der Partition _window-partition, falls angegeben. Der Rang einer Zeile entspricht der Anzahl der Rangwerte in der Partition vor der aktuellen Zeile plus eins.

RANK-Beispiele

```
select
  id,
  salary,
  rank() over (order by salary)
from employee
order by salary;
```

Ergebnis

```
id salary rank
- ------ -----
3   8.00     1
4   9.00     2
1  10.00     3
```

```
5  10.00     3
2  12.00     5
```

Siehe auch

DENSE_RANK, ROW_NUMBER

10.4.3. ROW_NUMBER

Verfügbar in

DSQL, PSQL

Ergebnistyp

BIGINT

Syntax

```
ROW_NUMBER () OVER <window-specification>
```

Gibt die fortlaufende Zeilennummer in der Partition der Ergebnismenge zurück, wobei '1' die erste Zeile in jeder der Partitionen ist.

ROW_NUMBER-Beispiele

```
select
  id,
  salary,
  row_number() over (order by salary)
from employee
order by salary;
```

Ergebnis

```
id salary rank
- ------ -----
3   8.00     1
4   9.00     2
1  10.00     3
5  10.00     4
2  12.00     5
```

Siehe auch

DENSE_RANK, RANK

10.5. Navigationsfunktionen

Die Navigationsfunktionen rufen den einfachen (nicht aggregierten) Wert eines Ausdrucks aus

einer anderen Zeile der Abfrage innerhalb derselben Partition ab.

FIRST_VALUE, LAST_VALUE und NTH_VALUE wirken auch auf einen Window-Frame. Derzeit wendet Firebird immer einen Frame von der ersten bis zur aktuellen Zeile der Partition an, nicht bis zur letzten. Dies entspricht der Verwendung der SQL-Standardsyntax (derzeit von Firebird nicht unterstützt):

```
ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
```

Dies führt wahrscheinlich zu seltsamen oder unerwarteten Ergebnissen für "NTH_VALUE" und insbesondere "LAST_VALUE".

Firebird 4 wird Unterstützung für die Angabe des Window-Frames einführen.

Beispiel für Navigationsfunktionen

```
select
    id,
    salary,
    first_value(salary) over (order by salary),
    last_value(salary) over (order by salary),
    nth_value(salary, 2) over (order by salary),
    lag(salary) over (order by salary),
    lead(salary) over (order by salary)
  from employee
  order by salary;
```

Ergebnisse

```
id  salary  first_value  last_value  nth_value     lag    lead
--  ------  -----------  ----------  ---------  ------  ------
3     8.00         8.00        8.00     <null>  <null>    9.00
4     9.00         8.00        9.00       9.00    8.00   10.00
1    10.00         8.00       10.00       9.00    9.00   10.00
5    10.00         8.00       10.00       9.00   10.00   12.00
2    12.00         8.00       12.00       9.00   10.00  <null>
```

10.5.1. FIRST_VALUE

Verfügbar in

DSQL, PSQL

Ergebnistyp

Das gleiche wie type wie *expr*

Syntax

```
FIRST_VALUE ( <expr> ) OVER <window-specification>
```

Tabelle 202. Arguments of `FIRST_VALUE`

Argument	Beschreibung
expr	Ausdruck. Kann eine Tabellenspalte, Konstante, Variable, Ausdruck, Skalarfunktion enthalten. Aggregatfunktionen sind als Ausdruck nicht zulässig.

Gibt den ersten Wert der aktuellen Partition zurück.

Siehe auch

`LAST_VALUE`, `NTH_VALUE`

10.5.2. LAG

Verfügbar in

DSQL, PSQL

Ergebnistyp

Das gleiche wie type wie *expr*

Syntax

```
LAG ( <expr> [, <offset [, <default>]])
  OVER <window-specification>
```

Tabelle 203. Arguments of `LAG`

Argument	Beschreibung
expr	Ausdruck. Kann eine Tabellenspalte, Konstante, Variable, Ausdruck, Skalarfunktion enthalten. Aggregatfunktionen sind als Ausdruck nicht zulässig.
offset	Der Offset in Zeilen vor der aktuellen Zeile, um den durch *expr* identifizierten Wert zu erhalten. Wenn *offset* nicht angegeben ist, ist der Standardwert 1. *offset* kann eine Spalte, eine Unterabfrage oder ein anderer Ausdruck sein, der zu einem positiven ganzzahligen Wert führt, oder ein anderer Typ, der implizit in `BIGINT` konvertiert werden kann. `offset` darf nicht negativ sein (verwenden Sie stattdessen `LEAD`).
default	Der Standardwert, der zurückgegeben werden soll, wenn *offset* außerhalb der Partition zeigt. Der Standardwert ist `NULL`.

Die `LAG`-Funktion ermöglicht den Zugriff auf die Zeile in der aktuellen Partition mit einem gegebenen *Offset* vor der aktuellen Zeile.

Wenn *offset* außerhalb der aktuellen Partition zeigt, wird *default* zurückgegeben, oder `NULL`, wenn kein Standard angegeben wurde.

offset kann ein Parameter sein, aber derzeit ist eine explizite Umwandlung in `INTEGER` oder `BIGINT` erforderlich (zB `LAG(somecolumn, cast(? as bigint))`). Siehe auch CORE-6421

LAG-Beispiele

Angenommen, Sie haben die Tabelle 'RATE', in der der Wechselkurs für jeden Tag gespeichert ist. Um die Änderung des Wechselkurses in den letzten fünf Tagen zu verfolgen, können Sie die folgende Abfrage verwenden.

```
select
  bydate,
  cost,
  cost - lag(cost) over (order by bydate) as change,
  100 * (cost - lag(cost) over (order by bydate)) /
    lag(cost) over (order by bydate) as percent_change
from rate
where bydate between dateadd(-4 day to current_date)
and current_date
order by bydate
```

Ergebnis

```
bydate     cost   change percent_change
---------- ------ ------ --------------
27.10.2014  31.00 <null>         <null>
28.10.2014  31.53   0.53         1.7096
29.10.2014  31.40  -0.13        -0.4123
30.10.2014  31.67   0.27         0.8598
31.10.2014  32.00   0.33         1.0419
```

Siehe auch

`LEAD`

10.5.3. LAST_VALUE

Verfügbar in

DSQL, PSQL

Ergebnistyp

Das gleiche wie type wie *expr*

Syntax

```
LAST_VALUE ( <expr> ) OVER <window-specification>
```

Tabelle 204. Argumente für `LAST_VALUE`

Argument	Beschreibung
expr	Ausdruck. Kann eine Tabellenspalte, Konstante, Variable, Ausdruck, Skalarfunktion enthalten. Aggregatfunktionen sind als Ausdruck nicht zulässig.

Gibt den letzten Wert der aktuellen Partition zurück.

Siehe auch

`FIRST_VALUE`, `NTH_VALUE`

10.5.4. LEAD

Verfügbar in

DSQL, PSQL

Ergebnistyp

Das gleiche wie type wie *expr*

Syntax

```
LEAD ( <expr> [, <offset [, <default>]])
  OVER <window-specification>
```

Tabelle 205. Argumente für `LEAD`

Argument	Beschreibung
expr	Ausdruck. Kann eine Tabellenspalte, Konstante, Variable, Ausdruck, Skalarfunktion enthalten. Aggregatfunktionen sind als Ausdruck nicht zulässig.
offset	Der Offset in Zeilen nach der aktuellen Zeile, um den durch *expr* identifizierten Wert zu erhalten. Wenn *offset* nicht angegeben ist, ist der Standardwert 1. *offset* kann eine Spalte, eine Unterabfrage oder ein anderer Ausdruck sein, der zu einem positiven ganzzahligen Wert führt, oder ein anderer Typ, der implizit in `BIGINT` konvertiert werden kann. `offset` darf nicht negativ sein (verwenden Sie stattdessen `LAG`).
default	Der Standardwert, der zurückgegeben werden soll, wenn *offset* außerhalb der Partition zeigt. Der Standardwert ist `NULL`.

Die `LEAD`-Funktion ermöglicht den Zugriff auf die Zeile in der aktuellen Partition mit einem gegebenen *Offset* nach der aktuellen Zeile.

Wenn *offset* außerhalb der aktuellen Partition zeigt, wird *default* zurückgegeben, oder `NULL`, wenn kein Standard angegeben wurde.

offset kann ein Parameter sein, aber derzeit ist explizites Casting in `INTEGER` oder `BIGINT` erforderlich (zB `LEAD(somecolumn, cast(? as bigint))`). Siehe auch CORE-6421

Siehe auch

`LAG`

10.5.5. NTH_VALUE

Verfügbar in

DSQL, PSQL

Ergebnistyp

Das gleiche wie type wie *expr*

Syntax

```
NTH_VALUE ( <expr>, <offset> )
  [FROM {FIRST | LAST}]
  OVER <window-specification>
```

Tabelle 206. Arguments of `NTH_VALUE`

Argument	Beschreibung
expr	Ausdruck. Kann eine Tabellenspalte, Konstante, Variable, Ausdruck, Skalarfunktion enthalten. Aggregatfunktionen sind als Ausdruck nicht zulässig.
offset	Der Versatz in Zeilen vom Anfang (`FROM FIRST`) oder dem letzten (`FROM LAST`), um den durch *expr* identifizierten Wert zu erhalten. *offset* kann eine Spalte, eine Unterabfrage oder ein anderer Ausdruck sein, der zu einem positiven ganzzahligen Wert führt, oder ein anderer Typ, der implizit in `BIGINT` konvertiert werden kann. `offset` kann nicht null oder negativ sein.

Die Funktion `NTH_VALUE` gibt den *N*ten Wert ab der ersten (`FROM FIRST`) oder der letzten (`FROM LAST`) Zeile des aktuellen Frames zurück, siehe auch note on Rahmen für Navigationsfunktionen. Offset 1 mit `FROM FIRST` entspricht `FIRST_VALUE` und Offset 1 mit `FROM LAST` entspricht `LAST_VALUE`.

offset kann ein Parameter sein, aber derzeit ist explizites Casting in `INTEGER` oder `BIGINT` erforderlich (zB `LEAD(somecolumn, cast(? as bigint))`). Siehe auch CORE-6421

Siehe auch

`FIRST_VALUE`, `LAST_VALUE`

10.6. Aggregatfunktionen innerhalb der Window-Spezifikation

Es ist möglich, Aggregatfunktionen (aber keine Window-Funktionen) innerhalb der `OVER`-Klausel zu verwenden. In diesem Fall wird zuerst die Aggregatfunktion angewendet, um die Windows zu bestimmen, und erst dann werden die Window-Funktionen auf diese Window- angewendet.

Bei Verwendung von Aggregatfunktionen innerhalb von `OVER` müssen alle Spalten, die nicht in Aggregatfunktionen verwendet werden, in der `GROUP BY`-Klausel von `SELECT` angegeben werden.

Verwenden einer Aggregatfunktion in einer Window-spezifikation

```
select
  code_employee_group,
  avg(salary) as avg_salary,
  rank() over (order by avg(salary)) as salary_rank
from employee
group by code_employee_group
```

Kapitel 11. Kontextvariablen

11.1. CURRENT_CONNECTION

Verfügbar in

DSQL, PSQL

Typ
`INTEGER`

Syntax

```
CURRENT_CONNECTION
```

`CURRENT_CONNECTION` enthält den eindeutigen Bezeichner der aktuellen Verbindung.

Sein Wert wird von einem Zähler auf der Kopfseite der Datenbank abgeleitet, der bei jeder neuen Verbindung inkrementiert wird. Wenn eine Datenbank wiederhergestellt wird, wird dieser Zähler auf Null zurückgesetzt.

Beispiele

```
select current_connection from rdb$database

execute procedure P_Login(current_connection)
```

11.2. CURRENT_DATE

Verfügbar in

DSQL, PSQL, ESQL

Typ
`DATE`

Syntax

```
CURRENT_DATE
```

`CURRENT_DATE` gibt das aktuelle Serverdatum zurück.

Innerhalb eines PSQL-Moduls (Prozedur, Trigger oder ausführbarer Block) bleibt der Wert von `CURRENT_DATE` bei jedem Lesen konstant. Wenn mehrere Module sich gegenseitig aufrufen oder auslösen, bleibt der Wert während der Dauer des äußersten Moduls konstant. Wenn Sie einen fortlaufenden Wert in PSQL benötigen (z. B. um Zeitintervalle zu messen), verwenden Sie `'TODAY'`.

Beispiele

```
select current_date from rdb$database
-- Ergebnis z.B. 2011-10-03
```

11.3. CURRENT_ROLE

Verfügbar in

DSQL, PSQL

Typ
`VARCHAR(31)`

Syntax

```
CURRENT_ROLE
```

`CURRENT_ROLE` ist eine Kontextvariable, die die Rolle des aktuell verbundenen Benutzers enthält. Wenn keine aktive Rolle vorhanden ist, ist `CURRENT_ROLE 'NONE'`.

`CURRENT_ROLE` repräsentiert immer eine gültige Rolle oder `'NONE'`. Wenn sich ein Benutzer mit einer nicht vorhandenen Rolle verbindet, setzt die Engine sie stillschweigend auf "NONE" zurück, ohne einen Fehler zurückzugeben.

Beispiel

```
if (current_role <> 'MANAGER')
  then exception only_managers_may_delete;
else
  delete from Customers where custno = :custno;
```

11.4. CURRENT_TIME

Verfügbar in

DSQL, PSQL, ESQL

Typ
`TIME`

Syntax

```
CURRENT_TIME [ (<precision>) ]

<precision> ::= 0 | 1 | 2 | 3
```

Das optionale Argument *precision* wird in ESQL nicht unterstützt.

Tabelle 207. CURRENT_TIME Parameter

Parameter	Beschreibung
precision	Präzision. Der Standardwert ist 0. In ESQL nicht unterstützt.

`CURRENT_TIME` gibt die aktuelle Serverzeit zurück. Der Standardwert ist 0 Dezimalstellen, d. h. Sekundengenauigkeit.

- `CURRENT_TIME` hat eine Standardgenauigkeit von 0 Dezimalstellen, wobei `CURRENT_TIMESTAMP` eine Standardgenauigkeit von 3 Dezimalstellen hat. Daher ist `CURRENT_TIMESTAMP` nicht die genaue Summe von `CURRENT_DATE` und `CURRENT_TIME`, es sei denn, Sie geben explizit eine Genauigkeit an (d. h. `CURRENT_TIME(3)` oder `CURRENT_TIMESTAMP(0)`).
- Innerhalb eines PSQL-Moduls (Prozedur, Trigger oder ausführbarer Block) bleibt der Wert von `CURRENT_TIME` bei jedem Lesen konstant. Wenn mehrere Module sich gegenseitig aufrufen oder auslösen, bleibt der Wert während der Dauer des äußersten Moduls konstant. Wenn Sie einen fortlaufenden Wert in PSQL benötigen (z. B. um Zeitintervalle zu messen), verwenden Sie `'NOW'`.

`CURRENT_TIME` und Firebird 4 Zeitzonenunterstützung

Firebird 4 unterstützt Zeitzonen. Als Teil dieser Unterstützung wird es eine Inkompatibilität mit dem `CURRENT_TIME`-Ausdruck geben.

In Firebird 4 gibt `CURRENT_TIME` den neuen Typ `TIME WITH TIME ZONE` zurück. Damit Ihre Abfragen mit dem Datenbankcode zukünftiger Firebird-Versionen kompatibel sind, hat Firebird 3.0.4 den `LOCALTIME`-Ausdruck eingeführt. In Firebird 3.0 ist `LOCALTIME` ein Synonym für `CURRENT_TIME`.

In Firebird 4 funktioniert `LOCALTIME` weiterhin wie bisher (und gibt `TIME [WITHOUT TIME ZONE]` zurück), während `CURRENT_TIME` einen anderen Datentyp zurückgibt, `TIME WITH TIME ZONE`.

Sofern Sie nicht in der Lage sein müssen, Ihre Datenbank auf Firebird 3.0.3 oder früher herunterzustufen, empfehlen wir, `LOCALTIME` statt `CURRENT_TIME` zu verwenden.

Beispiele

```
select current_time from rdb$database
-- Ergebnis z.B. 14:20:19.0000

select current_time(2) from rdb$database
-- Ergebnis z.B. 14:20:23.1200
```

Siehe auch

`CURRENT_TIMESTAMP`, `LOCALTIME`, `LOCALTIMESTAMP`

11.5. CURRENT_TIMESTAMP

Verfügbar in

DSQL, PSQL, ESQL

Typ

`TIMESTAMP`

Syntax

```
CURRENT_TIMESTAMP [ (<precision>) ]

<precision> ::= 0 | 1 | 2 | 3
```

Das optionale Argument *precision* wird in ESQL nicht unterstützt.

Tabelle 208. CURRENT_TIMESTAMP Parameter

Parameter	Beschreibung
precision	Präzision. Der Standardwert ist 0. In ESQL nicht unterstützt.

`CURRENT_TIMESTAMP` gibt das aktuelle Serverdatum und die aktuelle Uhrzeit zurück. Der Standardwert ist 3 Dezimalstellen, d. h. Millisekunden-Genauigkeit.

- Die Standardgenauigkeit von `CURRENT_TIME` beträgt 0 Dezimalstellen, also ist `CURRENT_TIMESTAMP` nicht die genaue Summe von `CURRENT_DATE` und `CURRENT_TIME`, es sei denn, Sie geben explizit eine Genauigkeit an (zB `CURRENT_TIME(3)` oder `` `CURRENT_TIMESTAMP(0) ` ``).
- Innerhalb eines PSQL-Moduls (Prozedur, Trigger oder ausführbarer Block) bleibt der Wert von `CURRENT_TIMESTAMP` bei jedem Lesen konstant. Wenn mehrere Module sich gegenseitig aufrufen oder auslösen, bleibt der Wert während der Dauer des äußersten Moduls konstant. Wenn Sie einen fortlaufenden Wert in PSQL benötigen (z. B. um Zeitintervalle zu messen), verwenden Sie `'NOW'`.

`CURRENT_TIMESTAMP` und Firebird 4 Zeitzonenunterstützung

Firebird 4 unterstützt Zeitzonen. Als Teil dieser Unterstützung wird es eine Inkompatibilität mit dem `CURRENT_TIMESTAMP`-Ausdruck geben.

In Firebird 4 gibt "CURRENT_TIMESTAMP" den neuen Typ "TIMESTAMP WITH TIME ZONE" zurück. Damit Ihre Abfragen mit dem Datenbankcode zukünftiger Firebird-Versionen kompatibel sind, hat Firebird 3.0.4 den `LOCALTIMESTAMP` -Ausdruck eingeführt. In Firebird 3.0 ist `LOCALTIMESTAMP` ein Synonym für `CURRENT_TIMESTAMP`.

In Firebird 4 funktioniert `LOCALTIMESTAMP` weiterhin wie bisher (gibt `TIMESTAMP [WITHOUT TIME ZONE]` zurück), während `CURRENT_TIMESTAMP` einen anderen Datentyp zurückgibt, `TIMESTAMP WITH TIME ZONE`.

> Sofern Sie nicht in der Lage sein müssen, Ihre Datenbank auf Firebird 3.0.3 oder früher herunterzustufen, empfehlen wir, LOCALTIMESTAMP statt CURRENT_TIMESTAMP zu verwenden.

Beispiele

```
select current_timestamp from rdb$database
-- Ergebnis z.B. 2008-08-13 14:20:19.6170

select current_timestamp(2) from rdb$database
-- Ergebnis z.B. 2008-08-13 14:20:23.1200
```

Siehe auch

CURRENT_TIME, LOCALTIME, LOCALTIMESTAMP

11.6. CURRENT_TRANSACTION

Verfügbar in

DSQL, PSQL

Typ

BIGINT

Syntax

```
CURRENT_TRANSACTION
```

CURRENT_TRANSACTION enthält die eindeutige Kennung der aktuellen Transaktion.

Sein Wert wird von einem Zähler auf der Kopfseite der Datenbank abgeleitet, der bei jeder neuen Transaktion inkrementiert wird. Wenn eine Datenbank wiederhergestellt wird, wird dieser Zähler auf Null zurückgesetzt.

Beispiele

```
select current_transaction from rdb$database

New.Txn_ID = current_transaction;
```

11.7. CURRENT_USER

Verfügbar in

DSQL, PSQL

Typ

VARCHAR(31)

Syntax

```
CURRENT_USER
```

CURRENT_USER ist eine Kontextvariable, die den Namen des aktuell verbundenen Benutzers enthält. Es ist völlig äquivalent zu USER.

Beispiel

```
create trigger bi_customers for customers before insert as
begin
    New.added_by  = CURRENT_USER;
    New.purchases = 0;
end
```

11.8. DELETING

Verfügbar in

PSQL

Typ

BOOLEAN

Syntax

```
DELETING
```

Nur in Triggern verfügbar, DELETING zeigt an, ob der Trigger für eine DELETE-Operation ausgelöst wurde. Vorgesehen für die Verwendung in multi-action triggers-de.

Beispiel

```
if (deleting) then
begin
  insert into Removed_Cars (id, make, model, removed)
    values (old.id, old.make, old.model, current_timestamp);
end
```

11.9. GDSCODE

Verfügbar in

PSQL

Typ

INTEGER

Syntax

```
GDSCODE
```

In einem "`WHEN ⋯ DO`"-Fehlerbehandlungsblock enthält die Kontextvariable `GDSCODE` die numerische Darstellung des aktuellen Firebird-Fehlercodes. Vor Firebird 2.0 wurde `GDSCODE` nur in `WHEN GDSCODE` -Handlern gesetzt. Jetzt kann es auch in den Blöcken `WHEN ANY`, `WHEN SQLCODE`, `WHEN SQLSTATE` und `WHEN EXCEPTION` ungleich Null sein, vorausgesetzt, die den Fehler auslösende Bedingung entspricht einem Firebird-Fehlercode. Außerhalb von Fehlerhandlern ist `GDSCODE` immer 0. Außerhalb von PSQL existiert es überhaupt nicht.

Nach `WHEN GDSCODE` müssen Sie symbolische Namen wie `grant_obj_notfound` usw. verwenden. Aber die Kontextvariable `GDSCODE` ist ein `INTEGER`. Wenn Sie es mit einem bestimmten Fehler vergleichen möchten, muss der Zahlenwert verwendet werden, z. `335544551` für `grant_obj_notfound`.

Beispiel

```
when gdscode grant_obj_notfound, gdscode grant_fld_notfound,
   gdscode grant_nopriv, gdscode grant_nopriv_on_base
do
begin
  execute procedure log_grant_error(gdscode);
  exit;
end
```

11.10. INSERTING

Verfügbar in

PSQL

Typ

`BOOLEAN`

Syntax

```
INSERTING
```

Nur in Triggern verfügbar, zeigt `INSERTING` an, ob der Trigger aufgrund einer `INSERT`-Operation ausgelöst wurde. Vorgesehen für die Verwendung in Multi-Action-Trigger. .Beispiel

```
if (inserting or updating) then
begin
  if (new.serial_num is null) then
    new.serial_num = gen_id(gen_serials, 1);
end
```

11.11. LOCALTIME

Verfügbar in

DSQL, PSQL, ESQL

Typ

TIME

Syntax

```
LOCALTIME [ (<precision>) ]

<precision> ::= 0 | 1 | 2 | 3
```

Das optionale Argument *precision* wird in ESQL nicht unterstützt.

Tabelle 209. LOCALTIME-Parameter

Parameter	Beschreibung
precision	Präzision. Der Standardwert ist 0. In ESQL nicht unterstützt

LOCALTIME gibt die aktuelle Serverzeit zurück. Der Standardwert ist 0 Dezimalstellen, d. h. Sekundengenauigkeit.

- LOCALTIME wurde in Firebird 3.0.4 als Alias von CURRENT_TIME eingeführt. In Firebird 4 gibt CURRENT_TIME eine TIME WITH TIME ZONE anstelle einer TIME [Without TIME ZONE] zurück, während LOCALTIME weiterhin TIME [Without TIME ZONE] zurückgibt. Es wird empfohlen, von CURRENT_TIME auf LOCALTIME zu wechseln, um die Vorwärtskompatibilität mit Firebird 4 zu gewährleisten.
- LOCALTIME hat eine Standardgenauigkeit von 0 Dezimalstellen, wobei LOCALTIMESTAMP eine Standardgenauigkeit von 3 Dezimalstellen hat. Daher ist LOCALTIMESTAMP nicht die genaue Summe von CURRENT_DATE und LOCALTIME, es sei denn, Sie geben explizit eine Genauigkeit an (d. h. LOCALTIME(3) oder LOCALTIMESTAMP(0)).
- Innerhalb eines PSQL-Moduls (Prozedur, Trigger oder ausführbarer Block) bleibt der Wert von LOCALTIME bei jedem Lesen konstant. Wenn sich mehrere Module gegenseitig aufrufen oder auslösen, bleibt der Wert während der Dauer des äußersten Moduls konstant. Wenn Sie in PSQL einen fortlaufenden Wert benötigen (z. B. um Zeitintervalle zu messen), verwenden Sie 'NOW'.====

Beispiele

```
select localtime from rdb$database
-- Ergebnis z.B. 14:20:19.0000

select localtime(2) from rdb$database
-- Ergebnis z.B. 14:20:23.1200
```

Siehe auch

`CURRENT_TIME`, `LOCALTIMESTAMP`

11.12. LOCALTIMESTAMP

Verfügbar in

DSQL, PSQL, ESQL

Typ

`TIMESTAMP`

Syntax

```
LOCALTIMESTAMP [ (<precision>) ]

<precision> ::= 0 | 1 | 2 | 3
```

Das optionale Argument *precision* wird in ESQL nicht unterstützt.

Tabelle 210. `LOCALTIMESTAMP` *Parameter*

Parameter	Beschreibung
precision	Präzision. Der Standardwert ist 3. In ESQL nicht unterstützt

`LOCALTIMESTAMP` gibt das aktuelle Serverdatum und die aktuelle Uhrzeit zurück. Der Standardwert ist 3 Dezimalstellen, d. h. Millisekunden-Genauigkeit.

- `LOCALTIMESTAMP` wurde in Firebird 3.0.4 als Synonym von `CURRENT_TIMESTAMP` eingeführt. In Firebird 4 gibt `CURRENT_TIMESTAMP` einen `TIMESTAMP WITH TIME ZONE` anstelle eines `TIMESTAMP [Without TIME ZONE]` zurück, während `LOCALTIMESTAMP` weiterhin `TIMESTAMP [OHNE TIME ZONE]` zurückgibt. Es wird empfohlen, von `CURRENT_TIMESTAMP` zu `LOCALTIMESTAMP` zu wechseln, um die Vorwärtskompatibilität mit Firebird 4 zu gewährleisten.
- Die Standardgenauigkeit von `LOCALTIME` beträgt 0 Dezimalstellen, also ist `LOCALTIMESTAMP` nicht die genaue Summe von `CURRENT_DATE` und `LOCALTIME`, es sei denn, Sie geben explizit eine Genauigkeit an (zB `LOCATIME(3)` oder `` `LOCALTIMESTAMP(0) ` ``).
- Innerhalb eines PSQL-Moduls (Prozedur, Trigger oder ausführbarer Block) bleibt der Wert von `LOCALTIMESTAMP` bei jedem Lesen konstant. Wenn sich mehrere Module gegenseitig aufrufen oder auslösen, bleibt der Wert während der Dauer des äußersten Moduls konstant. Wenn Sie in PSQL einen fortlaufenden Wert benötigen (z. B. um Zeitintervalle zu messen), verwenden Sie `'NOW'`.

Beispiele

```
select localtimestamp from rdb$database
```

```
-- Ergebnis z.B. 2008-08-13 14:20:19.6170

select localtimestamp(2) from rdb$database
-- Ergebnis z.B. 2008-08-13 14:20:23.1200
```

Siehe auch

CURRENT_TIMESTAMP, LOCALTIME

11.13. NEW

Verfügbar in

PSQL, nur Trigger

Typ

Datensatz

Syntax

```
NEW.column_name
```

Tabelle 211. NEW-Parameter

Parameter	Beschreibung
column_name	Spaltenname für den Zugriff

NEU enthält die neue Version eines Datenbankeintrags, der gerade eingefügt oder aktualisiert wurde. Ab Firebird 2.0 ist es in 'AFTER'-Triggern schreibgeschützt.

Bei Multi-Action-Triggern — eingeführt in Firebird 1.5 — ist NEU immer verfügbar. Wird der Trigger jedoch durch ein DELETE ausgelöst, gibt es keine neue Version des Datensatzes. In dieser Situation wird beim Lesen von NEW immer NULL zurückgegeben; das Schreiben in sie führt zu einer Laufzeitausnahme.

11.14. 'NOW'

Verfügbar in

DSQL, PSQL, ESQL

Typ

CHAR(3)

'NOW' ist keine Variable, sondern ein String-Literal. Es ist jedoch in dem Sinne besonders, dass Sie, wenn Sie es in einen Datums-/Uhrzeittyp CAST() geben, das aktuelle Datum und/oder die aktuelle Uhrzeit erhalten. Seit Firebird 2.0 beträgt die Genauigkeit 3 Dezimalstellen, also Millisekunden. Bei 'NOW' wird die Groß-/Kleinschreibung nicht beachtet und die Engine ignoriert führende oder nachfolgende Leerzeichen beim Casting.

Bitte beachten Sie, dass die Abkürzungsausdrücke sofort beim Parsen ausgewertet werden und gleich bleiben, solange die Anweisung vorbereitet ist. Somit bleibt auch bei mehrfacher Ausführung einer Abfrage der Wert für z.B. “`timestamp 'now'`” ändert sich nicht, egal wie viel Zeit vergeht. Wenn Sie den Wert benötigen, um fortzufahren (d. h. bei jedem Anruf ausgewertet werden), verwenden Sie eine vollständige Besetzung.

- `'NOW'` gibt immer das aktuelle Datum/die aktuelle Uhrzeit zurück, auch in PSQL-Modulen, wo `CURRENT_DATE`, `CURRENT_TIME` und << fblangref30-contextvars-current-timestamp-de>> gibt während der gesamten Dauer der äußersten Routine den gleichen Wert zurück. Dies macht `'NOW'` zum Messen von Zeitintervallen in Triggern, Prozeduren und ausführbaren Blöcken nützlich.
- Außer in der oben genannten Situation, Lesen von `CURRENT_DATE`, `CURRENT_TIME` und `CURRENT_TIMESTAMP` ist im Allgemeinen der Übertragung von `'JETZT'` vorzuziehen. Beachten Sie jedoch, dass `CURRENT_TIME` standardmäßig auf Sekunden genau eingestellt ist; Um eine Genauigkeit in Millisekunden zu erhalten, verwenden Sie `CURRENT_TIME(3)`.

Beispiele

```
select 'Now' from rdb$database
-- Ergebnis 'Now'

select cast('Now' as date) from rdb$database
-- Ergebnis z.B. 2008-08-13

select cast('now' as time) from rdb$database
-- Ergebnis z.B. 14:20:19.6170

select cast('NOW' as timestamp) from rdb$database
-- Ergebnis z.B. 2008-08-13 14:20:19.6170
```

Kurzformumwandlungen für Datums- und Uhrzeitdatentypen für die letzten drei Aussagen:

```
select date 'Now' from rdb$database
select time 'now' from rdb$database
select timestamp 'NOW' from rdb$database
```

11.15. OLD

Verfügbar in

PSQL, nur Trigger

Typ

Datensatz

Syntax

```
OLD.column_name
```

Tabelle 212. OLD Parameters

Parameter	Beschreibung
column_name	Spaltenname für den Zugriff

OLD enthält die vorhandene Version eines Datenbankeintrags kurz vor einer Löschung oder Aktualisierung. Ab Firebird 2.0 ist es schreibgeschützt.

Bei Multi-Action-Triggern — eingeführt in Firebird 1.5 — ist 'OLD' immer verfügbar. Wenn der Trigger jedoch durch ein 'INSERT' ausgelöst wird, gibt es offensichtlich keine bereits vorhandene Version des Datensatzes. In dieser Situation wird beim Lesen von OLD immer NULL zurückgegeben; das Schreiben in sie führt zu einer Laufzeitausnahme.

11.16. ROW_COUNT

Verfügbar in

PSQL

Typ

INTEGER

Syntax

```
ROW_COUNT
```

Die Kontextvariable ROW_COUNT enthält die Anzahl der Zeilen, die von der letzten DML-Anweisung (INSERT, UPDATE, DELETE, SELECT oder FETCH) im aktuellen Trigger, in der gespeicherten Prozedur oder im ausführbaren Block betroffen sind.

Verhalten bei SELECT *und* FETCH

- Nach einem Singleton SELECT ist ROW_COUNT 1, wenn eine Datenzeile abgerufen wurde, andernfalls 0.
- In einer FOR SELECT Schleife wird ROW_COUNT bei jeder Iteration inkrementiert (beginnend bei 0 vor der ersten).
- Nach einem FETCH von einem Cursor ist ROW_COUNT 1, wenn eine Datenzeile abgerufen wurde, andernfalls 0. Wenn mehr Datensätze vom gleichen Cursor abgerufen werden, wird ROW_COUNT *nicht* über 1 hinaus erhöht.
- In Firebird 1.5.x ist ROW_COUNT 0 nach jeder Art von SELECT-Anweisung.

ROW_COUNT kann nicht verwendet werden, um die Anzahl der Zeilen zu bestimmen, die von einem EXECUTE STATEMENT- oder EXECUTE PROCEDURE-Befehl betroffen sind.

Beispiel

```
update Figures set Number = 0 where id = :id;
if (row_count = 0) then
  insert into Figures (id, Number) values (:id, 0);
```

11.17. SQLCODE

Verfügbar in

PSQL

Eingestellt in

2.5.1

Typ

`INTEGER`

Syntax

```
SQLCODE
```

In einem "`WHEN ⋯ DO`"-Fehlerbehandlungsblock enthält die Kontextvariable `SQLCODE` den aktuellen SQL-Fehlercode. Vor Firebird 2.0 wurde `SQLCODE` nur in den `WHEN SQLCODE`- und `WHEN ANY`-Handlern gesetzt. Sie darf jetzt auch in den Blöcken `WHEN GDSCODE`, `WHEN SQLSTATE` und `WHEN EXCEPTION` ungleich Null sein, sofern die fehlerauslösende Bedingung einem SQL-Fehlercode entspricht. Außerhalb von Fehlerhandlern ist `SQLCODE` immer 0. Außerhalb von PSQL existiert es überhaupt nicht.

`SQLCODE` wird nun zugunsten des SQL-2003-kompatiblen Statuscodes `SQLSTATE` veraltet. Die Unterstützung für `SQLCODE` und `WHEN SQLCODE` wird in einer zukünftigen Version von Firebird eingestellt.

Beispiel

```
when any
do
begin
  if (sqlcode <> 0) then
    Msg = 'An SQL error occurred!';
  else
    Msg = 'Something bad happened!';
  exception ex_custom Msg;
end
```

11.18. SQLSTATE

Verfügbar in

PSQL

Aufgenommen in

2.5.1

Typ

`CHAR(5)`

Syntax

```
SQLSTATE
```

In einer "`WHEN … DO`"-Fehlerbehandlung enthält die Kontextvariable `SQLSTATE` den 5-stelligen, SQL-2003-konformen Statuscode, der sich aus der Anweisung ergibt, die den Fehler ausgelöst hat. Außerhalb von Fehlerhandlern ist `SQLSTATE` immer `'00000'`. Außerhalb von PSQL ist es überhaupt nicht verfügbar.

- `SQLSTATE` soll `SQLCODE` ersetzen. Letzteres ist jetzt in Firebird veraltet und wird in einer zukünftigen Version verschwinden.
- Firebird unterstützt (noch) nicht die Syntax "`WHEN SQLSTATE … DO`". Sie müssen `WHEN ANY` verwenden und die Variable `SQLSTATE` innerhalb des Handlers testen.
- Jeder `SQLSTATE`-Code ist die Verkettung einer 2-Zeichen-Klasse und einer 3-Zeichen-Unterklasse. Die Klassen *00* (erfolgreicher Abschluss), *01* (Warnung) und *02* (keine Daten) repräsentieren *Abschlussbedingungen*. Jeder Statuscode außerhalb dieser Klassen ist ein *Exception*. Da die Klassen *00*, *01* und *02* keinen Fehler auslösen, werden sie niemals in der Variablen `SQLSTATE` angezeigt.
- Eine vollständige Liste der `SQLSTATE`-Codes finden Sie im Abschnitt SQLSTATE-Codes und Nachrichtentexte in *Anhang B: Ausnahmebehandlung, Codes und Nachrichten*.

Beispiel

```
when any
do
begin
  Msg = case sqlstate
          when '22003' then 'Numeric value out of range.'
          when '22012' then 'Division by zero.'
          when '23000' then 'Integrity constraint violation.'
          else 'Something bad happened! SQLSTATE = ' || sqlstate
        end;
  exception ex_custom Msg;
end
```

11.19. 'TODAY'

Verfügbar in

DSQL, PSQL, ESQL

Typ

`CHAR(5)`

`'TODAY'` ist keine Variable, sondern ein String-Literal. Es ist jedoch in dem Sinne besonders, dass Sie, wenn Sie es in einen Datums-/Uhrzeittyp `CAST()` geben, das aktuelle Datum erhalten. Bei `'TODAY'` wird die Groß-/Kleinschreibung nicht beachtet und die Engine ignoriert führende oder nachfolgende Leerzeichen beim Casting.

- `'TODAY'` Ergebnis immer das aktuelle Datum, auch in PSQL-Modulen, wo `CURRENT_DATE`, `CURRENT_TIME` und `CURRENT_TIMESTAMP` gibt während der gesamten Dauer der äußersten Routine denselben Wert zurück. Dies macht `'TODAY'` nützlich, um Zeitintervalle in Triggern, Prozeduren und ausführbaren Blöcken zu messen (zumindest wenn Ihre Prozeduren tagelang laufen).
- Außer in der oben erwähnten Situation ist das Lesen von `CURRENT_DATE` im Allgemeinen vorzuziehen, `'NOW'` zu gießen.

Beispiele

```
select 'Today' from rdb$database
-- Ergebnis 'Today'

select cast('Today' as date) from rdb$database
-- Ergebnis z.B. 2011-10-03

select cast('TODAY' as timestamp) from rdb$database
-- Ergebnis z.B. 2011-10-03 00:00:00.0000
```

Kurzschreibweise für die letzten beiden Anweisungen:

```
select date 'Today' from rdb$database;
select timestamp 'TODAY' from rdb$database;
```

11.20. 'TOMORROW'

Verfügbar in

DSQL, PSQL, ESQL

Typ

`CHAR(8)`

`'TOMORROW'` ist keine Variable, sondern ein String-Literal. Es ist jedoch in dem Sinne besonders, dass

Sie, wenn Sie es in einen Datums-/Uhrzeittyp `CAST()` geben, das Datum des nächsten Tages erhalten. Siehe auch `'TODAY'`.

Beispiele

```
select 'Tomorrow' from rdb$database
-- Ergebnis 'Tomorrow'

select cast('Tomorrow' as date) from rdb$database
-- Ergebnis z.B. 2011-10-04

select cast('TOMORROW' as timestamp) from rdb$database
-- Ergebnis z.B. 2011-10-04 00:00:00.0000
```

Kurzschreibweise für die letzten beiden Anweisungen:

```
select date 'Tomorrow' from rdb$database;
select timestamp 'TOMORROW' from rdb$database;
```

11.21. UPDATING

Verfügbar in

PSQL

Typ

`BOOLEAN`

Syntax

```
UPDATING
```

Nur in Triggern verfügbar, 'UPDATING' zeigt an, ob der Trigger aufgrund einer 'UPDATE'-Operation ausgelöst wurde. Vorgesehen für die Verwendung in Multi-Action-Trigger.

Beispiel

```
if (inserting or updating) then
begin
  if (new.serial_num is null) then
    new.serial_num = gen_id(gen_serials, 1);
end
```

11.22. 'YESTERDAY'

Verfügbar in

DSQL, PSQL, ESQL

Typ
`CHAR(9)`

`'YESTERDAY'` ist keine Variable, sondern ein String-Literal. Es ist jedoch in dem Sinne besonders, dass Sie, wenn Sie es in einen Datums-/Uhrzeittyp `CAST()` geben, das Datum des Vortages erhalten. Siehe auch `'TODAY'`.

Beispiele

```
select 'Yesterday' from rdb$database
-- Ergebnis 'Yesterday'

select cast('Yesterday as date) from rdb$database
-- Ergebnis z.B. 2011-10-02

select cast('YESTERDAY' as timestamp) from rdb$database
-- Ergebnis z.B. 2011-10-02 00:00:00.0000
```

Kurzschreibweise für die letzten beiden Anweisungen:

```
select date 'Yesterday' from rdb$database;
select timestamp 'YESTERDAY' from rdb$database;
```

11.23. USER

Verfügbar in
DSQL, PSQL

Typ
`VARCHAR(31)`

Syntax

```
USER
```

`USER` ist eine Kontextvariable, die den Namen des aktuell verbundenen Benutzers enthält. Es entspricht vollständig `CURRENT_USER`.

Beispiel

```
create trigger bi_customers for customers before insert as
begin
  New.added_by  = USER;
  New.purchases = 0;
end
```

Kapitel 12. Transaktionssteuerung

Alles in Firebird geschieht in Transaktionen. Arbeitseinheiten werden zwischen einem Startpunkt und einem Endpunkt isoliert. Änderungen an Daten bleiben reversibel, bis die Client-Anwendung den Server anweist, sie festzuschreiben.

12.1. Transaktionsanweisungen

Firebird verfügt über ein kleines Lexikon von SQL-Anweisungen, die von Clientanwendungen verwendet werden, um die Transaktionen zu starten, zu verwalten, festzuschreiben und rückgängig zu machen (Rollback), die die Grenzen aller Datenbankaufgaben bilden:

SET TRANSACTION
: zum Konfigurieren und Starten einer Transaktion

COMMIT
: das Ende einer Arbeitseinheit signalisieren und Änderungen dauerhaft in die Datenbank schreiben

ROLLBACK
: um die in der Transaktion vorgenommenen Änderungen rückgängig zu machen

SAVEPOINT
: um eine Position im Arbeitsprotokoll zu markieren, falls ein teilweises Rollback erforderlich ist

RELEASE SAVEPOINT
: einen Speicherpunkt löschen

12.1.1. SET TRANSACTION

Verwendet für

Transaktion konfigurieren und starten

Verfügbar in

DSQL, ESQL

Syntax

```
SET TRANSACTION
   [NAME tr_name]
   [<tr_option> ...]

<tr_option> ::=
     READ {ONLY | WRITE}
   | [NO] WAIT
   | [ISOLATION LEVEL]
       { SNAPSHOT [TABLE [STABILITY]]
       | READ {UNCOMMITED | COMMITTED }
```

```
          [[NO] RECORD_VERSION] }
    | NO AUTO UNDO
    | RESTART REQUESTS
    | IGNORE LIMBO
    | LOCK TIMEOUT seconds
    | RESERVING <tables>
    | USING <dbhandles>

<tables> ::= <table_spec> [, <table_spec> ...]

<table_spec> ::= tablename [, tablename ...]
      [FOR [SHARED | PROTECTED] {READ | WRITE}]

<dbhandles> ::= dbhandle [, dbhandle ...]
```

Tabelle 213. `SET TRANSACTION`-*Anweisungsparameter*

Parameter	Beschreibung
tr_name	Transaktionsname. Nur in ESQL verfügbar
tr_option	Optionale Transaktionsoption. Jede Option sollte höchstens einmal angegeben werden, einige Optionen schließen sich gegenseitig aus (z.B. `READ ONLY` vs `READ WRITE`, `WAIT` vs `NO WAIT`)
seconds	Die Zeit in Sekunden, die die Anweisung warten soll, falls ein Konflikt auftritt. Muss größer oder gleich `0` sein.
tables	Die Liste der zu reservierenden Tische
dbhandles	Die Liste der Datenbanken, auf die die Datenbank zugreifen kann. Nur in ESQL verfügbar
table_spec	Spezifikationen für Tabellenreservierungen
tablename	Der Name der zu reservierenden Tabelle
dbhandle	Das Handle der Datenbank, auf die die Datenbank zugreifen kann. Nur in ESQL verfügbar

Die Anweisung `SET TRANSACTION` konfiguriert die Transaktion und startet sie. In der Regel starten nur Client-Anwendungen Transaktionen. Die Ausnahmen sind die Fälle, in denen der Server eine autonome Transaktion oder Transaktionen für bestimmte Hintergrundsystem-Threads/-Prozesse startet, wie z. B. Sweeps.

Eine Clientanwendung kann eine beliebige Anzahl gleichzeitig laufender Transaktionen starten. Eine einzelne Verbindung kann über mehrere gleichzeitig aktive Transaktionen verfügen (obwohl dies nicht von allen Treibern oder Zugriffskomponenten unterstützt wird). Für die Gesamtzahl der ausgeführten Transaktionen in allen Clientanwendungen, die mit einer bestimmten Datenbank arbeiten, gibt es ab dem Zeitpunkt, an dem die Datenbank aus ihrer Sicherungskopie wiederhergestellt wurde, oder ab dem Zeitpunkt, an dem die Datenbank ursprünglich erstellt wurde, eine Grenze. Das Limit beträgt 2^{48} — 281.474.976.710.656 — in Firebird 3 und 2^{31}-1 — oder 2.147.483.647 — in früheren Versionen.

Alle Klauseln in der `SET TRANSACTION`-Anweisung sind optional. Wenn die Anweisung, die eine Transaktion startet, keine Klauseln enthält, wird die Transaktion mit Standardwerten für Zugriffsmodus, Sperrenauflösungsmodus und Isolationsstufe gestartet, die wie folgt lauten:

```
SET TRANSACTION
  READ WRITE
  WAIT
  ISOLATION LEVEL SNAPSHOT;
```

Datenbanktreiber oder Zugriffskomponenten können andere Standardwerte verwenden. Weitere Informationen finden Sie in deren Dokumentation.

Der Server weist Transaktionen sequentiell Ganzzahlen zu. Immer wenn ein Client eine Transaktion startet, entweder explizit definiert oder standardmäßig, sendet der Server die Transaktions-ID an den Client. Diese Nummer kann in SQL mit der Kontextvariablen `CURRENT_TRANSACTION` abgerufen werden.

Einige Datenbanktreiber – oder ihre maßgeblichen Spezifikationen – erfordern, dass Sie die Transaktion über API-Methoden konfigurieren und starten. In diesem Fall wird die Verwendung von `SET TRANSACTION` entweder nicht unterstützt oder kann zu unspezifischem Verhalten führen. Ein Beispiel hierfür ist JDBC und der Firebird JDBC-Treiber Jaybird.

Weitere Informationen finden Sie in der Dokumentation Ihres Treibers.

Die Klauseln `NAME` und `USING` sind nur in ESQL gültig.

Transaktionsname

Das optionale Attribut `NAME` definiert den Namen einer Transaktion. Die Verwendung dieses Attributs ist nur in Embedded SQL verfügbar. In ESQL-Anwendungen ermöglichen benannte Transaktionen, dass mehrere Transaktionen gleichzeitig in einer Anwendung aktiv sind. Wenn benannte Transaktionen verwendet werden, muss für jede benannte Transaktion eine Hostsprachenvariable mit demselben Namen deklariert und initialisiert werden. Dies ist eine Einschränkung, die eine dynamische Angabe von Transaktionsnamen verhindert und somit eine Transaktionsbenennung in DSQL ausschließt.

Transaction Parameters

Die wichtigsten Parameter einer Transaktion sind:

- Datenzugriffsmodus (`READ WRITE`, `READ ONLY`)
- Auflösungsmodus sperren (`WAIT`, `NO WAIT`) mit einer optionalen `LOCK TIMEOUT` Spezifikation
- Isolationsstufe (`READ COMMITTED`, `SNAPSHOT`, `SNAPSHOT TABLE STABILITY`).

Die Isolationsstufe `READ UNCOMMITTED` ist ein Synonym für `READ COMMITTED` und wird nur aus Gründen der Syntaxkompatibilität bereitgestellt. Es bietet genau

> dieselbe Semantik wie `READ COMMITTED` und erlaubt Ihnen nicht, nicht festgeschriebene Änderungen anderer Transaktionen anzuzeigen.

- ein Mechanismus zum Reservieren oder Freigeben von Tabellen (die `RESERVING`-Klausel)

Zugriffsmodus

Die beiden Datenbankzugriffsmodi für Transaktionen sind `READ WRITE` und `READ ONLY`.

- Wenn der Zugriffsmodus `READ WRITE` ist, können Operationen im Kontext dieser Transaktion sowohl Leseoperationen als auch Datenaktualisierungsoperationen sein. Dies ist der Standardmodus.
- Wenn der Zugriffsmodus `READ ONLY` ist, können im Kontext dieser Transaktion nur `SELECT` -Operationen ausgeführt werden. Jeder Versuch, Daten im Kontext einer solchen Transaktion zu ändern, führt zu Datenbankausnahmen. Dies gilt jedoch nicht für globale temporäre Tabellen (GTT), die in `READ ONLY`-Transaktionen geändert werden dürfen, siehe *Globale temporäre Tabellen (GTT)* im Kapitel *Daten Definitions-(DDL)-Anweisungen* für Details.

Lock Resolution-Modus

Wenn mehrere Clientprozesse mit derselben Datenbank arbeiten, können Sperren auftreten, wenn ein Prozess nicht festgeschriebene Änderungen in einer Tabellenzeile vornimmt oder eine Zeile löscht und ein anderer Prozess versucht, dieselbe Zeile zu aktualisieren oder zu löschen. Solche Sperren werden als *Aktualisierungskonflikte* bezeichnet.

Sperren können in anderen Situationen auftreten, wenn mehrere Transaktionsisolationsstufen verwendet werden.

Die beiden Lock-Auflösungsmodi sind `WAIT` und `NO WAIT`.

`WAIT`-Modus

Wenn im `WAIT`-Modus (dem Standardmodus) ein Konflikt zwischen zwei parallelen Prozessen auftritt, die gleichzeitige Datenaktualisierungen in derselben Datenbank ausführen, wartet eine `WAIT`-Transaktion, bis die andere Transaktion beendet ist — durch Festschreiben (`COMMIT`) oder Rollback (`ROLLBACK`). Die Client-Anwendung mit der Transaktion `WAIT` wird angehalten, bis der Konflikt gelöst ist.

Wenn für die Transaktion `WAIT` ein `LOCK TIMEOUT` angegeben ist, wird nur für die in dieser Klausel angegebene Anzahl von Sekunden gewartet. Wenn die Sperre am Ende des angegebenen Intervalls nicht aufgelöst wird, wird die Fehlermeldung “Lock timeout on wait transaction” an den Client zurückgegeben.

Das Verhalten der Sperrenauflösung kann je nach Transaktionsisolationsstufe geringfügig variieren.

`NO WAIT` Mode

In the `NO WAIT` mode, a transaction will immediately throw a database exception if a conflict occurs.

`LOCK TIMEOUT` ist eine separate Transaktionsoption, kann aber nur für `WAIT`-Transaktionen verwendet werden. Die Angabe von `LOCK TIMEOUT` mit einer `NO WAIT`-Transaktion führt zum Fehler "*invalid parameter in transaction parameter block -Option isc_tpb_lock_timeout is not valid if isc_tpb_nowait was used previously in TPB`*".

Isolationsstufe

Bei der Isolation geht es darum, die Arbeit einer Datenbankaufgabe von anderen getrennt zu halten. Änderungen, die von einer Anweisung vorgenommen werden, werden für alle verbleibenden Anweisungen sichtbar, die innerhalb derselben Transaktion ausgeführt werden, unabhängig von ihrer Isolationsstufe. Änderungen, die in anderen Transaktionen ausgeführt werden, bleiben für die aktuelle Transaktion unsichtbar, solange sie nicht festgeschrieben sind. Die Isolationsstufe und manchmal andere Attribute bestimmen, wie Transaktionen interagieren, wenn eine andere Transaktion Arbeit festschreiben möchte.

Das Attribut `ISOLATION LEVEL` definiert die Isolationsstufe für die gestartete Transaktion. Es ist der wichtigste Transaktionsparameter, um sein Verhalten gegenüber anderen gleichzeitig laufenden Transaktionen zu bestimmen.

Die drei von Firebird unterstützten Isolationsstufen sind:

- `SNAPSHOT`
- `SNAPSHOT TABLE STABILITY`
- `READ COMMITTED` mit zwei Angaben (`NO RECORD_VERSION` und `RECORD_VERSION`)

`SNAPSHOT`-Isolationsstufe

Die Isolationsstufe `SNAPSHOT` – die Standardstufe – ermöglicht es der Transaktion, nur die Änderungen zu sehen, die vor dem Start festgeschrieben wurden. Alle festgeschriebenen Änderungen, die durch gleichzeitige Transaktionen vorgenommen werden, werden in einer `SNAPSHOT`-Transaktion nicht angezeigt, solange diese aktiv ist. Die Änderungen werden für eine neue Transaktion sichtbar, sobald die aktuelle Transaktion entweder festgeschrieben oder vollständig zurückgesetzt wurde, jedoch nicht, wenn sie nur auf einen Sicherungspunkt zurückgesetzt wurde.

Die Isolationsstufe `SNAPSHOT` wird auch als “*concurrency*” bezeichnet.

Autonome Transaktionen

Änderungen durch autonome Transaktionen werden nicht im Kontext der Transaktion `SNAPSHOT` gesehen, die sie gestartet hat.

`SNAPSHOT TABLE STABILITY`-Isolationsstufe

Die Isolationsstufe `SNAPSHOT TABLE STABILITY` oder `SNAPSHOT TABLE` – `ist die restriktivste. Wie in `SNAPSHOT` sieht eine Transaktion in der `SNAPSHOT TABLE STABILITY`-Isolation nur die Änderungen, die vor dem Start der aktuellen Transaktion festgeschrieben wurden. Nachdem eine `SNAPSHOT TABLE STABILITY` gestartet wurde, können keine anderen Transaktionen Änderungen an Tabellen in der Datenbank vornehmen, deren Änderungen für diese Transaktion anstehen. Andere Transaktionen

können andere Daten lesen, aber jeder Versuch des Einfügens, Aktualisierens oder Löschens durch einen parallelen Prozess führt zu Konfliktausnahmen.

Die RESERVING-Klausel kann verwendet werden, um anderen Transaktionen zu erlauben, Daten in einigen Tabellen zu ändern.

Wenn bei einer anderen Transaktion eine nicht festgeschriebene Änderung von Daten in einer Datenbanktabelle ansteht, bevor eine Transaktion mit der Isolationsstufe SNAPSHOT TABLE STABILITY gestartet wird, führt der Versuch, eine SNAPSHOT TABLE STABILITY-Transaktion zu starten, zu einer Ausnahme.

Die Isolationsstufe SNAPSHOT TABLE STABILITY wird auch als "*consistency*" bezeichnet.

READ COMMITTED-Isolationsstufe

Die Isolationsstufe READ COMMITTED ermöglicht, dass alle Datenänderungen, die andere Transaktionen seit ihrem Beginn festgeschrieben haben, sofort von der nicht festgeschriebenen aktuellen Transaktion gesehen werden. Nicht festgeschriebene Änderungen sind für eine 'READ COMMITTED'-Transaktion nicht sichtbar.

Um die aktualisierte Liste der Zeilen in der Tabelle, an der Sie interessiert sind - "aktualisiert" - abzurufen, muss nur die SELECT-Anweisung erneut angefordert werden, während sie sich noch in der nicht festgeschriebenen Transaktion READ COMMITTED befindet.

RECORD_VERSION

Für READ COMMITTED-Transaktionen kann je nach Art der gewünschten Konfliktlösung einer von zwei modifizierenden Parametern angegeben werden: RECORD_VERSION und NO RECORD_VERSION. Wie die Namen vermuten, schließen sie sich gegenseitig aus.

- NO RECORD_VERSION (der Standardwert) ist eine Art Zwei-Phasen-Sperrmechanismus: Er macht die Transaktion nicht in der Lage, in eine Zeile zu schreiben, für die eine Aktualisierung von einer anderen Transaktion aussteht.
  - Wenn NO WAIT die angegebene Lock-Resolution-Strategie ist, wird sofort ein Lock-Konflikt-Fehler ausgegeben
  - Wenn WAIT angegeben ist, wird gewartet, bis die andere Transaktion entweder festgeschrieben oder zurückgesetzt wird. Wenn die andere Transaktion zurückgesetzt oder festgeschrieben wird und ihre Transaktions-ID älter ist als die ID der aktuellen Transaktion, ist die Änderung der aktuellen Transaktion zulässig. Ein Sperrkonfliktfehler wird zurückgegeben, wenn die andere Transaktion festgeschrieben wurde und ihre ID neuer war als die der aktuellen Transaktion.
- Wenn RECORD_VERSION angegeben ist, liest die Transaktion die letzte festgeschriebene Version der Zeile, unabhängig von anderen ausstehenden Versionen der Zeile. Die Lock-Resolution-Strategie (WAIT oder NO WAIT) beeinflusst das Verhalten der Transaktion beim Start in keiner Weise.

NO AUTO UNDO

Die Option NO AUTO UNDO beeinflusst die Behandlung von Datensatzversionen (Garbage), die von der Transaktion im Fall eines Rollbacks erzeugt werden. Wenn NO AUTO UNDO markiert ist, markiert die

ROLLBACK-Anweisung die Transaktion nur als Rollback, ohne die in der Transaktion erstellten Datensatzversionen zu löschen. Sie werden später von der Müllabfuhr weggewischt.

NO AUTO UNDO kann nützlich sein, wenn viele separate Anweisungen ausgeführt werden, die Daten unter Bedingungen ändern, bei denen die Transaktion wahrscheinlich die meiste Zeit erfolgreich festgeschrieben wird.

Die Option NO AUTO UNDO wird bei Transaktionen ignoriert, bei denen keine Änderungen vorgenommen werden.

RESTART REQUESTS

Laut den Firebird-Quellen wird dies

> Alle Anfragen der aktuellen Verbindungen (Attachment) neustarten, um die übergebene Transaktion zu verwenden.
>
> — src/jrd/tra.cpp

Die genaue Semantik und die Auswirkungen dieser Klausel sind nicht klar, und wir empfehlen, diese Klausel nicht zu verwenden.

IGNORE LIMBO

Dieses Flag wird verwendet, um zu signalisieren, dass Datensätze, die von Limbo-Transaktionen erstellt wurden, ignoriert werden sollen. Transaktionen bleiben "in der Schwebe", wenn die zweite Stufe eines zweiphasigen Commits fehlschlägt.

Historischer Hinweis

IGNORE LIMBO liefert den TPB-Parameter isc_tpb_ignore_limbo, der seit InterBase-Zeiten in der API verfügbar ist und hauptsächlich von *gfix* verwendet wird.

RESERVING

Die RESERVING-Klausel in der SET TRANSACTION-Anweisung reserviert Tabellen, die in der Tabellenliste angegeben sind. Das Reservieren einer Tabelle verhindert, dass andere Transaktionen Änderungen daran vornehmen oder sogar unter Einbeziehung bestimmter Parameter Daten aus ihnen lesen, während diese Transaktion läuft.

Eine RESERVING-Klausel kann auch verwendet werden, um eine Liste von Tabellen anzugeben, die von anderen Transaktionen geändert werden können, selbst wenn die Transaktion mit der Isolationsstufe SNAPSHOT TABLE STABILITY gestartet wird.

Eine RESERVING-Klausel wird verwendet, um beliebig viele reservierte Tabellen anzugeben.

Optionen für die RESERVING-Klausel

Wird eines der Schlüsselwörter SHARED oder PROTECTED weggelassen, wird SHARED angenommen. Wenn die gesamte FOR-Klausel weggelassen wird, wird FOR SHARED READ angenommen. Die Namen und die Kompatibilität der vier Zugriffsoptionen zum Reservieren von Tabellen sind nicht

offensichtlich.

Tabelle 214. Kompatibilität der Zugriffsoptionen für `RESERVING`

	SHARED READ	SHARED WRITE	PROTECTED READ	PROTECTED WRITE
SHARED READ	Ja	Ja	Ja	Ja
SHARED WRITE	Ja	Ja	Nein	Nein
PROTECTED READ	Ja	Nein	Ja	Nein
PROTECTED WRITE	Ja	Nein	Nein	Nein

Die Kombinationen dieser `RESERVING`-Klausel-Flags für den gleichzeitigen Zugriff hängen von den Isolationsstufen der gleichzeitigen Transaktionen ab:

- `SNAPSHOT`-Isolierung
  - Gleichzeitige `SNAPSHOT`-Transaktionen mit `SHARED READ` haben keinen Einfluss auf den Zugriff des anderen
  - Eine gleichzeitige Mischung aus `SNAPSHOT`- und `READ COMMITTED`-Transaktionen mit `SHARED WRITE` hat keinen Einfluss auf den gegenseitigen Zugriff, aber sie blockieren Transaktionen mit der `SNAPSHOT TABLE STABILITY`-Isolation entweder vom Lesen aus oder Schreiben in die angegebene(n) Tabelle(n).)
  - Gleichzeitige Transaktionen mit beliebiger Isolationsstufe und `PROTECTED READ` können nur Daten aus den reservierten Tabellen lesen. Jeder Versuch, auf sie zu schreiben, führt zu einer Ausnahme
  - Mit `PROTECTED WRITE` können gleichzeitige Transaktionen mit `SNAPSHOT` und `READ COMMITTED` Isolation nicht in die angegebenen Tabellen schreiben. Transaktionen mit `SNAPSHOT TABLE STABILITY`-Isolation können überhaupt nicht aus den reservierten Tabellen lesen oder in sie schreiben.
- Isolierung "SNAPSHOT TABLE STABILITY"
  - Alle gleichzeitigen Transaktionen mit `SHARED READ` können unabhängig von ihrer Isolationsstufe aus den reservierten Tabellen lesen oder schreiben (wenn im `READ WRITE` Modus)
  - Gleichzeitige Transaktionen mit den Isolationsstufen `SNAPSHOT` und `READ COMMITTED` und `SHARED WRITE` können Daten aus den angegebenen Tabellen lesen und schreiben (wenn im `READ WRITE`-Modus) aber gleichzeitig auf diese Tabellen von Transaktionen mit `SNAPSHOT . zugreifen TABLE STABILITY` ist komplett gesperrt, während diese Transaktionen aktiv sind
  - Gleichzeitige Transaktionen mit beliebiger Isolationsstufe und `PROTECTED READ` können nur aus den reservierten Tabellen lesen
  - Mit `PROTECTED WRITE` können gleichzeitige `SNAPSHOT`- und `READ COMMITTED`-Transaktionen aus den reservierten Tabellen lesen, aber nicht in sie schreiben. Der Zugriff durch Transaktionen mit der Isolationsstufe `SNAPSHOT TABLE STABILITY` wird vollständig blockiert.
- Isolation "READ COMMITTED"

- Mit SHARED READ können alle gleichzeitigen Transaktionen mit beliebiger Isolationsstufe sowohl von den reservierten Tabellen lesen als auch schreiben (wenn im READ WRITE Modus)
- SHARED WRITE erlaubt allen Transaktionen in der SNAPSHOT- und READ COMMITTED-Isolation das Lesen und Schreiben (wenn im READ WRITE-Modus) in die angegebenen Tabellen und blockiert den Zugriff vollständig von Transaktionen mit der SNAPSHOT TABLE STABILITY -Isolation
- Mit PROTECTED READ können gleichzeitige Transaktionen mit beliebiger Isolationsstufe nur aus den reservierten Tabellen lesen
- Mit PROTECTED WRITE können gleichzeitige Transaktionen in SNAPSHOT und READ COMMITTED Isolation aus den angegebenen Tabellen lesen, aber nicht in sie schreiben. Der Zugriff von Transaktionen in der Isolation SNAPSHOT TABLE STABILITY wird vollständig blockiert.

In Embedded SQL kann die USING-Klausel verwendet werden, um Systemressourcen zu schonen, indem Begrenzung der Anzahl der Datenbanken, auf die eine Transaktion zugreifen kann. USING schließt sich mit RESERVING gegenseitig aus. Eine USING-Klausel in der SET TRANSACTION-Syntax wird in DSQL nicht unterstützt.

Siehe auch

COMMIT, ROLLBACK

12.1.2. COMMIT

Verwendet für

Bestätigen einer Transaktion

Verfügbar in

DSQL, ESQL

Syntax

```
COMMIT [TRANSACTION tr_name] [WORK]
  [RETAIN [SNAPSHOT] | RELEASE];
```

Tabelle 215. COMMIT-Anweisungsparameter

Parameter	Beschreibung
tr_name	Transaktionsname. Nur in ESQL verfügbar

Die COMMIT-Anweisung verpflichtet alle Arbeiten, die im Rahmen dieser Transaktion ausgeführt werden (Einfügungen, Aktualisierungen, Löschungen, Auswahlen, Ausführen von Prozeduren). Neue Datensatzversionen werden für andere Transaktionen verfügbar, und wenn die 'RETAIN'-Klausel nicht verwendet wird, werden alle Serverressourcen, die seiner Arbeit zugewiesen sind, freigegeben.

Wenn während des Festschreibens der Transaktion Konflikte oder andere Fehler in der Datenbank auftreten, wird die Transaktion nicht festgeschrieben und die Gründe werden zur Bearbeitung an

die Benutzeranwendung zurückgesendet, und die Möglichkeit, einen weiteren Festschreibungsversuch oder ein Rollback der Transaktion zu versuchen .

Die Klauseln TRANSACTION und RELEASE sind nur in ESQL gültig.

COMMIT-Optionen

- Die optionale TRANSACTION tr_name-Klausel, die nur in Embedded SQL verfügbar ist, gibt den Namen der Transaktion an, die festgeschrieben werden soll. Ohne TRANSACTION-Klausel wird COMMIT auf die Standardtransaktion angewendet.

In ESQL-Anwendungen ermöglichen benannte Transaktionen, dass mehrere Transaktionen gleichzeitig in einer Anwendung aktiv sind. Wenn benannte Transaktionen verwendet werden, muss für jede benannte Transaktion eine Hostsprachenvariable mit demselben Namen deklariert und initialisiert werden. Dies ist eine Einschränkung, die eine dynamische Angabe von Transaktionsnamen verhindert und somit eine Transaktionsbenennung in DSQL ausschließt.

- Das optionale Schlüsselwort WORK wird nur aus Kompatibilitätsgründen mit anderen relationalen Datenbankverwaltungssystemen unterstützt, die es erfordern.
- Das Schlüsselwort RELEASE ist nur in Embedded SQL verfügbar und ermöglicht die Trennung von allen Datenbanken, nachdem die Transaktion festgeschrieben wurde. RELEASE wird in Firebird nur aus Kompatibilitätsgründen mit älteren Versionen von InterBase beibehalten. Es wurde in ESQL durch die DISCONNECT-Anweisung ersetzt.
- Die RETAIN [SNAPSHOT]-Klausel wird für das “soft”-Commit verwendet, das unter Hostsprachen und ihren Praktikern verschiedentlich als COMMIT WITH RETAIN, “CommitRetaining”, “warm commit”, etc. bezeichnet wird. Die Transaktion wird festgeschrieben, aber einige Serverressourcen werden beibehalten und eine neue Transaktion wird transparent mit derselben Transaktions-ID neu gestartet. Der Zustand von Zeilencaches und Cursors wird so beibehalten, wie er vor dem Soft Commit war.

  Bei Transaktionen mit Soft-Committed, deren Isolationsstufe SNAPSHOT oder SNAPSHOT TABLE STABILITY ist, wird die Ansicht des Datenbankstatus nicht aktualisiert, um Änderungen durch andere Transaktionen widerzuspiegeln, und der Benutzer der Anwendungsinstanz hat weiterhin dieselbe Ansicht wie beim Transaktion wurde ursprünglich gestartet. Änderungen, die während der Laufzeit der einbehaltenen Transaktion vorgenommen wurden, sind natürlich für diese Transaktion sichtbar.

Empfehlung

Die Verwendung der COMMIT-Anweisung anstelle von ROLLBACK wird empfohlen, um Transaktionen zu beenden, die nur Daten aus der Datenbank lesen, da COMMIT weniger Serverressourcen verbraucht und hilft, die Leistung nachfolgender Transaktionen zu optimieren.

Siehe auch

SET TRANSACTION, ROLLBACK

12.1.3. ROLLBACK

Verwendet für

Rollback einer Transaktion

Verfügbar in

DSQL, ESQL

Syntax

```
  ROLLBACK [TRANSACTION tr_name] [WORK]
    [RETAIN [SNAPSHOT] | RELEASE]
| ROLLBACK [WORK] TO [SAVEPOINT] sp_name
```

Tabelle 216. ROLLBACK-Anweisungsparameter

Parameter	Beschreibung
tr_name	Transaktionsname. Nur in ESQL verfügbar
sp_name	Name des Sicherungspunkts. Nur in SQL verfügbar

Die ROLLBACK-Anweisung macht alle im Kontext dieser Transaktion ausgeführten Arbeiten (inserts, update, deletes, selects, Ausführung von Prozeduren) rückgängig. ROLLBACK schlägt nie fehl und verursacht daher keine Ausnahmen. Sofern die 'RETAIN'-Klausel nicht verwendet wird, werden alle der Arbeit der Transaktion zugeordneten Serverressourcen freigegeben.

Die Klauseln TRANSACTION und RELEASE sind nur in ESQL gültig. Die Anweisung ROLLBACK TO SAVEPOINT ist in ESQL nicht verfügbar.

ROLLBACK Options

- Die optionale TRANSACTION tr_name-Klausel, die nur in Embedded SQL verfügbar ist, gibt den Namen der Transaktion an, die festgeschrieben werden soll. Ohne TRANSACTION-Klausel wird ROLLBACK auf die Standardtransaktion angewendet.

In ESQL-Anwendungen ermöglichen benannte Transaktionen, dass mehrere Transaktionen gleichzeitig in einer Anwendung aktiv sind. Wenn benannte Transaktionen verwendet werden, muss für jede benannte Transaktion eine Hostsprachenvariable mit demselben Namen deklariert und initialisiert werden. Dies ist eine Einschränkung, die eine dynamische Angabe von Transaktionsnamen verhindert und somit eine Transaktionsbenennung in DSQL ausschließt.

- Das optionale Schlüsselwort WORK wird nur aus Kompatibilitätsgründen mit anderen relationalen Datenbankverwaltungssystemen unterstützt, die es benötigen.
- Das Schlüsselwort RETAIN gibt an, dass der Transaktionskontext beibehalten werden soll, obwohl die gesamte Arbeit der Transaktion rückgängig gemacht werden soll. Einige Serverressourcen werden beibehalten und die Transaktion wird transparent mit derselben Transaktions-ID neu gestartet. Der Zustand von Zeilencaches und Cursors wird so beibehalten, wie er vor dem

"sanften" Rollback war.

Bei Transaktionen, deren Isolationsstufe SNAPSHOT oder SNAPSHOT TABLE STABILITY ist, wird die Ansicht des Datenbankstatus durch das weiche Rollback nicht aktualisiert, um Änderungen durch andere Transaktionen widerzuspiegeln. Der Benutzer der Anwendungsinstanz hat weiterhin dieselbe Ansicht wie beim ursprünglichen Start der Transaktion. Änderungen, die während der Laufzeit der einbehaltenen Transaktion vorgenommen und mit einem Soft-Commit versehen wurden, sind natürlich für diese Transaktion sichtbar.

Siehe auch

SET TRANSACTION, COMMIT

ROLLBACK TO SAVEPOINT

Die alternative Anweisung ROLLBACK TO SAVEPOINT gibt den Namen eines Sicherungspunkts an, an dem Änderungen rückgängig gemacht werden sollen. Der Effekt besteht darin, alle innerhalb der Transaktion vorgenommenen Änderungen rückgängig zu machen, vom angegebenen Sicherungspunkt vorwärts bis zu dem Punkt, an dem ROLLBACK TO SAVEPOINT angefordert wird.

ROLLBACK TO SAVEPOINT führt die folgenden Operationen aus:

- Alle Datenbankmutationen, die seit der Erstellung des Sicherungspunkts durchgeführt wurden, werden rückgängig gemacht. Mit RDB$SET_CONTEXT() gesetzte Benutzervariablen bleiben unverändert.
- Alle Sicherungspunkte, die nach dem benannten erstellt wurden, werden zerstört. Savepoints vor dem benannten werden zusammen mit dem benannten Savepoint selbst beibehalten. Wiederholte Rollbacks auf denselben Sicherungspunkt sind somit zulässig.
- Alle impliziten und expliziten Datensatzsperren, die seit dem Sicherungspunkt erworben wurden, werden aufgehoben. Andere Transaktionen, die Zugriff auf nach dem Sicherungspunkt gesperrte Zeilen angefordert haben, müssen weiterhin warten, bis die Transaktion festgeschrieben oder zurückgesetzt wird. Andere Transaktionen, die die Zeilen noch nicht angefordert haben, können die entsperrten Zeilen sofort anfordern und darauf zugreifen.

Siehe auch

SAVEPOINT, RELEASE SAVEPOINT

12.1.4. SAVEPOINT

Verwendet für

Erstellen eines Sicherungspunkts

Verfügbar in

DSQL

Syntax

```
SAVEPOINT sp_name
```

Tabelle 217. SAVEPOINT-Anweisungsparameter

Parameter	Beschreibung
sp_name	Name des Sicherungspunkts. Nur in SQL verfügbar

Die SAVEPOINT-Anweisung erstellt einen SQL:99-konformen Savepoint, der als Marker im „Stack“ von Datenaktivitäten innerhalb einer Transaktion fungiert. Anschließend können die im “Stack” ausgeführten Aufgaben bis zu diesem Sicherungspunkt rückgängig gemacht werden, wobei die frühere Arbeit und ältere Sicherungspunkte unberührt bleiben. Savepoint-Mechanismen werden manchmal als “veschachtelte Transaktionen” bezeichnet.

Wenn bereits ein Sicherungspunkt mit demselben Namen wie dem für den neuen angegebenen Sicherungspunkt vorhanden ist, wird der vorhandene Sicherungspunkt freigegeben und ein neuer mit dem angegebenen Namen erstellt.

Um Änderungen zum Savepoint zurückzurollen, wird die Anweisung ROLLBACK TO SAVEPOINT verwendet.

Erwägungen zum Speicher

Der interne Mechanismus unter Sicherungspunkten kann viel Speicher beanspruchen, insbesondere wenn dieselben Zeilen mehrere Aktualisierungen in einer Transaktion erhalten. Wenn ein Sicherungspunkt nicht mehr benötigt wird, die Transaktion aber noch Arbeit zu erledigen hat, wird er durch eine RELEASE SAVEPOINT-Anweisung gelöscht und somit die Ressourcen freigegeben.

Beispiel-DSQL-Sitzung mit Sicherungspunkten

```
CREATE TABLE TEST (ID INTEGER);
COMMIT;
INSERT INTO TEST VALUES (1);
COMMIT;
INSERT INTO TEST VALUES (2);
SAVEPOINT Y;
DELETE FROM TEST;
SELECT * FROM TEST; -- returns no rows
ROLLBACK TO Y;
SELECT * FROM TEST; -- returns two rows
ROLLBACK;
SELECT * FROM TEST; -- returns one row
```

Siehe auch

ROLLBACK TO SAVEPOINT, RELEASE SAVEPOINT

12.1.5. RELEASE SAVEPOINT

Verwendet für

Speicherpunkt löschen

Verfügbar in

DSQL

Syntax

```
RELEASE SAVEPOINT sp_name [ONLY]
```

Tabelle 218. RELEASE SAVEPOINT Statement Parameter

Parameter	Beschreibung
sp_name	Name des Sicherungspunkts. Nur in SQL verfügbar

Die Anweisung `RELEASE SAVEPOINT` löscht einen benannten Savepoint und gibt alle darin enthaltenen Ressourcen frei. Standardmäßig werden alle Sicherungspunkte, die nach dem benannten Sicherungspunkt erstellt wurden, ebenfalls freigegeben. Der Qualifier `ONLY` weist die Engine an, nur den benannten Savepoint freizugeben.

Siehe auch

`SAVEPOINT`

12.1.6. Interne Sicherungspunkte

Standardmäßig verwendet die Engine einen automatischen Sicherungspunkt auf Transaktionsebene, um ein Transaktions-Rollback durchzuführen. Wenn eine `ROLLBACK`-Anweisung ausgegeben wird, werden alle in dieser Transaktion durchgeführten Änderungen über einen Sicherungspunkt auf Transaktionsebene zurückgesetzt und die Transaktion wird dann festgeschrieben. Diese Logik reduziert die Menge der durch Rollback-Transaktionen verursachten Garbage Collection.

Wenn das Volumen der Änderungen, die unter einem Sicherungspunkt auf Transaktionsebene durchgeführt werden, groß wird (~50000 betroffene Datensätze), gibt die Engine den Sicherungspunkt auf Transaktionsebene frei und verwendet die Transaktionsbestandsseite (TIP) als Mechanismus, um die Transaktion bei Bedarf zurückzusetzen.

Wenn Sie erwarten, dass das Volumen der Änderungen in Ihrer Transaktion groß ist, können Sie die Option `NO AUTO UNDO` in Ihrer `SET TRANSACTION`-Anweisung angeben, um die Erstellung des Sicherungspunkts auf Transaktionsebene zu blockieren. Wenn Sie stattdessen die API verwenden, würden Sie das TPB-Flag `isc_tpb_no_auto_undo` setzen.

12.1.7. Savepoints und PSQL

Anweisungen zur Transaktionssteuerung sind in PSQL nicht zulässig, da dies die Atomarität der Anweisung, die die Prozedur aufruft, zerstören würde. Firebird unterstützt jedoch das Auslösen und Behandeln von Ausnahmen in PSQL, sodass Aktionen, die in gespeicherten Prozeduren und Triggern ausgeführt werden, selektiv rückgängig gemacht werden können, ohne dass die gesamte Prozedur fehlschlägt.

Intern werden automatische Sicherungspunkte verwendet, um:

- alle Aktionen im `BEGIN···END` Block rückgängig machen, bei denen eine Ausnahme auftritt
- alle von der Prozedur oder dem Trigger ausgeführten Aktionen rückgängig machen oder, in einer wählbaren Prozedur, alle Aktionen, die seit dem letzten `SUSPEND` ausgeführt wurden, wenn die Ausführung aufgrund eines nicht abgefangenen Fehlers oder einer Ausnahme vorzeitig beendet wird

Jeder PSQL-Ausnahmebehandlungsblock ist außerdem durch automatische Systemsicherungspunkte begrenzt.

Ein `BEGIN···END`-Block erzeugt selbst keinen automatischen Sicherungspunkt. Ein Sicherungspunkt wird nur in Blöcken erstellt, die die WHEN-Anweisung zur Behandlung von Ausnahmen enthalten.

Kapitel 13. Sicherheit

Datenbanken müssen sicher sein, ebenso wie die darin gespeicherten Daten. Firebird bietet drei Ebenen der Datensicherheit: Benutzerauthentifizierung auf Serverebene, SQL-Berechtigungen in Datenbanken und - optional - Datenbankverschlüsselung. In diesem Kapitel wird beschrieben, wie Sie die Sicherheit auf diesen drei Ebenen verwalten.

Es gibt auch eine vierte Stufe der Datensicherheit: die drahtgebundene Protokollverschlüsselung, mit der Daten während der Übertragung zwischen Client und Server verschlüsselt werden. Die Verschlüsselung des Wire-Protokolls ist nicht Gegenstand dieser Sprachreferenz.

13.1. Benutzerauthentifizierung

Die Sicherheit der gesamten Datenbank hängt davon ab, einen Benutzer zu identifizieren und seine Berechtigung zu überprüfen, ein Verfahren, das als *Authentifizierung* bekannt ist. Die Benutzerauthentifizierung kann auf verschiedene Weise durchgeführt werden, abhängig von der Einstellung des Parameters `AuthServer` in der Konfigurationsdatei `firebird.conf`. Dieser Parameter enthält die Liste der Authentifizierungs-Plugins, die beim Herstellen einer Verbindung zum Server verwendet werden können. Wenn das erste Plugin bei der Authentifizierung fehlschlägt, kann der Client mit dem nächsten Plugin fortfahren usw. Wenn kein Plugin den Benutzer authentifizieren konnte, erhält der Benutzer eine Fehlermeldung.

Die Informationen über Benutzer, die berechtigt sind, auf einen bestimmten Firebird-Server zuzugreifen, werden in einer speziellen Sicherheitsdatenbank namens `security3.fdb` gespeichert. Jeder Eintrag in `security3.fdb` ist ein Benutzerkonto für einen Benutzer. Für jede Datenbank kann die Sicherheitsdatenbank in der Datei `database.conf` (Parameter `SecurityDatabase`) überschrieben werden. Jede Datenbank kann eine Sicherheitsdatenbank sein, sogar für diese Datenbank selbst.

Ein Benutzername, bestehend aus bis zu 31 Zeichen, ist ein Bezeichner, der den normalen Regeln für Bezeichner folgt (Groß-/Kleinschreibung nicht in Anführungszeichen, Groß-/Kleinschreibung in doppelten Anführungszeichen). Aus Gründen der Abwärtskompatibilität akzeptieren einige Anweisungen (z. B. *isql*s `CONNECT`) Benutzernamen in einfachen Anführungszeichen, die sich wie normale Bezeichner ohne Anführungszeichen verhalten.

Die maximale Passwortlänge hängt vom User-Manager-Plugin ab (Parameter `UserManager`, in `firebird.conf` oder `databases.conf`). Bei Kennwörtern muss die Groß-/Kleinschreibung beachtet werden. Der Standardbenutzermanager ist das erste Plugin in der `UserManager`-Liste, kann aber in den SQL-Benutzerverwaltungsanweisungen überschrieben werden. Für das `Srp`-Plugin beträgt die maximale Passwortlänge 255 Zeichen, bei einer effektiven Länge von 20 Bytes (siehe auch Warum beträgt die effektive Kennwortlänge SRP 20 Byte?). Für das Legacy_UserManager-Plugin sind nur die ersten acht Bytes von Bedeutung. Während es für `Legacy_UserManager` gültig ist, ein Passwort mit mehr als acht Bytes einzugeben, werden alle nachfolgenden Zeichen ignoriert.

Warum beträgt die effektive Kennwortlänge SRP 20 Byte?

Das SRP-Plugin hat keine 20-Byte-Beschränkung für die Kennwortlänge, und längere

Kennwörter können verwendet werden. Hashes verschiedener Passwörter, die länger als 20 Byte sind, sind auch – normalerweise – unterschiedlich. Diese effektive Grenze ergibt sich aus der begrenzten Hash-Länge in SHA1 (wird in der Firebirds SRP-Implementierung verwendet), 20 Byte oder 160 Bit und dem pigeonhole-Prinzip. Früher oder später wird es ein kürzeres (oder längeres) Passwort geben, das den gleichen Hash hat (z.B. bei einem Brute-Force-Angriff). Aus diesem Grund wird die effektive Kennwortlänge für den SHA1-Algorithmus häufig als 20 Byte bezeichnet.

Die eingebettete Version des Servers verwendet keine Authentifizierung. Allerdings müssen in den Verbindungsparametern der Benutzername und ggf. die Rolle angegeben werden, da sie den Zugriff auf Datenbankobjekte steuern.

SYSDBA oder der Eigentümer der Datenbank erhalten uneingeschränkten Zugriff auf alle Objekte der Datenbank. Benutzer mit der Rolle RDB$ADMIN erhalten einen ähnlichen uneingeschränkten Zugriff, wenn sie die Rolle beim Verbinden angeben.

13.1.1. Besonders privilegierte Benutzer

In Firebird ist das SYSDBA-Konto ein Superuser, der über alle Sicherheitsbeschränkungen hinaus existiert. Es hat vollständigen Zugriff auf alle Objekte in allen regulären Datenbanken auf dem Server und vollen Lese-/Schreibzugriff auf die Konten in der Sicherheitsdatenbank security3.fdb. Kein Benutzer hat Zugriff auf die Metadaten der Sicherheitsdatenbank.

Für Srp existiert das SYSDBA-Konto standardmäßig nicht; Es muss über eine eingebettete Verbindung erstellt werden. Für Legacy_Auth ist das Standard-SYSDBA-Passwort unter Windows und MacOS masterkey — oder masterke, um genau zu sein, wegen der Längenbeschränkung von 8 Zeichen.

Extrem wichtig!

Das Standardpasswort masterkey ist im ganzen Universum bekannt. Dieses sollte geändert werden, sobald die Installation des Firebird-Servers abgeschlossen ist.

Andere Benutzer können auf verschiedene Weise erhöhte Berechtigungen erwerben, von denen einige von der Betriebssystemplattform abhängig sind. Diese werden in den folgenden Abschnitten besprochen und in Administratoren zusammengefasst.

POSIX Hosts

Auf POSIX-Systemen, einschließlich MacOS, wird der POSIX-Benutzername als Firebird Embedded-Benutzername verwendet, wenn der Benutzername nicht explizit angegeben wird.

Der SYSDBA-Benutzer auf POSIX

Auf anderen POSIX-Hosts als MacOSX hat der SYSDBA-Benutzer kein Standardkennwort. Wenn die vollständige Installation mit den Standardskripten erfolgt, wird ein einmaliges Passwort erstellt und in einer Textdatei im gleichen Verzeichnis wie security3.fdb gespeichert, üblicherweise` /opt/firebird/. Der Name der Passwortdatei ist `SYSDBA.password.

Bei einer Installation, die von einem verteilungsspezifischen Installationsprogramm durchgeführt wird, kann der Speicherort der Sicherheitsdatenbank und der Kennwortdatei vom Standard abweichen.

Der *root*-Benutzer

Der Benutzer * root * kann in Firebird Embedded direkt als SYSDBA fungieren. Firebird behandelt **root** als wäre es SYSDBA und bietet Zugriff auf alle Datenbanken auf dem Server.

Windows-Hosts

Auf Windows-Server-fähigen Betriebssystemen können Betriebssystemkonten verwendet werden. Die Windows-Authentifizierung (auch bekannt als `Trusted Authentication`) kann aktiviert werden, indem das `Win_Sspi`-Plugin in die `AuthServer`-Liste in `firebird.conf` aufgenommen wird. Das Plugin muss auch clientseitig in der Einstellung `AuthClient` vorhanden sein.

Administratoren des Windows-Betriebssystems werden beim Herstellen einer Verbindung mit einer Datenbank nicht automatisch SYSDBA-Berechtigungen gewährt. Dazu muss die intern erstellte Rolle `RDB$ADMIN` von SYSDBA oder dem Datenbankbesitzer geändert werden, um sie zu aktivieren. Einzelheiten finden Sie im späteren Abschnitt mit dem Titel `AUTO ADMIN MAPPING`.

Vor Firebird 3.0 wurden mit aktivierter vertrauenswürdiger Authentifizierung Benutzer, die die Standardprüfungen bestanden haben, automatisch 'CURRENT_USER' zugeordnet. In Firebird 3.0 und höher muss die Zuordnung explizit mit `CREATE MAPPING` erfolgen.

Der Datenbankbesitzer

Der `Besitzer` (engl. `Owner`) einer Datenbank ist entweder der Benutzer, der zum Zeitpunkt der Erstellung (oder Wiederherstellung) der Datenbank `CURRENT_USER` war, oder, falls der `USER` -Parameter in der `CREATE DATABASE`-Anweisung angegeben wurde, der angegebene Benutzer.

`Owner` ist kein Benutzername. Der Benutzer, der Eigentümer einer Datenbank ist, verfügt über vollständige Administratorrechte in Bezug auf diese Datenbank, einschließlich des Rechts, sie zu löschen, aus einer Sicherung wiederherzustellen und die `AUTO ADMIN MAPPING`-Fähigkeit zu aktivieren oder zu deaktivieren.

Vor Firebird 2.1 hatte der Besitzer keine automatischen Berechtigungen für Datenbankobjekte, die von anderen Benutzern erstellt wurden.

13.1.2. `RDB$ADMIN`-Rolle

Die intern erstellte Rolle "RDB$ADMIN" ist in allen Datenbanken vorhanden. Die Zuweisung der Rolle `RDB$ADMIN` an einen regulären Benutzer in einer Datenbank gewährt diesem Benutzer die Privilegien des `SYSDBA` nur in dieser Datenbank.

Die erhöhten Berechtigungen werden wirksam, wenn der Benutzer unter der Rolle `RDB$ADMIN` bei dieser regulären Datenbank angemeldet ist und die vollständige Kontrolle über alle Objekte in dieser Datenbank bietet.

Die Zuweisung der Rolle RDB$ADMIN in der Sicherheitsdatenbank verleiht die Berechtigung zum Erstellen, Bearbeiten und Löschen von Benutzerkonten.

In beiden Fällen kann der Benutzer mit den erhöhten Rechten jedem anderen Benutzer die Rolle RDB$ADMIN zuweisen. Mit anderen Worten, die Angabe von WITH ADMIN OPTION ist unnötig, da dies in die Rolle integriert ist.

Gewähren der Rolle RDB$ADMIN in der Sicherheitsdatenbank

Da sich niemand – nicht einmal SYSDBA – aus der Ferne mit der Sicherheitsdatenbank verbinden kann, sind die Anweisungen GRANT und REVOKE für diese Aufgabe nutzlos. Stattdessen wird die Rolle RDB$ADMIN mit den SQL-Anweisungen für die Benutzerverwaltung gewährt und entzogen:

```
CREATE USER new_user
  PASSWORD 'password'
  GRANT ADMIN ROLE;

ALTER USER existing_user
  GRANT ADMIN ROLE;

ALTER USER existing_user
  REVOKE ADMIN ROLE;
```

GRANT ADMIN ROLE und REVOKE ADMIN ROLE sind keine Anweisungen im GRANT und REVOKE Lexikon. Es handelt sich um Drei-Wort-Klauseln zu den Anweisungen CREATE USER und ALTER USER.

Tabelle 219. Parameter für die RDB$ADMIN*-Rollen* GRANT *und* REVOKE

Parameter	Beschreibung
new_user	Name für den neuen Benutzer
existing_user	Name eines bestehenden Benutzers
password	Benutzerkennwort

Der Benutzer, der die Rechte vergibt (engl. grantor) muss als Administrator angemeldet sein.

Siehe auch

CREATE USER, ALTER USER, GRANT, REVOKE

Die gleiche Aufgabe mit *gsec* ausführen

Mit Firebird 3.0 gilt *gsec* als veraltet. Es wird empfohlen, stattdessen die SQL-Benutzerverwaltungsanweisungen zu verwenden.

Eine Alternative besteht darin, *gsec* mit dem Parameter -admin zu verwenden, um das Attribut RDB$ADMIN im Datensatz des Benutzers zu speichern:

```
gsec -add new_user -pw password -admin yes
gsec -mo existing_user -admin yes
gsec -mo existing_user -admin no
```

Abhängig vom administrativen Status des aktuellen Benutzers können beim Aufruf von *gsec* weitere Parameter benötigt werden, z. `-user` und `-pass` oder `-trusted`.

Verwenden der Rolle `RDB$ADMIN` in der Sicherheitsdatenbank

Um Benutzerkonten über SQL zu verwalten, muss der Stipendiat die Rolle `RDB$ADMIN` beim Verbinden oder über `SET ROLE` angeben. Kein Benutzer kann remote eine Verbindung zur Sicherheitsdatenbank herstellen. Die Lösung besteht daher darin, dass der Benutzer eine Verbindung zu einer regulären Datenbank herstellt, in der er auch die Rechte `RDB$ADMIN` hat und die Rolle `RDB$ADMIN` in seinen Anmeldeparametern angibt. Von dort aus können sie jeden beliebigen SQL-Benutzerverwaltungsbefehl senden.

Wenn es keine reguläre Datenbank gibt, in der der Benutzer die Rolle `RDB$ADMIN` hat, ist eine Kontoverwaltung über SQL-Abfragen nicht möglich, es sei denn, sie verbinden sich direkt über eine eingebettete Verbindung mit der Sicherheitsdatenbank.

Verwenden von *gsec* mit `RDB$ADMIN`-Rechten

Um die Benutzerverwaltung mit *gsec* durchzuführen, muss der Benutzer den zusätzlichen Schalter `-role rdb$admin` bereitstellen.

Gewähren der Rolle "RDB$ADMIN" in einer regulären Datenbank

In einer regulären Datenbank wird die Rolle `RDB$ADMIN` mit der üblichen Syntax zum Gewähren und Entziehen von Rollen gewährt und entzogen:

```
GRANT [ROLE] RDB$ADMIN TO username

REVOKE [ROLE] RDB$ADMIN FROM username
```

Tabelle 220. Parameters for `RDB$ADMIN` *Role* `GRANT` *and* `REVOKE`

Parameter	Beschreibung
username	Name des Benutzers

Um die Rolle `RDB$ADMIN` zu erteilen und zu entziehen, muss der Erteilender als Administrator angemeldet sein. .Siehe auch `GRANT`, `REVOKE`

Verwenden der Rolle `RDB$ADMIN` in einer regulären Datenbank

Um seine `RDB$ADMIN`-Privilegien auszuüben, muss der Stipendiat die Rolle bei der Verbindung mit der Datenbank in die Verbindungsattribute aufnehmen oder später mit `SET ROLE` angeben.

AUTO ADMIN MAPPING

Windows-Administratoren werden nicht automatisch RDB$ADMIN-Berechtigungen gewährt, wenn sie sich mit einer Datenbank verbinden (natürlich wenn Win_Sspi aktiviert ist) Der Schalter AUTO ADMIN MAPPING bestimmt nun datenbankweise, ob Administratoren über automatische RDB$ADMIN-Rechte verfügen. Wenn eine Datenbank erstellt wird, ist sie standardmäßig deaktiviert.

Wenn AUTO ADMIN MAPPING in der Datenbank aktiviert ist, wird es immer wirksam, wenn ein Windows-Administrator eine Verbindung herstellt:

a. mit Win_Sspi-Authentifizierung und
b. ohne eine Rolle anzugeben

Nach einer erfolgreichen auto admin-Verbindung wird die aktuelle Rolle auf RDB$ADMIN gesetzt.

Wenn beim Connect eine explizite Rolle angegeben wurde, kann die Rolle RDB$ADMIN später in der Sitzung mit SET TRUSTED ROLE übernommen werden.

Auto-Admin-Mapping in regulären Datenbanken

So aktivieren und deaktivieren Sie die automatische Zuordnung in einer regulären Datenbank:

```
ALTER ROLE RDB$ADMIN
  SET AUTO ADMIN MAPPING;  -- aktivieren

ALTER ROLE RDB$ADMIN
  DROP AUTO ADMIN MAPPING; -- deaktivieren
```

Beide Anweisungen müssen von einem Benutzer mit ausreichenden Rechten ausgegeben werden, d. h.:

- Der Datenbankbesitzer
- Ein Administrator
- Ein Benutzer mit der Berechtigung ALTER ANY ROLE

Die Anweisung

```
ALTER ROLE RDB$ADMIN
  SET AUTO ADMIN MAPPING;
```

ist eine vereinfachte Form einer CREATE MAPPING-Anweisung, um ein Mapping der vordefinierten Gruppe DOMAIN_ANY_RID_ADMINS auf die Rolle von RDB$ADMIN zu erstellen:

```
CREATE MAPPING WIN_ADMINS
  USING PLUGIN WIN_SSPI
  FROM Predefined_Group DOMAIN_ANY_RID_ADMINS
```

```
  TO ROLE RDB$ADMIN;
```

Dementsprechend ist die Anweisung

```
ALTER ROLE RDB$ADMIN
  DROP AUTO ADMIN MAPPING
```

gleichbedeutend zum Statement

```
DROP MAPPING WIN_ADMINS;
```

Für weitere Details, siehe auch Zuordnung von Benutzern zu Objekten

In einer regulären Datenbank wird der Status von `AUTO ADMIN MAPPING` nur zur Verbindungszeit überprüft. Wenn ein Administrator die Rolle "RDB$ADMIN" hat, weil die automatische Zuordnung bei der Anmeldung aktiviert war, behält er diese Rolle für die Dauer der Sitzung bei, auch wenn er oder eine andere Person die Zuordnung in der Zwischenzeit deaktiviert.

Ebenso ändert das Einschalten von "AUTO ADMIN MAPPING" die aktuelle Rolle für Administratoren, die bereits verbunden waren, nicht in `RDB$ADMIN`.

Auto Admin Mapping in der Sicherheitsdatenbank

Die Anweisung `ALTER ROLE RDB$ADMIN` kann `AUTO ADMIN MAPPING` in der Sicherheitsdatenbank nicht aktivieren oder deaktivieren. Sie können jedoch ein globales Mapping für die vordefinierte Gruppe `DOMAIN_ANY_RID_ADMINS` auf die Rolle `RDB$ADMIN` wie folgt erstellen:

```
CREATE GLOBAL MAPPING WIN_ADMINS
  USING PLUGIN WIN_SSPI
  FROM Predefined_Group DOMAIN_ANY_RID_ADMINS
  TO ROLE RDB$ADMIN;
```

Außerdem können Sie *gsec* verwenden:

```
gsec -mapping set

gsec -mapping drop
```

Abhängig vom administrativen Status des aktuellen Benutzers können beim Aufruf von *gsec* weitere Parameter benötigt werden, z. `-user` und `-pass` oder `-trusted`.

Nur SYSDBA kann `AUTO ADMIN MAPPING` aktivieren, wenn es deaktiviert ist, aber jeder Administrator kann es deaktivieren.

Wenn AUTO ADMIN MAPPING in *gsec* deaktiviert wird, schaltet der Benutzer den Mechanismus selbst aus, der ihm den Zugriff gewährt hat, und somit wäre er nicht in der Lage, AUTO ADMIN MAPPING wieder zu aktivieren. Auch in einer interaktiven *gsec*-Sitzung wird die neue Flag-Einstellung sofort wirksam.

13.1.3. Administratoren

Als allgemeine Beschreibung ist ein Administrator ein Benutzer mit ausreichenden Rechten zum Lesen, Schreiben, Erstellen, Ändern oder Löschen von Objekten in einer Datenbank, für die der Administratorstatus dieses Benutzers gilt. Die Tabelle fasst zusammen, wie Superuser-Rechte in den verschiedenen Firebird-Sicherheitskontexten aktiviert werden.

Tabelle 221. Administrator- (Superuser-) Eigenschaften

Benutzer	RDB$ADMIN-Rolle	Hinweis
SYSDBA	Auto	Existiert automatisch auf Serverebene. Verfügt über alle Berechtigungen für alle Objekte in allen Datenbanken. Kann Benutzer erstellen, ändern und löschen, hat jedoch keinen direkten Fernzugriff auf die Sicherheitsdatenbank
root-Benutzer unter POSIX	Auto	Genau wie SYSDBA. Nur Firebird Embedded.
Superuser unter POSIX	Auto	Genau wie SYSDBA. Nur Firebird Embedded.
Windows-Administrator	Als CURRENT_ROLE festlegen, wenn die Anmeldung erfolgreich ist	Genau wie SYSDBA, wenn alle der folgenden Bedingungen zutreffen: • In der Datei firebird.conf enthält AuthServer, Win_Sspi und Win_Sspi ist in der Konfiguration der clientseitigen Plugins (AuthClient) vorhanden • In Datenbanken, in denen AUTO ADMIN MAPPING aktiviert ist oder eine entsprechende Zuordnung der vordefinierten Gruppe DOMAIN_ANY_RID_ADMINS für die Rolle RDB$ADMIN existiertMIN • Bei der Anmeldung ist keine Rolle angegeben
Datenbankbesitzer	Auto	Wie SYSDBA, aber nur in den Datenbanken, die sie besitzen
Normaler Benutzer	Muss vorher erteilt werden; muss beim Login angegeben werden	Wie SYSDBA, aber nur in den Datenbanken, in denen die Rolle zugewiesen ist

Benutzer	RDB$ADMIN-Rolle	Hinweis
Benutzer unter POSIX-Betriebssystemen	Muss vorher erteilt werden; muss beim Login angegeben werden	Wie SYSDBA, aber nur in den Datenbanken, in denen die Rolle zugewiesen ist. Nur Firebird Embedded.
Windows-Benutzer	Muss vorher erteilt werden; muss beim Login angegeben werden	Wie SYSDBA, aber nur in den Datenbanken, in denen die Rolle zugewiesen ist. Nur verfügbar, wenn in der Datei `firebird.conf` `AuthServer` `Win_Sspi` enthält und `Win_Sspi` in der Konfiguration der clientseitigen Plugins (`AuthClient`) vorhanden ist

13.2. SQL-Anweisungen für die Benutzerverwaltung

Dieser Abschnitt beschreibt die SQL-Anweisungen zum Erstellen, Ändern und Löschen von Firebird-Benutzerkonten. Diese Anweisungen können von folgenden Benutzern ausgeführt werden:

- SYSDBA
- Jeder Benutzer mit der RDB$ADMIN-Rolle in der Sicherheitsdatenbank und der RDB$ADMIN-Rolle in der Datenbank der aktiven Verbindung (der Benutzer muss die RDB$ADMIN-Rolle beim Verbinden angeben oder `SET ROLE` verwenden)
- Wenn das Flag `AUTO ADMIN MAPPING` in der Sicherheitsdatenbank aktiviert ist (`security3.fdb` oder was auch immer die Sicherheitsdatenbank ist, die für die aktuelle Datenbank in der `databases.conf` konfiguriert ist), jeder Windows-Administrator - vorausgesetzt, `Win_Sspi` wurde verwendet zu verbinden, ohne Rollen anzugeben.

Für einen Windows-Administrator reicht die nur in einer regulären Datenbank aktivierte `AUTO ADMIN MAPPING` nicht aus, um die Verwaltung anderer Benutzer zu ermöglichen. Anweisungen zum Aktivieren in der Sicherheitsdatenbank finden Sie unter Auto Admin Mapping in der Sicherheitsdatenbank.

Nicht-privilegierte Benutzer können nur die `ALTER USER`-Anweisung verwenden und dann nur einige Daten ihres eigenen Kontos bearbeiten.

13.2.1. CREATE USER

Verwendet für

Erstellen eines Firebird-Benutzerkontos

Verfügbar in

DSQL

Syntax

```
CREATE USER username
  <user_option> [<user_option> ...]
  [TAGS (<user_var> [, <user_var> ...]]

<user_option> ::=
    PASSWORD 'password'
  | FIRSTNAME 'firstname'
  | MIDDLENAME 'middlename'
  | LASTNAME 'lastname'
  | {GRANT | REVOKE} ADMIN ROLE
  | {ACTIVE | INACTIVE}
  | USING PLUGIN plugin_name

<user_var> ::=
    tag_name = 'tag_value'
  | DROP tag_name
```

Tabelle 222. CREATE USER-Anweisungsparameter

Parameter	Beschreibung
username	Benutzername. Die maximale Länge beträgt 31 Zeichen gemäß den Regeln für Firebird-Identifikatoren.
password	Benutzerkennwort Die gültige oder effektive Passwortlänge hängt vom Benutzermanager-Plugin ab. Groß-/Kleinschreibung beachten.
firstname	Optional: Vorname des Benutzers. Maximale Länge 31 Zeichen
middlename	Optional: Zuname des Benutzers. Maximale Länge 31 Zeichen
lastname	Optional: Nachname des Benutzers. Maximale Länge 31 Zeichen.
plugin_name	Name des Benutzermanager-Plugins.
tag_name	Name eines benutzerdefinierten Attributs. Die maximale Länge beträgt 31 Zeichen gemäß den Regeln für reguläre Firebird-Identifikatoren.
tag_value	Wert des benutzerdefinierten Attributs. Die maximale Länge beträgt 255 Zeichen.

Die CREATE USER-Anweisung erstellt ein neues Firebird-Benutzerkonto. Wenn der Benutzer bereits in der Firebird-Sicherheitsdatenbank für das angegebene Benutzermanager-Plugin vorhanden ist, wird ein entsprechender Fehler ausgegeben. Es ist möglich, mehrere Benutzer mit demselben Namen zu erstellen: einen pro Benutzermanager-Plugin.

Das Argument *username* muss den Regeln für reguläre Firebird-Identifikatoren folgen: siehe *Bezeichner* im Kapitel *Struktur*. Bei Benutzernamen muss die Groß-/Kleinschreibung beachtet werden, wenn sie in doppelte Anführungszeichen gesetzt werden (mit anderen Worten, sie folgen denselben Regeln wie andere Bezeichner mit Trennzeichen).

Seit Firebird 3.0 folgen Benutzernamen den allgemeinen Namenskonventionen

von Bezeichnern. Somit unterscheidet sich ein Benutzer mit dem Namen "Alex" von einem Benutzer mit dem Namen "ALEX"

```
CREATE USER ALEX PASSWORD 'bz23ds';

- dieser Benutzer ist der gleiche wie der erste first
CREATE USER Alex PASSWORD 'bz23ds';

- dieser Benutzer ist der gleiche wie der erste first
CREATE USER "ALEX" PASSWORD 'bz23ds';

- und das ist ein anderer Benutzer
CREATE USER "Alex" PASSWORD 'bz23ds';
```

Die PASSWORD-Klausel gibt das Kennwort des Benutzers an und ist erforderlich. Die gültige bzw. effektive Passwortlänge hängt vom User Manager Plugin ab, siehe auch Benutzerauthentifizierung.

Die optionalen Klauseln FIRSTNAME, MIDDLENAME und LASTNAME können verwendet werden, um zusätzliche Benutzereigenschaften anzugeben, wie den Vornamen, den zweiten Vornamen bzw. den Nachnamen der Person. Sie sind nur einfache VARCHAR(31)-Felder und können verwendet werden, um alles zu speichern, was Sie bevorzugen.

Wenn die GRANT ADMIN ROLE-Klausel angegeben ist, wird das neue Benutzerkonto mit den Rechten der RDB$ADMIN-Rolle in der Sicherheitsdatenbank (security3.fdb oder datenbankspezifisch) erstellt. Es ermöglicht dem neuen Benutzer, Benutzerkonten von jeder regulären Datenbank aus zu verwalten, bei der er sich anmeldet, gewährt dem Benutzer jedoch keine besonderen Berechtigungen für Objekte in diesen Datenbanken.

Die REVOKE ADMIN ROLE-Klausel ist in einer CREATE USER-Anweisung syntaktisch gültig, hat aber keine Wirkung. Es ist nicht möglich, GRANT ADMIN ROLE und REVOKE ADMIN ROLE in einer Anweisung anzugeben.

Die ACTIVE-Klausel gibt an, dass der Benutzer aktiv ist und sich anmelden kann, dies ist die Standardeinstellung.

Die INACTIVE-Klausel gibt an, dass der Benutzer inaktiv ist und sich nicht anmelden kann. Es ist nicht möglich, ACTIVE und INACTIVE in einer Anweisung anzugeben. Die Option ACTIVE/INACTIVE wird vom Legacy_UserManager nicht unterstützt und wird ignoriert.

Die USING PLUGIN-Klausel gibt explizit das Benutzer-Manager-Plugin an, das zum Erstellen des Benutzers verwendet werden soll. Nur Plugins, die in der UserManager-Konfiguration für diese Datenbank aufgelistet sind (firebird.conf, oder überschrieben in databases.conf) sind gültig. Der Standardbenutzermanager (erster in der UserManager-Konfiguration) wird angewendet, wenn diese Klausel nicht angegeben wird.

Benutzer mit demselben Namen, die mit verschiedenen Benutzermanager-Plugins erstellt wurden, sind unterschiedliche Objekte. Daher kann der Benutzer, der mit einem Benutzermanager-Plugin erstellt wurde, nur von demselben Plugin

geändert oder gelöscht werden.

Aus der Perspektive des Besitzes und der in einer Datenbank gewährten Berechtigungen und Rollen werden verschiedene Benutzerobjekte mit demselben Namen als ein und derselbe Benutzer betrachtet.

Die `TAGS`-Klausel kann verwendet werden, um zusätzliche Benutzerattribute anzugeben. Benutzerdefinierte Attribute werden vom `Legacy_UserManager` nicht unterstützt (stillschweigend ignoriert). Benutzerdefinierte Attributnamen folgen den Regeln von Firebird-Identifikatoren, werden jedoch nicht zwischen Groß- und Kleinschreibung gehandhabt (wenn Sie beispielsweise sowohl `"A BC"` als auch `"a bc"` angeben, wird ein Fehler ausgelöst.)

Der Wert eines benutzerdefinierten Attributs kann eine Zeichenfolge mit maximal 255 Zeichen sein. Die Option `DROP tag_name` ist in `CREATE USER` syntaktisch gültig, verhält sich aber so, als ob die Eigenschaft nicht angegeben wäre.

Benutzer können ihre eigenen benutzerdefinierten Attribute anzeigen und ändern.

`CREATE/ALTER/DROP USER` sind DDL-Anweisungen und werden erst beim Festschreiben wirksam. Denken Sie daran, Ihre Arbeit zu `COMMIT`. In *isql* aktiviert der Befehl `SET AUTO ON` Autocommit für DDL-Anweisungen. In Tools von Drittanbietern und anderen Benutzeranwendungen ist dies möglicherweise nicht der Fall.

Wer kann einen Benutzer erstellen

Um ein Benutzerkonto zu erstellen, muss der aktuelle Benutzer in der Sicherheitsdatenbank über Administratorprivilegien verfügen. Administratorrechte nur in regulären Datenbanken reichen nicht aus.

`CREATE USER`-Beispiele

1. Erstellen eines Benutzers mit dem Benutzernamen `bigshot`:

```
CREATE USER bigshot PASSWORD 'buckshot';
```

2. Erstellen eines Benutzers mit dem `Legacy_UserManager`-Benutzermanager-Plugin

```
CREATE USER godzilla PASSWORD 'robot'
  USING PLUGIN Legacy_UserManager;
```

3. Erstellen des Benutzers `john` mit benutzerdefinierten Attributen:

```
CREATE USER john PASSWORD 'fYe_3Ksw'
  FIRSTNAME 'John' LASTNAME 'Doe'
```

```
  TAGS (BIRTHYEAR='1970', CITY='New York');
```

4. Erstellen eines inaktiven Benutzers:

```
CREATE USER john PASSWORD 'fYe_3Ksw'
  INACTIVE;
```

5. Erstellen des Benutzers `superuser` mit Benutzerverwaltungsrechten:

```
CREATE USER superuser PASSWORD 'kMn8Kjh'
GRANT ADMIN ROLE;
```

Siehe auch

`ALTER USER`, `CREATE OR ALTER USER`, `DROP USER`

13.2.2. ALTER USER

Verwendet für

Ändern eines Firebird-Benutzerkontos

Verfügbar in

DSQL

Syntax

```
ALTER {USER username | CURRENT USER}
  [SET] [<user_option> [<user_option> ...]]
  [TAGS (<user_var> [, <user_var> ...]]

<user_option> ::=
    PASSWORD 'password'
  | FIRSTNAME 'firstname'
  | MIDDLENAME 'middlename'
  | LASTNAME 'lastname'
  | {GRANT | REVOKE} ADMIN ROLE
  | {ACTIVE | INACTIVE}
  | USING PLUGIN plugin_name

<user_var> ::=
    tag_name = 'tag_value'
  | DROP tag_name
```

Vgl. `CREATE USER` für Details der Anweisungsparameter.

Die `ALTER USER`-Anweisung ändert die Details im benannten Firebird-Benutzerkonto. Die `ALTER USER` -Anweisung muss mindestens eine der optionalen Klauseln außer `USING PLUGIN` enthalten.

Jeder Benutzer kann sein eigenes Konto ändern, mit der Ausnahme, dass nur ein Administrator "ADMIN ROLE GRANT/REVOKE" und "ACTIVE/INACTIVE" verwenden kann.

Alle Klauseln sind optional, aber mindestens eine andere als `USING PLUGIN` muss vorhanden sein:

- Der Parameter 'PASSWORD' dient zum Ändern des Passworts für den Benutzer
- `FIRSTNAME`, `MIDDLENAME` und `LASTNAME` aktualisieren diese optionalen Benutzereigenschaften, wie den Vornamen, zweiten Vornamen bzw. Nachnamen der Person
- `GRANT ADMIN ROLE` gewährt dem Benutzer die Privilegien der `RDB$ADMIN` Rolle in der Sicherheitsdatenbank (`security3.fdb`) und ermöglicht es ihm, die Konten anderer Benutzer zu verwalten. Es gewährt dem Benutzer keine besonderen Privilegien in regulären Datenbanken.
- `REVOKE ADMIN ROLE` entfernt den Administrator des Benutzers in der Sicherheitsdatenbank, die diesem Benutzer, sobald die Transaktion festgeschrieben ist, die Möglichkeit verweigert, Benutzerkonten außer seinem eigenen zu ändern
- `ACTIVE` aktiviert ein deaktiviertes Konto (nicht unterstützt für `Legacy_UserManager`)
- `INACTIVE` deaktiviert ein Konto (nicht unterstützt für `Legacy_UserManager`). Dies ist praktisch, um ein Konto vorübergehend zu deaktivieren, ohne es zu löschen.
- `USING PLUGIN` gibt das zu verwendende Benutzermanager-Plugin an
- `TAGS` kann verwendet werden, um zusätzliche benutzerdefinierte Attribute hinzuzufügen, zu aktualisieren oder zu entfernen (`DROP`) (nicht unterstützt für `Legacy_UserManager`). Nicht aufgeführte Attribute werden nicht geändert.

Vgl. `CREATE USER` für weitere Details dieser Klausel.

Wenn Sie Ihr eigenes Konto ändern müssen, können Sie anstelle des Namens des aktuellen Benutzers die Klausel `CURRENT USER` verwenden.

Die Anweisung `ALTER CURRENT USER` folgt den normalen Regeln für die Auswahl des Benutzermanager-Plugins. Wenn der aktuelle Benutzer mit einem nicht standardmäßigen Benutzermanager-Plugin erstellt wurde, müssen die Benutzermanager-Plugins explizit `USING PLUGIN plugin_name` angeben, oder es wird eine Fehlermeldung ausgegeben, die anzeigt, dass der Benutzer nicht gefunden wurde. Wenn ein Benutzer mit demselben Namen für den Standardbenutzermanager vorhanden ist, ändern Sie stattdessen diesen Benutzer.

Denken Sie daran, Ihre Arbeit festzuschreiben (mittels Commit), wenn Sie in einer Anwendung arbeiten, die DDL nicht automatisch festschreibt.

Wer kann einen Benutzer ändern?

Um das Konto eines anderen Benutzers zu ändern, muss der aktuelle Benutzer über Administratorrechte in der Sicherheitsdatenbank verfügen. Jeder kann sein eigenes Konto ändern, mit Ausnahme der Optionen "GRANT/REVOKE ADMIN ROLE" und "ACTIVE/INACTIVE", die zum Ändern Administratorrechte erfordern.

`ALTER USER`-Beispiele

1. Ändern des Passworts für den Benutzer `bobby` und Erteilen von Benutzerverwaltungsrechten:

```
ALTER USER bobby PASSWORD '67-UiT_G8'
GRANT ADMIN ROLE;
```

2. Bearbeiten der optionalen Eigenschaften (der Vor- und Nachnamen) des Benutzers `dan`:

```
ALTER USER dan
FIRSTNAME 'No_Jack'
LASTNAME 'Kennedy';
```

3. Entziehen der Benutzerverwaltungsberechtigungen des Benutzers `Dumbbell`:

```
ALTER USER dumbbell
DROP ADMIN ROLE;
```

Siehe auch

`CREATE USER`, `DROP USER`

13.2.3. `CREATE OR ALTER USER`

Verwendet für

Erstellen eines neuen oder Ändern eines bestehenden Firebird-Benutzerkontos

Verfügbar in

DSQL

Syntax

```
CREATE OR ALTER USER username
  [SET] [<user_option> [<user_option> ...]]
  [TAGS (<user_var> [, <user_var> ...]]

<user_option> ::=
    PASSWORD 'password'
  | FIRSTNAME 'firstname'
  | MIDDLENAME 'middlename'
  | LASTNAME 'lastname'
  | {GRANT | REVOKE} ADMIN ROLE
  | {ACTIVE | INACTIVE}
  | USING PLUGIN plugin_name

<user_var> ::=
    tag_name = 'tag_value'
```

```
  | DROP tag_name
```

Vgl. `CREATE USER` and `ALTER USER` für Details der Anweisungsparameter.

Die Anweisung `CREATE OR ALTER USER` erstellt ein neues Firebird-Benutzerkonto oder ändert die Details des angegebenen. Wenn der Benutzer nicht existiert, wird er erstellt, als ob die Anweisung `CREATE USER` ausgeführt würde. Wenn der Benutzer bereits existiert, wird er so geändert, als ob die Anweisung `ALTER USER` ausgeführt würde. Die Anweisung `CREATE OR ALTER USER` muss mindestens eine der optionalen Klauseln außer `USING PLUGIN` enthalten. Wenn der Benutzer noch nicht existiert, ist die Klausel 'PASSWORD' erforderlich.

Denken Sie daran, Ihre Arbeit festzuschreiben, wenn Sie in einer Anwendung arbeiten, die DDL nicht automatisch festschreibt.

`CREATE OR ALTER USER`-Beispiele

Erstellen oder Ändern eines Benutzers

```
CREATE OR ALTER USER john PASSWORD 'fYe_3Ksw'
FIRSTNAME 'John'
LASTNAME 'Doe'
INACTIVE;
```

Vgl.

`CREATE USER`, `ALTER USER`, `DROP USER`

13.2.4. `DROP USER`

Verwendet für

Löschen eines Firebird-Benutzerkontos

Verfügbar in

DSQL

Syntax

```
DROP USER username
  [USING PLUGIN plugin_name]
```

Tabelle 223. `DROP USER`-Anweisungsparameter

Parameter	Beschreibung
username	Benutzername
plugin_name	Name des Benutzermanager-Plugins

Die Anweisung `DROP USER` löscht ein Firebird-Benutzerkonto.

Die optionale USING PLUGIN-Klausel gibt explizit das User-Manager-Plugin an, das zum Löschen des Benutzers verwendet werden soll. Nur Plugins, die in der UserManager-Konfiguration für diese Datenbank aufgelistet sind (firebird.conf, oder überschrieben in databases.conf) sind gültig. Der Standardbenutzermanager (erster in der UserManager-Konfiguration) wird angewendet, wenn diese Klausel nicht angegeben wird.

Benutzer mit demselben Namen, die mit verschiedenen Benutzermanager-Plugins erstellt wurden, sind unterschiedliche Objekte. Daher kann der Benutzer, der mit einem Benutzermanager-Plugin erstellt wurde, nur von demselben Plugin gelöscht werden.

Denken Sie daran, Ihre Arbeit festzuschreiben, wenn Sie in einer Anwendung arbeiten, die DDL nicht automatisch festschreibt.

Wer kann einen Benutzer löschen?

Um einen Benutzer zu löschen, muss der aktuelle Benutzer über Administratorrechte verfügen.

DROP USER-Beispiele

1. Benutzer bobby löschen:

```
DROP USER bobby;
```

2. Entfernen eines Benutzers, der mit dem Legacy_UserManager-Plugin erstellt wurde:

```
DROP USER Godzilla
  USING PLUGIN Legacy_UserManager;
```

Vgl.

CREATE USER, ALTER USER

13.3. SQL-Privilegien

Die zweite Ebene des Firebird-Sicherheitsmodells sind SQL-Berechtigungen. Während eine erfolgreiche Anmeldung - die erste Ebene - den Zugriff eines Benutzers auf den Server und auf alle Datenbanken unter diesem Server autorisiert, bedeutet dies nicht, dass der Benutzer Zugriff auf Objekte in Datenbanken hat. Wenn ein Objekt erstellt wird, haben nur der Benutzer, der es erstellt hat (sein Besitzer) und Administratoren Zugriff darauf. Der Benutzer benötigt *privileges* für jedes Objekt, auf das er zugreifen muss. Als allgemeine Regel müssen Rechte einem Benutzer explizit vom Objektbesitzer oder einem administrator der Datenbank *gewährt* werden.

Ein Privileg besteht aus einer DML-Zugriffsart (SELECT, INSERT, UPDATE, DELETE, EXECUTE und REFERENCES), dem Namen eines Datenbankobjekts (Tabelle, View, Prozedur, Rolle) und der Name des Zuwendungsempfängers (Benutzer, Verfahren, Auslöser, Rolle). Es stehen verschiedene Mittel zur Verfügung, um mehreren Benutzern in einer einzigen "GRANT"-Anweisung mehrere Arten von

Zugriff auf ein Objekt zu gewähren. Berechtigungen können von einem Benutzer mit "REVOKE" -Anweisungen widerrufen werden.

Eine zusätzliche Art von Berechtigungen, DDL-Berechtigungen, bietet Rechte zum Erstellen, Ändern oder Löschen bestimmter Arten von Metadatenobjekten

Berechtigungen werden in der Datenbank gespeichert, für die sie gelten, und gelten nicht für andere Datenbanken, mit Ausnahme der DDL-Berechtigungen `DATABASE`, die in der Sicherheitsdatenbank gespeichert sind.

13.3.1. Der Objektbesitzer

Der Benutzer, der ein Datenbankobjekt erstellt hat, wird sein Besitzer. Nur der Eigentümer eines Objekts und Benutzer mit Administratorrechten in der Datenbank, einschließlich des Datenbankeigentümers, können das Datenbankobjekt ändern oder löschen.

Administratoren, der Datenbankeigentümer oder der Objekteigentümer können anderen Benutzern Berechtigungen erteilen und diese widerrufen, einschließlich der Berechtigungen, anderen Benutzern Berechtigungen zu erteilen. Der Prozess des Erteilens und Widerrufens von SQL-Berechtigungen wird mit zwei Anweisungen implementiert, `GRANT` und `REVOKE`.

13.4. ROLE

Eine *Rolle* ist ein Datenbankobjekt, das einen Satz von Privilegien verpackt. Rollen implementieren das Konzept der Zugriffskontrolle auf Gruppenebene. Der Rolle werden mehrere Berechtigungen gewährt, und diese Rolle kann dann einem oder mehreren Benutzern gewährt oder entzogen werden.

Ein Benutzer, dem eine Rolle zugewiesen wurde, muss diese Rolle in seinen Anmeldeinformationen angeben, um die zugehörigen Berechtigungen auszuüben. Alle anderen Berechtigungen, die dem Benutzer direkt gewährt werden, sind von seiner Anmeldung mit der Rolle nicht betroffen. Die gleichzeitige Anmeldung mit mehreren Rollen wird nicht unterstützt.

In diesem Abschnitt werden die Aufgaben zum Erstellen und Löschen von Rollen erläutert.

13.4.1. CREATE ROLE

Verwendet für

Erstellen eines neuen `ROLE`-Objekts

Verfügbar in

DSQL, ESQL

Syntax

```
CREATE ROLE rolename
```

Tabelle 224. `CREATE ROLE`*-Anweisungsparameter*

Parameter	Beschreibung
rolename	Rollenname Die maximale Länge beträgt 31 Zeichen

Die Anweisung `CREATE ROLE` erzeugt ein neues Rollenobjekt, dem nachträglich ein oder mehrere Privilegien erteilt werden können. Der Name einer Rolle muss unter den Rollennamen in der aktuellen Datenbank eindeutig sein.

Es ist ratsam, den Namen einer Rolle auch unter den Benutzernamen eindeutig zu machen. Das System verhindert nicht die Erstellung einer Rolle, deren Name mit einem bestehenden Benutzernamen kollidiert, aber in diesem Fall kann der Benutzer keine Verbindung zur Datenbank herstellen.

Wer kann eine Rolle erstellen?

Die `CREATE ROLE`-Anweisung kann ausgeführt werden durch:

- Administratoren
- Benutzer mit der Berechtigung `CREATE ROLE`

Der Benutzer, der die `CREATE ROLE`-Anweisung ausführt, wird Eigentümer der Rolle.

`CREATE ROLE`-Beispiele

Erstellen einer Rolle namens `SELLERS`

```
CREATE ROLE SELLERS;
```

Siehe auch

`DROP ROLE`, `GRANT`, `REVOKE`

13.4.2. `ALTER ROLE`

Verwendet für

Ändern einer Rolle (Aktivieren oder Deaktivieren der automatischen Administratorzuordnung)

Verfügbar in

DSQL

Syntax

```
ALTER ROLE rolename
  {SET | DROP} AUTO ADMIN MAPPING
```

Tabelle 225. `ALTER ROLE`*-Anweisungsparameter*

Parameter	Beschreibung
rolename	Rollenname; die Angabe von etwas anderem als `RDB$ADMIN` schlägt fehl

`ALTER ROLE` hat keinen Platz im Create-Alter-Drop-Paradigma für Datenbankobjekte, da eine Rolle keine Attribute hat, die geändert werden können. Seine eigentliche Wirkung besteht darin, ein Attribut der Datenbank zu ändern: Firebird verwendet es, um die Fähigkeit für Windows-Administratoren zu aktivieren und zu deaktivieren, beim Anmelden automatisch administratorprivilegien zu übernehmen.

Diese Fähigkeit kann nur eine Rolle betreffen: die vom System generierte Rolle `RDB$ADMIN`, die in jeder Datenbank von ODS 11.2 oder höher vorhanden ist. Mehrere Faktoren sind an der Aktivierung dieser Funktion beteiligt.

Weitere Informationen finden Sie unter `AUTO ADMIN MAPPING`.

Wer kann eine Rolle ändern?

Die `ALTER ROLE`-Anweisung kann ausgeführt werden durch:

- Administratoren

Obwohl ein DDL-Privileg "ALTER ANY ROLE" vorhanden ist, gilt es nicht, da das Erstellen oder Löschen von Zuordnungen Administratorrechte erfordert.

13.4.3. DROP ROLE

Verwendet für

Eine Rolle löschen

Verfügbar in

DSQL, ESQL

Syntax

```
DROP ROLE rolename
```

Die Anweisung `DROP ROLE` löscht eine bestehende Rolle. Es braucht nur ein einziges Argument, den Namen der Rolle. Nachdem die Rolle gelöscht wurde, wird allen Benutzern und Objekten, denen die Rolle gewährt wurde, der gesamte Satz von Berechtigungen entzogen.

Wer kann eine Rolle löschen?

Die `DROP ROLE`-Anweisung kann ausgeführt werden durch:

- Administratoren
- Der Besitzer der Rolle
- Benutzer mit dem `DROP ANY ROLE`-Privileg

`DROP ROLE`-Beispiele

Löschen der Rolle SELLERS

```
DROP ROLE SELLERS;
```

Siehe auch

`CREATE ROLE`, `GRANT`, `REVOKE`

13.5. Anweisungen zum Erteilen von Rechten

Eine `GRANT`-Anweisung wird verwendet, um Benutzern und anderen Datenbankobjekten Berechtigungen – einschließlich Rollen – zu erteilen.

13.5.1. GRANT

Verwendet für

Berechtigungen erteilen und Rollen zuweisen

Verfügbar in

DSQL, ESQL

Syntax (Rechte erteilen)

```
GRANT <privileges>
  TO <grantee_list>
  [WITH GRANT OPTION]
  [{GRANTED BY | AS} [USER] grantor]

<privileges> ::=
    <table_privileges> | <execute_privilege>
  | <usage_privilege>  | <ddl_privileges>
  | <db_ddl_privilege>

<table_privileges> ::=
  {ALL [PRIVILEGES] | <table_privilege_list> }
    ON [TABLE] {table_name | view_name}

<table_privilege_list> ::=
  <table_privilege> [, <tableprivilege> ...]

<table_privilege> ::=
    SELECT | DELETE | INSERT
  | UPDATE [(col [, col ...])]
  | REFERENCES [(col [, col ...)]

<execute_privilege> ::= EXECUTE ON
  { PROCEDURE proc_name | FUNCTION func_name
  | PACKAGE package_name }

<usage_privilege> ::= USAGE ON
```

```
  { EXCEPTION exception_name
  | {GENERATOR | SEQUENCE} sequence_name }

<ddl_privileges> ::=
  {ALL [PRIVILEGES] | <ddl_privilege_list>} <object_type>

<ddl_privilege_list> ::=
  <ddl_privilege> [, <ddl_privilege> ...]

<ddl_privilege> ::= CREATE | ALTER ANY | DROP ANY

<object_type> ::=
    CHARACTER SET | COLLATION | DOMAIN | EXCEPTION
  | FILTER | FUNCTION | GENERATOR | PACKAGE
  | PROCEDURE | ROLE | SEQUENCE | TABLE | VIEW

<db_ddl_privileges> ::=
  {ALL [PRIVILEGES] | <db_ddl_privilege_list>} {DATABASE | SCHEMA}

<db_ddl_privilege_list> ::=
  <db_ddl_privilege> [, <db_ddl_privilege> ...]

<db_ddl_privilege> ::= CREATE | ALTER | DROP

<grantee_list> ::= <grantee> [, <grantee> ...]

<grantee> ::=
    PROCEDURE proc_name  | FUNCTION func_name
  | PACKAGE package_name | TRIGGER trig_name
  | VIEW view_name       | ROLE role_name
  | [USER] username      | GROUP Unix_group
```

Syntax (Rollenzzuweisung)

```
GRANT <role_granted>
  TO <role_grantee_list>
  [WITH ADMIN OPTION]
  [{GRANTED BY | AS} [USER] grantor]

<role_granted> ::= role_name [, role_name ...]

<role_grantee_list> ::=
  <role_grantee> [, <role_grantee> ...]

<role_grantee> ::= [USER] username
```

Tabelle 226. GRANT-*Anweisungsparameter*

Parameter	Beschreibung
grantor	Der Benutzer, der die Berechtigung(en) gewährt

Parameter	Beschreibung
table_name	Tabellenname
view_name	Der Name der View
col	Name der Tabellenspalte
proc_name	Name der Stored Procedure
func_name	Name der Stored Function (oder UDF)
package_name	Name des Pakets
exception_name	Der Name einer Ausnahme (Exception)
sequence_name	Der Name einer Sequenz (Generator)
object_type	Der Typ des Metadatenobjekts
trig_name	Der Name eines Triggers
role_name	Rollenname
username	Der Benutzername, dem die Berechtigungen erteilt werden oder dem die Rolle zugewiesen ist. Wenn das Schlüsselwort `USER` fehlt, kann es auch eine Rolle sein.
Unix_group	Der Name einer Benutzergruppe in einem POSIX-Betriebssystem

Die `GRANT`-Anweisung gewährt Benutzern, Rollen oder anderen Datenbankobjekten ein oder mehrere Privilegien für Datenbankobjekte.

Ein normaler, authentifizierter Benutzer hat keine Privilegien für irgendein Datenbankobjekt, bis sie ihm explizit gewährt werden, entweder diesem einzelnen Benutzer oder allen Benutzern, die als Benutzer "PUBLIC" gebündelt sind. Wenn ein Objekt erstellt wird, haben nur sein Ersteller (der Eigentümer) und Administratoren Berechtigungen dafür und können anderen Benutzern, Rollen oder Objekten Berechtigungen erteilen.

Für unterschiedliche Typen von Metadatenobjekten gelten unterschiedliche Berechtigungen. Die verschiedenen Arten von Berechtigungen werden später in diesem Abschnitt separat beschrieben.

`SCHEMA` ist derzeit ein Synonym für `DATABASE`; dies kann sich in einer zukünftigen Version ändern, daher empfehlen wir immer `DATABASE` zu verwenden

Die `TO`-Klausel

Die `TO`-Klausel gibt die Benutzer, Rollen und anderen Datenbankobjekte an, denen die in *privileges* aufgezählten Privilegien gewährt werden sollen. Die Klausel ist obligatorisch.

Mit dem optionalen Schlüsselwort `USER` in der `TO`-Klausel können Sie genau angeben, wem oder was die Berechtigung erteilt wird. Wenn kein Schlüsselwort `USER` (oder `ROLE`) angegeben ist, sucht der Server zuerst nach einer Rolle mit diesem Namen, und wenn keine solche Rolle vorhanden ist, werden dem Benutzer mit diesem Namen die Privilegien ohne weitere Prüfung gewährt.

Es wird empfohlen, `USER` und `ROLE` immer explizit anzugeben, um

Mehrdeutigkeiten zu vermeiden. Zukünftige Versionen von Firebird können 'BENUTZER' obligatorisch machen.

- Wenn eine GRANT-Anweisung ausgeführt wird, wird die Sicherheitsdatenbank nicht auf die Existenz des Grantee-Benutzers überprüft. Dies ist kein Fehler: SQL-Berechtigungen betreffen die Kontrolle des Datenzugriffs für authentifizierte Benutzer, sowohl native als auch vertrauenswürdige, und vertrauenswürdige Betriebssystembenutzer werden nicht in der Sicherheitsdatenbank gespeichert.
- Wenn Sie einem anderen Datenbankobjekt als einem Benutzer oder einer Rolle, wie einer Prozedur, einem Auslöser oder einer Ansicht, eine Berechtigung erteilen, müssen Sie den Objekttyp angeben.
- Obwohl das Schlüsselwort USER optional ist, empfiehlt es sich, es zu verwenden, um Mehrdeutigkeiten bei Rollen zu vermeiden.

Verpacken von Privilegien in einem ROLE-Objekt

Eine Rolle ist ein “Container”-Objekt, das verwendet werden kann, um eine Sammlung von Berechtigungen zu packen. Die Verwendung der Rolle wird dann jedem Benutzer gewährt, der diese Berechtigungen benötigt. Eine Rolle kann auch einer Liste von Benutzern zugewiesen werden.

Die Rolle muss vorhanden sein, bevor ihr Berechtigungen erteilt werden können. Syntax und Regeln finden Sie unter CREATE ROLE. Die Rolle wird aufrechterhalten, indem ihr Berechtigungen erteilt und ihr bei Bedarf Berechtigungen entzogen werden. Wenn eine Rolle gelöscht wird (siehe DROP ROLE), verlieren alle Benutzer die durch die Rolle erworbenen Berechtigungen. Alle Rechte, die einem betroffenen Benutzer zusätzlich durch eine andere grant-Anweisung gewährt wurden, bleiben erhalten.

Ein Benutzer, dem eine Rolle zugewiesen wurde, muss diese Rolle mit seinen Anmeldeinformationen angeben, um die zugehörigen Berechtigungen auszuüben. Alle anderen dem Benutzer gewährten Berechtigungen werden durch die Anmeldung mit einer Rolle nicht beeinflusst.

Einem Benutzer kann mehr als eine Rolle zugewiesen werden, die gleichzeitige Anmeldung mit mehreren Rollen wird jedoch nicht unterstützt.

Eine Rolle kann nur einem Benutzer zugewiesen werden.

Der Benutzer PUBLIC

Firebird hat einen vordefinierten Benutzer namens PUBLIC, der alle Benutzer repräsentiert. Privilegien für Operationen an einem bestimmten Objekt, die dem Benutzer "PUBLIC" gewährt werden, können von jedem authentifizierten Benutzer ausgeübt werden.

Wenn dem Benutzer PUBLIC Privilegien gewährt werden, sollten diese auch dem Benutzer PUBLIC entzogen werden.

Die `WITH GRANT OPTION`-Klausel

Die optionale `WITH GRANT OPTION`-Klausel ermöglicht es den in der Benutzerliste angegebenen Benutzern, anderen Benutzern die in der Berechtigungsliste angegebenen Berechtigungen zu erteilen.

Diese Option kann dem Benutzer `PUBLIC` zugewiesen werden. Mach das nicht!

Die `GANTED BY`-Klausel

Wenn Berechtigungen in einer Datenbank gewährt werden, wird standardmäßig der aktuelle Benutzer als Erteilender aufgezeichnet. Die `GRANTED BY`-Klausel ermöglicht es dem aktuellen Benutzer, diese Privilegien als anderen Benutzer zu erteilen.

Wenn die `REVOKE`-Anweisung verwendet wird, schlägt sie fehl, wenn der aktuelle Benutzer nicht der Benutzer ist, der in der `GRANTED BY`-Klausel genannt wurde.

Die `GRANTED BY` (und `AS`)-Klausel kann nur vom Datenbankbesitzer und anderen Administratoren verwendet werden. Der Objektbesitzer kann `GRANTED BY` nur verwenden, wenn er auch über Administratorrechte verfügt.

Alternative Syntax mit `AS username`

Die nicht standardmäßige `AS`-Klausel wird als Synonym der `GRANTED BY`-Klausel unterstützt, um die Migration von anderen Datenbanksystemen zu vereinfachen.

Berechtigungen für Tabellen und Ansichten (Views)

Für Tabellen und Views ist es im Gegensatz zu anderen Metadatenobjekten möglich, mehrere Privilegien gleichzeitig zu erteilen.

Liste der Berechtigungen für Tabellen

`SELECT`
: Erlaubt dem Benutzer oder Objekt, Daten aus der Tabelle oder Ansicht auszuwählen

`INSERT`
: Erlaubt dem Benutzer oder Objekt, Zeilen in die Tabelle oder Ansicht einzufügen

`DELETE`
: Ermöglicht dem Benutzer oder Objekt das Löschen von Zeilen aus der Tabelle oder Ansicht or

`UPDATE`
: Erlaubt dem Benutzer oder Objekt, Zeilen in der Tabelle oder Ansicht zu aktualisieren, optional auf bestimmte Spalten beschränkt specific

`REFERENCES`
: Erlaubt dem Benutzer oder Objekt, die Tabelle über einen Fremdschlüssel zu referenzieren, optional beschränkt auf die angegebenen Spalten. Wenn der primäre oder eindeutige Schlüssel, auf den der Fremdschlüssel der anderen Tabelle verweist, zusammengesetzt ist, müssen alle

Spalten des Schlüssels angegeben werden.

`ALL [PRIVILEGES]`

Kombiniert die Privilegien `SELECT`, `INSERT`, `UPDATE`, `DELETE` und `REFERENCES` in einem einzigen Paket

Beispiele für `GRANT <privilege>` auf Tabellen

1. `SELECT`- und `INSERT`-Berechtigungen für Benutzer `ALEX`:

```
GRANT SELECT, INSERT ON TABLE SALES
  TO USER ALEX;
```

2. Das `SELECT`-Privileg für die Rollen `MANAGER` sowie `ENGINEER` und für den Benutzer `IVAN`:

```
GRANT SELECT ON TABLE CUSTOMER
  TO ROLE MANAGER, ROLE ENGINEER, USER IVAN;
```

3. Alle Berechtigungen für die Rolle "ADMINISTRATOR", zusammen mit der Berechtigung, anderen dieselben Berechtigungen zu erteilen:

```
GRANT ALL ON TABLE CUSTOMER
  TO ROLE ADMINISTRATOR
  WITH GRANT OPTION;
```

4. Die `SELECT`- sowie `REFERENCES`-Privilegien in der Spalte `NAME` für alle Benutzer und Objekte:

```
GRANT SELECT, REFERENCES (NAME) ON TABLE COUNTRY
TO PUBLIC;
```

5. Das `SELECT`-Privileg wird dem Benutzer `IVAN` vom Benutzer `ALEX` gewährt:

```
GRANT SELECT ON TABLE EMPLOYEE
  TO USER IVAN
  GRANTED BY ALEX;
```

6. Gewähren der Berechtigung `UPDATE` für die Spalten `FIRST_NAME`, `LAST_NAME`:

```
GRANT UPDATE (FIRST_NAME, LAST_NAME) ON TABLE EMPLOYEE
  TO USER IVAN;
```

7. Gewähren der `INSERT`-Berechtigung für die gespeicherte Prozedur `ADD_EMP_PROJ`:

```
GRANT INSERT ON EMPLOYEE_PROJECT
```

```
  TO PROCEDURE ADD_EMP_PROJ;
```

Die EXECUTE-Berechtigung

Das Privileg EXECUTE gilt für gespeicherte Prozeduren, gespeicherte Funktionen (einschließlich UDFs) und Pakete. Es ermöglicht dem Empfänger, das angegebene Objekt auszuführen und gegebenenfalls seine Ausgabe abzurufen.

Im Fall von auswählbaren gespeicherten Prozeduren verhält es sich insofern wie ein SELECT -Privileg, insofern diese Art von gespeicherter Prozedur als Reaktion auf eine SELECT-Anweisung ausgeführt wird.

Bei Paketen kann das `EXECUTE'-Privileg nur für das gesamte Paket vergeben werden, nicht für einzelne Unterprogramme.

Beispiele für die Gewährung des EXECUTE-Privilegs

1. Einer Rolle das Privileg EXECUTE für eine gespeicherte Prozedur gewähren:

```
GRANT EXECUTE ON PROCEDURE ADD_EMP_PROJ
  TO ROLE MANAGER;
```

2. Einer Rolle das Privileg EXECUTE für eine gespeicherte Funktion gewähren:

```
GRANT EXECUTE ON FUNCTION GET_BEGIN_DATE
  TO ROLE MANAGER;
```

3. Gewähren des EXECUTE-Privilegs für ein Paket an den Benutzer PUBLIC:

```
GRANT EXECUTE ON PACKAGE APP_VAR
  TO USER PUBLIC;
```

4. Erteilen des EXECUTE-Privilegs für eine Funktion an ein Paket:

```
GRANT EXECUTE ON FUNCTION GET_BEGIN_DATE
  TO PACKAGE APP_VAR;
```

Das USAGE-Privileg

Um andere Metadatenobjekte als Tabellen, Ansichten, gespeicherte Prozeduren oder Funktionen, Trigger und Pakete verwenden zu können, ist es notwendig, dem Benutzer (oder Datenbankobjekten wie Trigger, Prozedur oder Funktion) das USAGE-Privileg für diese Objekte zu gewähren.

Da Firebird gespeicherte Prozeduren und Funktionen, Trigger und Paketroutinen mit den Rechten

des Aufrufers ausführt, ist es notwendig, dass entweder der Benutzer oder die Routine selbst das USAGE-Privileg besitzt.

In Firebird 3.0 ist das Privileg USAGE nur für Ausnahmen und Sequenzen verfügbar (in gen_id(gen_name, n) oder `nächster Wert für gen_name). Die Unterstützung des USAGE-Privilegs für andere Metadatenobjekte könnte in zukünftigen Versionen hinzugefügt werden.

Für Sequenzen (Generatoren) gewährt das Privileg USAGE nur das Recht, die Sequenz mit der Funktion GEN_ID oder NEXT VALUE FOR zu inkrementieren. Die Anweisung SET GENERATOR ist ein Synonym für ALTER SEQUENCE ⋯ RESTART WITH ⋯ und gilt als DDL-Anweisung. Standardmäßig haben nur der Besitzer der Sequenz und Administratoren die Rechte für solche Operationen. Das Recht, den Anfangswert einer beliebigen Sequenz zu setzen, kann mit GRANT ALTER ANY SEQUENCE gewährt werden, was für allgemeine Benutzer nicht empfohlen wird.

Beispiele für die Gewährung des USAGE-Privilegs

1. Einer Rolle das Privileg USAGE für eine Sequenz gewähren:

```
GRANT USAGE ON SEQUENCE GEN_AGE
  TO ROLE MANAGER;
```

2. Gewähren des USAGE-Privilegs für eine Sequenz an einen Trigger:

```
GRANT USAGE ON SEQUENCE GEN_AGE
  TO TRIGGER TR_AGE_BI;
```

3. Gewähren des Privilegs USAGE für eine Ausnahme für ein Paket:

```
GRANT USAGE ON EXCEPTION
  TO PACKAGE PKG_BILL;
```

DDL Privileges

Standardmäßig können nur Administratoren neue Metadatenobjekte erstellen; Das Ändern oder Löschen dieser Objekte ist auf den Eigentümer des Objekts (seinen Ersteller) und Administratoren beschränkt. DDL-Berechtigungen können verwendet werden, um anderen Benutzern Berechtigungen für diese Vorgänge zu erteilen.

Verfügbare DDL-Berechtigungen

`CREATE`

Ermöglicht die Erstellung eines Objekts des angegebenen Typs

`ALTER ANY`
: Ermöglicht die Änderung jedes Objekts des angegebenen Typs

`DROP ANY`
: Ermöglicht das Löschen jedes Objekts des angegebenen Typs

`ALL [PRIVILEGES]`
: Kombiniert die Berechtigungen `CREATE`, `ALTER ANY` und `DROP ANY` für den angegebenen Typ

Es gibt keine separaten DDL-Berechtigungen für Trigger und Indizes. Die erforderlichen Berechtigungen werden von der Tabelle oder Sicht geerbt. Das Erstellen, Ändern oder Löschen eines Triggers oder Index erfordert das Privileg `ALTER ANY TABLE` oder `ALTER ANY VIEW`.

Beispiele für die Gewährung von DDL-Berechtigungen

1. Erlaube dem Benutzer `JOE`, Tabellen zu erstellen

```
GRANT CREATE TABLE
  TO USER Joe;
```

2. Erlaube dem Benutzer `JOE`, jede Prozedur zu ändern

```
GRANT ALTER ANY PROCEDURE
  TO USER Joe;
```

Datenbank-DDL-Berechtigungen

Die Syntax für die Vergabe von Berechtigungen zum Erstellen, Ändern oder Löschen einer Datenbank weicht von der normalen Syntax für die Vergabe von DDL-Berechtigungen für andere Objekttypen ab.

Verfügbare Datenbank-DDL-Berechtigungen

`CREATE`
: Ermöglicht die Erstellung einer Datenbank

`ALTER`
: Ermöglicht die Änderung der aktuellen Datenbank

`DROP`
: Ermöglicht das Löschen der aktuellen Datenbank

`ALL [PRIVILEGES]`
: Kombiniert die Berechtigungen `ALTER` und `DROP`. `ALL` beinhaltet nicht das `CREATE`-Privileg.

Die Berechtigungen `ALTER DATABASE` und `DROP DATABASE` gelten nur für die aktuelle Datenbank,

während die DDL-Berechtigungen ALTER ANY und DROP ANY für andere Objekttypen für alle Objekte des angegebenen Typs in der aktuellen Datenbank gelten. Die Berechtigung zum Ändern oder Löschen der aktuellen Datenbank kann nur von Administratoren erteilt werden.

Das Privileg CREATE DATABASE ist ein besonderes Privileg, da es in der Sicherheitsdatenbank gespeichert wird. Eine Liste von Benutzern mit dem Privileg CREATE DATABASE ist in der virtuellen Tabelle SEC$DB_CREATORS verfügbar. Nur Administratoren in der Sicherheitsdatenbank kann die Berechtigung zum Anlegen einer neuen Datenbank erteilen.

SCHEMA ist derzeit ein Synonym für DATABASE; dies kann sich in einer zukünftigen Version ändern, daher empfehlen wir immer DATABASE zu verwenden

Beispiele für die Gewährung von Datenbank-DDL-Berechtigungen

1. Gewähren von SUPERUSER die Berechtigung zum Erstellen von Datenbanken:

```
GRANT CREATE DATABASE
  TO USER Superuser;
```

2. Gewähren Sie JOE das Recht, ALTER DATABASE für die aktuelle Datenbank auszuführen:

```
GRANT ALTER DATABASE
  TO USER Joe;
```

3. Gewähren Sie FEDOR das Recht, die aktuelle Datenbank zu löschen:

```
GRANT DROP DATABASE
  TO USER Fedor;
```

Rollen zuweisen

Das Zuweisen einer Rolle ähnelt dem Erteilen einer Berechtigung. Eine oder mehrere Rollen können einem oder mehreren Benutzern zugewiesen werden, einschließlich des Benutzer PUBLIC, mit einer GRANT-Anweisung.

Die WITH ADMIN OPTION-Klausel

Die optionale WITH ADMIN OPTION-Klausel ermöglicht es den in der Benutzerliste angegebenen Benutzern, anderen Benutzern die angegebene(n) Rolle(n) zu erteilen.

Es ist möglich, diese Option PUBLIC zuzuweisen. Tun Sie das nicht!

Beispiele für Rollenzuweisungen

1. Dem Benutzer IVAN die Rollen DIRECTOR und MANAGER zuweisen:

```
GRANT DIRECTOR, MANAGER
  TO USER IVAN;
```

2. Zuweisen der Rolle MANAGER an den Benutzer ALEX mit der Berechtigung, diese Rolle anderen Benutzern zuzuweisen:

```
GRANT MANAGER
  TO USER ALEX WITH ADMIN OPTION;
```

Siehe auch

REVOKE

13.6. Anweisungen zum Widerrufen von Berechtigungen

Eine REVOKE-Anweisung wird verwendet, um Berechtigungen – einschließlich Rollen – von Benutzern und anderen Datenbankobjekten zu entziehen.

13.6.1. REVOKE

Verwendet für

Widerrufen von Berechtigungen oder Rollenzuweisungen

Verfügbar in

DSQL, ESQL

Syntax (Privilegien widerrufen)

```
REVOKE [GRANT OPTION FOR] <privileges>
  FROM <grantee_list>
  [{GRANTED BY | AS} [USER] grantor]

<privileges> ::=
  !! Vgl. GRANT-Syntax !!
```

Syntax (Privilegien widerrufen)

```
REVOKE [ADMIN OPTION FOR] <role_granted>
  FROM <role_grantee_list>
  [{GRANTED BY | AS} [USER] grantor]

<role_granted> ::=
  !! Vgl. GRANT-Syntax !!

<role_grantee_list> ::=
```

```
  !! Vgl. GRANT-Syntax !!
```

Syntax (revoking all)

```
REVOKE ALL ON ALL FROM <grantee_list>

<grantee_list> ::=
  !! Vgl. GRANT-Syntax !!
```

Tabelle 227. REVOKE-Anweisungsparameter

Parameter	Beschreibung
grantor	Der erteilende Benutzer, in dessen Namen die Berechtigung(en) widerrufen werden

Die REVOKE-Anweisung entzieht Benutzern, Rollen und anderen Datenbankobjekten Berechtigungen, die mit der GRANT-Anweisung gewährt wurden. Siehe GRANT für detaillierte Beschreibungen der verschiedenen Rechtetypen.

Nur der Benutzer, der die Berechtigung erteilt hat, kann sie widerrufen.

Die FROM-Klausel

Die FROM-Klausel gibt eine Liste von Benutzern, Rollen und anderen Datenbankobjekten an, denen die aufgezählten Berechtigungen entzogen werden. Mit dem optionalen Schlüsselwort USER in der FROM-Klausel können Sie genau angeben, welchem Typ die Berechtigung entzogen werden soll. Wenn kein Schlüsselwort USER (oder ROLE) angegeben ist, sucht der Server zunächst nach einer Rolle mit diesem Namen, und wenn keine solche Rolle vorhanden ist, werden dem Benutzer mit diesem Namen die Berechtigungen ohne weitere Prüfung entzogen.

- Obwohl das Schlüsselwort USER optional ist, ist es ratsam, es zu verwenden, um Mehrdeutigkeiten bei Rollen zu vermeiden.
- Die REVOKE-Anweisung prüft nicht, ob der Benutzer existiert, dem die Privilegien entzogen werden.
- Wenn Sie einem anderen Datenbankobjekt als USER oder ROLE ein Privileg entziehen, müssen Sie dessen Objekttyp angeben

Widerrufen von Privilegien von Benutzer PUBLIC

Privilegien, die dem speziellen Benutzer mit dem Namen PUBLIC gewährt wurden, müssen dem Benutzer PUBLIC entzogen werden. Der Benutzer PUBLIC bietet eine Möglichkeit, allen Benutzern gleichzeitig Berechtigungen zu erteilen, aber es ist keine „Gruppe von Benutzern“.

Widerrufen von GRANT OPTION

Die optionale GRANT OPTION FOR-Klausel entzieht dem Benutzer die Berechtigung, anderen Benutzern, Rollen oder Datenbankobjekten die angegebenen Berechtigungen zu gewähren (wie

zuvor mit WITH GRANT OPTION gewährt). Es entzieht die angegebene Berechtigung nicht selbst.

Entfernen der Berechtigung für eine oder mehrere Rollen

Eine Verwendung der REVOKE-Anweisung besteht darin, Rollen zu entfernen, die einem Benutzer oder einer Benutzergruppe durch eine GRANT-Anweisung zugewiesen wurden. Bei mehreren Rollen und/oder mehreren Grantees folgt auf das Verb REVOKE die Liste der Rollen, die aus der nach der FROM-Klausel angegebenen Benutzerliste entfernt werden.

Die optionale ADMIN OPTION FOR-Klausel bietet die Möglichkeit, dem Berechtigten die Berechtigung "Administrator" zu entziehen, die Möglichkeit, anderen Benutzern dieselbe Rolle zuzuweisen, ohne die Berechtigung des Berechtigten für die Rolle zu widerrufen.

Mehrere Rollen und Empfänger können in einer einzigen Anweisung verarbeitet werden.

Widerrufen von GRANTED BY-Rechten

Ein Privileg, das unter Verwendung der GRANTED BY-Klausel gewährt wurde, wird intern explizit dem durch diese ursprüngliche GRANT-Anweisung bezeichneten Gewährer zugewiesen. Nur dieser Benutzer kann die gewährte Berechtigung widerrufen. Mit der GRANTED BY-Klausel können Sie Berechtigungen entziehen, als ob Sie der angegebene Benutzer wären. Um eine Berechtigung mit GRANTED BY zu entziehen, muss der aktuelle Benutzer entweder mit vollen Administratorrechten oder als der Benutzer, der durch diese GRANTED BY-Klausel als *grantor* bezeichnet wird, angemeldet sein.

Nicht einmal der Besitzer einer Rolle kann GRANTED BY verwenden, es sei denn, er hat Administratorrechte.

Die nicht standardmäßige AS-Klausel wird als Synonym der GRANTED BY-Klausel unterstützt, um die Migration von anderen Datenbanksystemen zu vereinfachen.

Widerrufen von ALL ON ALL

Die REVOKE ALL ON ALL-Anweisung ermöglicht es einem Benutzer, alle Privilegien (einschließlich Rollen) für alle Objekte von einem oder mehreren Benutzern, Rollen oder anderen Datenbankobjekten zu widerrufen. Es ist eine schnelle Möglichkeit zum "Löschen" von Berechtigungen, wenn der Zugriff auf die Datenbank für einen bestimmten Benutzer oder eine bestimmte Rolle gesperrt werden muss.

Wenn der aktuelle Benutzer mit vollen administrator Berechtigungen in der Datenbank angemeldet ist, entfernt REVOKE ALL ON ALL alle Berechtigungen, egal wer sie gewährt hat. Andernfalls werden nur die vom aktuellen Benutzer gewährten Berechtigungen entfernt.

Die Klausel GRANTED BY wird nicht unterstützt

REVOKE-Beispiele

1. Widerruf der Privilegien zum Auswählen und Einfügen in die Tabelle (oder View) SALES

```
REVOKE SELECT, INSERT ON TABLE SALES
  FROM USER ALEX;
```

2. Widerruf der Berechtigung zum Auswählen aus der Tabelle `CUSTOMER` der Rollen `MANAGER` und `ENGINEER` und dem Benutzer `IVAN`:

```
REVOKE SELECT ON TABLE CUSTOMER
  FROM ROLE MANAGER, ROLE ENGINEER, USER IVAN;
```

3. Entziehen der Rolle `ADMINISTRATOR` die Berechtigung, anderen Benutzern oder Rollen Berechtigungen für die Tabelle `CUSTOMER` zu erteilen:

```
REVOKE GRANT OPTION FOR ALL ON TABLE CUSTOMER
  FROM ROLE ADMINISTRATOR;
```

4. Widerruf der Berechtigung zum Auswählen aus der Tabelle `COUNTRY` und der Berechtigung zum Verweisen auf die Spalte `NAME` der Tabelle 'COUNTRY' von jedem Benutzer über den speziellen Benutzer `PUBLIC`:

```
REVOKE SELECT, REFERENCES (NAME) ON TABLE COUNTRY
  FROM PUBLIC;
```

5. Entzug des Privilegs zur Auswahl aus der Tabelle `EMPLOYEE` von dem Benutzer `IVAN`, das dem Benutzer `ALEX` gewährt wurde:

```
REVOKE SELECT ON TABLE EMPLOYEE
  FROM USER IVAN GRANTED BY ALEX;
```

6. Widerruf der Berechtigung zum Aktualisieren der Spalten `FIRST_NAME` und `LAST_NAME` der Tabelle `EMPLOYEE` von dem Benutzer `IVAN`:

```
REVOKE UPDATE (FIRST_NAME, LAST_NAME) ON TABLE EMPLOYEE
  FROM USER IVAN;
```

7. Widerruf der Berechtigung zum Einfügen von Datensätzen in die Tabelle `EMPLOYEE_PROJECT` aus der Prozedur `ADD_EMP_PROJ`:

```
REVOKE INSERT ON EMPLOYEE_PROJECT
  FROM PROCEDURE ADD_EMP_PROJ;
```

8. Widerruf der Berechtigung zum Ausführen der Prozedur `ADD_EMP_PROJ` aus der Rolle `MANAGER`:

```
REVOKE EXECUTE ON PROCEDURE ADD_EMP_PROJ
  FROM ROLE MANAGER;
```

9. Widerruf der Berechtigung, anderen Benutzern aus der Rolle `MANAGER` das `EXECUTE`-Privileg für die Funktion `GET_BEGIN_DATE` zu erteilen:

```
REVOKE GRANT OPTION FOR EXECUTE
  ON FUNCTION GET_BEGIN_DATE
  FROM ROLE MANAGER;
```

10. Widerrufen des `EXECUTE`-Privilegs für das Paket `DATE_UTILS` von Benutzer `ALEX`:

```
REVOKE EXECUTE ON PACKAGE DATE_UTILS
  FROM USER ALEX;
```

11. Aufheben des `USAGE`-Privilegs für die Sequenz `GEN_AGE` aus der Rolle `MANAGER`:

```
REVOKE USAGE ON SEQUENCE GEN_AGE
  FROM ROLE MANAGER;
```

12. Widerrufen des `USAGE`-Privilegs für die Sequenz `GEN_AGE` vom Trigger `TR_AGE_BI`:

```
REVOKE USAGE ON SEQUENCE GEN_AGE
  FROM TRIGGER TR_AGE_BI;
```

13. Widerrufen des `USAGE`-Privilegs für die Ausnahme `E_ACCESS_DENIED` aus dem Paket `PKG_BILL`:

```
REVOKE USAGE ON EXCEPTION E_ACCESS_DENIED
  FROM PACKAGE PKG_BILL;
```

14. Widerruf der Berechtigung zum Erstellen von Tabellen von Benutzer `JOE`:

```
REVOKE CREATE TABLE
  FROM USER Joe;
```

15. Widerruf der Berechtigung zum Ändern einer Prozedur von Benutzer `JOE`:

```
REVOKE ALTER ANY PROCEDURE
  FROM USER Joe;
```

16. Widerruf der Berechtigung zum Erstellen von Datenbanken vom Benutzer `SUPERUSER`:

```
REVOKE CREATE DATABASE
  FROM USER Superuser;
```

17. Entziehen der Rollen DIRECTOR und MANAGER vom Benutzer IVAN:

```
REVOKE DIRECTOR, MANAGER FROM USER IVAN;
```

18. Entziehen Sie dem Benutzer ALEX das Recht, anderen Benutzern die Rolle MANAGER zu erteilen:

```
REVOKE ADMIN OPTION FOR MANAGER FROM USER ALEX;
```

19. Entziehen aller Privilegien (einschließlich Rollen) für alle Objekte von dem Benutzer IVAN:

```
REVOKE ALL ON ALL
  FROM USER IVAN;
```

    Nachdem diese Anweisung von einem Administrator ausgeführt wurde, hat der Benutzer IVAN keinerlei Privilegien, außer denen, die durch PUBLIC gewährt wurden.

Siehe auch
GRANT

13.7. Zuordnung von Benutzern zu Objekten

Da Firebird nun mehrere Sicherheitsdatenbanken unterstützt, treten einige neue Probleme auf, die mit einer einzigen, globalen Sicherheitsdatenbank nicht auftreten könnten. Cluster von Datenbanken, die dieselbe Sicherheitsdatenbank verwenden, wurden effizient getrennt. Mappings bieten die Möglichkeit, dieselbe Effizienz zu erzielen, wenn mehrere Datenbanken ihre eigenen Sicherheitsdatenbanken verwenden. Einige Fälle erfordern eine Kontrolle für eine begrenzte Interaktion zwischen solchen Clustern. Beispielsweise:

- wenn EXECUTE STATEMENT ON EXTERNAL DATA SOURCE einen Datenaustausch zwischen Clustern erfordert
- wenn serverweiter SYSDBA-Zugriff auf Datenbanken von anderen Clustern über Dienste benötigt wird.
- Vergleichbare Probleme, die bei Firebird 2.1 und 2.5 für Windows aufgrund der Unterstützung der vertrauenswürdigen Benutzerauthentifizierung bestanden haben: zwei separate Listen von Benutzern – eine in der Sicherheitsdatenbank und eine andere in Windows, mit Fällen, in denen eine Verknüpfung erforderlich war. Ein Beispiel ist die Forderung nach einer ROLE, die einer Windows-Gruppe gewährt wird, um den Mitgliedern dieser Gruppe automatisch zugewiesen zu werden.

Die einzige Lösung für all diese Fälle ist das **Mapping** der Login-Informationen, die einem Benutzer zugewiesen werden, wenn er sich mit einem Firebird-Server verbindet, auf interne

Sicherheitsobjekte in einer Datenbank — `CURRENT_USER` und `CURRENT_ROLE`.

13.7.1. Die Zuordnungsregel

Die Abbildungsregel besteht aus vier Informationen:

1. Mapping-Bereich — ob die Zuordnung lokal zur aktuellen Datenbank ist oder ob sie global wirkt und alle Datenbanken im Cluster betrifft, einschließlich Sicherheitsdatenbanken
2. Mappingname — ein SQL-Bezeichner, da Mappings Objekte in einer Datenbank sind, wie alle anderen auch
3. das Objekt **FROM**, das das Mapping abbildet. Es besteht aus vier Elementen:
   - Die Authentifizierungsquelle
     - Pluginname **oder**
     - das Produkt einer Zuordnung in einer anderen Datenbank **oder**
     - Verwendung einer serverweiten Authentifizierung **oder**
     - beliebige Methode
   - Der Name der Datenbank, in der die Authentifizierung erfolgreich war
   - Der Name des Objekts, von dem das Mapping durchgeführt wird
   - Der **Typ** dieses Namens – Benutzername, Rolle oder Betriebssystemgruppe – abhängig von dem Plugin, das diesen Namen während der Authentifizierung hinzugefügt hat.

     Jeder Artikel wird akzeptiert, aber nur **Typ** ist erforderlich.
4. das Objekt **TO**, das das Mapping abbildet. Es besteht aus zwei Elementen:
   - Der Name des Objekts **TO** welches Mapping durchgeführt wird
   - Der **Typ**, für den nur `USER` oder `ROLE` gültig ist

13.7.2. `CREATE MAPPING`

Verwendet für

Erstellen einer Zuordnung eines Sicherheitsobjekts

Verfügbar in

DSQL

Syntax

```
CREATE [GLOBAL] MAPPING name
  USING
    { PLUGIN plugin_name [IN database]
    | ANY PLUGIN [IN database | SERVERWIDE]
    | MAPPING [IN database] | '*' [IN database] }
  FROM {ANY type | type from_name}
  TO {USER | ROLE} [to_name]
```

Tabelle 228. `CREATE MAPPING`-*Anweisungsparameter*

Parameter	Beschreibung
name	Mappingname Die maximale Länge beträgt 31 Zeichen. Muss unter allen Mapping-Namen im Kontext (lokal oder `GLOBAL`) eindeutig sein.
plugin_name	Name des Authentifizierungs-Plugins
database	Name der Datenbank, gegen die sich authentifiziert wird
type	Der Typ des zuzuordnenden Objekts. Mögliche Typen sind pluginspezifisch.
from_name	Der Name des zuzuordnenden Objekts
to_name	Der Name des Benutzers oder der Rolle, dem bzw. der zugeordnet werden soll

Die `CREATE MAPPING`-Anweisung erstellt eine Zuordnung von Sicherheitsobjekten (z. B. Benutzer, Gruppen, Rollen) eines oder mehrerer Authentifizierungs-Plugins zu internen Sicherheitsobjekten - `CURRENT_USER` und `CURRENT_ROLE`.

Wenn die `GLOBAL`-Klausel vorhanden ist, wird die Zuordnung nicht nur für die aktuelle Datenbank, sondern für alle Datenbanken im selben Cluster, einschließlich Sicherheitsdatenbanken, angewendet.

Es kann globale und lokale Zuordnungen mit demselben Namen geben. Sie sind unterschiedliche Objekte.

Die globale Zuordnung funktioniert am besten, wenn als Sicherheitsdatenbank eine Firebird 3.0- oder höhere Versionsdatenbank verwendet wird. Wenn Sie zu diesem Zweck eine andere Datenbank verwenden möchten – zum Beispiel mit Ihrem eigenen Provider – sollten Sie darin eine Tabelle namens `RDB$MAP` erstellen, mit der gleichen Struktur wie `RDB$MAP` in einem Firebird 3.0 Datenbank und nur mit SYSDBA-Schreibzugriff.

Die `USING`-Klausel beschreibt die Mapping-Quelle. Es hat eine sehr komplexe Reihe von Optionen:

- Ein expliziter Plugin-Name (`PLUGIN plugin_name`) bedeutet, dass dieser nur für dieses Plugin gilt
- es kann jedes verfügbare Plugin verwenden (`ANY PLUGIN`); allerdings nicht, wenn die Quelle das Produkt einer vorherigen Zuordnung ist
- es kann nur mit serverweiten Plugins (`SERVERWIDE`) zum Laufen gebracht werden
- es kann nur mit früheren Mapping-Ergebnissen (`MAPPING`) funktionieren
- Sie können die Verwendung einer bestimmten Methode unterlassen, indem Sie das Sternchen-Argument (*) verwenden
- es kann den Namen der Datenbank angeben, die das Mapping für das `FROM`-Objekt definiert hat (`IN database`)

Dieses Argument ist für die Zuordnung der serverweiten Authentifizierung

> nicht gültig.

Die FROM-Klausel beschreibt das abzubildende Objekt. Die FROM-Klausel hat ein obligatorisches Argument, den *type* des benannten Objekts. Es hat die folgenden Optionen:

- Beim Zuordnen von Namen über Plugins wird *type* vom Plugin definiert
- Beim Mapping des Produkts eines vorherigen Mappings kann *type* nur USER oder ROLE sein
- Wenn ein expliziter *from_name* angegeben wird, wird dieser von diesem Mapping berücksichtigt
- Verwenden Sie das Schlüsselwort ANY, um mit einem beliebigen Namen des angegebenen Typs zu arbeiten.

Die TO-Klausel gibt den Benutzer oder die Rolle an, die das Ergebnis der Zuordnung ist. Der *to_name* ist optional. Wenn er nicht angegeben wird, wird der ursprüngliche Name des zugeordneten Objekts verwendet.

Bei Rollen wird die durch eine Zuordnungsregel definierte Rolle nur angewendet, wenn der Benutzer beim Verbinden nicht explizit eine Rolle angibt. Die zugeordnete Rolle kann später in der Sitzung mit `SET TRUSTED ROLE` übernommen werden, auch wenn die zugeordnete Rolle dem Benutzer nicht explizit gewährt wird.

Wer kann Zuordnungen erstellen?

Die CREATE MAPPING-Anweisung kann ausgeführt werden durch:

- Administratoren
- Der Datenbankbesitzer – wenn die Zuordnung lokal ist

CREATE MAPPING-Beispiele

1. Aktivieren Sie die Verwendung der vertrauenswürdigen Windows-Authentifizierung in allen Datenbanken, die die aktuelle Sicherheitsdatenbank verwenden:

```
CREATE GLOBAL MAPPING TRUSTED_AUTH
  USING PLUGIN WIN_SSPI
  FROM ANY USER
  TO USER;
```

2. Aktivieren Sie den RDB$ADMIN-Zugriff für Windows-Administratoren in der aktuellen Datenbank:

```
CREATE MAPPING WIN_ADMINS
  USING PLUGIN WIN_SSPI
  FROM Predefined_Group
  DOMAIN_ANY_RID_ADMINS
  TO ROLE RDB$ADMIN;
```

Die Gruppe DOMAIN_ANY_RID_ADMINS existiert in Windows nicht, aber ein solcher Name würde vom Win_Sspi Plugin hinzugefügt, um eine genaue Abwärtskompatibilität zu gewährleisten.

3. Ermöglichen Sie einem bestimmten Benutzer aus einer anderen Datenbank, mit einem anderen Namen auf die aktuelle Datenbank zuzugreifen:

```
CREATE MAPPING FROM_RT
  USING PLUGIN SRP IN "rt"
  FROM USER U1 TO USER U2;
```

Datenbanknamen oder Aliase müssen auf Betriebssystemen mit Dateinamen, bei denen die Groß-/Kleinschreibung beachtet wird, in doppelte Anführungszeichen gesetzt werden.

4. Aktivieren Sie den SYSDBA des Servers (von der Hauptsicherheitsdatenbank), um auf die aktuelle Datenbank zuzugreifen. (Angenommen, die Datenbank verwendet eine nicht standardmäßige Sicherheitsdatenbank):

```
CREATE MAPPING DEF_SYSDBA
  USING PLUGIN SRP IN "security.db"
  FROM USER SYSDBA
  TO USER;
```

5. Stellen Sie sicher, dass Benutzer, die sich mit dem Legacy-Authentifizierungs-Plugin angemeldet haben, nicht zu viele Berechtigungen haben:

```
CREATE MAPPING LEGACY_2_GUEST
  USING PLUGIN legacy_auth
  FROM ANY USER
  TO USER GUEST;
```

Siehe auch

ALTER MAPPING, CREATE OR ALTER MAPPING, DROP MAPPING

13.7.3. ALTER MAPPING

Verwendet für

Ändern einer Zuordnung eines Sicherheitsobjekts

Verfügbar in

DSQL

Syntax

```
ALTER [GLOBAL] MAPPING name
  USING
    { PLUGIN plugin_name [IN database]
    | ANY PLUGIN [IN database | SERVERWIDE]
    | MAPPING [IN database] | '*' [IN database] }
  FROM {ANY type | type from_name}
  TO {USER | ROLE} [to_name]
```

Einzelheiten zu den Optionen finden Sie unter `CREATE MAPPING`.

Mit der `ALTER MAPPING`-Anweisung können Sie jede der vorhandenen Mapping-Optionen ändern, aber eine lokale Mapping kann nicht in `GLOBAL` geändert werden oder umgekehrt.

Globale und lokale Mappings gleichen Namens sind unterschiedliche Objekte.

Wer kann ein Mapping ändern?

The `ALTER MAPPING` statement can be executed by:

- Administratoren
- Der Datenbankbesitzer – wenn die Zuordnung lokal ist

`ALTER MAPPING`-Beispiele

Zuordnung ändern

```
ALTER MAPPING FROM_RT
  USING PLUGIN SRP IN "rt"
  FROM USER U1 TO USER U3;
```

Siehe auch

`CREATE MAPPING`, `CREATE OR ALTER MAPPING`, `DROP MAPPING`

13.7.4. `CREATE OR ALTER MAPPING`

Verwendet für

Erstellen eines neuen oder Ändern eines bestehenden Mappings eines Sicherheitsobjekts

Verfügbar in

DSQL

Syntax

```
CREATE OR ALTER [GLOBAL] MAPPING name
  USING
    { PLUGIN plugin_name [IN database]
```

```
    | ANY PLUGIN [IN database | SERVERWIDE]
    | MAPPING [IN database] | '*' [IN database] }
  FROM {ANY type | type from_name}
  TO {USER | ROLE} [to_name]
```

Einzelheiten zu den Optionen finden Sie unter `CREATE MAPPING`.

Die Anweisung `CREATE OR ALTER MAPPING` erstellt eine neue oder modifiziert eine vorhandene Zuordnung.

Globale und lokale Mappings gleichen Namens sind unterschiedliche Objekte.

`CREATE OR ALTER MAPPING`-Beispiele

Erstellen oder Ändern einer Zuordnung

```
CREATE OR ALTER MAPPING FROM_RT
  USING PLUGIN SRP IN "rt"
  FROM USER U1 TO USER U4;
```

Siehe auch

`CREATE MAPPING`, `ALTER MAPPING`, `DROP MAPPING`

13.7.5. DROP MAPPING

Verwendet für

Löschen (Entfernen) einer Zuordnung eines Sicherheitsobjekts

Verfügbar in

DSQL

Syntax

```
DROP [GLOBAL] MAPPING name
```

Tabelle 229. `DROP MAPPING`-Anweisungsparameter

Parameter	Beschreibung
name	Name der Zuordnung

Die `DROP MAPPING`-Anweisung entfernt ein vorhandenes Mapping. Wenn `GLOBAL` angegeben ist, wird eine globale Zuordnung entfernt.

Globale und lokale Mappings gleichen Namens sind unterschiedliche Objekte.

Wer kann ein Mapping löschen?

Die `DROP MAPPING`-Anweisung kann ausgeführt werden durch:

- Administratoren
- Der Datenbankbesitzer – wenn die Zuordnung lokal ist

`DROP MAPPING`-Beispiele

Zuordnung ändern

```
DROP MAPPING FROM_RT;
```

Siehe auch
`CREATE MAPPING`

13.8. Datenbankverschlüsselung

Firebird bietet einen Plugin-Mechanismus zum Verschlüsseln der in der Datenbank gespeicherten Daten. Dieser Mechanismus verschlüsselt nicht die gesamte Datenbank, sondern nur Datenseiten, Indexseiten und Blobseiten.

Um die Datenbankverschlüsselung zu ermöglichen, müssen Sie ein Datenbankverschlüsselungs-Plugin erwerben oder schreiben.

Firebird enthält standardmäßig kein Plugin zur Datenbankverschlüsselung.

Das Verschlüsselungs-Plugin-Beispiel in `examples/dbcrypt` führt keine echte Verschlüsselung durch, es ist nur als Beispiel gedacht, wie ein solches Plugin geschrieben werden kann.

Unter Linux befindet sich ein Beispiel-Plugin namens `libDbCrypt_example.so` in `plugins/`.

Das Hauptproblem bei der Datenbankverschlüsselung ist die Speicherung des geheimen Schlüssels. Firebird bietet Unterstützung für die Übertragung des Schlüssels vom Client, dies bedeutet jedoch nicht, dass die Speicherung des Schlüssels auf dem Client der beste Weg ist; es ist nur eine der möglichen Alternativen. Das Speichern von Verschlüsselungsschlüsseln auf derselben Festplatte wie die Datenbank ist jedoch eine unsichere Option.

Zur effizienten Trennung von Verschlüsselung und Schlüsselzugriff sind die Daten des Datenbankverschlüsselungs-Plugins in zwei Teile unterteilt, die Verschlüsselung selbst und den Inhaber des geheimen Schlüssels. Dies kann ein effizienter Ansatz sein, wenn Sie einen guten Verschlüsselungsalgorithmus verwenden möchten, aber Ihre eigene benutzerdefinierte Methode zum Speichern der Schlüssel haben.

Nachdem Sie sich für Plugin und Schlüsselhalter entschieden haben, können Sie die Verschlüsselung durchführen.

13.8.1. Eine Datenbank verschlüsseln

Syntax

```
ALTER {DATABASE | SCHEMA}
  ENCRYPT WITH plugin_name [KEY key_name]
```

Tabelle 230. `ALTER DATABASE ENCRYPT`-*Anweisungsparameter*

Parameter	Beschreibung
plugin_name	Der Name des Verschlüsselungs-Plugins
key_name	Der Name des Verschlüsselungsschlüssels

Verschlüsselt die Datenbank mit dem angegebenen Verschlüsselungs-Plugin. Die Verschlüsselung beginnt unmittelbar nach Abschluss dieser Anweisung und wird im Hintergrund ausgeführt. Der normale Betrieb der Datenbank wird während der Verschlüsselung nicht gestört.

Die optionale `KEY`-Klausel gibt den Namen des Schlüssels für das Verschlüsselungs-Plugin an. Das Plugin entscheidet, was mit diesem Schlüsselnamen geschehen soll.

Der Verschlüsselungsprozess kann mit dem Feld `MON$CRYPT_PAGE` in der virtuellen Tabelle `MON$DATABASE` überwacht oder mit `gstat -e` auf der Kopfseite der Datenbank angezeigt werden. `gstat -h` liefert auch begrenzte Informationen über den Verschlüsselungsstatus.

Die folgende Abfrage zeigt beispielsweise den Fortschritt des Verschlüsselungsprozesses in Prozent an.

```
select MON$CRYPT_PAGE * 100 / MON$PAGES
  from MON$DATABASE;
```

`SCHEMA` ist derzeit ein Synonym für `DATABASE`; dies kann sich in einer zukünftigen Version ändern, daher empfehlen wir immer `DATABASE` zu verwenden

Siehe auch

Eine Datenbank entschlüsseln, `ALTER DATABASE`

13.8.2. Eine Datenbank entschlüsseln

Syntax

```
ALTER {DATABASE | SCHEMA} DECRYPT
```

Entschlüsselt die Datenbank mit dem konfigurierten Plugin und Schlüssel. Die Entschlüsselung beginnt unmittelbar nach Abschluss dieser Anweisung und wird im Hintergrund ausgeführt. Der normale Betrieb der Datenbank wird während der Entschlüsselung nicht gestört.

`SCHEMA` ist derzeit ein Synonym für `DATABASE`; dies kann sich in einer zukünftigen Version ändern, daher empfehlen wir immer `DATABASE` zu verwenden

Siehe auch

Eine Datenbank verschlüsseln, `ALTER DATABASE`

Kapitel 14. Managementanweisungen

Seit Firebird 3.0 ist eine neue Klasse von DSQL-Anweisungen in Firebirds SQL-Lexikon entstanden, normalerweise für die Verwaltung von Aspekten der Client/Server-Umgebung. Typischerweise beginnen solche Anweisungen mit dem Verb SET.

Das *isql*-Tool hat auch eine Sammlung von SET-Befehlen. Diese Befehle sind nicht Teil des SQL-Lexikons von Firebird. Informationen zu *isql*s SET-Befehlen finden Sie unter *Isql* Set Commands in *Firebird Interactive SQL Utility*.

Die meisten Verwaltungsanweisungen betreffen nur die aktuelle Verbindung (Anhang oder "Sitzung") und erfordern keine Autorisierung über die Anmeldeberechtigungen des aktuellen Benutzers ohne erhöhte Berechtigungen hinaus.

14.1. Ändern der aktuellen Rolle

14.1.1. SET ROLE

Benutzt für

Ändern der Rolle der aktuellen Sitzung

Verfügbar in

DSQL

Syntax

```
SET ROLE {role_name | NONE}
```

Tabelle 231. SET ROLE*-Anweisungsparameter*

Parameter	Beschreibung
role_name	Der Name der anzuwendenden Rolle role

Die SET ROLE-Anweisung ermöglicht es einem Benutzer, eine andere Rolle anzunehmen; es setzt die Kontextvariable CURRENT_ROLE auf *role_name*, wenn diese Rolle dem CURRENT_USER gewährt wurde. Für diese Sitzung erhält der Benutzer die von dieser Rolle gewährten Berechtigungen. Alle Rechte, die der vorherigen Rolle gewährt wurden, werden aus der Sitzung entfernt. Verwenden Sie NONE anstelle von *role_name*, um die CURRENT_ROLE zu löschen.

Wenn die angegebene Rolle nicht existiert oder dem Benutzer nicht explizit zugewiesen wurde, wird der Fehler "*Role **role_name** is invalid or unavailable*" ausgegeben.

SET ROLE-Beispiele

1. Ändern Sie die aktuelle Rolle in MANAGER

```
SET ROLE manager;
```

```
select current_role from rdb$database;

ROLE
======================
MANAGER
```

2. Löschen Sie die aktuelle Rolle

```
SET ROLE NONE;
select current_role from rdb$database;

ROLE
======================
NONE
```

Siehe auch

`SET TRUSTED ROLE`, `GRANT`

14.1.2. SET TRUSTED ROLE

Verwendet für

Ändert die Rolle der aktuellen Sitzung in die vertrauenswürdige Rolle

Verfügbar in

DSQL

Syntax

```
SET TRUSTED ROLE
```

Die Anweisung `SET TRUSTED ROLE` ermöglicht es, die dem Benutzer durch eine Mapping-Regel zugewiesene Rolle einzunehmen (siehe Mapping von Benutzern auf Objekte). Die durch eine Zuordnungsregel zugewiesene Rolle wird beim Verbinden automatisch übernommen, wenn der Benutzer keine explizite Rolle angegeben hat. Die Anweisung `SET TRUSTED ROLE` ermöglicht es, die zugeordnete (oder `trusted`) Rolle zu einem späteren Zeitpunkt oder nach Änderung der aktuellen Rolle mit `SET ROLE` wieder einzunehmen.

Eine vertrauenswürdige Rolle ist kein bestimmter Rollentyp, sondern kann eine beliebige Rolle sein, die mit `CREATE ROLE` erstellt wurde, oder eine vordefinierte Systemrolle wie `RDB$ADMIN`. Ein Anhang (Sitzung) hat eine vertrauenswürdige Rolle, wenn das Sicherheitsobjekt-Mapping-Subsystem eine Übereinstimmung zwischen dem vom Plugin übergebenen Authentifizierungsergebnis und einer lokalen oder globalen Zuordnung zu einer Rolle für die aktuelle Datenbank findet. Die Rolle kann diesem Benutzer nicht explizit zugewiesen werden.

Wenn eine Sitzung keine vertrauenswürdige Rolle hat, wird die Ausführung von `SET TRUSTED ROLE` den Fehler „`Your attachment has no trusted role`“ auslösen.

Während die CURRENT_ROLE mit SET ROLE geändert werden kann, ist es nicht immer möglich, mit demselben Befehl zu einer vertrauenswürdigen Rolle zurückzukehren, da SET ROLE prüft, ob die Rolle dem Benutzer zugewiesen wurde. Mit SET TRUSTED ROLE kann die Trusted Rolle auch dann wieder übernommen werden, wenn SET ROLE fehlschlägt.

SET TRUSTED ROLE-Beispiele

1. Angenommen, eine Zuordnungsregel weist einem Benutzer "ALEX" die Rolle "ROLE1" zu:

```
CONNECT 'employee' USER ALEX PASSWORD 'password';
SELECT CURRENT_ROLE FROM RDB$DATABASE;

ROLE
==============================
ROLE1

SET ROLE ROLE2;
SELECT CURRENT_ROLE FROM RDB$DATABASE;

ROLE
==============================
ROLE2

SET TRUSTED ROLE;
SELECT CURRENT_ROLE FROM RDB$DATABASE;

ROLE
==============================
ROLE1
```

Siehe auch

SET ROLE, Zuordnung von Benutzern zu Objekten

Anhang A: Zusatzinformationen

In diesem Anhang finden Sie Themen, auf die sich Entwickler beziehen können, um das Verständnis für Funktionen oder Änderungen zu verbessern.

Das Feld RDB$VALID_BLR

Das Feld RDB$VALID_BLR wurde zu den Systemtabellen RDB$PROCEDURES und RDB$TRIGGERS in Firebird 2.1 hinzugefügt. Sein Zweck ist, eine mögliche Ungültigkeit eines PSQL-Moduls nach einer Änderung einer Domäne oder Tabellenspalte, von der das Modul abhängt, zu signalisieren. RDB$VALID_BLR wird auf 0 gesetzt, sobald der Code einer Prozedur oder eines Triggers ungültig durch eine solche Änderung wird.

Funktionsweise der Invalidierung

Bei Triggern und Prozeduren ergeben sich Abhängigkeiten durch die Definitionen der Tabellenspalten, auf die zugegriffen wird, und auch über alle Parameter oder Variablen, die im Modul mit der TYPE OF-Klausel definiert wurden.

Nachdem die Engine jede Domain, einschließlich der impliziten Domains, die intern hinter den Spaltendefinitionen und den Ausgabeparametern erstellt wurden, geändert hat, werden alle ihre Abhängigkeiten intern neu kompiliert.

In V.2.x beinhaltet dies Prozeduren und Trigger, jedoch keine Blöcke, die in DML-Anweisungen für die Laufzeitausführung mit EXECUTE BLOCK codiert sind. Firebird 3 umfasst weitere Modultypen (gespeicherte Funktionen, Pakete).

Jedes Modul, das aufgrund einer Inkompatibilität aufgrund einer Domänenänderung nicht neu kompiliert werden kann, wird als ungültig markiert (“invalidated” durch Setzen von RDB$VALID_BLR in seinem System Record (in RDB$PROCEDURES oder RDB$TRIGGERS , wie zutreffend) auf Null.

Eine erneute Validierung (Einstellung von RDB$VALID_BLR auf 1) erfolgt, wenn

1. die Domäne wird erneut geändert und die neue Definition ist mit der zuvor ungültig gemachten Moduldefinition kompatibel; ODER
2. das zuvor ungültig gemachte Modul wird geändert, um der neuen Domänendefinition zu entsprechen

Die folgende Abfrage findet die Module, die von einer bestimmten Domäne abhängen und meldet den Status ihrer RDB$VALID_BLR-Felder:

```
SELECT * FROM (
  SELECT
    'Procedure',
    rdb$procedure_name,
    rdb$valid_blr
  FROM rdb$procedures
```

```
  UNION ALL
  SELECT
    'Trigger',
    rdb$trigger_name,
    rdb$valid_blr
  FROM rdb$triggers
) (type, name, valid)
WHERE EXISTS
  (SELECT * from rdb$dependencies
   WHERE rdb$dependent_name = name
     AND rdb$depended_on_name = 'MYDOMAIN')

/* Replace MYDOMAIN with the actual domain name.
   Use all-caps if the domain was created
   case-insensitively. Otherwise, use the exact
   capitalisation. */
```

Die folgende Abfrage zeigt alle Module an, die von einer bestimmten Tabellenspalte abhängig sind, und gibt deren RDB$VALID_BLR-Status aus:

```
SELECT * FROM (
  SELECT
    'Procedure',
    rdb$procedure_name,
    rdb$valid_blr
  FROM rdb$procedures
  UNION ALL
  SELECT
    'Trigger',
    rdb$trigger_name,
    rdb$valid_blr
  FROM rdb$triggers) (type, name, valid)
WHERE EXISTS
  (SELECT *
   FROM rdb$dependencies
   WHERE rdb$dependent_name = name
     AND rdb$depended_on_name = 'MYTABLE'
     AND rdb$field_name = 'MYCOLUMN')
```

Alle durch Domänen-/Spaltenänderungen verursachten PSQL-Ungültigkeiten werden im Feld RDB$VALID_BLR wiedergegeben. Andere Arten von Änderungen, wie die Anzahl der Eingabe- oder Ausgabeparameter, aufgerufene Routinen usw., wirken sich jedoch nicht auf das Validierungsfeld aus, obwohl sie das Modul möglicherweise ungültig machen. Ein typisches solches Szenario könnte eines der folgenden sein:

1. Es wird eine Prozedur (B) definiert, die eine andere Prozedur (A) aufruft und daraus Ausgabeparameter liest. In diesem Fall wird eine Abhängigkeit in

RDB$DEPENDENCIES registriert. Anschließend wird die aufgerufene Prozedur (A) geändert, um einen oder mehrere dieser Ausgabeparameter zu ändern oder zu entfernen. Die Anweisung ALTER PROCEDURE A schlägt mit einem Fehler fehl, wenn ein Commit versucht wird.

2. Eine Prozedur (B) ruft Prozedur A auf und liefert Werte für ihre Eingabeparameter. In RDB$DEPENDENCIES ist keine Abhängigkeit registriert. Eine nachträgliche Änderung der Eingabeparameter in Prozedur A ist zulässig. Ein Fehler tritt zur Laufzeit auf, wenn B A mit dem nicht übereinstimmenden Eingabeparametersatz aufruft.

Weitere Hinweise

- Für PSQL-Module aus früheren Firebird-Versionen (gilt für einige Systemtrigger, sogar wenn die Datenbank unter Firebird 2.1. oder höher erstellt wurde), ist RDB$VALID_BLR NULL. Dies heißt nicht, dass ihre BLR ungültig ist.
- Die *isql*-Befehle SHOW PROCEDURES und SHOW TRIGGERS zeigen ein Sternchen in der RDB$VALID_BLR-Spalte für alle Module, deren Wert 0 ist (ungültig). Für SHOW PROCEDURE <procname> und SHOW TRIGGER <trigname>, welche einzelne PSQL-Module sind, werden gar keien ungültigen BLR signalisiert.

Ein Hinweis zur Gleichheit

Diese Anmerkung über Gleichheits- und Ungleichheits-Operatoren gilt überall in der Firebird SQL-Sprache.

Der “=”-Operator, welcher ausdrücklich in vielen Bedingungen verwendet wird, prüft nur Werte gegen Werte. Entsprechend dem SQL-Standard, ist NULL kein Wert und somit sind zwei NULLs weder gleich noch ungleich zueinander. Wenn Sie NULLs in Bedingungen vergleichen müssen, nutzen Sie den Operator IS NOT DISTINCT FROM. Dieser gibt wahr zurück, falls die Operanden den gleichen Wert besitzen *oder* beide NULL sind.

```
select *
  from A join B
  on A.id is not distinct from B.code
```

In Fällen in denen Sie gegen NULL innerhalb einer *ungleich*-Bedingung testen wollen, nutzen Sie IS DISTINCT FROM, nicht “<>”. Möchten Sie NULL unterschiedlich zu anderen Werten und zwei NULLs als gleich betrachten:

```
select *
  from A join B
  on A.id is distinct from B.code
```

Anhang B: Fehlercodes und Meldungen

Dieser Anhang enthält:

- SQLSTATE Fehlercodes und Beschreibungen
- GDSCODE Fehlercodes, SQLCODEs und Beschreibungen

Benutzerdefinierte Ausnahmen

Firebird DDL bietet eine einfache Syntax zum Erstellen benutzerdefinierter Ausnahmen für die Verwendung in PSQL-Modulen mit einem Nachrichtentext von bis zu 1.021 Zeichen. Weitere Informationen finden Sie unter `CREATE EXCEPTION` in *DDL Statements* und zur Verwendung in der Anweisung `EXCEPTION` in *PSQL-Anweisungen*.

Die Firebird `SQLCODE` Fehlercodes korrelieren nicht mit den Standardkonformen `SQLSTATE`-Codes. `SQLCODE` wurde viele Jahre verwendet und wird als veraltet angesehen. Der Support für `SQLCODE` wird vermutlich in zukünftigen Versionen entfernt.

SQLSTATE Fehlercodes und Beschreibungen

Die folgende Tabelle zeigt die Fehlercodes und Meldungen für die `SQLSTATE` Kontextvariablen.

Die Struktur eines `SQLSTATE` Fehlercodes besteht aus fünf Zeichen, die die SQL-Fehlerklasse (2 Zeichen) und die SQL-Subklasse (3 Zeichen) charakterisieren.

Tabelle 232. SQLSTATE Fehlercodes und Meldungen

SQLSTATE	Zugewiesene Meldungen
SQLCLASS 00 (Success)	
00000	Success
SQLCLASS 01 (Warning)	
01000	General warning
01001	Cursor operation conflict
01002	Disconnect error
01003	NULL value eliminated in set function
01004	String data, right-truncated
01005	Insufficient item descriptor areas
01006	Privilege not revoked
01007	Privilege not granted
01008	Implicit zero-bit padding
01100	Statement reset to unprepared
01101	Ongoing transaction has been committed

SQLSTATE	Zugewiesene Meldungen
01102	Ongoing transaction has been rolled back
SQLCLASS 02 (No Data)	
02000	No data found or no rows affected
SQLCLASS 07 (Dynamic SQL error)	
07000	Dynamic SQL error
07001	Wrong number of input parameters
07002	Wrong number of output parameters
07003	Cursor specification cannot be executed
07004	USING clause required for dynamic parameters
07005	Prepared statement not a cursor-specification
07006	Restricted data type attribute violation
07007	USING clause required for result fields
07008	Invalid descriptor count
07009	Invalid descriptor index
SQLCLASS 08 (Connection Exception)	
08001	Client unable to establish connection
08002	Connection name in use
08003	Connection does not exist
08004	Server rejected the connection
08006	Connection failure
08007	Transaction resolution unknown
SQLCLASS 0A (Feature Not Supported)	
0A000	Feature Not Supported
SQLCLASS 0B (Invalid Transaction Initiation)	
0B000	Invalid transaction initiation
SQLCLASS 0L (Invalid Grantor)	
0L000	Invalid grantor
SQLCLASS 0P (Invalid Role Specification)	
0P000	Invalid role specification
SQLCLASS 0U (Attempt to Assign to Non-Updatable Column)	
0U000	Attempt to assign to non-updatable column
SQLCLASS 0V (Attempt to Assign to Ordering Column)	
0V000	Attempt to assign to Ordering column

SQLSTATE	Zugewiesene Meldungen
SQLCLASS 20 (Case Not Found For Case Statement)	
20000	Case not found for case statement
SQLCLASS 21 (Cardinality Violation)	
21000	Cardinality violation
21S01	Insert value list does not match column list
21S02	Degree of derived table does not match column list
SQLCLASS 22 (Data Exception)	
22000	Data exception
22001	String data, right truncation
22002	Null value, no indicator parameter
22003	Numeric value out of range
22004	Null value not allowed
22005	Error in assignment
22006	Null value in field reference
22007	Invalid datetime format
22008	Datetime field overflow
22009	Invalid time zone displacement value
2200A	Null value in reference target
2200B	Escape character conflict
2200C	Invalid use of escape character
2200D	Invalid escape octet
2200E	Null value in array target
2200F	Zero-length character string
2200G	Most specific type mismatch
22010	Invalid indicator parameter value
22011	Substring error
22012	Division by zero
22014	Invalid update value
22015	Interval field overflow
22018	Invalid character value for cast
22019	Invalid escape character
2201B	Invalid regular expression
2201C	Null row not permitted in table

SQLSTATE	Zugewiesene Meldungen
22012	Division by zero
22020	Invalid limit value
22021	Character not in repertoire
22022	Indicator overflow
22023	Invalid parameter value
22024	Character string not properly terminated
22025	Invalid escape sequence
22026	String data, length mismatch
22027	Trim error
22028	Row already exists
2202D	Null instance used in mutator function
2202E	Array element error
2202F	Array data, right truncation
SQLCLASS 23 (Integrity Constraint Violation)	
23000	Integrity constraint violation
SQLCLASS 24 (Invalid Cursor State)	
24000	Invalid cursor state
24504	The cursor identified in the UPDATE, DELETE, SET, or GET statement is not positioned on a row
SQLCLASS 25 (Invalid Transaction State)	
25000	Invalid transaction state
25S01	Transaction state
25S02	Transaction is still active
25S03	Transaction is rolled back
SQLCLASS 26 (Invalid SQL Statement Name)	
26000	Invalid SQL statement name
SQLCLASS 27 (Triggered Data Change Violation)	
27000	Triggered data change violation
SQLCLASS 28 (Invalid Authorization Specification)	
28000	Invalid authorization specification
SQLCLASS 2B (Dependent Privilege Descriptors Still Exist)	
2B000	Dependent privilege descriptors still exist
SQLCLASS 2C (Invalid Character Set Name)	
2C000	Invalid character set name

SQLSTATE	Zugewiesene Meldungen
SQLCLASS 2D (Invalid Transaction Termination)	
2D000	Invalid transaction termination
SQLCLASS 2E (Invalid Connection Name)	
2E000	Invalid connection name
SQLCLASS 2F (SQL Routine Exception)	
2F000	SQL routine exception
2F002	Modifying SQL-data not permitted
2F003	Prohibited SQL-statement attempted
2F004	Reading SQL-data not permitted
2F005	Function executed no return statement
SQLCLASS 33 (Invalid SQL Descriptor Name)	
33000	Invalid SQL descriptor name
SQLCLASS 34 (Invalid Cursor Name)	
34000	Invalid cursor name
SQLCLASS 35 (Invalid Condition Number)	
35000	Invalid condition number
SQLCLASS 36 (Cursor Sensitivity Exception)	
36001	Request rejected
36002	Request failed
SQLCLASS 37 (Invalid Identifier)	
37000	Invalid identifier
37001	Identifier too long
SQLCLASS 38 (External Routine Exception)	
38000	External routine exception
SQLCLASS 39 (External Routine Invocation Exception)	
39000	External routine invocation exception
SQLCLASS 3B (Invalid Save Point)	
3B000	Invalid save point
SQLCLASS 3C (Ambiguous Cursor Name)	
3C000	Ambiguous cursor name
SQLCLASS 3D (Invalid Catalog Name)	
3D000	Invalid catalog name
3D001	Catalog name not found

SQLSTATE	Zugewiesene Meldungen
SQLCLASS 3F (Invalid Schema Name)	
3F000	Invalid schema name
SQLCLASS 40 (Transaction Rollback)	
40000	Ongoing transaction has been rolled back
40001	Serialization failure
40002	Transaction integrity constraint violation
40003	Statement completion unknown
SQLCLASS 42 (Syntax Error or Access Violation)	
42000	Syntax error or access violation
42702	Ambiguous column reference
42725	Ambiguous function reference
42818	The operands of an operator or function are not compatible
42S01	Base table or view already exists
42S02	Base table or view not found
42S11	Index already exists
42S12	Index not found
42S21	Column already exists
42S22	Column not found
SQLCLASS 44 (With Check Option Violation)	
44000	WITH CHECK OPTION Violation
SQLCLASS 45 (Unhandled User-defined Exception)	
45000	Unhandled user-defined exception
SQLCLASS 54 (Program Limit Exceeded)	
54000	Program limit exceeded
54001	Statement too complex
54011	Too many columns
54023	Too many arguments
SQLCLASS HY (CLI-specific Condition)	
HY000	CLI-specific condition
HY001	Memory allocation error
HY003	Invalid data type in application descriptor
HY004	Invalid data type
HY007	Associated statement is not prepared

SQLSTATE	Zugewiesene Meldungen
HY008	Operation canceled
HY009	Invalid use of null pointer
HY010	Function sequence error
HY011	Attribute cannot be set now
HY012	Invalid transaction operation code
HY013	Memory management error
HY014	Limit on the number of handles exceeded
HY015	No cursor name available
HY016	Cannot modify an implementation row descriptor
HY017	Invalid use of an automatically allocated descriptor handle
HY018	Server declined the cancellation request
HY019	Non-string data cannot be sent in pieces
HY020	Attempt to concatenate a null value
HY021	Inconsistent descriptor information
HY024	Invalid attribute value
HY055	Non-string data cannot be used with string routine
HY090	Invalid string length or buffer length
HY091	Invalid descriptor field identifier
HY092	Invalid attribute identifier
HY095	Invalid Function ID specified
HY096	Invalid information type
HY097	Column type out of range
HY098	Scope out of range
HY099	Nullable type out of range
HY100	Uniqueness option type out of range
HY101	Accuracy option type out of range
HY103	Invalid retrieval code
HY104	Invalid Length/Precision value
HY105	Invalid parameter type
HY106	Invalid fetch orientation
HY107	Row value out of range
HY109	Invalid cursor position
HY110	Invalid driver completion

SQLSTATE	Zugewiesene Meldungen
HY111	Invalid bookmark value
HYC00	Optional feature not implemented
HYT00	Timeout expired
HYT01	Connection timeout expired
SQLCLASS XX (Internal Error)	
XX000	Internal error
XX001	Data corrupted
XX002	Index corrupted

SQLCODE- und GDSCODE-Fehlercodes sowie ihre Beschreibungen

Die Tabelle enthält die SQLCODE-Gruppierungen, die numerischen und symbolischen Werte für die GDSCODE-Fehler und die Nachrichtentexte.

`SQLCODE` wird seit vielen Jahren verwendet und sollte jetzt als veraltet angesehen werden. Die Unterstützung für `SQLCODE` wird wahrscheinlich in einer zukünftigen Version eingestellt.

Tabelle 233. SQLCODE- und GDSCODE-Fehlercodes sowie Meldungen

SQL-CODE	GDSCODE	Symbol	Message Text
501	335544802	dialect_reset_warning	Database dialect being changed from 3 to 1
301	335544808	dtype_renamed	DATE data type is now called TIMESTAMP
301	336003076	dsql_dialect_warning_expr	Use of @1 expression that returns different results in dialect 1 and dialect 3
301	336003080	dsql_warning_number_ambiguous	WARNING: Numeric literal @1 is interpreted as a floating-point
301	336003081	dsql_warning_number_ambiguous1	value in SQL dialect 1, but as an exact numeric value in SQL dialect 3.
301	336003082	dsql_warn_precision_ambiguous	WARNING: NUMERIC and DECIMAL fields with precision 10 or greater are stored
301	336003083	dsql_warn_precision_ambiguous1	as approximate floating-point values in SQL dialect 1, but as 64-bit
301	336003084	dsql_warn_precision_ambiguous2	integers in SQL dialect 3.

SQL-CODE	GDSCODE	Symbol	Message Text
300	335544807	sqlwarn	SQL warning code = @1
106	336068855	dyn_miss_priv_warning	Warning: @1 on @2 is not granted to @3.
101	335544366	segment	segment buffer length shorter than expected
100	335544338	from_no_match	no match for first value expression
100	335544354	no_record	invalid database key
100	335544367	segstr_eof	attempted retrieval of more segments than exist
0	335544875	bad_debug_format	Bad debug info format
0	335544931	montabexh	Monitoring table space exhausted
-84	335544554	nonsql_security_rel	object has non-SQL security class defined
-84	335544555	nonsql_security_fld	column has non-SQL security class defined
-84	335544668	dsql_procedure_use_err	procedure @1 does not return any values
-85	335544747	usrname_too_long	The username entered is too long. Maximum length is 31 bytes.
-85	335544748	password_too_long	The password specified is too long. Maximum length is 8 bytes.
-85	335544749	usrname_required	A username is required for this operation.
-85	335544750	password_required	A password is required for this operation
-85	335544751	bad_protocol	The network protocol specified is invalid
-85	335544752	dup_usrname_found	A duplicate user name was found in the security database
-85	335544753	usrname_not_found	The user name specified was not found in the security database
-85	335544754	error_adding_sec_record	An error occurred while attempting to add the user.
-85	335544755	error_modifying_sec_record	An error occurred while attempting to modify the user record.
-85	335544756	error_deleting_sec_record	An error occurred while attempting to delete the user record.

SQL-CODE	GDSCODE	Symbol	Message Text
-85	335544757	error_updating_sec_db	An error occurred while updating the security database.
-103	335544571	dsql_constant_err	Data type for constant unknown
-104	335544343	invalid_blr	invalid request BLR at offset @1
-104	335544390	syntaxerr	BLR syntax error: expected @1 at offset @2, encountered @3
-104	335544425	ctxinuse	context already in use (BLR error)
-104	335544426	ctxnotdef	context not defined (BLR error)
-104	335544429	badparnum	undefined parameter number
-104	335544440	bad_msg_vec	
-104	335544456	invalid_sdl	invalid slice description language at offset @1
-104	335544570	dsql_command_err	Invalid command
-104	335544579	dsql_internal_err	Internal error
-104	335544590	dsql_dup_option	Option specified more than once
-104	335544591	dsql_tran_err	Unknown transaction option
-104	335544592	dsql_invalid_array	Invalid array reference
-104	335544608	command_end_err	Unexpected end of command
-104	335544612	token_err	Token unknown
-104	335544634	dsql_token_unk_err	Token unknown - line @1, column @2
-104	335544709	dsql_agg_ref_err	Invalid aggregate reference
-104	335544714	invalid_array_id	invalid blob id
-104	335544730	cse_not_supported	Client/Server Express not supported in this release
-104	335544743	token_too_long	token size exceeds limit
-104	335544763	invalid_string_constant	a string constant is delimited by double quotes
-104	335544764	transitional_date	DATE must be changed to TIMESTAMP
-104	335544796	sql_dialect_datatype_unsupport	Client SQL dialect @1 does not support reference to @2 datatype
-104	335544798	depend_on_uncommitted_rel	You created an indirect dependency on uncommitted metadata. You must roll back the current transaction.
-104	335544821	dsql_column_pos_err	Invalid column position used in the @1 clause

SQL-CODE	GDSCODE	Symbol	Message Text
-104	335544822	dsql_agg_where_err	Cannot use an aggregate or window function in a WHERE clause, use HAVING (for aggregate only) instead
-104	335544823	dsql_agg_group_err	Cannot use an aggregate or window function in a GROUP BY clause
-104	335544824	dsql_agg_column_err	Invalid expression in the @1 (not contained in either an aggregate function or the GROUP BY clause)
-104	335544825	dsql_agg_having_err	Invalid expression in the @1 (neither an aggregate function nor a part of the GROUP BY clause)
-104	335544826	dsql_agg_nested_err	Nested aggregate and window functions are not allowed
-104	335544849	malformed_string	Malformed string
-104	335544851	command_end_err2	Unexpected end of command - line @1, column @2
-104	335544930	too_big_blr	BLR stream length @1 exceeds implementation limit @2
-104	335544980	internal_rejected_params	Incorrect parameters provided to internal function @1
-104	335545022	cannot_copy_stmt	Cannot copy statement @1
-104	335545023	invalid_boolean_usage	Invalid usage of boolean expression
-104	335545035	svc_no_stdin	No isc_info_svc_stdin in user request, but service thread requested stdin data
-104	335545037	svc_no_switches	All services except for getting server log require switches
-104	335545038	svc_bad_size	Size of stdin data is more than was requested from client
-104	335545039	no_crypt_plugin	Crypt plugin @1 failed to load
-104	335545040	cp_name_too_long	Length of crypt plugin name should not exceed @1 bytes
-104	335545045	null_spb	NULL data with non-zero SPB length
-104	336003075	dsql_transitional_numeric	Precision 10 to 18 changed from DOUBLE PRECISION in SQL dialect 1 to 64-bit scaled integer in SQL dialect 3
-104	336003077	sql_db_dialect_dtype_unsupport	Database SQL dialect @1 does not support reference to @2 datatype
-104	336003087	dsql_invalid_label	Label @1 @2 in the current scope

SQL-CODE	GDSCODE	Symbol	Message Text
-104	336003088	dsql_datatypes_not_comparable	Datatypes @1are not comparable in expression @2
-104	336397215	dsql_max_sort_items	cannot sort on more than 255 items
-104	336397216	dsql_max_group_items	cannot group on more than 255 items
-104	336397217	dsql_conflicting_sort_field	Cannot include the same field (@1.@2) twice in the ORDER BY clause with conflicting sorting options
-104	336397218	dsql_derived_table_more_columns	column list from derived table @1 has more columns than the number of items in its SELECT statement
-104	336397219	dsql_derived_table_less_columns	column list from derived table @1 has less columns than the number of items in its SELECT statement
-104	336397220	dsql_derived_field_unnamed	no column name specified for column number @1 in derived table @2
-104	336397221	dsql_derived_field_dup_name	column @1 was specified multiple times for derived table @2
-104	336397222	dsql_derived_alias_select	Internal dsql error: alias type expected by pass1_expand_select_node
-104	336397223	dsql_derived_alias_field	Internal dsql error: alias type expected by pass1_field
-104	336397224	dsql_auto_field_bad_pos	Internal dsql error: column position out of range in pass1_union_auto_cast
-104	336397225	dsql_cte_wrong_reference	Recursive CTE member (@1) can refer itself only in FROM clause
-104	336397226	dsql_cte_cycle	CTE '@1' has cyclic dependencies
-104	336397227	dsql_cte_outer_join	Recursive member of CTE can’t be member of an outer join
-104	336397228	dsql_cte_mult_references	Recursive member of CTE can’t reference itself more than once
-104	336397229	dsql_cte_not_a_union	Recursive CTE (@1) must be an UNION
-104	336397230	dsql_cte_nonrecurs_after_recurs	CTE '@1' defined non-recursive member after recursive
-104	336397231	dsql_cte_wrong_clause	Recursive member of CTE '@1' has @2 clause
-104	336397232	dsql_cte_union_all	Recursive members of CTE (@1) must be linked with another members via UNION ALL

SQL-CODE	GDSCODE	Symbol	Message Text
-104	336397233	dsql_cte_miss_nonrecursive	Non-recursive member is missing in CTE '@1'
-104	336397234	dsql_cte_nested_with	WITH clause can’t be nested
-104	336397235	dsql_col_more_than_once_using	column @1 appears more than once in USING clause
-104	336397237	dsql_cte_not_used	CTE "@1" is not used in query
-104	336397238	dsql_col_more_than_once_view	column @1 appears more than once in ALTER VIEW
-104	336397257	dsql_max_distinct_items	Cannot have more than 255 items in DISTINCT list
-104	336397321	dsql_cte_recursive_aggregate	Recursive member of CTE cannot use aggregate or window function
-104	336397326	dsql_wlock_simple	WITH LOCK can be used only with a single physical table
-104	336397327	dsql_firstskip_rows	FIRST/SKIP cannot be used with OFFSET/FETCH or ROWS
-104	336397328	dsql_wlock_aggregates	WITH LOCK cannot be used with aggregates
-104	336397329	dsql_wlock_conflict	WITH LOCK cannot be used with @1
-105	335544702	escape_invalid	Invalid ESCAPE sequence
-105	335544789	extract_input_mismatch	Specified EXTRACT part does not exist in input datatype
-105	335544884	invalid_similar_pattern	Invalid SIMILAR TO pattern
-150	335544360	read_only_rel	attempted update of read-only table
-150	335544362	read_only_view	cannot update read-only view @1
-150	335544446	non_updatable	not updatable
-150	335544546	constaint_on_view	Cannot define constraints on views
-151	335544359	read_only_field	attempted update of read-only column
-155	335544658	dsql_base_table	@1 is not a valid base table of the specified view
-157	335544598	specify_field_err	must specify column name for view select expression
-158	335544599	num_field_err	number of columns does not match select list
-162	335544685	no_dbkey	dbkey not available for multi-table views

SQL-CODE	GDSCODE	Symbol	Message Text
-170	335544512	prcmismat	Input parameter mismatch for procedure @1
-170	335544619	extern_func_err	External functions cannot have more than 10 parameters
-170	335544850	prc_out_param_mismatch	Output parameter mismatch for procedure @1
-170	335545101	fun_param_mismatch	Input parameter mismatch for function @1
-171	335544439	funmismat	function @1 could not be matched
-171	335544458	invalid_dimension	column not array or invalid dimensions (expected @1, encountered @2)
-171	335544618	return_mode_err	Return mode by value not allowed for this data type
-171	335544873	array_max_dimensions	Array data type can use up to @1 dimensions
-172	335544438	funnotdef	function @1 is not defined
-172	335544932	modnotfound	module name or entrypoint could not be found
-203	335544708	dyn_fld_ambiguous	Ambiguous column reference.
-204	335544463	gennotdef	generator @1 is not defined
-204	335544502	stream_not_defined	reference to invalid stream number
-204	335544509	charset_not_found	CHARACTER SET @1 is not defined
-204	335544511	prcnotdef	procedure @1 is not defined
-204	335544515	codnotdef	status code @1 unknown
-204	335544516	xcpnotdef	exception @1 not defined
-204	335544532	ref_cnstrnt_notfound	Name of Referential Constraint not defined in constraints table.
-204	335544551	grant_obj_notfound	could not find object for GRANT
-204	335544568	text_subtype	Implementation of text subtype @1 not located.
-204	335544573	dsql_datatype_err	Data type unknown
-204	335544580	dsql_relation_err	Table unknown
-204	335544581	dsql_procedure_err	Procedure unknown
-204	335544588	collation_not_found	COLLATION @1 for CHARACTER SET @2 is not defined

SQL-CODE	GDSCODE	Symbol	Message Text
-204	335544589	collation_not_for_charset	COLLATION @1 is not valid for specified CHARACTER SET
-204	335544595	dsql_trigger_err	Trigger unknown
-204	335544620	alias_conflict_err	alias @1 conflicts with an alias in the same statement
-204	335544621	procedure_conflict_error	alias @1 conflicts with a procedure in the same statement
-204	335544622	relation_conflict_err	alias @1 conflicts with a table in the same statement
-204	335544635	dsql_no_relation_alias	there is no alias or table named @1 at this scope level
-204	335544636	indexname	there is no index @1 for table @2
-204	335544640	collation_requires_text	Invalid use of CHARACTER SET or COLLATE
-204	335544662	dsql_blob_type_unknown	BLOB SUB_TYPE @1 is not defined
-204	335544759	bad_default_value	can not define a not null column with NULL as default value
-204	335544760	invalid_clause	invalid clause --- '@1'
-204	335544800	too_many_contexts	Too many Contexts of Relation/Procedure/Views. Maximum allowed is 256
-204	335544817	bad_limit_param	Invalid parameter to FETCH or FIRST. Only integers >= 0 are allowed.
-204	335544818	bad_skip_param	Invalid parameter to OFFSET or SKIP. Only integers >= 0 are allowed.
-204	335544837	bad_substring_offset	Invalid offset parameter @1 to SUBSTRING. Only positive integers are allowed.
-204	335544853	bad_substring_length	Invalid length parameter @1 to SUBSTRING. Negative integers are not allowed.
-204	335544854	charset_not_installed	CHARACTER SET @1 is not installed
-204	335544855	collation_not_installed	COLLATION @1 for CHARACTER SET @2 is not installed
-204	335544867	subtype_for_internal_use	Blob sub_types bigger than 1 (text) are for internal use only
-204	335545104	invalid_attachment_charset	CHARACTER SET @1 cannot be used as a attachment character set

SQL-CODE	GDSCODE	Symbol	Message Text
-204	336003085	dsql_ambiguous_field_name	Ambiguous field name between @1 and @2
-205	335544396	fldnotdef	column @1 is not defined in table @2
-205	335544552	grant_fld_notfound	could not find column for GRANT
-205	335544883	fldnotdef2	column @1 is not defined in procedure @2
-206	335544578	dsql_field_err	Column unknown
-206	335544587	dsql_blob_err	Column is not a BLOB
-206	335544596	dsql_subselect_err	Subselect illegal in this context
-206	336397208	dsql_line_col_error	At line @1, column @2
-206	336397209	dsql_unknown_pos	At unknown line and column
-206	336397210	dsql_no_dup_name	Column @1 cannot be repeated in @2 statement
-208	335544617	order_by_err	invalid ORDER BY clause
-219	335544395	relnotdef	table @1 is not defined
-219	335544872	domnotdef	domain @1 is not defined
-230	335544487	walw_err	WAL Writer error
-231	335544488	logh_small	Log file header of @1 too small
-232	335544489	logh_inv_version	Invalid version of log file @1
-233	335544490	logh_open_flag	Log file @1 not latest in the chain but open flag still set
-234	335544491	logh_open_flag2	Log file @1 not closed properly; database recovery may be required
-235	335544492	logh_diff_dbname	Database name in the log file @1 is different
-236	335544493	logf_unexpected_eof	Unexpected end of log file @1 at offset @2
-237	335544494	logr_incomplete	Incomplete log record at offset @1 in log file @2
-238	335544495	logr_header_small	Log record header too small at offset @1 in log file @2
-239	335544496	logb_small	Log block too small at offset @1 in log file @2
-239	335544691	cache_too_small	Insufficient memory to allocate page buffer cache
-239	335544693	log_too_small	Log size too small

SQL-CODE	GDSCODE	Symbol	Message Text
-239	335544694	partition_too_small	Log partition size too small
-240	335544497	wal_illegal_attach	Illegal attempt to attach to an uninitialized WAL segment for @1
-241	335544498	wal_invalid_wpb	Invalid WAL parameter block option @1
-242	335544499	wal_err_rollover	Cannot roll over to the next log file @1
-243	335544500	no_wal	database does not use Write-ahead Log
-244	335544503	wal_subsys_error	WAL subsystem encountered error
-245	335544504	wal_subsys_corrupt	WAL subsystem corrupted
-246	335544513	wal_bugcheck	Database @1: WAL subsystem bug for pid @2 @3
-247	335544514	wal_cant_expand	Could not expand the WAL segment for database @1
-248	335544521	wal_err_rollover2	Unable to roll over please see Firebird log.
-249	335544522	wal_err_logwrite	WAL I/O error. Please see Firebird log.
-250	335544523	wal_err_jrn_comm	WAL writer - Journal server communication error. Please see Firebird log.
-251	335544524	wal_err_expansion	WAL buffers cannot be increased. Please see Firebird log.
-252	335544525	wal_err_setup	WAL setup error. Please see Firebird log.
-253	335544526	wal_err_ww_sync	obsolete
-254	335544527	wal_err_ww_start	Cannot start WAL writer for the database @1
-255	335544556	wal_cache_err	Write-ahead Log without shared cache configuration not allowed
-257	335544566	start_cm_for_wal	WAL defined; Cache Manager must be started first
-258	335544567	wal_ovflow_log_required	Overflow log specification required for round-robin log
-259	335544629	wal_shadow_err	Write-ahead Log with shadowing configuration not allowed
-260	335544690	cache_redef	Cache redefined
-260	335544692	log_redef	Log redefined

SQL-CODE	GDSCODE	Symbol	Message Text
-261	335544695	partition_not_supp	Partitions not supported in series of log file specification
-261	335544696	log_length_spec	Total length of a partitioned log must be specified
-281	335544637	no_stream_plan	table @1 is not referenced in plan
-282	335544638	stream_twice	table @1 is referenced more than once in plan; use aliases to distinguish
-282	335544643	dsql_self_join	the table @1 is referenced twice; use aliases to differentiate
-282	335544659	duplicate_base_table	table @1 is referenced twice in view; use an alias to distinguish
-282	335544660	view_alias	view @1 has more than one base table; use aliases to distinguish
-282	335544710	complex_view	navigational stream @1 references a view with more than one base table
-283	335544639	stream_not_found	table @1 is referenced in the plan but not the from list
-284	335544642	index_unused	index @1 cannot be used in the specified plan
-291	335544531	primary_key_notnull	Column used in a PRIMARY constraint must be NOT NULL.
-291	335545103	domain_primary_key_notnull	Domain used in the PRIMARY KEY constraint of table @1 must be NOT NULL
-292	335544534	ref_cnstrnt_update	Cannot update constraints (RDB$REF_CONSTRAINTS).
-293	335544535	check_cnstrnt_update	Cannot update constraints (RDB$CHECK_CONSTRAINTS).
-294	335544536	check_cnstrnt_del	Cannot delete CHECK constraint entry (RDB$CHECK_CONSTRAINTS)
-295	335544545	rel_cnstrnt_update	Cannot update constraints (RDB$RELATION_CONSTRAINTS).
-296	335544547	invld_cnstrnt_type	internal Firebird consistency check (invalid RDB$CONSTRAINT_TYPE)
-297	335544558	check_constraint	Operation violates CHECK constraint @1 on view or table @2
-313	335544669	dsql_count_mismatch	count of column list and variable list do not match

SQL-CODE	GDSCODE	Symbol	Message Text
-313	336003099	upd_ins_doesnt_match_pk	UPDATE OR INSERT field list does not match primary key of table @1
-313	336003100	upd_ins_doesnt_match_matching	UPDATE OR INSERT field list does not match MATCHING clause
-313	336003111	dsql_wrong_param_num	Wrong number of parameters (expected @1, got @2)
-314	335544565	transliteration_failed	Cannot transliterate character between character sets
-315	336068815	dyn_dtype_invalid	Cannot change datatype for column @1. Changing datatype is not supported for BLOB or ARRAY columns.
-383	336068814	dyn_dependency_exists	Column @1 from table @2 is referenced in @3
-401	335544647	invalid_operator	invalid comparison operator for find operation
-402	335544368	segstr_no_op	attempted invalid operation on a BLOB
-402	335544414	blobnotsup	BLOB and array data types are not supported for @1 operation
-402	335544427	datnotsup	data operation not supported
-406	335544457	out_of_bounds	subscript out of bounds
-406	335545028	ss_out_of_bounds	Subscript @1 out of bounds [@2, @3]
-407	335544435	nullsegkey	null segment of UNIQUE KEY
-413	335544334	convert_error	conversion error from string "@1"
-413	335544454	nofilter	filter not found to convert type @1 to type @2
-413	335544860	blob_convert_error	Unsupported conversion to target type BLOB (subtype @1)
-413	335544861	array_convert_error	Unsupported conversion to target type ARRAY
-501	335544577	dsql_cursor_close_err	Attempt to reclose a closed cursor
-502	335544574	dsql_decl_err	Invalid cursor declaration
-502	335544576	dsql_cursor_open_err	Attempt to reopen an open cursor
-502	336003090	dsql_cursor_redefined	Statement already has a cursor @1 assigned
-502	336003091	dsql_cursor_not_found	Cursor @1 is not found in the current context

SQL-CODE	GDSCODE	Symbol	Message Text
-502	336003092	dsql_cursor_exists	Cursor @1 already exists in the current context
-502	336003093	dsql_cursor_rel_ambiguous	Relation @1 is ambiguous in cursor @2
-502	336003094	dsql_cursor_rel_not_found	Relation @1 is not found in cursor @2
-502	336003095	dsql_cursor_not_open	Cursor is not open
-504	335544572	dsql_cursor_err	Invalid cursor reference
-504	336003089	dsql_cursor_invalid	Empty cursor name is not allowed
-508	335544348	no_cur_rec	no current record for fetch operation
-510	335544575	dsql_cursor_update_err	Cursor @1 is not updatable
-518	335544582	dsql_request_err	Request unknown
-519	335544688	dsql_open_cursor_request	The prepare statement identifies a prepare statement with an open cursor
-530	335544466	foreign_key	violation of FOREIGN KEY constraint "@1" on table "@2"
-530	335544838	foreign_key_target_doesnt_exist	Foreign key reference target does not exist
-530	335544839	foreign_key_references_present	Foreign key references are present for the record
-531	335544597	dsql_crdb_prepare_err	Cannot prepare a CREATE DATABASE/SCHEMA statement
-532	335544469	trans_invalid	transaction marked invalid and cannot be committed
-532	335545002	attachment_in_use	Attachment is in use
-532	335545003	transaction_in_use	Transaction is in use
-532	335545017	async_active	Asynchronous call is already running for this attachment
-551	335544352	no_priv	no permission for @1 access to @2 @3
-551	335544790	insufficient_svc_privileges	Service @1 requires SYSDBA permissions. Reattach to the Service Manager using the SYSDBA account.
-551	335545033	trunc_limits	expected length @1, actual @2
-551	335545034	info_access	Wrong info requested in isc_svc_query() for anonymous service
-551	335545036	svc_start_failed	Start request for anonymous service is impossible

SQL-CODE	GDSCODE	Symbol	Message Text
-552	335544550	not_rel_owner	only the owner of a table may reassign ownership
-552	335544553	grant_nopriv	user does not have GRANT privileges for operation
-552	335544707	grant_nopriv_on_base	user does not have GRANT privileges on base table/view for operation
-552	335545058	protect_ownership	Only the owner can change the ownership
-553	335544529	existing_priv_mod	cannot modify an existing user privilege
-595	335544645	stream_crack	the current position is on a crack
-596	335544374	stream_eof	attempt to fetch past the last record in a record stream
-596	335544644	stream_bof	attempt to fetch before the first record in a record stream
-596	335545092	cursor_not_positioned	Cursor @1 is not positioned in a valid record
-597	335544632	dsql_file_length_err	Preceding file did not specify length, so @1 must include starting page number
-598	335544633	dsql_shadow_number_err	Shadow number must be a positive integer
-599	335544607	node_err	gen.c: node not supported
-599	335544625	node_name_err	A node name is not permitted in a secondary, shadow, cache or log file name
-600	335544680	crrp_data_err	sort error: corruption in data structure
-601	335544646	db_or_file_exists	database or file exists
-604	335544593	dsql_max_arr_dim_exceeded	Array declared with too many dimensions
-604	335544594	dsql_arr_range_error	Illegal array dimension range
-605	335544682	dsql_field_ref	Inappropriate self-reference of column
-607	335544351	no_meta_update	unsuccessful metadata update
-607	335544549	systrig_update	cannot modify or erase a system trigger
-607	335544657	dsql_no_blob_array	Array/BLOB/DATE data types not allowed in arithmetic
-607	335544746	reftable_requires_pk	"REFERENCES table" without "(column)" requires PRIMARY KEY on referenced table

SQL-CODE	GDSCODE	Symbol	Message Text
-607	335544815	generator_name	GENERATOR @1
-607	335544816	udf_name	Function @1
-607	335544858	must_have_phys_field	Can’t have relation with only computed fields or constraints
-607	336003074	dsql_dbkey_from_non_table	Cannot SELECT RDB$DB_KEY from a stored procedure.
-607	336003086	dsql_udf_return_pos_err	External function should have return position between 1 and @1
-607	336003096	dsql_type_not_supp_ext_tab	Data type @1 is not supported for EXTERNAL TABLES. Relation '@2', field '@3'
-607	336003104	dsql_record_version_table	To be used with RDB$RECORD_VERSION, @1 must be a table or a view of single table
-607	336068845	dyn_cannot_del_syscoll	Cannot delete system collation
-607	336068866	dyn_cannot_mod_sysproc	Cannot ALTER or DROP system procedure @1
-607	336068867	dyn_cannot_mod_systrig	Cannot ALTER or DROP system trigger @1
-607	336068868	dyn_cannot_mod_sysfunc	Cannot ALTER or DROP system function @1
-607	336068869	dyn_invalid_ddl_proc	Invalid DDL statement for procedure @1
-607	336068870	dyn_invalid_ddl_trig	Invalid DDL statement for trigger @1
-607	336068878	dyn_invalid_ddl_func	Invalid DDL statement for function @1
-607	336397206	dsql_table_not_found	Table @1 does not exist
-607	336397207	dsql_view_not_found	View @1 does not exist
-607	336397212	dsql_no_array_computed	Array and BLOB data types not allowed in computed field
-607	336397214	dsql_only_can_subscript_array	scalar operator used on field @1 which is not an array
-612	336068812	dyn_domain_name_exists	Cannot rename domain @1 to @2. A domain with that name already exists.
-612	336068813	dyn_field_name_exists	Cannot rename column @1 to @2. A column with that name already exists in table @3.

SQL-CODE	GDSCODE	Symbol	Message Text
-615	335544475	relation_lock	lock on table @1 conflicts with existing lock
-615	335544476	record_lock	requested record lock conflicts with existing lock
-615	335544501	drop_wal	cannot drop log file when journaling is enabled
-615	335544507	range_in_use	refresh range number @1 already in use
-616	335544530	primary_key_ref	Cannot delete PRIMARY KEY being used in FOREIGN KEY definition.
-616	335544539	integ_index_del	Cannot delete index used by an Integrity Constraint
-616	335544540	integ_index_mod	Cannot modify index used by an Integrity Constraint
-616	335544541	check_trig_del	Cannot delete trigger used by a CHECK Constraint
-616	335544543	cnstrnt_fld_del	Cannot delete column being used in an Integrity Constraint.
-616	335544630	dependency	there are @1 dependencies
-616	335544674	del_last_field	last column in a table cannot be deleted
-616	335544728	integ_index_deactivate	Cannot deactivate index used by an integrity constraint
-616	335544729	integ_deactivate_primary	Cannot deactivate index used by a PRIMARY/UNIQUE constraint
-617	335544542	check_trig_update	Cannot update trigger used by a CHECK Constraint
-617	335544544	cnstrnt_fld_rename	Cannot rename column being used in an Integrity Constraint.
-618	335544537	integ_index_seg_del	Cannot delete index segment used by an Integrity Constraint
-618	335544538	integ_index_seg_mod	Cannot update index segment used by an Integrity Constraint
-625	335544347	not_valid	validation error for column @1, value "@2"
-625	335544879	not_valid_for_var	validation error for variable @1, value "@2"
-625	335544880	not_valid_for	validation error for @1, value "@2"

SQL-CODE	GDSCODE	Symbol	Message Text
-637	335544664	dsql_duplicate_spec	duplicate specification of @1 - not supported
-637	336397213	dsql_implicit_domain_name	Implicit domain name @1 not allowed in user created domain
-660	335544533	foreign_key_notfound	Non-existent PRIMARY or UNIQUE KEY specified for FOREIGN KEY.
-660	335544628	idx_create_err	cannot create index @1
-660	336003098	primary_key_required	Primary key required on table @1
-663	335544624	idx_seg_err	segment count of 0 defined for index @1
-663	335544631	idx_key_err	too many keys defined for index @1
-663	335544672	key_field_err	too few key columns found for index @1 (incorrect column name?)
-664	335544434	keytoobig	key size exceeds implementation restriction for index "@1"
-677	335544445	ext_err	@1 extension error
-685	335544465	bad_segstr_type	invalid BLOB type for operation
-685	335544670	blob_idx_err	attempt to index BLOB column in index @1
-685	335544671	array_idx_err	attempt to index array column in index @1
-689	335544403	badpagtyp	page @1 is of wrong type (expected @2, found @3)
-689	335544650	page_type_err	wrong page type
-690	335544679	no_segments_err	segments not allowed in expression index @1
-691	335544681	rec_size_err	new record size of @1 bytes is too big
-692	335544477	max_idx	maximum indexes per table (@1) exceeded
-693	335544663	req_max_clones_exceeded	Too many concurrent executions of the same request
-694	335544684	no_field_access	cannot access column @1 in view @2
-802	335544321	arith_except	arithmetic exception, numeric overflow, or string truncation
-802	335544836	concat_overflow	Concatenation overflow. Resulting string cannot exceed 32765 bytes in length.

SQL-CODE	GDSCODE	Symbol	Message Text
-802	335544914	string_truncation	string right truncation
-802	335544915	blob_truncation	blob truncation when converting to a string: length limit exceeded
-802	335544916	numeric_out_of_range	numeric value is out of range
-802	336003105	dsql_invalid_sqlda_version	SQLDA version expected between @1 and @2, found @3
-802	336003106	dsql_sqlvar_index	at SQLVAR index @1
-802	336003107	dsql_no_sqlind	empty pointer to NULL indicator variable
-802	336003108	dsql_no_sqldata	empty pointer to data
-802	336003109	dsql_no_input_sqlda	No SQLDA for input values provided
-802	336003110	dsql_no_output_sqlda	No SQLDA for output values provided
-803	335544349	no_dup	attempt to store duplicate value (visible to active transactions) in unique index "@1"
-803	335544665	unique_key_violation	violation of PRIMARY or UNIQUE KEY constraint "@1" on table "@2"
-804	335544380	wronumarg	wrong number of arguments on call
-804	335544583	dsql_sqlda_err	SQLDA error
-804	335544584	dsql_var_count_err	Count of read-write columns does not equal count of values
-804	335544586	dsql_function_err	Function unknown
-804	335544713	dsql_sqlda_value_err	Incorrect values within SQLDA structure
-804	335545050	wrong_message_length	Message length passed from user application does not match set of columns
-804	335545051	no_output_format	Resultset is missing output format information
-804	335545052	item_finish	Message metadata not ready - item @1 is not finished
-804	335545100	interface_version_too_old	Interface @3 version too old: expected @1, found @2
-804	336003097	dsql_feature_not_supported_ods	Feature not supported on ODS version older than @1.@2
-804	336397205	dsql_too_old_ods	ODS versions before ODS@1 are not supported

SQL-CODE	GDSCODE	Symbol	Message Text
-806	335544600	col_name_err	Only simple column names permitted for VIEW WITH CHECK OPTION
-807	335544601	where_err	No WHERE clause for VIEW WITH CHECK OPTION
-808	335544602	table_view_err	Only one table allowed for VIEW WITH CHECK OPTION
-809	335544603	distinct_err	DISTINCT, GROUP or HAVING not permitted for VIEW WITH CHECK OPTION
-810	335544605	subquery_err	No subqueries permitted for VIEW WITH CHECK OPTION
-811	335544652	sing_select_err	multiple rows in singleton select
-816	335544651	ext_readonly_err	Cannot insert because the file is readonly or is on a read only medium.
-816	335544715	extfile_uns_op	Operation not supported for EXTERNAL FILE table @1
-817	335544361	read_only_trans	attempted update during read-only transaction
-817	335544371	segstr_no_write	attempted write to read-only BLOB
-817	335544444	read_only	operation not supported
-817	335544765	read_only_database	attempted update on read-only database
-817	335544766	must_be_dialect_2_and_up	SQL dialect @1 is not supported in this database
-817	335544793	ddl_not_allowed_by_db_sql_dial	Metadata update statement is not allowed by the current database SQL dialect @1
-817	336003079	sql_dialect_conflict_num	DB dialect @1 and client dialect @2 conflict with respect to numeric precision @3.
-817	336003101	upd_ins_with_complex_view	UPDATE OR INSERT without MATCHING could not be used with views based on more than one table
-817	336003102	dsql_incompatible_trigger_type	Incompatible trigger type
-817	336003103	dsql_db_trigger_type_cant_change	Database trigger type can’t be changed
-820	335544356	obsolete_metadata	metadata is obsolete
-820	335544379	wrong_ods	unsupported on-disk structure for file @1; found @2.@3, support @4.@5

SQL-CODE	GDSCODE	Symbol	Message Text
-820	335544437	wrodynver	wrong DYN version
-820	335544467	high_minor	minor version too high found @1 expected @2
-820	335544881	need_difference	Difference file name should be set explicitly for database on raw device
-823	335544473	invalid_bookmark	invalid bookmark handle
-824	335544474	bad_lock_level	invalid lock level @1
-825	335544519	bad_lock_handle	invalid lock handle
-826	335544585	dsql_stmt_handle	Invalid statement handle
-827	335544655	invalid_direction	invalid direction for find operation
-827	335544718	invalid_key	Invalid key for find operation
-828	335544678	inval_key_posn	invalid key position
-829	335544616	field_ref_err	invalid column reference
-829	336068816	dyn_char_fld_too_small	New size specified for column @1 must be at least @2 characters.
-829	336068817	dyn_invalid_dtype_conversion	Cannot change datatype for @1. Conversion from base type @2 to @3 is not supported.
-829	336068818	dyn_dtype_conv_invalid	Cannot change datatype for column @1 from a character type to a non-character type.
-829	336068829	max_coll_per_charset	Maximum number of collations per character set exceeded
-829	336068830	invalid_coll_attr	Invalid collation attributes
-829	336068852	dyn_scale_too_big	New scale specified for column @1 must be at most @2.
-829	336068853	dyn_precision_too_small	New precision specified for column @1 must be at least @2.
-829	336068857	dyn_cannot_addrem_computed	Cannot add or remove COMPUTED from column @1
-830	335544615	field_aggregate_err	column used with aggregate
-831	335544548	primary_key_exists	Attempt to define a second PRIMARY KEY for the same table
-832	335544604	key_field_count_err	FOREIGN KEY column count does not match PRIMARY KEY
-833	335544606	expression_eval_err	expression evaluation not supported

SQL-CODE	GDSCODE	Symbol	Message Text
-833	335544810	date_range_exceeded	value exceeds the range for valid dates
-833	335544912	time_range_exceeded	value exceeds the range for a valid time
-833	335544913	datetime_range_exceeded	value exceeds the range for valid timestamps
-833	335544937	invalid_type_datetime_op	Invalid data type in DATE/TIME/TIMESTAMP addition or subtraction in add_datettime()
-833	335544938	onlycan_add_timetodate	Only a TIME value can be added to a DATE value
-833	335544939	onlycan_add_datetotime	Only a DATE value can be added to a TIME value
-833	335544940	onlycansub_tstampfromtstamp	TIMESTAMP values can be subtracted only from another TIMESTAMP value
-833	335544941	onlyoneop_mustbe_tstamp	Only one operand can be of type TIMESTAMP
-833	335544942	invalid_extractpart_time	Only HOUR, MINUTE, SECOND and MILLISECOND can be extracted from TIME values
-833	335544943	invalid_extractpart_date	HOUR, MINUTE, SECOND and MILLISECOND cannot be extracted from DATE values
-833	335544944	invalidarg_extract	Invalid argument for EXTRACT() not being of DATE/TIME/TIMESTAMP type
-833	335544945	sysf_argmustbe_exact	Arguments for @1 must be integral types or NUMERIC/DECIMAL without scale
-833	335544946	sysf_argmustbe_exact_or_fp	First argument for @1 must be integral type or floating point type
-833	335544947	sysf_argviolates_uuidtype	Human readable UUID argument for @1 must be of string type
-833	335544948	sysf_argviolates_uuidlen	Human readable UUID argument for @2 must be of exact length @1
-833	335544949	sysf_argviolates_uuidfmt	Human readable UUID argument for @3 must have "-" at position @2 instead of "@1"
-833	335544950	sysf_argviolates_guidigits	Human readable UUID argument for @3 must have hex digit at position @2 instead of "@1"

SQL-CODE	GDSCODE	Symbol	Message Text
-833	335544951	sysf_invalid_addpart_time	Only HOUR, MINUTE, SECOND and MILLISECOND can be added to TIME values in @1
-833	335544952	sysf_invalid_add_datetime	Invalid data type in addition of part to DATE/TIME/TIMESTAMP in @1
-833	335544953	sysf_invalid_addpart_dtime	Invalid part @1 to be added to a DATE/TIME/TIMESTAMP value in @2
-833	335544954	sysf_invalid_add_dtime_rc	Expected DATE/TIME/TIMESTAMP type in evlDateAdd() result
-833	335544955	sysf_invalid_diff_dtime	Expected DATE/TIME/TIMESTAMP type as first and second argument to @1
-833	335544956	sysf_invalid_timediff	The result of TIME-<value> in @1 cannot be expressed in YEAR, MONTH, DAY or WEEK
-833	335544957	sysf_invalid_tstamptimediff	The result of TIME-TIMESTAMP or TIMESTAMP-TIME in @1 cannot be expressed in HOUR, MINUTE, SECOND or MILLISECOND
-833	335544958	sysf_invalid_datetimediff	The result of DATE-TIME or TIME-DATE in @1 cannot be expressed in HOUR, MINUTE, SECOND and MILLISECOND
-833	335544959	sysf_invalid_diffpart	Invalid part @1 to express the difference between two DATE/TIME/TIMESTAMP values in @2
-833	335544960	sysf_argmustbe_positive	Argument for @1 must be positive
-833	335544961	sysf_basemustbe_positive	Base for @1 must be positive
-833	335544962	sysf_argnmustbe_nonneg	Argument #@1 for @2 must be zero or positive
-833	335544963	sysf_argnmustbe_positive	Argument #@1 for @2 must be positive
-833	335544964	sysf_invalid_zeropowneg	Base for @1 cannot be zero if exponent is negative
-833	335544965	sysf_invalid_negpowfp	Base for @1 cannot be negative if exponent is not an integral value
-833	335544966	sysf_invalid_scale	The numeric scale must be between -128 and 127 in @1
-833	335544967	sysf_argmustbe_nonneg	Argument for @1 must be zero or positive

SQL-CODE	GDSCODE	Symbol	Message Text
-833	335544968	sysf_binuuid_mustbe_str	Binary UUID argument for @1 must be of string type
-833	335544969	sysf_binuuid_wrongsize	Binary UUID argument for @2 must use @1 bytes
-833	335544976	sysf_argmustbe_nonzero	Argument for @1 must be different than zero
-833	335544977	sysf_argmustbe_range_inc1_1	Argument for @1 must be in the range [-1, 1]
-833	335544978	sysf_argmustbe_gteq_one	Argument for @1 must be greater or equal than one
-833	335544979	sysf_argmustbe_range_exc1_1	Argument for @1 must be in the range]-1, 1[
-833	335544981	sysf_fp_overflow	Floating point overflow in built-in function @1
-833	335545009	sysf_invalid_trig_namespace	Invalid usage of context namespace DDL_TRIGGER
-833	335545024	sysf_argscant_both_be_zero	Arguments for @1 cannot both be zero
-833	335545046	max_args_exceeded	Maximum (@1) number of arguments exceeded for function @2
-833	336397240	dsql_eval_unknode	Unknown node type @1 in dsql/GEN_expr
-833	336397241	dsql_agg_wrongarg	Argument for @1 in dialect 1 must be string or numeric
-833	336397242	dsql_agg2_wrongarg	Argument for @1 in dialect 3 must be numeric
-833	336397243	dsql_nodateortime_pm_string	Strings cannot be added to or subtracted from DATE or TIME types
-833	336397244	dsql_invalid_datetime_subtract	Invalid data type for subtraction involving DATE, TIME or TIMESTAMP types
-833	336397245	dsql_invalid_dateortime_add	Adding two DATE values or two TIME values is not allowed
-833	336397246	dsql_invalid_type_minus_date	DATE value cannot be subtracted from the provided data type
-833	336397247	dsql_nostring_addsub_dial3	Strings cannot be added or subtracted in dialect 3
-833	336397248	dsql_invalid_type_addsub_dial3	Invalid data type for addition or subtraction in dialect 3

SQL-CODE	GDSCODE	Symbol	Message Text
-833	336397249	dsql_invalid_type_multip_dial1	Invalid data type for multiplication in dialect 1
-833	336397250	dsql_nostring_multip_dial3	Strings cannot be multiplied in dialect 3
-833	336397251	dsql_invalid_type_multip_dial3	Invalid data type for multiplication in dialect 3
-833	336397252	dsql_mustuse_numeric_div_dial1	Division in dialect 1 must be between numeric data types
-833	336397253	dsql_nostring_div_dial3	Strings cannot be divided in dialect 3
-833	336397254	dsql_invalid_type_div_dial3	Invalid data type for division in dialect 3
-833	336397255	dsql_nostring_neg_dial3	Strings cannot be negated (applied the minus operator) in dialect 3
-833	336397256	dsql_invalid_type_neg	Invalid data type for negation (minus operator)
-834	335544508	range_not_found	refresh range number @1 not found
-835	335544649	bad_checksum	bad checksum
-836	335544517	except	exception @1
-836	335544848	except2	exception @1
-836	335545016	formatted_exception	@1
-837	335544518	cache_restart	restart shared cache manager
-838	335544560	shutwarn	database @1 shutdown in @2 seconds
-839	335544686	jrn_format_err	journal file wrong format
-840	335544687	jrn_file_full	intermediate journal file full
-841	335544677	version_err	too many versions
-842	335544697	precision_err	Precision must be from 1 to 18
-842	335544698	scale_nogt	Scale must be between zero and precision
-842	335544699	expec_short	Short integer expected
-842	335544700	expec_long	Long integer expected
-842	335544701	expec_ushort	Unsigned short integer expected
-842	335544712	expec_positive	Positive value expected
-901	335544322	bad_dbkey	invalid database key
-901	335544326	bad_dpb_form	unrecognized database parameter block
-901	335544327	bad_req_handle	invalid request handle
-901	335544328	bad_segstr_handle	invalid BLOB handle

SQL-CODE	GDSCODE	Symbol	Message Text
-901	335544329	bad_segstr_id	invalid BLOB ID
-901	335544330	bad_tpb_content	invalid parameter in transaction parameter block
-901	335544331	bad_tpb_form	invalid format for transaction parameter block
-901	335544332	bad_trans_handle	invalid transaction handle (expecting explicit transaction start)
-901	335544337	excess_trans	attempt to start more than @1 transactions
-901	335544339	infinap	information type inappropriate for object specified
-901	335544340	infona	no information of this type available for object specified
-901	335544341	infunk	unknown information item
-901	335544342	integ_fail	action cancelled by trigger (@1) to preserve data integrity
-901	335544345	lock_conflict	lock conflict on no wait transaction
-901	335544350	no_finish	program attempted to exit without finishing database
-901	335544353	no_recon	transaction is not in limbo
-901	335544355	no_segstr_close	BLOB was not closed
-901	335544357	open_trans	cannot disconnect database with open transactions (@1 active)
-901	335544358	port_len	message length error (encountered @1, expected @2)
-901	335544363	req_no_trans	no transaction for request
-901	335544364	req_sync	request synchronization error
-901	335544365	req_wrong_db	request referenced an unavailable database
-901	335544369	segstr_no_read	attempted read of a new, open BLOB
-901	335544370	segstr_no_trans	attempted action on BLOB outside transaction
-901	335544372	segstr_wrong_db	attempted reference to BLOB in unavailable database
-901	335544376	unres_rel	table @1 was omitted from the transaction reserving list

SQL-CODE	GDSCODE	Symbol	Message Text
-901	335544377	uns_ext	request includes a DSRI extension not supported in this implementation
-901	335544378	wish_list	feature is not supported
-901	335544382	random	@1
-901	335544383	fatal_conflict	unrecoverable conflict with limbo transaction @1
-901	335544392	bdbincon	internal error
-901	335544407	dbbnotzer	database handle not zero
-901	335544408	tranotzer	transaction handle not zero
-901	335544418	trainlim	transaction in limbo
-901	335544419	notinlim	transaction not in limbo
-901	335544420	traoutsta	transaction outstanding
-901	335544428	badmsgnum	undefined message number
-901	335544431	blocking_signal	blocking signal has been received
-901	335544442	noargacc_read	database system cannot read argument @1
-901	335544443	noargacc_write	database system cannot write argument @1
-901	335544450	misc_interpreted	@1
-901	335544468	tra_state	transaction @1 is @2
-901	335544485	bad_stmt_handle	invalid statement handle
-901	335544510	lock_timeout	lock time-out on wait transaction
-901	335544559	bad_svc_handle	invalid service handle
-901	335544561	wrospbver	wrong version of service parameter block
-901	335544562	bad_spb_form	unrecognized service parameter block
-901	335544563	svcnotdef	service @1 is not defined
-901	335544609	index_name	INDEX @1
-901	335544610	exception_name	EXCEPTION @1
-901	335544611	field_name	COLUMN @1
-901	335544613	union_err	union not supported
-901	335544614	dsql_construct_err	Unsupported DSQL construct
-901	335544623	dsql_domain_err	Illegal use of keyword VALUE
-901	335544626	table_name	TABLE @1

SQL-CODE	GDSCODE	Symbol	Message Text
-901	335544627	proc_name	PROCEDURE @1
-901	335544641	dsql_domain_not_found	Specified domain or source column @1 does not exist
-901	335544656	dsql_var_conflict	variable @1 conflicts with parameter in same procedure
-901	335544666	srvr_version_too_old	server version too old to support all CREATE DATABASE options
-901	335544673	no_delete	cannot delete
-901	335544675	sort_err	sort error
-901	335544703	svcnoexe	service @1 does not have an associated executable
-901	335544704	net_lookup_err	Failed to locate host machine.
-901	335544705	service_unknown	Undefined service @1/@2.
-901	335544706	host_unknown	The specified name was not found in the hosts file or Domain Name Services.
-901	335544711	unprepared_stmt	Attempt to execute an unprepared dynamic SQL statement.
-901	335544716	svc_in_use	Service is currently busy: @1
-901	335544719	net_init_error	Error initializing the network software.
-901	335544720	loadlib_failure	Unable to load required library @1.
-901	335544731	tra_must_sweep	
-901	335544740	udf_exception	A fatal exception occurred during the execution of a user defined function.
-901	335544741	lost_db_connection	connection lost to database
-901	335544742	no_write_user_priv	User cannot write to RDB$USER_PRIVILEGES
-901	335544767	blob_filter_exception	A fatal exception occurred during the execution of a blob filter.
-901	335544768	exception_access_violation	Access violation. The code attempted to access a virtual address without privilege to do so.
-901	335544769	exception_datatype_missalignment	Datatype misalignment. The attempted to read or write a value that was not stored on a memory boundary.
-901	335544770	exception_array_bounds_exceeded	Array bounds exceeded. The code attempted to access an array element that is out of bounds.

SQL-CODE	GDSCODE	Symbol	Message Text
-901	335544771	exception_float_denormal_operand	Float denormal operand. One of the floating-point operands is too small to represent a standard float value.
-901	335544772	exception_float_divide_by_zero	Floating-point divide by zero. The code attempted to divide a floating-point value by zero.
-901	335544773	exception_float_inexact_result	Floating-point inexact result. The result of a floating-point operation cannot be represented as a deciaml fraction.
-901	335544774	exception_float_invalid_operand	Floating-point invalid operand. An indeterminant error occurred during a floating-point operation.
-901	335544775	exception_float_overflow	Floating-point overflow. The exponent of a floating-point operation is greater than the magnitude allowed.
-901	335544776	exception_float_stack_check	Floating-point stack check. The stack overflowed or underflowed as the result of a floating-point operation.
-901	335544777	exception_float_underflow	Floating-point underflow. The exponent of a floating-point operation is less than the magnitude allowed.
-901	335544778	exception_integer_divide_by_zero	Integer divide by zero. The code attempted to divide an integer value by an integer divisor of zero.
-901	335544779	exception_integer_overflow	Integer overflow. The result of an integer operation caused the most significant bit of the result to carry.
-901	335544780	exception_unknown	An exception occurred that does not have a description. Exception number @1.
-901	335544781	exception_stack_overflow	Stack overflow. The resource requirements of the runtime stack have exceeded the memory available to it.
-901	335544782	exception_sigsegv	Segmentation Fault. The code attempted to access memory without privileges.
-901	335544783	exception_sigill	Illegal Instruction. The Code attempted to perfrom an illegal operation.
-901	335544784	exception_sigbus	Bus Error. The Code caused a system bus error.

SQL-CODE	GDSCODE	Symbol	Message Text
-901	335544785	exception_sigfpe	Floating Point Error. The Code caused an Arithmetic Exception or a floating point exception.
-901	335544786	ext_file_delete	Cannot delete rows from external files.
-901	335544787	ext_file_modify	Cannot update rows in external files.
-901	335544788	adm_task_denied	Unable to perform operation. You must be either SYSDBA or owner of the database
-901	335544794	cancelled	operation was cancelled
-901	335544797	svcnouser	user name and password are required while attaching to the services manager
-901	335544801	datype_notsup	data type not supported for arithmetic
-901	335544803	dialect_not_changed	Database dialect not changed.
-901	335544804	database_create_failed	Unable to create database @1
-901	335544805	inv_dialect_specified	Database dialect @1 is not a valid dialect.
-901	335544806	valid_db_dialects	Valid database dialects are @1.
-901	335544811	inv_client_dialect_specified	passed client dialect @1 is not a valid dialect.
-901	335544812	valid_client_dialects	Valid client dialects are @1.
-901	335544814	service_not_supported	Services functionality will be supported in a later version of the product
-901	335544820	invalid_savepoint	Unable to find savepoint with name @1 in transaction context
-901	335544835	bad_shutdown_mode	Target shutdown mode is invalid for database "@1"
-901	335544840	no_update	cannot update
-901	335544842	stack_trace	@1
-901	335544843	ctx_var_not_found	Context variable @1 is not found in namespace @2
-901	335544844	ctx_namespace_invalid	Invalid namespace name @1 passed to @2
-901	335544845	ctx_too_big	Too many context variables
-901	335544846	ctx_bad_argument	Invalid argument passed to @1
-901	335544847	identifier_too_long	BLR syntax error. Identifier @1... is too long

SQL-CODE	GDSCODE	Symbol	Message Text
-901	335544859	invalid_time_precision	Time precision exceeds allowed range (0-@1)
-901	335544866	met_wrong_gtt_scope	@1 cannot depend on @2
-901	335544868	illegal_prc_type	Procedure @1 is not selectable (it does not contain a SUSPEND statement)
-901	335544869	invalid_sort_datatype	Datatype @1 is not supported for sorting operation
-901	335544870	collation_name	COLLATION @1
-901	335544871	domain_name	DOMAIN @1
-901	335544874	max_db_per_trans_allowed	A multi database transaction cannot span more than @1 databases
-901	335544876	bad_proc_BLR	Error while parsing procedure @1's BLR
-901	335544877	key_too_big	index key too big
-901	335544885	bad_teb_form	Invalid TEB format
-901	335544886	tpb_multiple_txn_isolation	Found more than one transaction isolation in TPB
-901	335544887	tpb_reserv_before_table	Table reservation lock type @1 requires table name before in TPB
-901	335544888	tpb_multiple_spec	Found more than one @1 specification in TPB
-901	335544889	tpb_option_without_rc	Option @1 requires READ COMMITTED isolation in TPB
-901	335544890	tpb_conflicting_options	Option @1 is not valid if @2 was used previously in TPB
-901	335544891	tpb_reserv_missing_tlen	Table name length missing after table reservation @1 in TPB
-901	335544892	tpb_reserv_long_tlen	Table name length @1 is too long after table reservation @2 in TPB
-901	335544893	tpb_reserv_missing_tname	Table name length @1 without table name after table reservation @2 in TPB
-901	335544894	tpb_reserv_corrup_tlen	Table name length @1 goes beyond the remaining TPB size after table reservation @2
-901	335544895	tpb_reserv_null_tlen	Table name length is zero after table reservation @1 in TPB
-901	335544896	tpb_reserv_relnotfound	Table or view @1 not defined in system tables after table reservation @2 in TPB

SQL-CODE	GDSCODE	Symbol	Message Text
-901	335544897	tpb_reserv_baserelnotfound	Base table or view @1 for view @2 not defined in system tables after table reservation @3 in TPB
-901	335544898	tpb_missing_len	Option length missing after option @1 in TPB
-901	335544899	tpb_missing_value	Option length @1 without value after option @2 in TPB
-901	335544900	tpb_corrupt_len	Option length @1 goes beyond the remaining TPB size after option @2
-901	335544901	tpb_null_len	Option length is zero after table reservation @1 in TPB
-901	335544902	tpb_overflow_len	Option length @1 exceeds the range for option @2 in TPB
-901	335544903	tpb_invalid_value	Option value @1 is invalid for the option @2 in TPB
-901	335544904	tpb_reserv_stronger_wng	Preserving previous table reservation @1 for table @2, stronger than new @3 in TPB
-901	335544905	tpb_reserv_stronger	Table reservation @1 for table @2 already specified and is stronger than new @3 in TPB
-901	335544906	tpb_reserv_max_recursion	Table reservation reached maximum recursion of @1 when expanding views in TPB
-901	335544907	tpb_reserv_virtualtbl	Table reservation in TPB cannot be applied to @1 because it’s a virtual table
-901	335544908	tpb_reserv_systbl	Table reservation in TPB cannot be applied to @1 because it’s a system table
-901	335544909	tpb_reserv_temptbl	Table reservation @1 or @2 in TPB cannot be applied to @3 because it’s a temporary table
-901	335544910	tpb_readtxn_after_writelock	Cannot set the transaction in read only mode after a table reservation isc_tpb_lock_write in TPB
-901	335544911	tpb_writelock_after_readtxn	Cannot take a table reservation isc_tpb_lock_write in TPB because the transaction is in read only mode
-901	335544917	shutdown_timeout	Firebird shutdown is still in progress after the specified timeout

SQL-CODE	GDSCODE	Symbol	Message Text
-901	335544918	att_handle_busy	Attachment handle is busy
-901	335544919	bad_udf_freeit	Bad written UDF detected: pointer returned in FREE_IT function was not allocated by ib_util_malloc
-901	335544920	eds_provider_not_found	External Data Source provider '@1' not found
-901	335544921	eds_connection	Execute statement error at @1 : @2Data source : @3
-901	335544922	eds_preprocess	Execute statement preprocess SQL error
-901	335544923	eds_stmt_expected	Statement expected
-901	335544924	eds_prm_name_expected	Parameter name expected
-901	335544925	eds_unclosed_comment	Unclosed comment found near '@1'
-901	335544926	eds_statement	Execute statement error at @1 : @2Statement : @3 Data source : @4
-901	335544927	eds_input_prm_mismatch	Input parameters mismatch
-901	335544928	eds_output_prm_mismatch	Output parameters mismatch
-901	335544929	eds_input_prm_not_set	Input parameter '@1' have no value set
-901	335544933	nothing_to_cancel	nothing to cancel
-901	335544934	ibutil_not_loaded	ib_util library has not been loaded to deallocate memory returned by FREE_IT function
-901	335544973	bad_epb_form	Unrecognized events block
-901	335544982	udf_fp_overflow	Floating point overflow in result from UDF @1
-901	335544983	udf_fp_nan	Invalid floating point value returned by UDF @1
-901	335544985	out_of_temp_space	No free space found in temporary directories
-901	335544986	eds_expl_tran_ctrl	Explicit transaction control is not allowed
-901	335544988	package_name	PACKAGE @1
-901	335544989	cannot_make_not_null	Cannot make field @1 of table @2 NOT NULL because there are NULLs present
-901	335544990	feature_removed	Feature @1 is not supported anymore
-901	335544991	view_name	VIEW @1

SQL-CODE	GDSCODE	Symbol	Message Text
-901	335544993	invalid_fetch_option	Fetch option @1 is invalid for a non-scrollable cursor
-901	335544994	bad_fun_BLR	Error while parsing function @1's BLR
-901	335544995	func_pack_not_implemented	Cannot execute function @1 of the unimplemented package @2
-901	335544996	proc_pack_not_implemented	Cannot execute procedure @1 of the unimplemented package @2
-901	335544997	eem_func_not_returned	External function @1 not returned by the external engine plugin @2
-901	335544998	eem_proc_not_returned	External procedure @1 not returned by the external engine plugin @2
-901	335544999	eem_trig_not_returned	External trigger @1 not returned by the external engine plugin @2
-901	335545000	eem_bad_plugin_ver	Incompatible plugin version @1 for external engine @2
-901	335545001	eem_engine_notfound	External engine @1 not found
-901	335545004	pman_cannot_load_plugin	Error loading plugin @1
-901	335545005	pman_module_notfound	Loadable module @1 not found
-901	335545006	pman_entrypoint_notfound	Standard plugin entrypoint does not exist in module @1
-901	335545007	pman_module_bad	Module @1 exists but can not be loaded
-901	335545008	pman_plugin_notfound	Module @1 does not contain plugin @2 type @3
-901	335545010	unexpected_null	Value is NULL but isNull parameter was not informed
-901	335545011	type_notcompat_blob	Type @1 is incompatible with BLOB
-901	335545012	invalid_date_val	Invalid date
-901	335545013	invalid_time_val	Invalid time
-901	335545014	invalid_timestamp_val	Invalid timestamp
-901	335545015	invalid_index_val	Invalid index @1 in function @2
-901	335545018	private_function	Function @1 is private to package @2
-901	335545019	private_procedure	Procedure @1 is private to package @2
-901	335545021	bad_events_handle	invalid events id (handle)
-901	335545025	spb_no_id	missing service ID in spb
-901	335545026	ee_blr_mismatch_null	External BLR message mismatch: invalid null descriptor at field @1

SQL-CODE	GDSCODE	Symbol	Message Text
-901	335545027	ee_blr_mismatch_length	External BLR message mismatch: length = @1, expected @2
-901	335545031	libtommath_generic	Libtommath error code @1 in function @2
-901	335545041	cp_process_active	Crypt failed - already crypting database
-901	335545042	cp_already_crypted	Crypt failed - database is already in requested state
-901	335545047	ee_blr_mismatch_names_count	External BLR message mismatch: names count = @1, blr count = @2
-901	335545048	ee_blr_mismatch_name_not_found	External BLR message mismatch: name @1 not found
-901	335545049	bad_result_set	Invalid resultset interface
-901	335545059	badvarnum	undefined variable number
-901	335545071	info_unprepared_stmt	Attempt to get information about an unprepared dynamic SQL statement.
-901	335545072	idx_key_value	Problematic key value is @1
-901	335545073	forupdate_virtualtbl	Cannot select virtual table @1 for update WITH LOCK
-901	335545074	forupdate_systbl	Cannot select system table @1 for update WITH LOCK
-901	335545075	forupdate_temptbl	Cannot select temporary table @1 for update WITH LOCK
-901	335545076	cant_modify_sysobj	System @1 @2 cannot be modified
-901	335545077	server_misconfigured	Server misconfigured - contact administrator please
-901	335545078	alter_role	Deprecated backward compatibility ALTER ROLE ... SET/DROP AUTO ADMIN mapping may be used only for RDB$ADMIN role
-901	335545079	map_already_exists	Mapping @1 already exists
-901	335545080	map_not_exists	Mapping @1 does not exist
-901	335545081	map_load	@1 failed when loading mapping cache
-901	335545082	map_aster	Invalid name <*> in authentication block
-901	335545083	map_multi	Multiple maps found for @1
-901	335545084	map_undefined	Undefined mapping result - more than one different results found

SQL-CODE	GDSCODE	Symbol	Message Text
-901	335545088	map_nodb	Global mapping is not available when database @1 is not present
-901	335545089	map_notable	Global mapping is not available when table RDB$MAP is not present in database @1
-901	335545090	miss_trusted_role	Your attachment has no trusted role
-901	335545091	set_invalid_role	Role @1 is invalid or unavailable
-901	335545093	dup_attribute	Duplicated user attribute @1
-901	335545094	dyn_no_priv	There is no privilege for this operation
-901	335545095	dsql_cant_grant_option	Using GRANT OPTION on @1 not allowed
-901	335545097	crdb_load	@1 failed when working with CREATE DATABASE grants
-901	335545098	crdb_nodb	CREATE DATABASE grants check is not possible when database @1 is not present
-901	335545099	crdb_notable	CREATE DATABASE grants check is not possible when table RDB$DB_CREATORS is not present in database @1
-901	335545102	savepoint_backout_err	Error during savepoint backout - transaction invalidated
-901	335545105	map_down	Some database(s) were shutdown when trying to read mapping data
-901	335545109	encrypt_error	Page requires encryption but crypt plugin is missing
-901	336068645	dyn_filter_not_found	BLOB Filter @1 not found
-901	336068649	dyn_func_not_found	Function @1 not found
-901	336068656	dyn_index_not_found	Index not found
-901	336068662	dyn_view_not_found	View @1 not found
-901	336068697	dyn_domain_not_found	Domain not found
-901	336068717	dyn_cant_modify_auto_trig	Triggers created automatically cannot be modified
-901	336068740	dyn_dup_table	Table @1 already exists
-901	336068748	dyn_proc_not_found	Procedure @1 not found
-901	336068752	dyn_exception_not_found	Exception not found

SQL-CODE	GDSCODE	Symbol	Message Text
-901	336068754	dyn_proc_param_not_found	Parameter @1 in procedure @2 not found
-901	336068755	dyn_trig_not_found	Trigger @1 not found
-901	336068759	dyn_charset_not_found	Character set @1 not found
-901	336068760	dyn_collation_not_found	Collation @1 not found
-901	336068763	dyn_role_not_found	Role @1 not found
-901	336068767	dyn_name_longer	Name longer than database column size
-901	336068784	dyn_column_does_not_exist	column @1 does not exist in table/view @2
-901	336068796	dyn_role_does_not_exist	SQL role @1 does not exist
-901	336068797	dyn_no_grant_admin_opt	user @1 has no grant admin option on SQL role @2
-901	336068798	dyn_user_not_role_member	user @1 is not a member of SQL role @2
-901	336068799	dyn_delete_role_failed	@1 is not the owner of SQL role @2
-901	336068800	dyn_grant_role_to_user	@1 is a SQL role and not a user
-901	336068801	dyn_inv_sql_role_name	user name @1 could not be used for SQL role
-901	336068802	dyn_dup_sql_role	SQL role @1 already exists
-901	336068803	dyn_kywd_spec_for_role	keyword @1 can not be used as a SQL role name
-901	336068804	dyn_roles_not_supported	SQL roles are not supported in on older versions of the database. A backup and restore of the database is required.
-901	336068820	dyn_zero_len_id	Zero length identifiers are not allowed
-901	336068822	dyn_gen_not_found	Sequence @1 not found
-901	336068840	dyn_wrong_gtt_scope	@1 cannot reference @2
-901	336068843	dyn_coll_used_table	Collation @1 is used in table @2 (field name @3) and cannot be dropped
-901	336068844	dyn_coll_used_domain	Collation @1 is used in domain @2 and cannot be dropped
-901	336068846	dyn_cannot_del_def_coll	Cannot delete default collation of CHARACTER SET @1
-901	336068849	dyn_table_not_found	Table @1 not found
-901	336068851	dyn_coll_used_procedure	Collation @1 is used in procedure @2 (parameter name @3) and cannot be dropped

SQL-CODE	GDSCODE	Symbol	Message Text
-901	336068856	dyn_ods_not_supp_feature	Feature '@1' is not supported in ODS @2.@3
-901	336068858	dyn_no_empty_pw	Password should not be empty string
-901	336068859	dyn_dup_index	Index @1 already exists
-901	336068864	dyn_package_not_found	Package @1 not found
-901	336068865	dyn_schema_not_found	Schema @1 not found
-901	336068871	dyn_funcnotdef_package	Function @1 has not been defined on the package body @2
-901	336068872	dyn_procnotdef_package	Procedure @1 has not been defined on the package body @2
-901	336068873	dyn_funcsignat_package	Function @1 has a signature mismatch on package body @2
-901	336068874	dyn_procsignat_package	Procedure @1 has a signature mismatch on package body @2
-901	336068875	dyn_defvaldecl_package_proc	Default values for parameters are allowed only in declaration of packaged procedure @1.@2
-901	336068877	dyn_package_body_exists	Package body @1 already exists
-901	336068879	dyn_newfc_oldsyntax	Cannot alter new style function @1 with ALTER EXTERNAL FUNCTION. Use ALTER FUNCTION instead.
-901	336068886	dyn_func_param_not_found	Parameter @1 in function @2 not found
-901	336068887	dyn_routine_param_not_found	Parameter @1 of routine @2 not found
-901	336068888	dyn_routine_param_ambiguous	Parameter @1 of routine @2 is ambiguous (found in both procedures and functions). Use a specifier keyword.
-901	336068889	dyn_coll_used_function	Collation @1 is used in function @2 (parameter name @3) and cannot be dropped
-901	336068890	dyn_domain_used_function	Domain @1 is used in function @2 (parameter name @3) and cannot be dropped
-901	336068891	dyn_alter_user_no_clause	ALTER USER requires at least one clause to be specified
-901	336068894	dyn_duplicate_package_item	Duplicate @1 @2
-901	336068895	dyn_cant_modify_sysobj	System @1 @2 cannot be modified

SQL-CODE	GDSCODE	Symbol	Message Text
-901	336068896	dyn_cant_use_zero_increment	INCREMENT BY 0 is an illegal option for sequence @1
-901	336068897	dyn_cant_use_in_foreignkey	Can’t use @1 in FOREIGN KEY constraint
-901	336068898	dyn_defvaldecl_package_func	Default values for parameters are allowed only in declaration of packaged function @1.@2
-901	336397211	dsql_too_many_values	Too many values (more than @1) in member list to match against
-901	336397236	dsql_unsupp_feature_dialect	feature is not supported in dialect @1
-901	336397239	dsql_unsupported_in_auto_trans	@1 is not supported inside IN AUTONOMOUS TRANSACTION block
-901	336397258	dsql_alter_charset_failed	ALTER CHARACTER SET @1 failed
-901	336397259	dsql_comment_on_failed	COMMENT ON @1 failed
-901	336397260	dsql_create_func_failed	CREATE FUNCTION @1 failed
-901	336397261	dsql_alter_func_failed	ALTER FUNCTION @1 failed
-901	336397262	dsql_create_alter_func_failed	CREATE OR ALTER FUNCTION @1 failed
-901	336397263	dsql_drop_func_failed	DROP FUNCTION @1 failed
-901	336397264	dsql_recreate_func_failed	RECREATE FUNCTION @1 failed
-901	336397265	dsql_create_proc_failed	CREATE PROCEDURE @1 failed
-901	336397266	dsql_alter_proc_failed	ALTER PROCEDURE @1 failed
-901	336397267	dsql_create_alter_proc_failed	CREATE OR ALTER PROCEDURE @1 failed
-901	336397268	dsql_drop_proc_failed	DROP PROCEDURE @1 failed
-901	336397269	dsql_recreate_proc_failed	RECREATE PROCEDURE @1 failed
-901	336397270	dsql_create_trigger_failed	CREATE TRIGGER @1 failed
-901	336397271	dsql_alter_trigger_failed	ALTER TRIGGER @1 failed
-901	336397272	dsql_create_alter_trigger_failed	CREATE OR ALTER TRIGGER @1 failed
-901	336397273	dsql_drop_trigger_failed	DROP TRIGGER @1 failed
-901	336397274	dsql_recreate_trigger_failed	RECREATE TRIGGER @1 failed
-901	336397275	dsql_create_collation_failed	CREATE COLLATION @1 failed
-901	336397276	dsql_drop_collation_failed	DROP COLLATION @1 failed
-901	336397277	dsql_create_domain_failed	CREATE DOMAIN @1 failed
-901	336397278	dsql_alter_domain_failed	ALTER DOMAIN @1 failed
-901	336397279	dsql_drop_domain_failed	DROP DOMAIN @1 failed

SQL-CODE	GDSCODE	Symbol	Message Text
-901	336397280	dsql_create_except_failed	CREATE EXCEPTION @1 failed
-901	336397281	dsql_alter_except_failed	ALTER EXCEPTION @1 failed
-901	336397282	dsql_create_alter_except_failed	CREATE OR ALTER EXCEPTION @1 failed
-901	336397283	dsql_recreate_except_failed	RECREATE EXCEPTION @1 failed
-901	336397284	dsql_drop_except_failed	DROP EXCEPTION @1 failed
-901	336397285	dsql_create_sequence_failed	CREATE SEQUENCE @1 failed
-901	336397286	dsql_create_table_failed	CREATE TABLE @1 failed
-901	336397287	dsql_alter_table_failed	ALTER TABLE @1 failed
-901	336397288	dsql_drop_table_failed	DROP TABLE @1 failed
-901	336397289	dsql_recreate_table_failed	RECREATE TABLE @1 failed
-901	336397290	dsql_create_pack_failed	CREATE PACKAGE @1 failed
-901	336397291	dsql_alter_pack_failed	ALTER PACKAGE @1 failed
-901	336397292	dsql_create_alter_pack_failed	CREATE OR ALTER PACKAGE @1 failed
-901	336397293	dsql_drop_pack_failed	DROP PACKAGE @1 failed
-901	336397294	dsql_recreate_pack_failed	RECREATE PACKAGE @1 failed
-901	336397295	dsql_create_pack_body_failed	CREATE PACKAGE BODY @1 failed
-901	336397296	dsql_drop_pack_body_failed	DROP PACKAGE BODY @1 failed
-901	336397297	dsql_recreate_pack_body_failed	RECREATE PACKAGE BODY @1 failed
-901	336397298	dsql_create_view_failed	CREATE VIEW @1 failed
-901	336397299	dsql_alter_view_failed	ALTER VIEW @1 failed
-901	336397300	dsql_create_alter_view_failed	CREATE OR ALTER VIEW @1 failed
-901	336397301	dsql_recreate_view_failed	RECREATE VIEW @1 failed
-901	336397302	dsql_drop_view_failed	DROP VIEW @1 failed
-901	336397303	dsql_drop_sequence_failed	DROP SEQUENCE @1 failed
-901	336397304	dsql_recreate_sequence_failed	RECREATE SEQUENCE @1 failed
-901	336397305	dsql_drop_index_failed	DROP INDEX @1 failed
-901	336397306	dsql_drop_filter_failed	DROP FILTER @1 failed
-901	336397307	dsql_drop_shadow_failed	DROP SHADOW @1 failed
-901	336397308	dsql_drop_role_failed	DROP ROLE @1 failed
-901	336397309	dsql_drop_user_failed	DROP USER @1 failed
-901	336397310	dsql_create_role_failed	CREATE ROLE @1 failed
-901	336397311	dsql_alter_role_failed	ALTER ROLE @1 failed

SQL-CODE	GDSCODE	Symbol	Message Text
-901	336397312	dsql_alter_index_failed	ALTER INDEX @1 failed
-901	336397313	dsql_alter_database_failed	ALTER DATABASE failed
-901	336397314	dsql_create_shadow_failed	CREATE SHADOW @1 failed
-901	336397315	dsql_create_filter_failed	DECLARE FILTER @1 failed
-901	336397316	dsql_create_index_failed	CREATE INDEX @1 failed
-901	336397317	dsql_create_user_failed	CREATE USER @1 failed
-901	336397318	dsql_alter_user_failed	ALTER USER @1 failed
-901	336397319	dsql_grant_failed	GRANT failed
-901	336397320	dsql_revoke_failed	REVOKE failed
-901	336397322	dsql_mapping_failed	@2 MAPPING @1 failed
-901	336397323	dsql_alter_sequence_failed	ALTER SEQUENCE @1 failed
-901	336397324	dsql_create_generator_failed	CREATE GENERATOR @1 failed
-901	336397325	dsql_set_generator_failed	SET GENERATOR @1 failed
-901	336397330	dsql_max_exception_arguments	Number of arguments (@1) exceeds the maximum (@2) number of EXCEPTION USING arguments
-901	336397331	dsql_string_byte_length	String literal with @1 bytes exceeds the maximum length of @2 bytes
-901	336397332	dsql_string_char_length	String literal with @1 characters exceeds the maximum length of @2 characters for the @3 character set
-901	336397333	dsql_max_nesting	Too many BEGIN...END nesting. Maximum level is @1
-902	335544333	bug_check	internal Firebird consistency check (@1)
-902	335544335	db_corrupt	database file appears corrupt (@1)
-902	335544344	io_error	I/O error during "@1" operation for file "@2"
-902	335544346	metadata_corrupt	corrupt system table
-902	335544373	sys_request	operating system directive @1 failed
-902	335544384	badblk	internal error
-902	335544385	invpoolcl	internal error
-902	335544387	relbadblk	internal error
-902	335544388	blktoobig	block size exceeds implementation restriction

SQL-CODE	GDSCODE	Symbol	Message Text
-902	335544394	badodsver	incompatible version of on-disk structure
-902	335544397	dirtypage	internal error
-902	335544398	waifortra	internal error
-902	335544399	doubleloc	internal error
-902	335544400	nodnotfnd	internal error
-902	335544401	dupnodfnd	internal error
-902	335544402	locnotmar	internal error
-902	335544404	corrupt	database corrupted
-902	335544405	badpage	checksum error on database page @1
-902	335544406	badindex	index is broken
-902	335544409	trareqmis	transaction—request mismatch (synchronization error)
-902	335544410	badhndcnt	bad handle count
-902	335544411	wrotpbver	wrong version of transaction parameter block
-902	335544412	wroblrver	unsupported BLR version (expected @1, encountered @2)
-902	335544413	wrodpbver	wrong version of database parameter block
-902	335544415	badrelation	database corrupted
-902	335544416	nodetach	internal error
-902	335544417	notremote	internal error
-902	335544422	dbfile	internal error
-902	335544423	orphan	internal error
-902	335544432	lockmanerr	lock manager error
-902	335544436	sqlerr	SQL error code = @1
-902	335544448	bad_sec_info	
-902	335544449	invalid_sec_info	
-902	335544470	buf_invalid	cache buffer for page @1 invalid
-902	335544471	indexnotdefined	there is no index in table @1 with id @2
-902	335544472	login	Your user name and password are not defined. Ask your database administrator to set up a Firebird login.

SQL-CODE	GDSCODE	Symbol	Message Text
-902	335544478	jrn_enable	enable journal for database before starting online dump
-902	335544479	old_failure	online dump failure. Retry dump
-902	335544480	old_in_progress	an online dump is already in progress
-902	335544481	old_no_space	no more disk/tape space. Cannot continue online dump
-902	335544482	no_wal_no_jrn	journaling allowed only if database has Write-ahead Log
-902	335544483	num_old_files	maximum number of online dump files that can be specified is 16
-902	335544484	wal_file_open	error in opening Write-ahead Log file during recovery
-902	335544486	wal_failure	Write-ahead log subsystem failure
-902	335544505	no_archive	must specify archive file when enabling long term journal for databases with round-robin log files
-902	335544506	shutinprog	database @1 shutdown in progress
-902	335544520	jrn_present	long-term journaling already enabled
-902	335544528	shutdown	database @1 shutdown
-902	335544557	shutfail	database shutdown unsuccessful
-902	335544564	no_jrn	long-term journaling not enabled
-902	335544569	dsql_error	Dynamic SQL Error
-902	335544653	psw_attach	cannot attach to password database
-902	335544654	psw_start_trans	cannot start transaction for password database
-902	335544717	err_stack_limit	stack size insufficent to execute current request
-902	335544721	network_error	Unable to complete network request to host "@1".
-902	335544722	net_connect_err	Failed to establish a connection.
-902	335544723	net_connect_listen_err	Error while listening for an incoming connection.
-902	335544724	net_event_connect_err	Failed to establish a secondary connection for event processing.
-902	335544725	net_event_listen_err	Error while listening for an incoming event connection request.

SQL-CODE	GDSCODE	Symbol	Message Text
-902	335544726	net_read_err	Error reading data from the connection.
-902	335544727	net_write_err	Error writing data to the connection.
-902	335544732	unsupported_network_drive	Access to databases on file servers is not supported.
-902	335544733	io_create_err	Error while trying to create file
-902	335544734	io_open_err	Error while trying to open file
-902	335544735	io_close_err	Error while trying to close file
-902	335544736	io_read_err	Error while trying to read from file
-902	335544737	io_write_err	Error while trying to write to file
-902	335544738	io_delete_err	Error while trying to delete file
-902	335544739	io_access_err	Error while trying to access file
-902	335544745	login_same_as_role_name	Your login @1 is same as one of the SQL role name. Ask your database administrator to set up a valid Firebird login.
-902	335544791	file_in_use	The file @1 is currently in use by another process. Try again later.
-902	335544795	unexp_spb_form	unexpected item in service parameter block, expected @1
-902	335544809	extern_func_dir_error	Function @1 is in @2, which is not in a permitted directory for external functions.
-902	335544819	io_32bit_exceeded_err	File exceeded maximum size of 2GB. Add another database file or use a 64 bit I/O version of Firebird.
-902	335544831	conf_access_denied	Use of @1 at location @2 is not allowed by server configuration
-902	335544834	cursor_not_open	Cursor is not open
-902	335544841	cursor_already_open	Cursor is already open
-902	335544856	att_shutdown	connection shutdown
-902	335544882	long_login	Login name too long (@1 characters, maximum allowed @2)
-902	335544936	psw_db_error	Security database error
-902	335544970	missing_required_spb	Missing required item @1 in service parameter block
-902	335544971	net_server_shutdown	@1 server is shutdown

SQL-CODE	GDSCODE	Symbol	Message Text
-902	335544974	no_threads	Could not start first worker thread - shutdown server
-902	335544975	net_event_connect_timeout	Timeout occurred while waiting for a secondary connection for event processing
-902	335544984	instance_conflict	Database is probably already opened by another engine instance in another Windows session
-902	335544987	no_trusted_spb	Use of TRUSTED switches in spb_command_line is prohibited
-902	335545029	missing_data_structures	Install incomplete, please read the Compatibility chapter in the release notes for this version
-902	335545030	protect_sys_tab	@1 operation is not allowed for system table @2
-902	335545032	wroblrver2	unsupported BLR version (expected between @1 and @2, encountered @3)
-902	335545043	decrypt_error	Missing crypt plugin, but page appears encrypted
-902	335545044	no_providers	No providers loaded
-902	335545053	miss_config	Missing configuration file: @1
-902	335545054	conf_line	@1: illegal line <@2>
-902	335545055	conf_include	Invalid include operator in @1 for <@2>
-902	335545056	include_depth	Include depth too big
-902	335545057	include_miss	File to include not found
-902	335545060	sec_context	Missing security context for @1
-902	335545061	multi_segment	Missing segment @1 in multisegment connect block parameter
-902	335545062	login_changed	Different logins in connect and attach packets - client library error
-902	335545063	auth_handshake_limit	Exceeded exchange limit during authentication handshake
-902	335545064	wirecrypt_incompatible	Incompatible wire encryption levels requested on client and server
-902	335545065	miss_wirecrypt	Client attempted to attach unencrypted but wire encryption is required

SQL-CODE	GDSCODE	Symbol	Message Text
-902	335545066	wirecrypt_key	Client attempted to start wire encryption using unknown key @1
-902	335545067	wirecrypt_plugin	Client attempted to start wire encryption using unsupported plugin @1
-902	335545068	secdb_name	Error getting security database name from configuration file
-902	335545069	auth_data	Client authentication plugin is missing required data from server
-902	335545070	auth_datalength	Client authentication plugin expected @2 bytes of @3 from server, got @1
-902	335545106	login_error	Error occurred during login, please check server firebird.log for details
-902	335545107	already_opened	Database already opened with engine instance, incompatible with current
-902	335545108	bad_crypt_key	Invalid crypt key @1
-904	335544324	bad_db_handle	invalid database handle (no active connection)
-904	335544375	unavailable	unavailable database
-904	335544381	imp_exc	Implementation limit exceeded
-904	335544386	nopoolids	too many requests
-904	335544389	bufexh	buffer exhausted
-904	335544391	bufinuse	buffer in use
-904	335544393	reqinuse	request in use
-904	335544424	no_lock_mgr	no lock manager available
-904	335544430	virmemexh	unable to allocate memory from operating system
-904	335544451	update_conflict	update conflicts with concurrent update
-904	335544453	obj_in_use	object @1 is in use
-904	335544455	shadow_accessed	cannot attach active shadow file
-904	335544460	shadow_missing	a file in manual shadow @1 is unavailable
-904	335544661	index_root_page_full	cannot add index, index root page is full.
-904	335544676	sort_mem_err	sort error: not enough memory

SQL-CODE	GDSCODE	Symbol	Message Text
-904	335544683	req_depth_exceeded	request depth exceeded. (Recursive definition?)
-904	335544758	sort_rec_size_err	sort record size of @1 bytes is too big
-904	335544761	too_many_handles	too many open handles to database
-904	335544762	optimizer_blk_exc	size of optimizer block exceeded
-904	335544792	service_att_err	Cannot attach to services manager
-904	335544799	svc_name_missing	The service name was not specified.
-904	335544813	optimizer_between_err	Unsupported field type specified in BETWEEN predicate.
-904	335544827	exec_sql_invalid_arg	Invalid argument in EXECUTE STATEMENT - cannot convert to string
-904	335544828	exec_sql_invalid_req	Wrong request type in EXECUTE STATEMENT '@1'
-904	335544829	exec_sql_invalid_var	Variable type (position @1) in EXECUTE STATEMENT '@2' INTO does not match returned column type
-904	335544830	exec_sql_max_call_exceeded	Too many recursion levels of EXECUTE STATEMENT
-904	335544832	wrong_backup_state	Cannot change difference file name while database is in backup mode
-904	335544833	wal_backup_err	Physical backup is not allowed while Write-Ahead Log is in use
-904	335544852	partner_idx_incompat_type	partner index segment no @1 has incompatible data type
-904	335544857	blobtoobig	Maximum BLOB size exceeded
-904	335544862	record_lock_not_supp	Stream does not support record locking
-904	335544863	partner_idx_not_found	Cannot create foreign key constraint @1. Partner index does not exist or is inactive.
-904	335544864	tra_num_exc	Transactions count exceeded. Perform backup and restore to make database operable again
-904	335544865	field_disappeared	Column has been unexpectedly deleted
-904	335544878	concurrent_transaction	concurrent transaction number is @1
-904	335544935	circular_computed	Cannot have circular dependencies with computed fields
-904	335544992	lock_dir_access	Can not access lock files directory @1

SQL-CODE	GDSCODE	Symbol	Message Text
-904	335545020	request_outdated	Request can’t access new records in relation @1 and should be recompiled
-904	335545096	read_conflict	read conflicts with concurrent update
-906	335544452	unlicensed	product @1 is not licensed
-906	335544744	max_att_exceeded	Maximum user count exceeded. Contact your database administrator.
-909	335544667	drdb_completed_with_errs	drop database completed with errors
-911	335544459	rec_in_limbo	record from transaction @1 is stuck in limbo
-913	335544336	deadlock	deadlock
-922	335544323	bad_db_format	file @1 is not a valid database
-923	335544421	connect_reject	connection rejected by remote interface
-923	335544461	cant_validate	secondary server attachments cannot validate databases
-923	335544462	cant_start_journal	secondary server attachments cannot start journaling
-923	335544464	cant_start_logging	secondary server attachments cannot start logging
-924	335544325	bad_dpb_content	bad parameters on attach or create database
-924	335544433	journerr	communication error with journal "@1"
-924	335544441	bad_detach	database detach completed with errors
-924	335544648	conn_lost	Connection lost to pipe server
-924	335544972	bad_conn_str	Invalid connection string
-924	335545085	baddpb_damaged_mode	Incompatible mode of attachment to damaged database
-924	335545086	baddpb_buffers_range	Attempt to set in database number of buffers which is out of acceptable range [@1:@2]
-924	335545087	baddpb_temp_buffers	Attempt to temporarily set number of buffers less than @1
-926	335544447	no_rollback	no rollback performed
-999	335544689	ib_error	Firebird error

Anhang C: Reservierte Wörter und Schlüsselwörter

Reservierte Wörter sind Teil der Firebird SQL-Sprache. Sie können nicht als Bezeichner verwendet werden (z.B. als Tabellen- oder Prozedurname), es sei denn sie werden in doppelte Anführungszeichen gesetzt. Dies gilt nur für Dialekt 3. Grundsätzlich sollte dies jedoch vermieden werden.

Schlüsselwörter sind ebenfalls Teil der Sprache. Sie besitzen eine spezielle Bedeutung, sofern sie ordnungsgemäß eingesetzt werden. Sie sind jedoch nicht für die eigene und exklusive Verwendung unter Firebird reserviert. Sie können diese als Bezeichner ohne doppelte Anführungszeichen nutzen.

Reservierte Wörter

Vollständige Liste reservierter Wörter in Firebird 3.0:

ADD	ADMIN	ALL
ALTER	AND	ANY
AS	AT	AVG
BEGIN	BETWEEN	BIGINT
BIT_LENGTH	BLOB	BOOLEAN
BOTH	BY	CASE
CAST	CHAR	CHARACTER
CHARACTER_LENGTH	CHAR_LENGTH	CHECK
CLOSE	COLLATE	COLUMN
COMMIT	CONNECT	CONSTRAINT
CORR	COUNT	COVAR_POP
COVAR_SAMP	CREATE	CROSS
CURRENT	CURRENT_CONNECTION	CURRENT_DATE
CURRENT_ROLE	CURRENT_TIME	CURRENT_TIMESTAMP
CURRENT_TRANSACTION	CURRENT_USER	CURSOR
DATE	DAY	DEC
DECIMAL	DECLARE	DEFAULT
DELETE	DELETING	DETERMINISTIC
DISCONNECT	DISTINCT	DOUBLE
DROP	ELSE	END
ESCAPE	EXECUTE	EXISTS
EXTERNAL	EXTRACT	FALSE
FETCH	FILTER	FLOAT
FOR	FOREIGN	FROM

FULL	FUNCTION	GDSCODE
GLOBAL	GRANT	GROUP
HAVING	HOUR	IN
INDEX	INNER	INSENSITIVE
INSERT	INSERTING	INT
INTEGER	INTO	IS
JOIN	LEADING	LEFT
LIKE	LONG	LOWER
MAX	MERGE	MIN
MINUTE	MONTH	NATIONAL
NATURAL	NCHAR	NO
NOT	NULL	NUMERIC
OCTET_LENGTH	OF	OFFSET
ON	ONLY	OPEN
OR	ORDER	OUTER
OVER	PARAMETER	PLAN
POSITION	POST_EVENT	PRECISION
PRIMARY	PROCEDURE	RDB$DB_KEY
RDB$RECORD_VERSION	REAL	RECORD_VERSION
RECREATE	RECURSIVE	REFERENCES
REGR_AVGX	REGR_AVGY	REGR_COUNT
REGR_INTERCEPT	REGR_R2	REGR_SLOPE
REGR_SXX	REGR_SXY	REGR_SYY
RELEASE	RETURN	RETURNING_VALUES
RETURNS	REVOKE	RIGHT
ROLLBACK	ROW	ROWS
ROW_COUNT	SAVEPOINT	SCROLL
SECOND	SELECT	SENSITIVE
SET	SIMILAR	SMALLINT
SOME	SQLCODE	SQLSTATE
START	STDDEV_POP	STDDEV_SAMP
SUM	TABLE	THEN
TIME	TIMESTAMP	TO
TRAILING	TRIGGER	TRIM
TRUE	UNION	UNIQUE
UNKNOWN	UPDATE	UPDATING
UPPER	USER	USING
VALUE	VALUES	VARCHAR
VARIABLE	VARYING	VAR_POP

VAR_SAMP	VIEW	WHEN
WHERE	WHILE	WITH
YEAR		

Schlüsselwörter

Die folgenden Terme (Zeichen, Zeichenkombinationen, Wörter) haben in der DSQL von Firebird 3.0 eine spezielle Bedeutung. Einige sind außerdem reservierte Wörter, andere nicht.

!<	^<	^=
^>	,	:=
!=	!>	(
)	<	⇐
<>	=	>
>=	||	~<
~=	~>	ABS
ABSOLUTE	ACCENT	ACOS
ACOSH	ACTION	ACTIVE
ADD	ADMIN	AFTER
ALL	ALTER	ALWAYS
AND	ANY	AS
ASC	ASCENDING	ASCII_CHAR
ASCII_VAL	ASIN	ASINH
AT	ATAN	ATAN2
ATANH	AUTO	AUTONOMOUS
AVG	BACKUP	BEFORE
BEGIN	BETWEEN	BIGINT
BIN_AND	BIN_NOT	BIN_OR
BIN_SHL	BIN_SHR	BIN_XOR
BIT_LENGTH	BLOB	BLOCK
BODY	BOOLEAN	BOTH
BREAK	BY	CALLER
CASCADE	CASE	CAST
CEIL	CEILING	CHAR
CHARACTER	CHARACTER_LENGTH	CHAR_LENGTH
CHAR_TO_UUID	CHECK	CLOSE
COALESCE	COLLATE	COLLATION
COLUMN	COMMENT	COMMIT
COMMITTED	COMMON	COMPUTED
CONDITIONAL	CONNECT	CONSTRAINT

CONTAINING	CONTINUE	CORR
COS	COSH	COT
COUNT	COVAR_POP	COVAR_SAMP
CREATE	CROSS	CSTRING
CURRENT	CURRENT_CONNECTION	CURRENT_DATE
CURRENT_ROLE	CURRENT_TIME	CURRENT_TIMESTAMP
CURRENT_TRANSACTION	CURRENT_USER	CURSOR
DATA	DATABASE	DATE
DATEADD	DATEDIFF	DAY
DB_KEY	DDL	DEC
DECIMAL	DECLARE	DECODE
DECRYPT	DEFAULT	DELETE
DELETING	DENSE_RANK	DESC
DESCENDING	DESCRIPTOR	DETERMINISTIC
DIFFERENCE	DISCONNECT	DISTINCT
DO	DOMAIN	DOUBLE
DROP	ELSE	ENCRYPT
END	ENGINE	ENTRY_POINT
ESCAPE	EXCEPTION	EXECUTE
EXISTS	EXIT	EXP
EXTERNAL	EXTRACT	FALSE
FETCH	FILE	FILTER
FIRST	FIRSTNAME	FIRST_VALUE
FLOAT	FLOOR	FOR
FOREIGN	FREE_IT	FROM
FULL	FUNCTION	GDSCODE
GENERATED	GENERATOR	GEN_ID
GEN_UUID	GLOBAL	GRANT
GRANTED	GROUP	HASH
HAVING	HOUR	IDENTITY
IF	IGNORE	IIF
IN	INACTIVE	INCREMENT
INDEX	INNER	INPUT_TYPE
INSENSITIVE	INSERT	INSERTING
INT	INTEGER	INTO
IS	ISOLATION	JOIN
KEY	LAG	LAST
LASTNAME	LAST_VALUE	LEAD
LEADING	LEAVE	LEFT

LENGTH	LEVEL	LIKE
LIMBO	LINGER	LIST
LN	LOCALTIME	LOCALTIMESTAMP
LOCK	LOG	LOG10
LONG	LOWER	LPAD
MANUAL	MAPPING	MATCHED
MATCHING	MAX	MAXVALUE
MERGE	MIDDLENAME	MILLISECOND
MIN	MINUTE	MINVALUE
MOD	MODULE_NAME	MONTH
NAME	NAMES	NATIONAL
NATURAL	NCHAR	NEXT
NO	NOT	NTH_VALUE
NULL	NULLIF	NULLS
NUMERIC	OCTET_LENGTH	OF
OFFSET	ON	ONLY
OPEN	OPTION	OR
ORDER	OS_NAME	OUTER
OUTPUT_TYPE	OVER	OVERFLOW
OVERLAY	PACKAGE	PAD
PAGE	PAGES	PAGE_SIZE
PARAMETER	PARTITION	PASSWORD
PI	PLACING	PLAN
PLUGIN	POSITION	POST_EVENT
POWER	PRECISION	PRESERVE
PRIMARY	PRIOR	PRIVILEGES
PROCEDURE	PROTECTED	RAND
RANK	RDB$DB_KEY	RDB$GET_CONTEXT
RDB$RECORD_VERSION	RDB$SET_CONTEXT	RDB_GET_CONTEXT
RDB_SET_CONTEXT	READ	REAL
RECORD_VERSION	RECREATE	RECURSIVE
REFERENCES	REGR_AVGX	REGR_AVGY
REGR_COUNT	REGR_INTERCEPT	REGR_R2
REGR_SLOPE	REGR_SXX	REGR_SXY
REGR_SYY	RELATIVE	RELEASE
REPLACE	REQUESTS	RESERV
RESERVING	RESTART	RESTRICT
RETAIN	RETURN	RETURNING
RETURNING_VALUES	RETURNS	REVERSE

REVOKE	RIGHT	ROLE
ROLLBACK	ROUND	ROW
ROWS	ROW_COUNT	ROW_NUMBER
RPAD	SAVEPOINT	SCALAR_ARRAY
SCHEMA	SCROLL	SECOND
SEGMENT	SELECT	SENSITIVE
SEQUENCE	SERVERWIDE	SET
SHADOW	SHARED	SIGN
SIMILAR	SIN	SINGULAR
SINH	SIZE	SKIP
SMALLINT	SNAPSHOT	SOME
SORT	SOURCE	SPACE
SQLCODE	SQLSTATE	SQRT
STABILITY	START	STARTING
STARTS	STATEMENT	STATISTICS
STDDEV_POP	STDDEV_SAMP	SUBSTRING
SUB_TYPE	SUM	SUSPEND
TABLE	TAGS	TAN
TANH	TEMPORARY	THEN
TIME	TIMEOUT	TIMESTAMP
TO	TRAILING	TRANSACTION
TRIGGER	TRIM	TRUE
TRUNC	TRUSTED	TWO_PHASE
TYPE	UNCOMMITTED	UNDO
UNION	UNIQUE	UNKNOWN
UPDATE	UPDATING	UPPER
USAGE	USER	USING
UUID_TO_CHAR	VALUE	VALUES
VARCHAR	VARIABLE	VARYING
VAR_POP	VAR_SAMP	VIEW
WAIT	WEEK	WEEKDAY
WHEN	WHERE	WHILE
WITH	WORK	WRITE
YEAR	YEARDAY	

Anhang D: Systemtabellen

Wenn Sie eine Datenbank erstellen, generiert die Firebird Engine einige Systemtabellen. Metadaten — die Beschreibungen und Eigenschaften aller Datenbankobjekte — werden in den Systemtabellen gespeichert.

Systemtabellen werden durch den Präfix `RDB$` gekennzeichnet.

Liste der Systemtabellen

`RDB$AUTH_MAPPING`

Speichert die Authentifizierung und andere Sicherheitszuordnungen

`RDB$BACKUP_HISTORY`

Historie der Backups, die mittels *nBackup* durchgeführt wurden.

`RDB$CHARACTER_SETS`

Benennt und beschreibt die in der Datenbank verfügbaren Zeichensätze

`RDB$CHECK_CONSTRAINTS`

Querverweise zwischen den Constraint-Namen (`NOT NULL`-Constraints, `CHECK`-Constraints sowie `ON UPDATE`- und `ON DELETE`-Klauseln eines Fremdschlüssel-Constraints) und ihren systemgenerierten Triggern.

`RDB$COLLATIONS`

Collation-Sequenzen für alle Zeichensätze

`RDB$DATABASE`

Basisinformationen über die Datenbank

`RDB$DB_CREATORS`

Eine Liste von Benutzern, denen das Privileg `CREATE DATABASE` gewährt wurde, wenn die angegebene Datenbank als Sicherheitsdatenbank verwendet wird

`RDB$DEPENDENCIES`

Informationen zu den Abhängigkeiten zwischen Datenbankobjekten

`RDB$EXCEPTIONS`

Benutzerdefinierte Datenbank-Exceptions

`RDB$FIELDS`

Spalten- und Domain-Definitionen, sowohl system- als auch benutzerdefiniert

`RDB$FIELD_DIMENSIONS`

Dimensionen der Arrayspalten

`RDB$FILES`

Informationen über sekundäre und Shadow-Dateien

RDB$FILTERS

Informationen über BLOB-Filter

RDB$FORMATS

Informationen über Änderungen der Tabellenformate

RDB$FUNCTIONS

Informationen über externe Funktionen

RDB$FUNCTION_ARGUMENTS

Parametereigenschaften externer Funktionen

RDB$GENERATORS

Informationen über Generatoren (Sequenzen)

RDB$INDEX_SEGMENTS

Segmente und Index-Positionen

RDB$INDICES

Definitionen aller Datenbankindizes (System- und Benutzerdefiniert)

RDB$LOG_FILES

In derzeitigen Versionen nicht verwendet

RDB$PACKAGES

Speichert die Definition (Header und Body) von SQL-Paketen

RDB$PAGES

Informationen über Datenbankseiten

RDB$PROCEDURES

Definitionen der Stored Procedures

RDB$PROCEDURE_PARAMETERS

Parameter der Stored Procedures

RDB$REF_CONSTRAINTS

Definitionen der referentiellen Constraints (Fremdschlüssel)

RDB$RELATIONS

Header für Tabellen und Views

RDB$RELATION_CONSTRAINTS

Definitionen aller Einschränkungen auf Tabellenebene

RDB$RELATION_FIELDS

Definitionen von Tabellenspalten auf oberster Ebene

RDB$ROLES

Rollendefinitionen

RDB$SECURITY_CLASSES

Zugriffslisten (ACL)

RDB$TRANSACTIONS

Status für Multi-Datenbank-Transaktionen

RDB$TRIGGERS

Trigger-Definitionen

RDB$TRIGGER_MESSAGES

Triggermeldungen

RDB$TYPES

Definitionen für enumerierte Datentypen

RDB$USER_PRIVILEGES

An Systembenutzer zugewiesene SQL-Privilegien

RDB$VIEW_RELATIONS

Tabellen denen View-Definitionen zugewiesen wurden: eine Zeile für jede Tabelle innerhalb einer View

RDB$AUTH_MAPPING

RDB$AUTH_MAPPING speichert Authentifizierung und andere Sicherheitszuordnungen.

Spaltenname	Datentyp	Beschreibung
RDB$MAP_NAME	CHAR(31)	Name des Mappings
RDB$MAP_USING	CHAR(1)	Verwendete Definitionen: P - Plugin (spezifisch oder beliebig) S - jedes Plugin serverweit M - Mapping * - jede Methode
RDB$MAP_PLUGIN	CHAR(31)	Die Zuordnung gilt für Authentifizierungsinformationen von diesem bestimmten Plugin
RDB$MAP_DB	CHAR(31)	Die Zuordnung gilt für Authentifizierungsinformationen aus dieser bestimmten Datenbank

Spaltenname	Datentyp	Beschreibung
RDB$MAP_FROM_TYPE	CHAR(31)	Der Typ des Authentifizierungsobjekts (definiert durch das Plugin), von dem die Zuordnung erfolgen soll, oder * für jeden Typ
RDB$MAP_FROM	CHAR(255)	Der Name des Authentifizierungsobjekts, von dem aus eine Zuordnung vorgenommen werden soll
RDB$MAP_TO_TYPE	SMALLINT	Der Typ, dem zugeordnet werden soll 0 - USER 1 - ROLE
RDB$MAP_TO	CHAR(31)	Der Name, dem zugeordnet werden soll
RDB$SYSTEM_FLAG	SMALLINT	Flag: 0 - benutzerdefiniert 1 oder höher - systemdefiniert
RDB$DESCRIPTION	BLOB TEXT	Optionale Beschreibung des Mappings (Kommentar)

RDB$BACKUP_HISTORY

RDB$BACKUP_HISTORY speichert die Historie der Backups, die mittels *nBackup* durchgeführt wurden.

Spaltenname	Datentyp	Beschreibung
RDB$BACKUP_ID	INTEGER	Durch die Engine vergebene Kennung
RDB$TIMESTAMP	TIMESTAMP	Zeitstempel des Backup
RDB$BACKUP_LEVEL	INTEGER	Backup-Level
RDB$GUID	CHAR(38)	Eindeutige Kennung
RDB$SCN	INTEGER	Systemnummer (Scan)
RDB$FILE_NAME	VARCHAR(255)	Vollständiger Pfad und Dateiname der Backupdatei

RDB$CHARACTER_SETS

RDB$CHARACTER_SETS benennt und beschreibt die in der Datenbank verfügbaren Zeichensätze.

Spaltenname	Datentyp	Beschreibung
RDB$CHARACTER_SET_NAME	CHAR(31)	Name des Zeichensatzes
RDB$FORM_OF_USE	CHAR(31)	Nicht verwendet

Spaltenname	Datentyp	Beschreibung
RDB$NUMBER_OF_CHARACTERS	INTEGER	Die Anzahl der Zeichen im Zeichensatz. Wird nicht für existente Zeichensätze verwendet.
RDB$DEFAULT_COLLATE_NAME	CHAR(31)	Der Name der Standard-Collation-Sequenz für den Zeichensatz
RDB$CHARACTER_SET_ID	SMALLINT	Eindeutige Kennung des Zeichensatzes
RDB$SYSTEM_FLAG	SMALLINT	Systemkennzeichen: Wert ist 1 wenn der Zeichensatz bei Erstellung der Datenbank festgelegt wurde; Wert ist 0 für einen benutzerdefinierten Zeichensatz.
RDB$DESCRIPTION	BLOB TEXT	Kann die Textbeschreibung des Zeichensatzes speichern
RDB$FUNCTION_NAME	CHAR(31)	Für benutzerdefinierte Zeichensätze, auf die über externe Funktionen zugegriffen wird, ist dies der Name der externen Funktion.
RDB$BYTES_PER_CHARACTER	SMALLINT	Die maximale Anzahl von Bytes, die ein Zeichen repräsentieren.
RDB$SECURITY_CLASS	CHAR(31)	Kann auf eine in der Tabelle RDB$SECURITY_CLASSES definierte Sicherheitsklasse verweisen, um Zugriffskontrollbeschränkungen auf alle Benutzer dieses Zeichensatzes anzuwenden.
RDB$OWNER_NAME	CHAR(31)	Der Benutzername des Benutzers, der den Zeichensatz ursprünglich erstellt hat

RDB$CHECK_CONSTRAINTS

RDB$CHECK_CONSTRAINTS enthält die Querverweise zwischen den systemgenerierten Triggern für Contraints sowie die Namen der zugewiesenen Constraints (NOT NULL-Constraints, CHECK-Constraints sowie die ON UPDATE- und ON DELETE-Klauseln in Fremdschlüssel-Constraints).

Spaltenname	Datentyp	Beschreibung
RDB$CONSTRAINT_NAME	CHAR(31)	Constraint-Name, der durch den Benutzer oder automatisch durch das System vergeben wurde.

Spaltenname	Datentyp	Beschreibung
RDB$TRIGGER_NAME	CHAR(31)	Für CHECK-Constraints ist dies der Name des Triggers, der diesen Constraint erzwingt. Für NOT NULL-Constraints ist dies der Name der Tabelle, die diesen Constraint enthält. Für Fremdschlüssel-Constraints ist dies der Name des Trigger, der die ON UPDATE- und ON DELETE -Klauseln erzwingt.

RDB$COLLATIONS

RDB$COLLATIONS speichert Kollatierungssequenzen für alle Zeichensätze.

Spaltenname	Datentyp	Beschreibung
RDB$COLLATION_NAME	CHAR(31)	Name der Collation-Sequenz
RDB$COLLATION_ID	SMALLINT	Kennung der Collation-Sequenz. Bildet zusammen mit der Kennung des Zeichensatzen eine eindeutige Kennung.
RDB$CHARACTER_SET_ID	SMALLINT	Kennung des Zeichensatzes. Bildet zusammen mit der Kennung der Collation-Sequenz eine eindeutige Kennung.
RDB$COLLATION_ATTRIBUTES	SMALLINT	Collation-Eigenschaften. Dies ist eine Bitmaske, wobei das erste Bit angibt, ob nachstehende Leerzeichen in Collations berücksichtigt werden sollen (0 - NO PAD; 1 - PAD SPACE); das zweite Bit gibt an, ob die Collation sensitiv für Groß- und Kleinschreibung ist (0 - CASE SENSITIVE, 1 - CASE INSENSITIVE); das dritte Bit gibt an, ob die Collation Akzent-sensitiv ist (0 - ACCENT SENSITIVE, 1 - ACCENT SENSITIVE). Hieraus ergibt sich, dass die Collation bei einem Wert von 5 nachstehende Leerzeichen nicht berücksichtigt und Akzent-sensitiv ist.
RDB$SYSTEM_FLAG	SMALLINT	Kennzeichen: der Wert 0 bedeutet benutzerdefiniert; der Wert 1 bedeutet systemdefiniert.
RDB$DESCRIPTION	BLOB TEXT	Kann Textbeschreibung der Collation speichern
RDB$FUNCTION_NAME	CHAR(31)	Derzeit nicht verwendet

Spaltenname	Datentyp	Beschreibung
RDB$BASE_COLLATION_NAME	CHAR(31)	Der Name der Basis-Collation für diese Collation-Sequenz.
RDB$SPECIFIC_ATTRIBUTES	BLOB TEXT	Beschreibt spezifische Eigenschaften.
RDB$SECURITY_CLASS	CHAR(31)	Kann auf eine in der Tabelle RDB$SECURITY_CLASSES definierte Sicherheitsklasse verweisen, um Zugriffskontrollbeschränkungen auf alle Benutzer dieser Kollation anzuwenden.
RDB$OWNER_NAME	CHAR(31)	Der Benutzername des Benutzers, der die Sortierung ursprünglich erstellt hat

RDB$DATABASE

RDB$DATABASE speichert grundlegende Informationen über die Datenbank. Es enthält nur einen Datensatz.

Spaltenname	Datentyp	Beschreibung
RDB$DESCRIPTION	BLOB TEXT	Datenbankkommentar.
RDB$RELATION_ID	SMALLINT	Zähler der durch jede neu erstellte Tabelle oder View um eins erhöht wird.
RDB$SECURITY_CLASS	CHAR(31)	Die Sicherheitsklasse, die in Tabelle RDB$SECURITY_CLASSES definiert wurde, um Zugriffe für die gesamte Datenbank zu begrenzen.
RDB$CHARACTER_SET_NAME	CHAR(31)	Der Name des Standardzeichensatzes, der mittels der DEFAULT CHARACTER SET -Klausel während der Datenbankerstellung gesetzt wurde. NULL für den Zeichensatz NONE.
RDB$LINGER	INTEGER	Die "Verzögerung" in Sekunden (festgelegt mit der Anweisung "ALTER DATABASE SET LINGER"), bis die Datenbankdatei geschlossen wird, nachdem die letzte Verbindung zu dieser Datenbank geschlossen wurde (in SuperServer). NULL, wenn keine Verzögerung eingestellt ist.

RDB$DB_CREATORS

RDB$DB_CREATORS enthält eine Liste von Benutzern, denen das CREATE DATABASE-Privileg gewährt wurde, wenn die angegebene Datenbank als Sicherheitsdatenbank verwendet wird.

Spaltenname	Datentyp	Beschreibung
RDB$USER	CHAR(31)	Benutzer- oder Rollenname
RDB$USER_TYPE	SMALLINT	Benutzertyp 8 - user 13 - role

RDB$DEPENDENCIES

RDB$DEPENDENCIES speichert die Abhängigkeiten zwischen Datenbankobjekten.

Spaltenname	Datentyp	Beschreibung
RDB$DEPENDENT_NAME	CHAR(31)	Der Name der View, Prozedur, Trigger, CHECK-Constraint oder Computed Column, für die die Abhängigkeit definiert ist, z.B. das *abhängige* Objekt.
RDB$DEPENDED_ON_NAME	CHAR(31)	Der Name des Objekts, von dem das definierte Objekt — Tabelle, View, Prozedur, Trigger, CHECK-Constraint oder Computed Column — abhängig ist.
RDB$FIELD_NAME	CHAR(31)	Der Spaltenname im abhängigen Objekt, das auf eine View, Prozedur, Trigger, CHECK-Constraint oder Computed Column verweist.
RDB$DEPENDENT_TYPE	SMALLINT	Kennzeichnet den Typ des abhängigen Objekts: 0 - Tabelle 1 - View 2 - Trigger 3 - Computed Column 4 - CHECK-Constraint 5 - Prozedur 6 - Index-Ausdruck 7 - Exception 8 - User 9 - Spalte 10 - Index 15 - Stored Function 18 - Package Header 19 - Package Body

Spaltenname	Datentyp	Beschreibung
RDB$DEPENDED_ON_TYPE	SMALLINT	Kennzeichnet den Typ des Objekts, auf das verwiesen wird: 0 - Tabelle (oder darin enthaltene Spalte) 1 - View 2 - Trigger 3 - Computed-Column 4 - CHECK-Constraint 5 - Prozedur (oder deren Parameter) 6 - Index-Anweisung 7 - Exception 8 - User 9 - Spalte 10 - Index 14 - Generator (Sequence) 15 - UDF 17 - Collation 18 - Package Header 19 - Package Body
RDB$PACKAGE_NAME	CHAR(31)	Das Paket einer Prozedur oder Funktion, für die dies die Abhängigkeit beschreibt.

RDB$EXCEPTIONS

RDB$EXCEPTIONS speichert benutzerdefinierte Datenbankausnahmen.

Spaltenname	Datentyp	Beschreibung
RDB$EXCEPTION_NAME	CHAR(31)	Benutzerdefinierter Exception-Name
RDB$EXCEPTION_NUMBER	INTEGER	Die eindeutige Nummer der Exception, die durch das System zugewiesen wurde
RDB$MESSAGE	VARCHAR(1021)	Exception-Meldungstext
RDB$DESCRIPTION	BLOB TEXT	Kann die Beschreibung der Exception speichern
RDB$SYSTEM_FLAG	SMALLINT	Kennzeichen: 0 - Benutzerdefiniert 1 oder höher - Systemdefiniert

Spaltenname	Datentyp	Beschreibung
RDB$SECURITY_CLASS	CHAR(31)	Kann auf eine in der Tabelle RDB$SECURITY_CLASSES definierte Sicherheitsklasse verweisen, um Zugriffskontrollbeschränkungen auf alle Benutzer dieser Ausnahme anzuwenden.
RDB$OWNER_NAME	CHAR(31)	Der Benutzername des Benutzers, der die Ausnahme ursprünglich erstellt hat

RDB$FIELDS

RDB$FIELDS speichert Definitionen von Spalten und Domänen, sowohl System- als auch benutzerdefiniert. Hier werden die detaillierten Datenattribute für alle Spalten gespeichert.

Die Spalte RDB$FIELDS.RDB$FIELD_NAME verlinkt auf RDB$RELATION_FIELDS.RDB$FIELD_SOURCE, nicht auf RDB$RELATION_FIELDS.RDB$FIELD_NAME.

Spaltenname	Datentyp	Beschreibung
RDB$FIELD_NAME	CHAR(31)	Der eindeutige Name der Domain. Wird durch den Benutzer festgelegt oder automatisch durch das System. Domains die durch das System erstellt wurden, beginnen mit dem Präfix “RDB$”.
RDB$QUERY_NAME	CHAR(31)	Nicht in Verwendung
RDB$VALIDATION_BLR	BLOB BLR	Die Binärsprachenrepräsentation (BLR) des SQL-Ausdrucks, der die Prüfung der CHECK-Werte in der Domain angibt
RDB$VALIDATION_SOURCE	BLOB TEXT	Der originale Quelltext in SQL, der die Prüfung des CHECK-Wertes angibt
RDB$COMPUTED_BLR	BLOB BLR	Die Binärsprachenrepräsentation (BLR) des SQL-Ausdrucks, welchen der Datenbankserver verwendet, wenn auf COMPUTED BY-Spalten zugegriffen wird.
RDB$COMPUTED_SOURCE	BLOB TEXT	Der originale Quelltext der Anweisung, der die COMPUTED BY-Spalte definiert.
RDB$DEFAULT_VALUE	BLOB BLR	Der Standardwert, sofern vorhanden, für das Feld oder die Domain, in Binarsprachenrepräsentation (BLR).
RDB$DEFAULT_SOURCE	BLOB TEXT	Der Vorgabewert als Quelltext, als SQL-Konstante oder -Ausdruck.

Spaltenname	Datentyp	Beschreibung
RDB$FIELD_LENGTH	SMALLINT	Spaltengröße in Bytes. BOOLEAN beansprucht 1 Byte. FLOAT, DATE, TIME, INTEGER beanspruchen 4 Bytes. DOUBLE PRECISION, BIGINT, TIMESTAMP und BLOB beanspruchen 8 Bytes. Für CHAR- und VARCHAR-Datentypen wird die größtmögliche Anzahl Bytes beansprucht, wenn eine String-Domain (Spalte) definiert wurde.
RDB$FIELD_SCALE	SMALLINT	Die negative Nummer, die die Präzision für DECIMAL- und NUMERIC-Spalten festlegt — die Anzahl der Stellen nach dem Dezimalkomma.
RDB$FIELD_TYPE	SMALLINT	Code des Datentyps für die Spalte: 7 - SMALLINT `8 - INTEGER `10 - FLOAT 12 - DATE 13 - TIME 14 - CHAR 16 - BIGINT 23 - BOOLEAN 27 - DOUBLE PRECISION 35 - TIMESTAMP 37 - VARCHAR 261 - BLOB Codes für DECIMAL und NUMERIC sind die gleichen wie für Integer-Typen, da diese als solche gespeichert werden.

Spaltenname	Datentyp	Beschreibung
RDB$FIELD_SUB_TYPE	SMALLINT	Gibt den Untertyp für BLOB-Datentypen an: 0 - undefiniert 1 - Text 2 - BLR 3 - ACL 4 - für zukünftigen Gebrauch reserviert 5 - Enkodierte Tabellenmetadatenbeschreibung 6 - Speicherung der Details für übergreifende Datenbanktransaktionen, die abnormal beendet wurden. 7 - Beschreibung der externen Datei 8 - Debug-Informationen (für PSQL) Spezifikationen für die CHAR-Datentypen: 0 - untypisierte Daten 1 - feste Binärdaten Spezifiziert einen bestimmten Datentyp für die Integer-Datentypen (SMALLINT, INTEGER, BIGINT) und für Festkomma-Zahlen (NUMERIC, DECIMAL): 0 oder NULL - der Datentyp passt zum Wert im Feld RDB$FIELD_TYPE 1 - NUMERIC 2 - DECIMAL
RDB$MISSING_VALUE	BLOB BLR	Nicht verwendet
RDB$MISSING_SOURCE	BLOB TEXT	Nicht verwendet
RDB$DESCRIPTION	BLOB TEXT	Beliebiger Kommentar für Domains (Tabellenspalten)
RDB$SYSTEM_FLAG	SMALLINT	Kennzeichen: der Wert 1 bedeutet, dass die Domain automatisch durch das System erstellt wurde, der Wert 0 bedeutet, die Domain wurde durch den Benutzer definiert.
RDB$QUERY_HEADER	BLOB TEXT	Nicht verwendet

Spaltenname	Datentyp	Beschreibung
RDB$SEGMENT_LENGTH	SMALLINT	Gibt die Länge der BLOB-Buffer in Bytes für BLOB-Spalten an. Verwendet NULL für alle anderen Datentypen.
RDB$EDIT_STRING	VARCHAR(127)	Nicht verwendet
RDB$EXTERNAL_LENGTH	SMALLINT	Die Länge der Spalte in Bytes, sofern diese zu einer externen Tabelle gehört. Für reguläre Tabellen immer NULL.
RDB$EXTERNAL_SCALE	SMALLINT	Der Skalierungsfaktor für Integer-Felder in einer externen Tabelle; repräsentiert die Potenz von 10, die mit dem Integer multipliziert wird
RDB$EXTERNAL_TYPE	SMALLINT	Der Datentype des Feldes, wie er in der externen Tabelle vorkommt: 7 - SMALLINT 8 - INTEGER 10 - FLOAT 12 - DATE 13 - TIME 14 - CHAR 16 - BIGINT 23 - BOOLEAN 27 - DOUBLE PRECISION 35 - TIMESTAMP 37 - VARCHAR 261 - BLOB
RDB$DIMENSIONS	SMALLINT	Gibt die Anzahl der Dimensionen in einem Array an, sofern die Spalte als Array definiert wurde, sonst immer NULL.
RDB$NULL_FLAG	SMALLINT	Gibt an, ob die Spalte einen leeren Wert annehmen darf (das Feld enthält dann NULL) oder nicht (das Feld enthält dann den Wert 1
RDB$CHARACTER_LENGTH	SMALLINT	Die Länge für CHAR- oder VARCHAR-Spalten in Zeichen (nicht in Bytes)
RDB$COLLATION_ID	SMALLINT	Die Kennung der Collation-Sequenz für eine Zeichenspalte oder -Domain. Wurde dies nicht definiert ist der Feldwert 0

Spaltenname	Datentyp	Beschreibung
RDB$CHARACTER_SET_ID	SMALLINT	Die Kennung des Zeichensatzes für eine Zeichenspalte, eine BLOB TEXT-Spalte oder -Domain
RDB$FIELD_PRECISION	SMALLINT	Gibt die Gesamtzahl der Stellen für Festkomma-Datentypen (DECIMAL und NUMERIC) an. Der Wert ist 0 für Integer-Datentypen, NULL für alle anderen.
RDB$SECURITY_CLASS	CHAR(31)	Kann auf eine in der Tabelle RDB$SECURITY_CLASSES definierte Sicherheitsklasse verweisen, um Zugriffskontrollbeschränkungen auf alle Benutzer dieser Domäne anzuwenden
RDB$OWNER_NAME	CHAR(31)	Der Benutzername des Benutzers, der die Domäne ursprünglich erstellt hat.

RDB$FIELD_DIMENSIONS

RDB$FIELD_DIMENSIONS speichert die Dimensionen für Array-Spalten.

Spaltenname	Datentyp	Beschreibung
RDB$FIELD_NAME	CHAR(31)	Der Name der Array-Spalte. Dieser muss im Feld RDB$FIELD_NAME innerhalb der Tabelle RDB$FIELDS.
RDB$DIMENSION	SMALLINT	Kennzeichnet eine Dimension in der Array-Spalte. Die Nummerierung der Dimensionen startet bei 0.
RDB$LOWER_BOUND	INTEGER	Die untere Grenze dieser Dimension.
RDB$UPPER_BOUND	INTEGER	Die obere Grenze dieser Dimension.

RDB$FILES

RDB$FILES speichert Informationen über sekundäre Dateien und Shadow-Dateien.

Spaltenname	Datentyp	Beschreibung
RDB$FILE_NAME	VARCHAR(255)	Der vollständige Pfad zur Datei und der Name einer der beiden • der sekundären Datenbankdatei in Multidatei-Datenbanken, oder • der Shadow-Datei

Spaltenname	Datentyp	Beschreibung
RDB$FILE_SEQUENCE	SMALLINT	Die fortlaufende Nummer der sekundären Datei in einer Sequenz oder der Shadow-Datei innerhalb einer Shadow-Dateien-Sammlung.
RDB$FILE_START	INTEGER	Die initiale Seitenzahl in der sekundären Datei oder der Shadow-Datei.
RDB$FILE_LENGTH	INTEGER	Dateilänge in Datenbankseiten.
RDB$FILE_FLAGS	SMALLINT	Für den internen Gebrauch
RDB$SHADOW_NUMBER	SMALLINT	Nummer der Shadow-Sammlung. Wenn die Zeile eine sekundäre Datenbankdatei beschreibt, ist der Feldwert NULL, andernfalls 0.

RDB$FILTERS

RDB$FILTERS speichert Informationen über BLOB-Filter.

Spaltenname	Datentyp	Beschreibung
RDB$FUNCTION_NAME	CHAR(31)	Das eindeutige Kennzeichen für BLOB -Filter
RDB$DESCRIPTION	BLOB TEXT	Dokumentation über die BLOB-Filter und die zwei Untertypen, die dieser nutzt. Geschrieben durch den Benutzer.
RDB$MODULE_NAME	VARCHAR(255)	Der Name der dynamischen Bibliothek oder des Shared Object, in der der Code des BLOB-Filters steht.
RDB$ENTRYPOINT	CHAR(31)	Der exportierte Name des BLOB-Filters in der Filterbibliothek. Beachten Sie, das dies ist üblicherweise nicht das Gleiche wie RDB$FUNCTION_NAME ist. Das ist die Kennung, womit der BLOB-Filter in der Datenbank deklariert wird.
RDB$INPUT_SUB_TYPE	SMALLINT	Der BLOB-Untertyp der Daten, die durch die Funktion konvertiert werden
RDB$OUTPUT_SUB_TYPE	SMALLINT	Der BLOB-Untertyp der konvertierten Daten.

Spaltenname	Datentyp	Beschreibung
RDB$SYSTEM_FLAG	SMALLINT	Dieses Kennzeichen gibt an, ob der Filter ist benutzerdefiniert oder intern definiert: 0 - benutzerdefiniert 1 oder größer - intern definiert
RDB$SECURITY_CLASS	CHAR(31)	Kann auf eine in der Tabelle RDB$SECURITY_CLASSES definierte Sicherheitsklasse verweisen, um Zugriffskontrollbeschränkungen auf alle Benutzer dieses Filters anzuwenden
RDB$OWNER_NAME	CHAR(31)	Der Benutzername des Benutzers, der den Filter ursprünglich erstellt hat

RDB$FORMATS

RDB$FORMATS speichert Informationen über Änderungen in Tabellen. Jedes Mal wenn Änderungen in den Metadaten einer Tabelle durchgeführt werden, bekommt diese eine neue Formatnummer. Wen die Formatnummer irgendeiner Tabelle die 255 erreicht, wird die gesamte Datenbank inoperabel. Um in den normalen Betrieb zu wechseln, müssen Sie zunächst ein Backup der Datenbank mit dem Werkzeug *gbak* und anschließend eine Wiederherstellung durchführen.

Spaltenname	Datentyp	Beschreibung
RDB$RELATION_ID	SMALLINT	Kennung der Tabelle oder View
RDB$FORMAT	SMALLINT	Kennung des Tabellenformats — maximal 255. Der kritische Punkt ist erreicht, wenn die Nummer 255 für eine *beliebige* Tabelle oder View erreicht.
RDB$DESCRIPTOR	BLOB FORMAT	Speichert Spaltennamen und Dateneigenschaften als BLOB, so wie sie zum Zeitpunkt der Erstellung des Format-Datensatzes war.

RDB$FUNCTIONS

RDB$FUNCTIONS speichert Informationen, die von der Engine für externe Funktionen (benutzerdefinierte Funktionen, UDFs) verwendet werden.

Spaltenname	Datentyp	Beschreibung
RDB$FUNCTION_NAME	CHAR(31)	Der eindeutige (deklarierte) Name der externen Funktion.
RDB$FUNCTION_TYPE	SMALLINT	Derzeit nicht verwendet

Spaltenname	Datentyp	Beschreibung
RDB$QUERY_NAME	CHAR(31)	Derzeit nicht verwendet
RDB$DESCRIPTION	BLOB TEXT	Beliebiger Textkommentar zur externen Funktion
RDB$MODULE_NAME	VARCHAR(255)	Der Name der dynamischen Bibliothek oder des Shared Object, die bzw. das den Code der externen Funktion vorhält.
RDB$ENTRYPOINT	CHAR(31)	Der exportierte Name der externen Funktion in der Funktionsbibliothek. Beachten Sie, dass dies üblicherweise nicht der gleiche Name wie in RDB$FUNCTION_NAME ist, welches wiederum die Kennung hält, mit der die externe Funktion in der Datenbank registriert ist.
RDB$RETURN_ARGUMENT	SMALLINT	Die Positionsnummer des zurückgegebenen Argumentes innerhalb der Parameterliste, die sich auf die Eingabeargumente bezieht.
RDB$SYSTEM_FLAG	SMALLINT	Kennzeichen zeigt an, ob der Filter benuntzer- oder intern definiert wurde: 0 - benutzerdefiniert 1 oder größer - intern definiert
RDB$ENGINE_NAME	CHAR(31)	Engine für externe Funktionen 'UDR' für UDR-Funktionen. NULL für ältere UDF- oder PSQL-Funktionen
RDB$PACKAGE_NAME	CHAR(31)	Paket, das diese Funktion enthält (oder NULL)
RDB$PRIVATE_FLAG	SMALLINT	NULL für normale (Haupt-) Funktionen, 0 für im Header definierte Paketfunktion, 1 für nur im Paketrumpf definierte Paketfunktion.
RDB$FUNCTION_SOURCE	BLOB TEXT	Der PSQL-Quellcode der Funktion
RDB$FUNCTION_ID	SMALLINT	Eindeutige Kennung der Funktion
RDB$FUNCTION_BLR	BLOB BLR	Die binäre Sprachdarstellung (BLR) des Funktionscodes (nur PSQL-Funktion)
RDB$VALID_BLR	SMALLINT	Gibt an, ob die Quell-PSQL der gespeicherten Prozedur nach der letzten ALTER FUNCTION-Änderung gültig bleibt

Spaltenname	Datentyp	Beschreibung
RDB$DEBUG_INFO	BLOB DEBUG_INFORMATION	Enthält Debugging-Informationen zu Variablen, die in der Funktion verwendet werden (nur PSQL-Funktion)
RDB$SECURITY_CLASS	CHAR(31)	Kann auf eine in der Tabelle RDB$SECURITY_CLASSES definierte Sicherheitsklasse verweisen, um Zugriffskontrollbeschränkungen auf alle Benutzer dieser Funktion anzuwenden
RDB$OWNER_NAME	CHAR(31)	Der Benutzername des Benutzers, der die Funktion ursprünglich erstellt hat
RDB$LEGACY_FLAG	SMALLINT	Das Legacy-Stilattribut der Funktion. 1 - wenn die Funktion im Legacy-Stil beschrieben ist (DECLARE EXTERNAL FUNCTION), andernfalls CREATE FUNCTION.
RDB$DETERMINISTIC_FLAG	SMALLINT	Deterministische Flagge. 1 - wenn die Funktion deterministisch ist

RDB$FUNCTION_ARGUMENTS

RDB$FUNCTION_ARGUMENTS speichert die Paramter externer Funktionen und ihrer Attribute.

Spaltenname	Datentyp	Beschreibung
RDB$FUNCTION_NAME	CHAR(31)	Der eindeutige Name (deklariertes Kennzeichen) der externen Funktion
RDB$ARGUMENT_POSITION	SMALLINT	Die Position des Arguments innerhalb der Argumentliste.
RDB$MECHANISM	SMALLINT	Kennzeichen: wie wird das Argument übergeben 0 - per Wert (by value) 1 - per Referenz (by reference) 2 - per Beschreibung (by descriptor) 3 - per BLOB-Beschreibung (by BLOB descriptor)

Spaltenname	Datentyp	Beschreibung
RDB$FIELD_TYPE	SMALLINT	Data type code defined for the column: 7 - SMALLINT 8 - INTEGER 12 - DATE 13 - TIME 14 - CHAR 16 - BIGINT 23 - BOOLEAN 27 - DOUBLE PRECISION 35 - TIMESTAMP 37 - VARCHAR 40 - CSTRING (null-terminierter Text) 45 - BLOB_ID 261 - BLOB
RDB$FIELD_SCALE	SMALLINT	Die Skalierung eines Integer- oder Festkomma-Arguments. Dies ist der Exponent von 10.
RDB$FIELD_LENGTH	SMALLINT	Argumentlänge in Bytes: BOOLEAN = 1 SMALLINT = 2 INTEGER = 4 DATE = 4 TIME = 4 BIGINT = 8 DOUBLE PRECISION = 8 TIMESTAMP = 8 BLOB_ID = 8
RDB$FIELD_SUB_TYPE	SMALLINT	Speichert den BLOB-Untertypen für ein Argument des BLOB-Datentyps.
RDB$CHARACTER_SET_ID	SMALLINT	Die Kennung des Zeichensatzes für Zeichenargumente.
RDB$FIELD_PRECISION	SMALLINT	Die Anzahl der Stelle für die Präzision, die für den Datentyp des Arguments verfügbar ist.
RDB$CHARACTER_LENGTH	SMALLINT	Die Länge eines CHAR- oder VARCHAR -Arguments in Zeichen (nicht in Bytes).
RDB$PACKAGE_NAME	CHAR(31)	Paketname der Funktion (oder NULL für eine Top-Level-Funktion)
RDB$ARGUMENT_NAME	CHAR(31)	Parametername

Spaltenname	Datentyp	Beschreibung
RDB$FIELD_SOURCE	CHAR(31)	Der Name der vom Benutzer erstellten Domäne, wenn auf eine Domäne anstelle eines Datentyps verwiesen wird. Beginnt der Name mit dem Präfix “RDB$”, ist dies der Name der vom System automatisch generierten Domäne für den Parameter.
RDB$DEFAULT_VALUE	BLOB BLR	Der Standardwert für den Parameter in der binären Sprachdarstellung (BLR)
RDB$DEFAULT_SOURCE	BLOB TEXT	Der Standardwert für den Parameter im PSQL-Code
RDB$COLLATION_ID	SMALLINT	Der Bezeichner der Kollatierungssequenz, die für einen Zeichenparameter verwendet wird
RDB$NULL_FLAG	SMALLINT	Das Flag, das angibt, ob NULL zulässig ist
RDB$ARGUMENT_MECHANISM	SMALLINT	Parameterübergabemechanismus für Nicht-Legacy-Funktionen: 0 - per Wert (by value) 1 - per Referenz (by reference) 2 - mittels einer Beschreibung (by descriptor) 3 - per BLOB-Beschreibung (by BLOB descriptor)
RDB$FIELD_NAME	CHAR(31)	Der Name der Spalte, auf die der Parameter verweist, wenn er mit "TYPE OF COLUMN" anstelle eines regulären Datentyps deklariert wurde. Wird in Verbindung mit RDB$RELATION_NAME verwendet (siehe nächstes).
RDB$RELATION_NAME	CHAR(31)	Der Name der Tabelle, auf die der Parameter verweist, wenn er mit "TYPE OF COLUMN" anstelle eines regulären Datentyps deklariert wurde
RDB$SYSTEM_FLAG	SMALLINT	Kennzeichen: 0 - benutzerdefiniert 1 oder höher - systemdefiniert
RDB$DESCRIPTION	BLOB TEXT	Optionale Beschreibung des Funktionsarguments (Kommentar)

RDB$GENERATORS

RDB$GENERATORS speichert Generatoren (Sequenzen) und hält diese aktuell.

Spaltenname	Datentyp	Beschreibung
RDB$GENERATOR_NAME	CHAR(31)	Der eindeutige Generatorname.
RDB$GENERATOR_ID	SMALLINT	Die eindeutige Kennung, die für den Generator durch das System vergeben wurde.
RDB$SYSTEM_FLAG	SMALLINT	Kennzeichen: 0 - benutzerdefiniert 1 oder größer - intern definiert 6 - interner Generator für Identitätsspalte
RDB$DESCRIPTION	BLOB TEXT	Kann Kommentartexte zum Generator speichern.
RDB$SECURITY_CLASS	CHAR(31)	Kann auf eine in der Tabelle RDB$SECURITY_CLASSES definierte Sicherheitsklasse verweisen, um Zugriffskontrollbeschränkungen auf alle Benutzer dieses Generators anzuwenden
RDB$OWNER_NAME	CHAR(31)	Der Benutzername des Benutzers, der den Generator ursprünglich erstellt hat
RDB$INITIAL_VALUE	BIGINT	Speichert den Anfangswert (START WITH Wert) des Generators
RDB$GENERATOR_INCREMENT	INTEGER	Speichert das Inkrement des Wertes (INCREMENT BY Wert) des Generators

RDB$INDICES

RDB$INDICES speichert die Definitionen benutzerdefinierter und systemdefinierter Indizes. Die Eigenschaften jeder Spalte, die zu einem Index gehören, werden in je einer Spalte innerhalb der Tabelle RDB$INDEX_SEGMENTS vorgehalten.

Spaltenname	Datentyp	Beschreibung
RDB$INDEX_NAME	CHAR(31)	Der eindeutige Indexname, der durch den Benutzer oder automatisch durch das System vergeben wurde.
RDB$RELATION_NAME	CHAR(31)	Der Name der Tabelle zu der der Index gehört. Dieser korrespondiert mit der Kennung in RDB$RELATION_NAME.RDB$RELATIONS

Spaltenname	Datentyp	Beschreibung
RDB$INDEX_ID	SMALLINT	Die interne (System-)Kennung des Index.
RDB$UNIQUE_FLAG	SMALLINT	Gibt an, ob der Index eindeutig ist: 1 - eindeutig (unique) 0 - nicht eindeutig (not unique)
RDB$DESCRIPTION	BLOB TEXT	Kann Kommentare zum Index speichern.
RDB$SEGMENT_COUNT	SMALLINT	Die Anzahl der Segment (Spalten) des Index.
RDB$INDEX_INACTIVE	SMALLINT	Gibt an, ob der Index derzeit aktiv ist: 1 - inaktiv 0 - aktiv
RDB$INDEX_TYPE	SMALLINT	Unterscheidet zwischen aufsteigendem (0 oder NULL) und absteigendem Index (1). Wird nicht in Datenbanken vor Firebird 2.0 verwendet; reguläre Indizes in aktualisierten (upgraded) Datenbanken werden üblicherweise NULL in dieser Spalte speichern.
RDB$FOREIGN_KEY	CHAR(31)	Der Name des zugewiesenen Fremdschlüssel-Constraints, falls vorhanden.
RDB$SYSTEM_FLAG	SMALLINT	Gibt an, ob der Index system- oder benutzerdefiniert ist: 0 - benutzerdefiniert 1 oder größer - intern definiert
RDB$EXPRESSION_BLR	BLOB BLR	Ausdruck für einen Anweisungsindex, geschrieben in Binärsprachenrepräsentation (BLR). Wird für die Berechnung der Indexwerte zur Laufzeit verwendet.
RDB$EXPRESSION_SOURCE	BLOB TEXT	Der Quellcode des Ausdrucks für einen Anweisungsindex.

Spaltenname	Datentyp	Beschreibung
RDB$STATISTICS	DOUBLE PRECISION	Speichert die letzte bekannte Selektivität des gesamten Index, die durch die Ausführung eines SET STATISTICS-Statements berechnet wird. Diese wird außerdem beim ersten Öffnen der Datenbank durch den Server neuberechnet. Die Selektivität jedes einzelnen Index-Segments wird in der Tabelle RDB$INDEX_SEGMENTS gespeichert.

RDB$INDEX_SEGMENTS

RDB$INDEX_SEGMENTS speichert die Segmente (Tabellenspalten) eines Index und ihre Position innerhalb des Schlüssels. Pro Spalte innerhalb des Index wird eine einzelne Zeile vorgehalten.

Spaltenname	Datentyp	Beschreibung
RDB$INDEX_NAME	CHAR(31)	Der Name des Index, dem dieses Seqment zugewiesen ist. Der Hauptdatensatz befindet sich in RDB$INDICES.RDB$INDEX_NAME.
RDB$FIELD_NAME	CHAR(31)	Der Name der Spalte, die zum Index gehört, korrespondierend zur Kennung für die Tabelle und dessen Spalte in RDB$RELATION_FIELDS.RDB$FIELD_NAME.
RDB$FIELD_POSITION	SMALLINT	Die Spaltenposition im Index. Die Positionen werden von links nach rechts festgelegt und starten bei 0.
RDB$STATISTICS	DOUBLE PRECISION	Die letzte bekannte (berechnete) Selektivität dieses Spaltenindex. Je größer die Zahl ist, desto kleiner die Selektivität.

RDB$LOG_FILES

RDB$LOG_FILES wird derzeit nicht verwendet.

RDB$PACKAGES

RDB$PACKAGES speichert die Definition (Header und Body) von SQL-Paketen.

Spaltenname	Datentyp	Beschreibung
RDB$PACKAGE_NAME	CHAR(31)	Name des Pakets
RDB$PACKAGE_HEADER_SOURCE	BLOB TEXT	Der PSQL-Quellcode des Paket-Headers

Spaltenname	Datentyp	Beschreibung
RDB$PACKAGE_BODY_SOURCE	BLOB TEXT	Der PSQL-Quellcode des Paketkörpers
RDB$VALID_BODY_FLAG	SMALLINT	Gibt an, ob der Hauptteil des Pakets noch gültig ist. NULL oder 0 zeigt an, dass der Body nicht gültig ist.
RDB$SECURITY_CLASS	CHAR(31)	Kann auf eine in der Tabelle RDB$SECURITY_CLASSES definierte Sicherheitsklasse verweisen, um Zugriffskontrollbeschränkungen auf alle Benutzer dieses Pakets anzuwenden
RDB$OWNER_NAME	CHAR(31)	Der Benutzername des Benutzers, der das Paket ursprünglich erstellt hat
RDB$SYSTEM_FLAG	SMALLINT	Flagge: 0 - benutzerdefiniert 1 oder höher - systemdefiniert
RDB$DESCRIPTION	BLOB TEXT	Optionale Beschreibung des Pakets (Kommentar)

RDB$PAGES

RDB$PAGES speichert Informationen über die Datenbankseiten und deren Nutzung.

Spaltenname	Datentyp	Beschreibung
RDB$PAGE_NUMBER	INTEGER	Die eindeutige Nummer der physikalisch erstellen Datenbankseiten.
RDB$RELATION_ID	SMALLINT	Die Kennung der Tabelle, zu der die Seite gehört.
RDB$PAGE_SEQUENCE	INTEGER	Die Nummer der Seite innerhalb der Sequenz aller Seiten in der zugehörigen Tabelle.
RDB$PAGE_TYPE	SMALLINT	Gibt den Seitentyp an (Daten, Index, BLOB, etc.). Informationen für das System.

RDB$PROCEDURES

RDB$PROCEDURES speichert die Definitionen für Stored Procedures, inklusive ihres PSQL-Quelltextes und ihrer Binärsprachenrepräsentation (BLR). Die nächste Tabelle RDB$PROCEDURE_PARAMETERS speichert die Definitionen der Eingabe- und Ausgabeparameter.

Spaltenname	Datentyp	Beschreibung
RDB$PROCEDURE_NAME	CHAR(31)	Name (Kennung) der Stored Procedure.

Spaltenname	Datentyp	Beschreibung
RDB$PROCEDURE_ID	SMALLINT	Die eindeutige system-generierte Kennung.
RDB$PROCEDURE_INPUTS	SMALLINT	Gibt die Anzahl der Eingabeparameter an. NULL wenn es keine gibt.
RDB$PROCEDURE_OUTPUTS	SMALLINT	Gibt die Anzahl der Ausgabeparameter an. NULL wenn es keine gibt.
RDB$DESCRIPTION	BLOB TEXT	Beliebiger Kommentartext, der die Prozedur beschreibt.
RDB$PROCEDURE_SOURCE	BLOB TEXT	Der PSQL-Quelltext der Prozedur.
RDB$PROCEDURE_BLR	BLOB BLR	Die Binärsprachenrepräsentation (BLR) des Prozedurcodes.
RDB$SECURITY_CLASS	CHAR(31)	Kann die definierte Sicherheitsklasse aus der Systemtabelle RDB$SECURITY_CLASSES aufnehmen, um Zugriffsbeschränkungen zu verwenden.
RDB$OWNER_NAME	CHAR(31)	Der Benutzername des Prozedurbesitzers — der Benutzer, der CURRENT_USER war, als die Prozedur erstellt wurde. Dies kann, muss aber nicht, der Benutzername des Autors sein.
RDB$RUNTIME	BLOB	Eine Metadatenbeschreibung der Prozedur, die intern für die Optimierung verwendet wird.
RDB$SYSTEM_FLAG	SMALLINT	Gibt an, ob die Prozedur durch einen Benutzer (Wert 0) oder durch das System (Wert 1 oder größer) erstellt wurde.
RDB$PROCEDURE_TYPE	SMALLINT	Prozedurtyp: 1 - selektierbare Stored Procedure (beinhaltet ein SUSPEND-Statement) 2 - ausführbare Stored Procedure NULL - unbekannt * * gilt für Prozeduren, die vor Firebird 1.5 erstellt wurden.
RDB$VALID_BLR	SMALLINT	Gibt an, ob der PSQL-Quelltext der Stored Procedure nach der letzten Anpassung mittels ALTER PROCEDURE gültig bleibt.

Spaltenname	Datentyp	Beschreibung
RDB$DEBUG_INFO	BLOB	Beinhaltet Debugging-Informationen über Variablen, die in der Stored Procedure Verwendung finden.
RDB$ENGINE_NAME	CHAR(31)	Engine für externe Funktionen. UDR für UDR-Verfahren. NULL für gespeicherte PSQL-Prozeduren
RDB$ENTRYPOINT	CHAR(255)	Der exportierte Name der externen Funktion in der Prozedurbibliothek. Beachten Sie, dass dies häufig nicht mit RDB$PROCEDURE_NAME identisch ist. Dies ist die Kennung, mit der die externe gespeicherte Prozedur in der Datenbank deklariert wird
RDB$PACKAGE_NAME	CHAR(31)	Paketname der Prozedur (oder NULL für eine gespeicherte Prozedur der obersten Ebene)
RDB$PRIVATE_FLAG	SMALLINT	NULL für normale (oberste) gespeicherte Prozeduren, 0 für im Header definierte Paketprozeduren, 1 für nur im Paketrumpf definierte Paketprozeduren.

RDB$PROCEDURE_PARAMETERS

RDB$PROCEDURE_PARAMETERS speichert die Parameter einer Stored Procedure und ihrer Eigenschaften. Je Parameter wird eine eigene Zeile vorgehalten.

Spaltenname	Datentyp	Beschreibung
RDB$PARAMETER_NAME	CHAR(31)	Parametername
RDB$PROCEDURE_NAME	CHAR(31)	Der Name der Prozedur, für die der Parameter definiert wurde.
RDB$PARAMETER_NUMBER	SMALLINT	Die Folgenummer des Paramters.
RDB$PARAMETER_TYPE	SMALLINT	Gibt an, ob dies ein Eingabe- (Wert 0) oder Ausgabeparameter (Wert 1) ist.
RDB$FIELD_SOURCE	CHAR(31)	Der Name der benutzerdefinierten Domain, wenn eine Domain anstelle eine Datentyps referenziert wurde. Beginnt der Name mit dem Präfix “RDB$”, wurde die Domain automatisch durch das Syste erstellt.
RDB$DESCRIPTION	BLOB TEXT	Kann Kommentartexte zum Parameter speichern.

Spaltenname	Datentyp	Beschreibung
RDB$SYSTEM_FLAG	SMALLINT	Gibt an, ob der Parameter durch das System (Wert 1 oder größer) oder durch den Benutzer definiert wurde (Wert 0)
RDB$DEFAULT_VALUE	BLOB BLR	Der Vorgabewert des Parameters in Binärsprachenrepräsentation (BLR).
RDB$DEFAULT_SOURCE	BLOB TEXT	Der Vorgabewert des Parameters als PSQL-Code.
RDB$COLLATION_ID	SMALLINT	Die Kennung der Collation-Sequenz, die für Zeichenparameter verwendet wird.
RDB$NULL_FLAG	SMALLINT	Gibt an, ob NULL erlaubt ist.
RDB$PARAMETER_MECHANISM	SMALLINT	Kennzeichen: gibt an wie der Parameter übergeben wird: 0 - per Wert (by value) 1 - per Referenz (by reference) 2 - per Beschreibung (by descriptor) 3 - per BLOB-Beschreibung (by BLOB descriptor)
RDB$FIELD_NAME	CHAR(31)	Der Name der Spalte, auf die der Parameter verweist, wenn er mit TYPE OF COLUMN anstelle eines regulären Datentyps deklariert wurde. Wird in Verbindung mit RDB$RELATION_NAME verwendet (siehe unten).
RDB$RELATION_NAME	CHAR(31)	Der Name der Tabelle, auf die der Parameter verweist, wenn er mit TYPE OF COLUMN anstelle eines regulären Datentyps deklariert wurde.
RDB$PACKAGE_NAME	CHAR(31)	Paketname der Prozedur (oder NULL für eine gespeicherte Prozedur der obersten Ebene)

RDB$REF_CONSTRAINTS

RDB$REF_CONSTRAINTS speichert die Eigenschaften für referentielle Constraints — Fremdschlüsselbeziehungen und referentielle Aktionen.

Spaltenname	Datentyp	Beschreibung
RDB$CONSTRAINT_NAME	CHAR(31)	Name des Fremdschlüssels, definiert durch den Benutzer oder automatisch durch das System.

Spaltenname	Datentyp	Beschreibung
RDB$CONST_NAME_UQ	CHAR(31)	Der Name der primären oder eindeutigen Schlüsselbedingung, die durch die REFERENCES-Klausel in der Constraint-Definition verknüpft ist.
RDB$MATCH_OPTION	CHAR(7)	Wird nicht verwendet. Der Wert ist in allen Fällen FULL.
RDB$UPDATE_RULE	CHAR(11)	Aktionen für die referentielle Integrität, die auf Fremschlüsseldatensätze angewendet wird, sobald der Primärschlüssel der Elterntabelle aktualisiert wird: RESTRICT, NO ACTION, CASCADE, SET NULL, SET DEFAULT
RDB$DELETE_RULE	CHAR(11)	Aktionen für die referentielle Integrität, die auf Fremschlüsseldatensätze angewendet wird, sobald der Primärschlüssel der Elterntabelle gelöscht wird: RESTRICT, NO ACTION, CASCADE, SET NULL, SET DEFAULT

RDB$RELATIONS

RDB$RELATIONS speichert die Top-Level-Definitionen und -Eigenschaften aller Tabellen und Views im System.

Spaltenname	Datentyp	Beschreibung
RDB$VIEW_BLR	BLOB BLR	Speichert die Abfragespezifikation einer View in Binärsprachenrepräsentation (BLR). Das Feld speichert NULL für Tabellen.
RDB$VIEW_SOURCE	BLOB TEXT	Beinhaltet den Originalquelltext der Abfrage für eine View, in SQL-Sprache. Benutzerkommentare sind inkludiert. Das Feld speichert NULL für Tabellen.
RDB$DESCRIPTION	BLOB TEXT	Speichert Kommentare für die Tabelle oder View.
RDB$RELATION_ID	SMALLINT	Interne Kennung der Tabelle oder View.
RDB$SYSTEM_FLAG	SMALLINT	Gibt an ob die Tabelle oder View benutzer- (Wert 0) oder systemdefiniert (Wert 1 oder größer) ist.

Spaltenname	Datentyp	Beschreibung
RDB$DBKEY_LENGTH	SMALLINT	Die Gesamtlänge des Datenbankschlüssels. Für eine Tabelle: 8 Bytes. Für eine View: die Anzahl aller beinhalteten Tabellen mit 8 multipliziert.
RDB$FORMAT	SMALLINT	Interne Verwendung, zeigt auf den verknüpften Datensatz in RDB$FORMATS — nicht anpassen.
RDB$FIELD_ID	SMALLINT	Die Feld-ID für die nächste anzufügende Spalte. Die Zahl wird nicht dekrementiert, wenn eine Spalte gelöscht wird.
RDB$RELATION_NAME	CHAR(31)	Name der Tabelle oder View.
RDB$SECURITY_CLASS	CHAR(31)	Kann eine Referenz zur Sicherheitsklasse aufnehmen, die in der Tabelle RDB$SECURITY_CLASSES definiert wurde. Damit lassen sich Zugriffsbeschränkungen für alle Benutzer dieser Tabelle oder View umsetzen.
RDB$EXTERNAL_FILE	VARCHAR(255)	Der vollständige Pfad der externen Datendatei, sofern die Tabelle mit der EXTERNAL FILE-Klausel definiert wurde.
RDB$RUNTIME	BLOB	Beschreibung der Tabellenmetadaten, intern für Optimierungen verwendet.
RDB$EXTERNAL_DESCRIPTION	BLOB	Kann Kommentare für die externe Datei einer externen Tabelle speichern.
RDB$OWNER_NAME	CHAR(31)	Der Benutzername des Benutzers, der die Tabelle oder View erstellt hat.
RDB$DEFAULT_CLASS	CHAR(31)	Standard-Sicherheitsklasse. Wird verwendet, wenn eine neue Spalte zur Tabelle hinzugefügt wurde.
RDB$FLAGS	SMALLINT	Internes Kennzeichen.
RDB$RELATION_TYPE	SMALLINT	Der Typ des Relationsobjekts: 0 - system- oder benutzerdefinierte Tabelle 1 - View 2 - Externe Tabelle 3 - Monitoring-Tabelle 4 - Verbindungslevel GTT (PRESERVE ROWS) 5 - Transaktionslevel GTT (DELETE ROWS)

RDB$RELATION_CONSTRAINTS

RDB$RELATION_CONSTRAINTS speichert die Definitionen aller Tabellen-Level Constraints: Primärschlüssel, UNIQUE, Fremdschlüssel, CHECK, NOT NULL.

Spaltenname	Datentyp	Beschreibung
RDB$CONSTRAINT_NAME	CHAR(31)	Der Name des Tabellen-Level Constraints. Definiert durch den Benutzer, oder automatisch durch das System erstellt.
RDB$CONSTRAINT_TYPE	CHAR(11)	Der Name des Constraint-Typs: PRIMARY KEY, UNIQUE, FOREIGN KEY, CHECK oder NOT NULL
RDB$RELATION_NAME	CHAR(31)	Der Name der Tabelle zu der der Constraint gehört.
RDB$DEFERRABLE	CHAR(3)	Derzeit in allen Fällen NO: Firebird unterstützt derzeit keine verögerten (deferrable) Constraints.
RDB$INITIALLY_DEFERRED	CHAR(3)	Derzeit in allen Fällen NO.
RDB$INDEX_NAME	CHAR(31)	Der Name des Index, der diesen Constraint unterstützt. Für einen CHECK- oder NOT NULL-Constraint ist der Wert NULL.

RDB$RELATION_FIELDS

RDB$RELATION_FIELDS speichert die Definitionen der Tabellen- und View-Spalten.

Spaltenname	Datentyp	Beschreibung
RDB$FIELD_NAME	CHAR(31)	Spaltenname
RDB$RELATION_NAME	CHAR(31)	Der Name der Tabelle oder View zu der die Spalte gehört.
RDB$FIELD_SOURCE	CHAR(31)	Name der Domain auf der die Spalte basiert, entweder benutzerdefiniert über die Tabellendefinition oder automatisch über das System erstellt, anhand der definierten Eigenschaften. Die Eigenschaften stehen in der Tabelle RDB$FIELDS: diese Spalte verweist auf RDB$FIELDS.RDB$FIELD_NAME.
RDB$QUERY_NAME	CHAR(31)	Derzeit nicht verwendet
RDB$BASE_FIELD	CHAR(31)	Nur bei Views gefüllt. Beinhaltet den Namen der Spalte aus der Basistabelle.

Spaltenname	Datentyp	Beschreibung
RDB$EDIT_STRING	VARCHAR(127)	Nicht verwendet.
RDB$FIELD_POSITION	SMALLINT	Die null-basierte Position der Spalten in der Tabelle oder View, Aufzählung von links nach rechts.
RDB$QUERY_HEADER	BLOB TEXT	Nicht verwendet.
RDB$UPDATE_FLAG	SMALLINT	Gibt an ober dies eine reguläre (Wert 1) oder berechnete (Wert 0) Spalte ist.
RDB$FIELD_ID	SMALLINT	Eine ID zugewiesen durch RDB$RELATIONS.RDB$FIELD_ID zum Zeitpunkt als die Spalte zur View oder Tabelle hinzugefügt wurde. Sollte immer als vergänglich angesehen werden.
RDB$VIEW_CONTEXT	SMALLINT	Für eine View-Spalte ist dies die interne Kennung der Basistabelle aus der das Feld stammt.
RDB$DESCRIPTION	BLOB TEXT	Kommentare zur Tabellen- oder View-Spalte.
RDB$DEFAULT_VALUE	BLOB BLR	Der Wert, der für die DEFAULT-Klausel der Spalte verwendet wurde, sofern einer vorhanden ist, gespeichert als Binärsprachenrepräsentation (BLR).
RDB$SYSTEM_FLAG	SMALLINT	Gibt an, ob die Spalte benutzer: (Wert 0) oder systemdefiniert (Wert 1 oder größer) ist.
RDB$SECURITY_CLASS	CHAR(31)	Kann auf eine in RDB$SECURITY_CLASSES definierte Sicherheitsklasse verweisen, um Zugriffsbeschränkungen für alle Benutzer dieser Spalte anzuwenden.
RDB$COMPLEX_NAME	CHAR(31)	Nicht verwendet.
RDB$NULL_FLAG	SMALLINT	Gibt an ob die Spalte null zulässt (NULL) oder nicht (Wert 1)
RDB$DEFAULT_SOURCE	BLOB TEXT	Der Quelltext einer DEFAULT-Klausel, wenn vorhanden.
RDB$COLLATION_ID	SMALLINT	Die Kennung der Collation-Sequenz des Zeichensatzen für die Spalte, sofern dies nicht die Vorgabe-Collation ist.
RDB$GENERATOR_NAME	CHAR(31)	Interner Generatorname zum Generieren eines Identitätswerts für die Spalte.

Spaltenname	Datentyp	Beschreibung
RDB$IDENTITY_TYPE	SMALLINT	Der Identitätstyp der Spalte NULL - keine Identitätsspalte 0 - Identitätsspalte, GENERATED ALWAYS (nicht unterstützt in Firebird 3.0, wird in Firebird 4.0 eingeführt) 1 - Identitätsspalte, GENERATED BY DEFAULT

RDB$ROLES

RDB$ROLES speichert die Rollen, die in der Datenbank definiert wurden.

Spaltenname	Datentyp	Beschreibung
RDB$ROLE_NAME	CHAR(31)	Rollenname
RDB$OWNER_NAME	CHAR(31)	Der Benutzername des Rolleneigentümers.
RDB$DESCRIPTION	BLOB TEXT	Speichert Kommentare zur Rolle.
RDB$SYSTEM_FLAG	SMALLINT	Systemkennzeichen.
RDB$SECURITY_CLASS	CHAR(31)	Kann auf eine Sicherheitsklasse verweisen, die in der Tabelle "RDB$SECURITY_CLASSES" definiert ist, um Zugriffssteuerungsbeschränkungen auf alle Benutzer dieser Rolle anzuwenden

RDB$SECURITY_CLASSES

RDB$SECURITY_CLASSES speichert die Zugriffskontrolllisten.

Spaltenname	Datentyp	Beschreibung
RDB$SECURITY_CLASS	CHAR(31)	Name der Sicherheitsklasse.
RDB$ACL	BLOB ACL	Die Zugriffsliste, die sich auf die Sicherheitsklasse bezieht. Listet Benutzer und ihre Berechtigungen auf.
RDB$DESCRIPTION	BLOB TEXT	Speichert Kommentare zur Sicherheitsklasse.

RDB$TRANSACTIONS

RDB$TRANSACTIONS stores the states of distributed transactions and other transactions that were prepared for two-phase commit with an explicit prepare message.

Spaltenname	Datentyp	Beschreibung
RDB$TRANSACTION_ID	INTEGER	Die eindeutige Kennung der verfolgten Transaktion.
RDB$TRANSACTION_STATE	SMALLINT	Transaktionsstatus: 0 - in limbo 1 - committed 2 - rolled back
RDB$TIMESTAMP	TIMESTAMP	Nicht verwendet.
RDB$TRANSACTION_DESCRIPTION	BLOB	Beschreibt die vorbereitete Transaktion und kann eine benutzerdefinierte Meldung sein, die an isc_prepare_transaction2 übergeben wurde, auch wenn diese keine verteilte Transaktion ist. Diese kann Verwendung finden, wenn eine verlorene Verbindung nicht wiederhergestellt werden kann.

RDB$TRIGGERS

RDB$TRIGGERS speichert Triggerdefinitionen für alle Tabellen und View.

Spaltenname	Datentyp	Beschreibung
RDB$TRIGGER_NAME	CHAR(31)	Triggernname
RDB$RELATION_NAME	CHAR(31)	Der Name der Tabelle oder View zu der der Trigger gehört. NULL wenn der Trigger auf ein Datenbankereignis angewandt wird (“database trigger”)
RDB$TRIGGER_SEQUENCE	SMALLINT	Position dieses Triggers in der Sequenz. Null bedeutet normalerweise, dass keine Sequenzposition angegeben wurde.
RDB$TRIGGER_TYPE	BIGINT	Der Ereignistyp, bei dem der Trigger ausgelöst wird, siehe RDB$TRIGGER_TYPE Wert
RDB$TRIGGER_SOURCE	BLOB TEXT	Speichert den Quellcode des Triggers in PSQL.
RDB$TRIGGER_BLR	BLOB BLR	Speichert den Quellcode des Triggers in Binärsprachenrepräsentation (BLR).
RDB$DESCRIPTION	BLOB TEXT	Kommentartext zum Trigger.
RDB$TRIGGER_INACTIVE	SMALLINT	Gibt an, ob der Trigger derzeit inaktiv (1) oder aktiv (0) ist.

Spaltenname	Datentyp	Beschreibung
RDB$SYSTEM_FLAG	SMALLINT	Kennzeichen: Gibt an, ob der Trigger benutzer- (Wert 0) oder systemdefiniert (Wert 1 oder größer) ist.
RDB$FLAGS	SMALLINT	Interne Verwendung
RDB$VALID_BLR	SMALLINT	Gibt an, ob der Text des Triggers nach der letzten Änderung mittels ALTER TRIGGER gültig bleibt.
RDB$DEBUG_INFO	BLOB	Beinhaltet Debugging-Informationen über die im Trigger genutzten Variablen.
RDB$ENGINE_NAME	CHAR(31)	Engine für externe Trigger. UDR für UDR-Trigger. NULL für PSQL-Trigger
RDB$ENTRYPOINT	CHAR(255)	Der exportierte Name des externen Triggers in der Triggerbibliothek. Beachten Sie, dass dies oft nicht dasselbe ist wie RDB$TRIGGER_NAME, was der Bezeichner ist, mit dem der Trigger in der Datenbank deklariert wird

RDB$TRIGGER_TYPE Wert

Der Wert von RDB$TRIGGER_TYPE wird gebildet aus:

1 before insert

2 after insert

3 before update

4 after update

5 before delete

6 after delete

17 before insert or update

18 after insert or update

25 before insert or delete

26 after insert or delete

27 before update or delete

28 after update or delete

113	before insert or update or delete
114	after insert or update or delete
8192	on connect
8193	on disconnect
8194	on transaction start
8195	on transaction commit
8196	on transaction rollback

Die Identifizierung des genauen RDB$TRIGGER_TYPE-Codes ist etwas komplizierter, da es sich um eine Bitmap handelt, die anhand der abgedeckten Phase und Ereignisse sowie der Reihenfolge ihrer Definition berechnet wird. Für Neugierige wird die Berechnung in https://tinyurl.com/fb-triggertype erklärt[Codekommentar von Mark Rotteveel]

Bei DDL-Triggern wird der Triggertyp durch bitweises ODER über der Ereignisphase (0 — BEFORE, 1 — AFTER) und allen aufgelisteten Ereignistypen ermittelt:

0x0000000000004002	CREATE TABLE
0x0000000000004004	ALTER TABLE
0x0000000000004008	DROP TABLE
0x0000000000004010	CREATE PROCEDURE
0x0000000000004020	ALTER PROCEDURE
0x0000000000004040	DROP PROCEDURE
0x0000000000004080	CREATE FUNCTION
0x0000000000004100	ALTER FUNCTION
0x0000000000004200	DROP FUNCTION
0x0000000000004400	CREATE TRIGGER
0x0000000000004800	ALTER TRIGGER
0x0000000000005000	DROP TRIGGER
0x0000000000014000	CREATE EXCEPTION
0x0000000000024000	ALTER EXCEPTION
0x0000000000044000	DROP EXCEPTION
0x0000000000084000	CREATE VIEW
0x0000000000104000	ALTER VIEW

0x0000000000204000	DROP VIEW
0x0000000000404000	CREATE DOMAIN
0x0000000000804000	ALTER DOMAIN
0x0000000001004000	DROP DOMAIN
0x0000000002004000	CREATE ROLE
0x0000000004004000	ALTER ROLE
0x0000000008004000	DROP ROLE
0x0000000010004000	CREATE INDEX
0x0000000020004000	ALTER INDEX
0x0000000040004000	DROP INDEX
0x0000000080004000	CREATE SEQUENCE
0x0000000100004000	ALTER SEQUENCE
0x0000000200004000	DROP SEQUENCE
0x0000000400004000	CREATE USER
0x0000000800004000	ALTER USER
0x0000001000004000	DROP USER
0x0000002000004000	CREATE COLLATION
0x0000004000004000	DROP COLLATION
0x0000008000004000	ALTER CHARACTER SET
0x0000010000004000	CREATE PACKAGE
0x0000020000004000	ALTER PACKAGE
0x0000040000004000	DROP PACKAGE
0x0000080000004000	CREATE PACKAGE BODY
0x0000100000004000	DROP PACKAGE BODY
0x0000200000004000	CREATE MAPPING
0x0000400000004000	ALTER MAPPING
0x0000800000004000	DROP MAPPING
0x7FFFFFFFFFFDFFE	ANY DDL STATEMENT

Zum Beispiel ein Trigger mit
`BEFORE CREATE PROCEDURE OR CREATE FUNCTION` ist vom Typ `0x0000000000004090`,
`AFTER CREATE PROCEDURE OR CREATE FUNCTION` — `0x0000000000004091`,
`BEFORE DROP FUNCTION OR DROP EXCEPTION` — `0x0000000000044200`,
`AFTER DROP FUNCTION OR DROP EXCEPTION` — `0x0000000000044201`,

BEFORE DROP TRIGGER OR DROP DOMAIN — 0x0000000000001005000,
AFTER DROP TRIGGER OR DROP DOMAIN — 0x0000000000001005001.

RDB$TRIGGER_MESSAGES

RDB$TRIGGER_MESSAGES speichert die Triggermeldungen.

Spaltenname	Datentyp	Beschreibung
RDB$TRIGGER_NAME	CHAR(31)	Der Name des Triggers, zu dem die Meldung gehört
RDB$MESSAGE_NUMBER	SMALLINT	Die Nummer der Meldung innerhalb des Triggers (von 1 bis 32.767)
RDB$MESSAGE	VARCHAR(1023)	Text der Triggermeldung

RDB$TYPES

RDB$TYPES speichert die definierten Listen enumerierter Typen, die im gesamten System verwendet werden.

Spaltenname	Datentyp	Beschreibung
RDB$FIELD_NAME	CHAR(31)	Enumerierter Typname. Jeder Typname beinhaltet seinen eigenen Typensatz, z.B. Objekttypen, Datentypen, Zeichensätze, Triggertypen, BLOB -Untertypen, etc.
RDB$TYPE	SMALLINT	The object type identifier. A unique series of numbers is used within each separate enumerated type. For example, in this selection from the set mastered under RDB$OBJECT_TYPE in RDB$FIELD_NAME, some object types are enumerated: 0 - TABLE 1 - VIEW 2 - TRIGGER 3 - COMPUTED_FIELD 4 - VALIDATION 5 - PROCEDURE ...

Spaltenname	Datentyp	Beschreibung
RDB$TYPE_NAME	CHAR(31)	Der Name eines Elements eines Aufzählungstyps, z. B. TABLE, VIEW, TRIGGER usw. im obigen Beispiel. Im Aufzählungstyp RDB$CHARACTER_SET, speichert RDB$TYPE_NAME die Namen der Zeichensätze.
RDB$DESCRIPTION	BLOB TEXT	Beliebige Kommentartexte zu den Aufzählungstypen.
RDB$SYSTEM_FLAG	SMALLINT	Kennzeichen: gibt an, ob das Typ-Element benutzer- (Wert 0) oder systemdefiniert (Wert 1 oder größer) ist.

RDB$USER_PRIVILEGES

RDB$USER_PRIVILEGES speichert die SQL-Zugriffsprivilegien der Firebird-Benutzer und Privilegobjekte.

Spaltenname	Datentyp	Beschreibung
RDB$USER	CHAR(31)	Der Benutzer oder das Objekt, dem bzw. der diese Berechtigung erteilt wird.
RDB$GRANTOR	CHAR(31)	Der Benutzer, der die Berechtigung erteilt.
RDB$PRIVILEGE	CHAR(6)	Das hier gewährte Privileg: A - alle (alle Privilegien) S - select (Abfrage von Daten) I - insert (Datensätze einfügen) D - delete (Datensätze löschen) R - references (Fremdschlüssel) U - update (Datensätze aktualisieren) X - executing (Prozeduren) G - usage (anderer Objekttypen) M - role membership (Rollenmitgliedschaft) C - DDL privilege create (Erstellberechtigung für DDL)+ L - DDL privilege alter (Aktualisierungs- und Änderungsberechtigung für DDL) O - DDL privilege drop (Löschberechtigung für DDL)

Spaltenname	Datentyp	Beschreibung
RDB$GRANT_OPTION	SMALLINT	Gibt an, ob die Berechtigung WITH GRANT OPTION im Privileg enthalten ist: 1 - enthalten 0 - nicht enthalten
RDB$RELATION_NAME	CHAR(31)	Der Objektname (Tabelle, View, Prozedur oder Rolle) dem das Privileg zugewiesen wurde (ON).
RDB$FIELD_NAME	CHAR(31)	Der Name der Spalte zu dem das Privileg gehört, für Spaltenbasierte Berechtigungen (ein UPDATE- oder REFERENCES-Privileg).
RDB$USER_TYPE	SMALLINT	Gibt den Typ des Benutzers (ein Benutzer, eine Prozedur, eine View, etc.) an, dem das Privileg zugewiesen wurde (TO).
RDB$OBJECT_TYPE	SMALLINT	Gibt den Typ des Objekts an, dem das Privileg zugewiesen wurde (ON). 0 - Tabelle 1 - View 2 - Trigger 5 - Prozedur 7 - Exception 8 - Benutzer 9 - Domain 11 - Zeichensatz 13 - Rolle 14 - Generator (Sequenz) 15 - Function 16 - BLOB-Filter 17 - Collation 18 - Paket

RDB$VIEW_RELATIONS

RDB$VIEW_RELATIONS speichert die Tabellen, die in der View-Definition referenziert werden. Pro Tabelle wird ein Datensatz verwendet.

Spaltenname	Datentyp	Beschreibung
RDB$VIEW_NAME	CHAR(31)	Viewname

Spaltenname	Datentyp	Beschreibung
RDB$RELATION_NAME	CHAR(31)	Der Name der Tabelle, die in der View referenziert wird.
RDB$VIEW_CONTEXT	SMALLINT	Der Alias, der für die View-Spalte im Code der Abfragedefinition in Binärsprachenrepräsentation (BLR) verwendet wird
RDB$CONTEXT_NAME	CHAR(255)	Der Text, der mit dem in der Spalte RDB$VIEW_CONTEXT gemeldeten Alias verknüpft ist.
RDB$CONTEXT_TYPE	SMALLINT	Kontexttyp: 0 - Tabelle 1 - View 2 - Stored Procedure
RDB$PACKAGE_NAME	CHAR(31)	Paketname für eine gespeicherte Prozedur in einem Paket

Anhang E: Monitoringtabellen

Die Firebird-Engine kann Aktivitäten in einer Datenbank überwachen und über die Monitoring-Tabellen für Benutzerabfragen zur Verfügung stellen. Die Definitionen dieser Tabellen sind immer in der Datenbank vorhanden, alle mit dem Präfix `MON$` benannt. Die Tabellen sind virtuell: Sie werden nur in dem Moment mit Daten gefüllt, wenn der Benutzer sie abfragt. Das ist auch ein guter Grund, warum es sinnlos ist, Trigger für sie zu erstellen!

Der Schlüsselbegriff zum Verständnis der Überwachungsfunktion ist ein *Aktivitäts-Snapshot*. Die Aktivitätsmomentaufnahme stellt den aktuellen Zustand der Datenbank zu Beginn der Transaktion dar, in der die Überwachungstabellenabfrage ausgeführt wird. Es liefert viele Informationen über die Datenbank selbst, aktive Verbindungen, Benutzer, vorbereitete Transaktionen, laufende Abfragen und mehr.

Der Snapshot wird erstellt, wenn eine Überwachungstabelle zum ersten Mal abgefragt wird. Sie wird bis zum Ende der aktuellen Transaktion beibehalten, um eine stabile, konsistente Ansicht für Abfragen über mehrere Tabellen hinweg aufrechtzuerhalten, z. B. eine Master-Detail-Abfrage. Mit anderen Worten, Monitoring-Tabellen verhalten sich immer so, als ob sie sich in der Isolation `SNAPSHOT TABLE STABILITY` (“consistency”) befinden, selbst wenn die aktuelle Transaktion mit einer niedrigeren Isolationsstufe gestartet wird.

Um den Snapshot zu aktualisieren, muss die aktuelle Transaktion abgeschlossen und die Überwachungstabellen in einem neuen Transaktionskontext erneut abgefragt werden.

Zugriffssicherheit

- SYSDBA und der Datenbankbesitzer haben vollen Zugriff auf alle Informationen, die in den Überwachungstabellen verfügbar sind
- Normale Benutzer können Informationen über ihre eigenen Verbindungen sehen; andere Verbindungen sind für sie nicht sichtbar

In stark belasteten Umgebungen kann das Sammeln von Informationen über die Monitoringtabellen einen negativen Einfluss auf die Systemleistung haben.

Liste der Monitoringtabellen

`MON$ATTACHMENTS`

Informationen über aktive Datenbankattachments

`MON$CALL_STACK`

Aufrufe des Stack durch aktive Abfragen gespeicherter Prozeduren und Trigger

`MON$CONTEXT_VARIABLES`

Informationen zu benutzerdefinierten Kontextvariablen

MON$DATABASE

Informationen über die Datenbank, an die die `CURRENT_CONNECTION` angehängt ist

MON$IO_STATS

Eingabe-/Ausgabestatistiken

MON$MEMORY_USAGE

Statistiken zur Speichernutzung

MON$RECORD_STATS

Recordlevelstatistiken

MON$STATEMENTS

Zur Ausführung vorbereitete Anweisungen

MON$TABLE_STATS

Statistiken auf Tabellenebene

MON$TRANSACTIONS

Gestartete Transaktionen

MON$ATTACHMENTS

`MON$ATTACHMENTS` zeigt Informationen über aktive Attachments der Datenbank an.

Spaltenname	Datentyp	Beschreibung
`MON$ATTACHMENT_ID`	`BIGINT`	Verbindungs-Kennung
`MON$SERVER_PID`	`INTEGER`	Serverprozess-Kennung
`MON$STATE`	`SMALLINT`	Verbindungsstatus: `0` - Leerlauf (idle) 1 - Aktiv (active)
`MON$ATTACHMENT_NAME`	`VARCHAR(255)`	Connection String — der Dateiname und volle Pfad zur primären Datenbankdatei
`MON$USER`	`CHAR(31)`	Der Name des Benutzers, der mit diese Verbindung nutzt
`MON$ROLE`	`CHAR(31)`	Der angegebene Rollenname zum Zeitpunkt des Verbindungsaufbaus. Wurde beim Aufbau der Verbindung keine Rolle angegeben, enthält das Feld den Text `NONE`
`MON$REMOTE_PROTOCOL`	`VARCHAR(10)`	Name des Remote-Protokolls
`MON$REMOTE_ADDRESS`	`VARCHAR(255)`	Remote-Adresse (Adresse und Servername)
`MON$REMOTE_PID`	`INTEGER`	Kennung des Client-Prozesses

Spaltenname	Datentyp	Beschreibung
MON$CHARACTER_SET_ID	SMALLINT	Kennung des Zeichensatzes (vgl. RDB$CHARACTER_SET in der Systemtabelle RDB$TYPES)
MON$TIMESTAMP	TIMESTAMP	Datum und Zeit zum Zeitpunkt des Verbindungsaufbaus.
MON$GARBAGE_COLLECTION	SMALLINT	Kennzeichen für Garbage Collection (wie in der Attachment DPB definiert): 1=erlaubt (allowed), 0=nicht erlaubt (not allowed)
MON$REMOTE_PROCESS	VARCHAR(255)	Der volle Dateiname und Pfad zu der ausführbaren Datei, die diese Verbindung aufgebaut hat
MON$STAT_ID	INTEGER	Statistik-Kennung
MON$CLIENT_VERSION	VARCHAR(255)	Version der Clientbibliothek
MON$REMOTE_VERSION	VARCHAR(255)	Remote-Protokollversion
MON$REMOTE_HOST	VARCHAR(255)	Name des entfernten Hosts
MON$REMOTE_OS_USER	VARCHAR(255)	Name des entfernten Benutzers
MON$AUTH_METHOD	VARCHAR(255)	Name des Authentifizierungs-Plugins, mit dem die Verbindung hergestellt wurde
MON$SYSTEM_FLAG	SMALLINT	Kennzeichen, das den Verbindungstyp angibt: 0 - normale Verbindung 1 - Systemverbindung

Abrufen von Informationen zu Clientanwendungen

```
SELECT MON$USER, MON$REMOTE_ADDRESS, MON$REMOTE_PID, MON$TIMESTAMP
FROM MON$ATTACHMENTS
WHERE MON$ATTACHMENT_ID <> CURRENT_CONNECTION
```

Verwendung von MON$ATTACHMENTS um eine Verbindung zu beenden

Monitoringtabellen sind nur-lesend. Jedoch hat der Server einen eingebauten Mechanismus, um Datensätze zu löschen (und nur zum Löschen) in der Tabelle MON$ATTACHMENTS, wodurch es möglich wird, Datenbankverbindungen zu beenden.

Hinweise

- Sämtliche Aktivitäten der beendeten Verbindung werden augenblicklich gestoppt und alle aktiven Transaktionen werden zurückgerollt

- Die beendete Verbindung gibt einen Fehler mit dem Code `isc_att_shutdown` zurück
- Versuche diese Verbindung weiterzuverwenden, wird ebenfalls Fehler zurückgeben.
- Das Beenden von Systemverbindungen (`MON$SYSTEM_FLAG = 1`) ist nicht möglich. Der Server überspringt Systemverbindungen in einem `DELETE FROM MON$ATTACHMENTS`.

Alle Verbindungen außer der eigenen (current) beenden:

```
DELETE FROM MON$ATTACHMENTS
WHERE MON$ATTACHMENT_ID <> CURRENT_CONNECTION
```

MON$CALL_STACK

`MON $ CALL_STACK` zeigt Aufrufe des Stacks von Abfragen an, die in gespeicherten Prozeduren und Triggern ausgeführt werden.

Spaltenname	Datentyp	Beschreibung
MON$CALL_ID	BIGINT	Aufruf-Kennung
MON$STATEMENT_ID	BIGINT	Der Bezeichner der SQL-Anweisung der obersten Ebene, die die Aufrufkette initiiert hat. Verwenden Sie diesen Bezeichner, um die Datensätze zur aktiven Anweisung in der Tabelle MON$STATEMENTS zu finden
MON$CALLER_ID	INTEGER	Die Kennung der aufrufenden Stored Procedure oder des aufrufenden Triggers
MON$OBJECT_NAME	CHAR(31)	PSQL-Objekt-Name (Module)
MON$OBJECT_TYPE	SMALLINT	PSQL-Objekt-Typ (Trigger oder Stored Procedure): 2 - Trigger 5 - Stored Procedure 15 - Stored Function
MON$TIMESTAMP	TIMESTAMP	Datum und Zeitpunkt des Aufrufs
MON$SOURCE_LINE	INTEGER	Die Zeilennummer im SQL-Statement, welches zum Zeitpunkt des Snapshots gestartet wurde
MON$SOURCE_COLUMN	INTEGER	Die Spaltennummer im SQL-Statement, welches zum Zeitpunkt des Snapshots gestartet wurde

Spaltenname	Datentyp	Beschreibung
MON$STAT_ID	INTEGER	Statistik-Kennung
MON$PACKAGE_NAME	CHAR(31)	Paketname für gespeicherte Prozeduren oder Funktionen in einem Paket

Informationen über Aufrufe während der Ausführung der `EXECUTE STATEMENT` -Anweisung gelangen nicht in die Aufrufliste.

Rufen Sie die Aufrufliste für alle Verbindungen außer Ihren eigenen ab

```
WITH RECURSIVE
  HEAD AS (
    SELECT
      CALL.MON$STATEMENT_ID, CALL.MON$CALL_ID,
      CALL.MON$OBJECT_NAME, CALL.MON$OBJECT_TYPE
    FROM MON$CALL_STACK CALL
    WHERE CALL.MON$CALLER_ID IS NULL
    UNION ALL
    SELECT
      CALL.MON$STATEMENT_ID, CALL.MON$CALL_ID,
      CALL.MON$OBJECT_NAME, CALL.MON$OBJECT_TYPE
    FROM MON$CALL_STACK CALL
      JOIN HEAD ON CALL.MON$CALLER_ID = HEAD.MON$CALL_ID
  )
SELECT MON$ATTACHMENT_ID, MON$OBJECT_NAME, MON$OBJECT_TYPE
FROM HEAD
  JOIN MON$STATEMENTS STMT ON STMT.MON$STATEMENT_ID = HEAD.MON$STATEMENT_ID
WHERE STMT.MON$ATTACHMENT_ID <> CURRENT_CONNECTION
```

MON$CONTEXT_VARIABLES

`MON$CONTEXT_VARIABLES` zeigt Infos über benutzerdefinierte Kontextvariablen an.

Spaltenname	Datentyp	Beschreibung
MON$ATTACHMENT_ID	INTEGER	Verbindungskennung. Gültiger Wert nur für Variablen auf Verbindungsebene. Für Transaktionsebenen ist der Variablenwert `NULL`.
MON$TRANSACTION_ID	BIGINT	Transaktionskennung. Gültiger Wert nur auf Transaktionsebene. Für Verbindungsebenen ist der Variablenwert `NULL`.
MON$VARIABLE_NAME	VARCHAR(80)	Name der Kontextvariable
MON$VARIABLE_VALUE	VARCHAR(255)	Wert der Kontextvariable

Abrufen aller Sitzungskontextvariablen für die aktuelle Verbindung

```
SELECT
  VAR.MON$VARIABLE_NAME,
  VAR.MON$VARIABLE_VALUE
FROM MON$CONTEXT_VARIABLES VAR
WHERE VAR.MON$ATTACHMENT_ID = CURRENT_CONNECTION
```

MON$DATABASE

MON$DATABASE zeigt Header-Daten der Datenbank an, mit der der aktuelle Benutzer verbunden ist.

Spaltenname	Datentyp	Beschreibung
MON$DATABASE_NAME	VARCHAR(255)	Name und voller Pfad der primären Datenbankdatei oder der Datenbank-Alias.
MON$PAGE_SIZE	SMALLINT	Datenbank Seitengröße in Bytes.
MON$ODS_MAJOR	SMALLINT	Haupt-ODS-Version, z.B. 11
MON$ODS_MINOR	SMALLINT	Unter-ODS-Version, z.B. 2
MON$OLDEST_TRANSACTION	INTEGER	Nummer der ältesten (relevanten) Transaktion (oldest [interesting] transaction (OIT))
MON$OLDEST_ACTIVE	INTEGER	Nummer der ältesten aktiven Transaktion (oldest active transaction (OAT))
MON$OLDEST_SNAPSHOT	INTEGER	Nummer der Transaktion, die zum Zeitpunkt der OAT aktiv war - älteste Snapshot Transaktion (oldest snapshot transaction (OST))
MON$NEXT_TRANSACTION	INTEGER	Nummer der nächsten Transaktion zum Zeitpunkt als der Monitoring-Snapshot erstellt wurde
MON$PAGE_BUFFERS	INTEGER	Die Anzahl der Seiten, die im Speicher für den Datenbank Seiten-Cache (page cache) zugewiesen wurden
MON$SQL_DIALECT	SMALLINT	SQL-Dialekt der Datenbank: 1 oder 3
MON$SHUTDOWN_MODE	SMALLINT	Der derzeitige Shutdown-Status der Datenbank: 0 - Die Datenbank ist online 1 - Multi-User Shutdown 2 - Single-User Shutdown 3 - Kompletter Shutdown

Spaltenname	Datentyp	Beschreibung
MON$SWEEP_INTERVAL	INTEGER	Sweep-Intervall
MON$READ_ONLY	SMALLINT	Dieses Kennzeichen gibt an, ob die Datenbank im Modus read-only (Wert 1) oder read-write (Wert 0) arbeitet.
MON$FORCED_WRITES	SMALLINT	Gibt an, ob der Schreibmodus der Datenbank auf synchrones Schreiben (forced writes ON, Wert ist 1) oder asynchrones Schreiben (forced writes OFF, Wert ist 0) gestellt ist
MON$RESERVE_SPACE	SMALLINT	Gibt an, ob reserve_space (Wert 1) oder use_all_space (Wert 0) zum Füllen der Datenbankseiten verwendet wird.
MON$CREATION_DATE	TIMESTAMP	Datum und Zeit zu der die Datenbank erstellt oder wiederhergestellt wurde.
MON$PAGES	BIGINT	Anzahl der zugewiesenen Seiten der Datenbank auf einem externen Gerät
MON$STAT_ID	INTEGER	Statistik-Kennung
MON$BACKUP_STATE	SMALLINT	Derzeitiger physikalischer Backup-Status (nBackup): 0 - normal 1 - stalled 2 - merge
MON$CRYPT_PAGE	BIGINT	Anzahl verschlüsselter Seiten
MON$OWNER	CHAR(31)	Benutzername des Datenbankbesitzers
MON$SEC_DATABASE	CHAR(7)	Zeigt an, welcher Typ von Sicherheitsdatenbank verwendet wird: Default - Standard-Sicherheitsdatenbank, d. h. security3.fdb Self - aktuelle Datenbank wird als Sicherheitsdatenbank verwendet Sonstige - eine andere Datenbank wird als Sicherheitsdatenbank verwendet (nicht sie selbst oder security3.fdb)

MON$IO_STATS

MON$IO_STATS zeigt Input/Output-Statistiken an. Die Zähler arbeiten kumulativ, gruppiert für jede Statistikgruppe.

Spaltenname	Datentyp	Beschreibung
MON$STAT_ID	INTEGER	Statistik-Kennung
MON$STAT_GROUP	SMALLINT	Statistik-Gruppe: 0 - Datenbank 1 - Verbindung 2 - Transaktion 3 - Statement 4 - Aufruf (Call)
MON$PAGE_READS	BIGINT	Anzahl der gelesenen Datenbankseiten
MON$PAGE_WRITES	BIGINT	Anzahl der geschriebenen Datenbankseiten
MON$PAGE_FETCHES	BIGINT	Anzahl der geholten (fetched) Datenbankseiten
MON$PAGE_MARKS	BIGINT	Anzahl der markierten Datenbankseiten

MON$MEMORY_USAGE

MON$MEMORY_USAGE zeigt Statistiken zu Speichernutzung an.

Spaltenname	Datentyp	Beschreibung
MON$STAT_ID	INTEGER	Statistik-Kennung
MON$STAT_GROUP	SMALLINT	Statistik-Gruppen: 0 - Datenbank 1 - Verbindung 2 - Transaktion 3 - Statement 4 - Aufruf (Call)
MON$MEMORY_USED	BIGINT	Die Größe des genutzten Speichers in Bytes. Diese Daten beziehen sich auf die höchste Speicherzuteilung, die vom Server abgerufen wird. Dies ist nützlich, um Speicherlecks und exzessiven Speicherverbrauch in Verbindungen, Prozeduren, etc. zu ermitteln.

Spaltenname	Datentyp	Beschreibung
MON$MEMORY_ALLOCATED	BIGINT	Die vom Betriebssystem zugewiesene Speichermenge in Byte. Diese Daten beziehen sich auf die Low-Level-Speicherzuweisung, die vom Firebird-Speichermanager durchgeführt wird - die vom Betriebssystem zugewiesene Speichermenge -, mit der Sie die physische Speichernutzung steuern können.
MON$MAX_MEMORY_USED	BIGINT	Der größte Speicherverbrauch für dieses Objekt in Bytes.
MON$MAX_MEMORY_ALLOCATED	BIGINT	Die größte Speicherreservierung für dieses Objekt durch das Betriebssystem in Bytes.

Zähler, die den Datensätzen auf Datenbankebene MON$DATABASE (MON$STAT_GROUP = 0) zugeordnet sind, zeigen die Speicherzuweisung für alle Verbindungen an. In Classic und SuperClassic zeigen Nullwerte der Zähler an, dass diese Architekturen keinen gemeinsamen Cache haben.

Kleinere Speicherzuweisungen werden hier nicht gesammelt, sondern stattdessen dem Datenbankspeicherpool hinzugefügt.

Erhalten von 10 Anfragen, die den meisten Speicher verbrauchen

```
SELECT
  STMT.MON$ATTACHMENT_ID,
  STMT.MON$SQL_TEXT,
  MEM.MON$MEMORY_USED
FROM MON$MEMORY_USAGE MEM
NATURAL JOIN MON$STATEMENTS STMT
ORDER BY MEM.MON$MEMORY_USED DESC
FETCH FIRST 10 ROWS ONLY
```

MON$RECORD_STATS

MON$RECORD_STATS zeigt Statistiken auf Datensatzebene an. Die Zähler sind kumulativ nach Gruppe für jede Statistikgruppe.

Spaltenname	Datentyp	Beschreibung
MON$STAT_ID	INTEGER	Statistik-Kennung

Spaltenname	Datentyp	Beschreibung
MON$STAT_GROUP	SMALLINT	Statistik-Gruppen: 0 - Datenbank 1 - Verbindung 2 - Transaktion 3 - Statement 4 - Aufruf (Call)
MON$RECORD_SEQ_READS	BIGINT	Anzahl der sequenziell gelesenen Datensätze
MON$RECORD_IDX_READS	BIGINT	Anzahl der mittels Index gelesenen Datensätze
MON$RECORD_INSERTS	BIGINT	Anzahl der eingefügten Datensätze
MON$RECORD_UPDATES	BIGINT	Anzahl der aktualisierten Datensätze
MON$RECORD_DELETES	BIGINT	Anzahl der gelöschten Datensätze
MON$RECORD_BACKOUTS	BIGINT	Anzahl der Datensätze für die eine neue primäre Datensatzversion während eines Rollbacks oder Savepoint-Undo erstellt wurde.
MON$RECORD_PURGES	BIGINT	Anzahl der Datensätze für die die Versionskette nicht länger von der OAT (oldest active transaction) oder jüngeren Transaktionen benötigt wird.
MON$RECORD_EXPUNGES	BIGINT	Anzahl der Datensätze, in denen die Versionskette aufgrund von Löschungen innerhalb von Transaktionen gelöscht wird, die älter als die OAT (oldest active transaction) sind
MON$RECORD_LOCKS	BIGINT	Anzahl gesperrter Datensätze records
MON$RECORD_WAITS	BIGINT	Anzahl der Aktualisierungs-, Lösch- oder Sperrversuche für Datensätze, die anderen aktiven Transaktionen gehören. Die Transaktion befindet sich im WAIT-Modus.
MON$RECORD_CONFLICTS	BIGINT	Anzahl der erfolglosen Aktualisierungs-, Lösch- oder Sperrversuche für Datensätze, die anderen aktiven Transaktionen gehören. Diese werden als Aktualisierungskonflikte gemeldet.
MON$BACKVERSION_READS	BIGINT	Anzahl der gelesenen Back-Versionen, um sichtbare Datensätze zu finden

Spaltenname	Datentyp	Beschreibung
MON$FRAGMENT_READS	BIGINT	Anzahl der gelesenen fragmentierten Datensätze
MON$RECORD_RPT_READS	BIGINT	Anzahl der wiederholten Lesevorgänge von Datensätzen

MON$STATEMENTS

MON$STATEMENTS zeigt Anweisungen an, die zur Ausführung vorbereitet sind.

Spaltenname	Datentyp	Beschreibung
MON$STATEMENT_ID	INTEGER	Statement-Kennung
MON$ATTACHMENT_ID	INTEGER	Verbindungs-Kennung
MON$TRANSACTION_ID	INTEGER	Transaktions-Kennung
MON$STATE	SMALLINT	Statement-Status: 0 - Leerlauf (idle) 1 - Aktiv 2 - verzögert (stalled)
MON$TIMESTAMP	TIMESTAMP	Der Zeitpunkt an dem das Statement vorbereitet wurde.
MON$SQL_TEXT	BLOB TEXT	Statement-Text in SQL
MON$STAT_ID	INTEGER	Statistik-Kennung
MON$EXPLAINED_PLAN	BLOB TEXT	Erklärter Ausführungsplan

Der Status STALLED zeigt an, dass die Anweisung zum Zeitpunkt des Snapshots einen geöffneten Cursor hatte und darauf wartete, dass der Client den Abruf von Zeilen wieder aufnimmt.

Aktive Abfragen anzeigen, ausgenommen diejenigen, die in Ihrer Verbindung ausgeführt werden

```
SELECT
  ATT.MON$USER,
  ATT.MON$REMOTE_ADDRESS,
  STMT.MON$SQL_TEXT,
  STMT.MON$TIMESTAMP
FROM MON$ATTACHMENTS ATT
JOIN MON$STATEMENTS STMT ON ATT.MON$ATTACHMENT_ID = STMT.MON$ATTACHMENT_ID
WHERE ATT.MON$ATTACHMENT_ID <> CURRENT_CONNECTION
AND STMT.MON$STATE = 1
```

Verwenden von MON$STATEMENTS zum Abbrechen einer Abfrage

Überwachungstabellen sind schreibgeschützt. Der Server verfügt jedoch über einen eingebauten Mechanismus zum Löschen (und nur zum Löschen) von Datensätzen in der Tabelle MON$STATEMENTS,

der es ermöglicht, eine laufende Abfrage abzubrechen.

Anmerkungen

- Wenn derzeit keine Anweisungen in der Verbindung ausgeführt werden, wird jeder Versuch, Abfragen abzubrechen, nicht fortgesetzt
- Nachdem eine Abfrage abgebrochen wurde, wird beim Aufrufen von API-Funktionen zum Ausführen/Abrufen ein Fehler mit dem Code `isc_cancelled` zurückgegeben
- Nachfolgende Abfragen von dieser Verbindung werden wie gewohnt fortgesetzt
- Der Abbruch der Anweisung erfolgt nicht synchron, sondern markiert nur die Anforderung der Stornierung, und die Stornierung selbst erfolgt asynchron durch den Server

Beispiel

Alle aktiven Abfragen für die angegebene Verbindung abbrechen:

```
DELETE FROM MON$STATEMENTS
  WHERE MON$ATTACHMENT_ID = 32
```

MON$TABLE_STATS

`MON$TABLE_STATS` gibt Statistiken auf Tabellenebene aus.

Spaltenname	Datentyp	Beschreibung
`MON$STAT_ID`	`INTEGER`	Statistikkennung
`MON$STAT_GROUP`	`SMALLINT`	Statistikgruppe: `0` - Datenbank `1` - Verbindung `2` - Transaktion `3` - Anweisung (Statement) `4` - Aufruf (Call)
`MON$TABLE_NAME`	`CHAR(31)`	Tabellenname
`MON$RECORD_STAT_ID`	`INTEGER`	Link zu `MON$RECORD_STATS`

Statistiken auf Datensatzebene für jede Tabelle für die aktuelle Verbindung abrufen

```
SELECT
  t.mon$table_name,
  r.mon$record_inserts,
  r.mon$record_updates,
  r.mon$record_deletes,
  r.mon$record_backouts,
```

```
  r.mon$record_purges,
  r.mon$record_expunges,
  ------------------------
  r.mon$record_seq_reads,
  r.mon$record_idx_reads,
  r.mon$record_rpt_reads,
  r.mon$backversion_reads,
  r.mon$fragment_reads,
  ------------------------
  r.mon$record_locks,
  r.mon$record_waits,
  r.mon$record_conflicts,
  ------------------------
  a.mon$stat_id
FROM mon$record_stats r
JOIN mon$table_stats t ON r.mon$stat_id = t.mon$record_stat_id
JOIN mon$attachments a ON t.mon$stat_id = a.mon$stat_id
WHERE a.mon$attachment_id = CURRENT_CONNECTION
```

MON$TRANSACTIONS

MON$TRANSACTIONS meldet gestartete Transaktionen.

Spaltenname	Datentyp	Beschreibung
MON$TRANSACTION_ID	INTEGER	Transaktionskennung
MON$ATTACHMENT_ID	INTEGER	Verbindungskennung
MON$STATE	SMALLINT	Transaktionsstatus: 0 - Leerlauf (idle) 1 - Aktiv
MON$TIMESTAMP	TIMESTAMP	Zeitpunkt an dem die Transaktion gestartet wurde
MON$TOP_TRANSACTION	INTEGER	Top-Level-Transaktionsnummer (Kennung)
MON$OLDEST_TRANSACTION	INTEGER	Kennung der ältesten relevanten Transaktion (oldest [interesting] transaction (OIT))
MON$OLDEST_ACTIVE	INTEGER	Kennung der ältesten aktiven Transaktion (oldest active transaction (OAT))

Spaltenname	Datentyp	Beschreibung
MON$ISOLATION_MODE	SMALLINT	Isolationsmodus (Level): 0 - Konsistenz (Snapshot für Tabellenstabilität) 1 - Konkurrierend (Snapshot) 2 - Read Committed mit Datensatzversion 3 - Read Committed ohne Datensatzversion
MON$LOCK_TIMEOUT	SMALLINT	Lock-Timeout: -1 - warten (ewig) 0 - nicht warten 1 oder größer - Lock-Timeout in Sekunden
MON$READ_ONLY	SMALLINT	Gibt an, ob die Transaktion nur-lesend (Wert 1) oder lesend-schreibend (Wert 0) läuft
MON$AUTO_COMMIT	SMALLINT	Gibt an, ob automatisches Commit für die Transaktion verwendet wird (Wert 1) oder nicht (Wert 0)
MON$AUTO_UNDO	SMALLINT	Gibt an, ob der Logging-Mechanismus *automatiches Undo* für die Transaktion verwendet wird (Wert 1) oder nicht (Wert 0)
MON$STAT_ID	INTEGER	Statistikkennung

Alle Verbindungen abrufen, die Read Write-Transaktionen mit Isolationsstufe über Read Committed starten

```
SELECT DISTINCT a. *
FROM mon$attachments a
JOIN mon$transactions t ON a.mon$attachment_id = t.mon$attachment_id
WHERE NOT (t.mon$read_only = 1 AND t.mon$isolation_mode >= 2)
```

Anhang F: Sicherheitstabellen

Die Namen der Sicherheitstabellen haben SEC$ als Präfix. Sie zeigen Daten aus der aktuellen Sicherheitsdatenbank an. Diese Tabellen sind virtuell in dem Sinne, dass vor dem Zugriff durch den Benutzer keine Daten in ihnen aufgezeichnet werden. Sie werden zum Zeitpunkt der Benutzeranfrage mit Daten gefüllt. Die Beschreibungen dieser Tabellen sind jedoch ständig in der Datenbank vorhanden. In diesem Sinne sind diese virtuellen Tabellen wie Tabellen der MON$ -Familie, die zur Überwachung des Servers verwendet werden.

Sicherheit

- SYSDBA, Benutzer mit der Rolle RDB$ADMIN in der Sicherheitsdatenbank und der aktuellen Datenbank sowie der Besitzer der Sicherheitsdatenbank haben vollen Zugriff auf alle von den Sicherheitstabellen bereitgestellten Informationen.
- Reguläre Benutzer können nur Informationen über sich selbst sehen, andere Benutzer sind nicht sichtbar.

Diese Funktionen sind stark vom Benutzerverwaltungs-Plugin abhängig. Beachten Sie, dass einige Optionen ignoriert werden, wenn Benutzer eines Legacy-Steuerelement-Plugins verwenden.

Liste der Sicherheitstabellen

SEC$DB_CREATORS

Listet Benutzer und Rollen auf, denen das Privileg CREATE DATABASE gewährt wurde

SEC$GLOBAL_AUTH_MAPPING

Informationen zu globalen Authentifizierungszuordnungen

SEC$USERS

Listet Benutzer in der aktuellen Sicherheitsdatenbank auf

SEC$USER_ATTRIBUTES

Zusätzliche Attribute von Benutzern

SEC$DB_CREATORS

Listet Benutzer und Rollen auf, denen die Berechtigung CREATE DATABASE zugewiesen wurde.

Spaltenname	Datentyp	Beschreibung
SEC$USER	CHAR(31)	Name des Benutzers oder der Rolle
SEC$USER_TYPE	SMALLINT	Benutzerart 8 - Benutzer 13 - Rolle

SEC$GLOBAL_AUTH_MAPPING

Listet Benutzer und Rollen auf, denen die Berechtigung `CREATE DATABASE` zugewiesen wurde.

Spaltenname	Datentyp	Beschreibung
SEC$MAP_NAME	CHAR(31)	Name der Zuweisung
SEC$MAP_USING	CHAR(1)	Verwendete Definitionen P - Plugin (spezifisch oder beliebig) S - jedes Plugin serverweit M - Zuordnung (Mapping) * - jede Methode
SEC$MAP_PLUGIN	CHAR(31)	Die Zuordnung gilt für Authentifizierungsinformationen von diesem bestimmten Plugin
SEC$MAP_DB	CHAR(31)	Die Zuordnung gilt für Authentifizierungsinformationen aus dieser speziellen Datenbank
SEC$MAP_FROM_TYPE	CHAR(31)	Der Typ des Authentifizierungsobjekts (definiert durch das Plugin), von dem die Zuordnung erfolgen soll, oder * für jeden Typ
SEC$MAP_FROM	CHAR(255)	Der Name des Authentifizierungsobjekts, von dem eine Zuordnung vorgenommen werden soll
SEC$MAP_TO_TYPE	SMALLINT Nullable	Der Typ, dem zugeordnet werden soll 0 - Benutzer 1 - Rolle
SEC$MAP_TO	CHAR(31)	Der Name, dem zugeordnet werden soll

SEC$USERS

Listet Benutzer in der aktuellen Sicherheitsdatenbank auf.

Column Name	Data Type	Beschreibung
SEC$USER_NAME	CHAR(31)	Benutzername
SEC$FIRST_NAME	VARCHAR(32)	Vorname
SEC$MIDDLE_NAME	VARCHAR(32)	Zusatzname
SEC$LAST_NAME	VARCHAR(32)	Nachname
SEC$ACTIVE	BOOLEAN	`true` - aktiv, `false` - inaktiv

Column Name	Data Type	Beschreibung
SEC$ADMIN	BOOLEAN	`true` - Benutzer hat eine Administratorrolle in der Sicherheitsdatenbank, andernfalls `false`
SEC$DESCRIPTION	BLOB TEXT	Beschreibung (Kommentar) zum Benutzer
SEC$PLUGIN	CHAR(31)	Name des Authentifizierungs-Plugins, das diesen Benutzer verwaltet

Es können mehrere Benutzer mit demselben Benutzernamen existieren, die jeweils von einem anderen Authentifizierungs-Plugin verwaltet werden.

SEC$USER_ATTRIBUTES

Zusätzliche Benutzerattribute

Spaltenname	Datentyp	Beschreibung
SEC$USER_NAME	CHAR(31)	Benutzername
SEC$KEY	VARCHAR(31)	Attributname
SEC$VALUE	VARCHAR(255)	Attributwert
SEC$PLUGIN	CHAR(31)	Name des Authentifizierungs-Plugins, das diesen Benutzer verwaltet

Anzeigen einer Liste von Benutzern und ihren Attributen

```
SELECT
  U.SEC$USER_NAME AS LOGIN,
  A.SEC$KEY AS TAG,
  A.SEC$VALUE AS "VALUE",
  U.SEC$PLUGIN AS "PLUGIN"
FROM SEC$USERS U
LEFT JOIN SEC$USER_ATTRIBUTES A
  ON U.SEC$USER_NAME = A.SEC$USER_NAME
    AND U.SEC$PLUGIN = A.SEC$PLUGIN;

LOGIN    TAG     VALUE   PLUGIN
======== ======= ======= ====================
SYSDBA   <null>  <null>  Srp
ALEX     B       x       Srp
ALEX     C       sample  Srp
SYSDBA   <null>  <null>  Legacy_UserManager
```

Anhang G: Zeichensätze und Collations

Tabelle 234. Zeichensätze und Collations

Zeichensatz	ID	Bytes pro Zeichen	Sortierung	Sprache
ASCII	2	1	ASCII	Englisch
BIG_5	56	2	BIG_5	Chinesisch, Vietnamesisch, Koreanisch
CP943C	68	2	CP943C	Japanisch
//	//	//	CP943C_UNICODE	Japanisch
CYRL	50	1	CYRL	Russisch
//	//	//	DB_RUS	Russisch dBase
//	//	//	PDOX_CYRL	Russisch Paradox
DOS437	10	1	DOS437	U.S. Englisch
//	//	//	DB_DEU437	Deutsch dBase
//	//	//	DB_ESP437	Spanisch dBase
//	//	//	DB_FIN437	Finnisch dBase
//	//	//	DB_FRA437	Französisch dBase
//	//	//	DB_ITA437	Italienisch dBase
//	//	//	DB_NLD437	Niederländisch dBase
//	//	//	DB_SVE437	Schwedisch dBase
//	//	//	DB_UK437	Englisch (Groß Britanien) dBase
//	//	//	DB_US437	U.S. Englisch dBase
//	//	//	PDOX_ASCII	Code page Paradox-ASCII
//	//	//	PDOX_SWEDFIN	Schwedisch / Finnisch Paradox
//	//	//	PDOX_INTL	International Englisch Paradox
DOS737	9	1	DOS737	Griechisch
DOS775	15	1	DOS775	Baltic
DOS850	11	1	DOS850	Latin I (ohne Euro-Zeichen)
//	//	//	DB_DEU850	Deutsch
//	//	//	DB_ESP850	Spanisch
//	//	//	DB_FRA850	Französisch
//	//	//	DB_FRC850	Französisch-Kanadisch
//	//	//	DB_ITA850	Italienisch

Zeichensatz	ID	Bytes pro Zeichen	Sortierung	Sprache
//	//	//	DB_NLD850	Niederländisch
//	//	//	DB_PTB850	Portugisisch - Brasilien
//	//	//	DB_SVE850	Schwedisch
//	//	//	DB_UK850	Englisch- Groß Britanien
//	//	//	DB_US850	U.S. Englisch
DOS852	45	1	DOS852	Latin II
//	//	//	DB_CSY	Tschechisch dBase
//	//	//	DB_PLK	Polnisch dBase
//	//	//	DB_SLO	Slowenisch dBase
//	//	//	PDOX_CSY	Tschechisch Paradox
//	//	//	PDOX_HUN	Hungarisch Paradox
//	//	//	PDOX_PLK	Polnisch Paradox
//	//	//	PDOX_SLO	Sloweniisch Paradox
DOS857	46	1	DOS857	Türkisch
//	//	//	DB_TRK	Türkisch dBase
DOS858	16	1	DOS858	Latin I (mit Euro-Zeichen)
DOS860	13	1	DOS860	Portugisisch
//	//	//	DB_PTG860	Portugisisch dBase
DOS861	47	1	DOS861	Isländisch
//	//	//	PDOX_ISL	Isländisch Paradox
DOS862	17	1	DOS862	Hebräisch
DOS863	14	1	DOS863	Französisch-Kanada
//	//	//	DB_FRC863	Französisch dBase-Kanada
DOS864	18	1	DOS864	Arabisch
DOS865	12	1	DOS865	Skandinavisch
//	//	//	DB_DAN865	Dänisch dBase
//	//	//	DB_NOR865	Norwegisch dBase
//	//	//	PDOX_NORDAN4	Paradox Norwegen und Dänemark
DOS866	48	1	DOS866	Russisch
DOS869	49	1	DOS869	Modernes Griechisch
EUCJ_0208	6	2	EUCJ_0208	Japanisch EUC

Zeichensatz	ID	Bytes pro Zeichen	Sortierung	Sprache
GB_2312	57	2	GB_2312	Vereinfachtes Chinesisch (Hong Kong, Korea)
GB18030	69	4	GB18030	Chinesisch
//	//	//	GB18030_UNICODE	Chinesisch
GBK	67	2	GBK	Chinesisch
//	//	//	GBK_UNICODE	Chinesisch
ISO8859_1	21	1	ISO8859_1	Latin I
//	//	//	DA_DA	Dänisch
//	//	//	DE_DE	Deutsch
//	//	//	DU_NL	Niederländisch
//	//	//	EN_UK	Englisch - Groß Britanien
//	//	//	EN_US	U.S. Englisch
//	//	//	ES_ES	Spanisch
//	//	//	ES_ES_CI_AI	Spanisch — groß- und kleinschreibungsunabhängig sowie akzentunabhängig
//	//	//	FI_FI	Finnisch
//	//	//	FR_CA	Französisch-Kanada
//	//	//	FR_FR	Französisch
//	//	//	FR_FR_CI_AI	Französisch — groß- und kleinschreibungsunabhängig sowie akzentunabhängig
//	//	//	IS_IS	Isländisch
//	//	//	IT_IT	Italienisch
//	//	//	NO_NO	Norwegisch
ISO8859_1	//	//	PT_PT	Portugisisch
//	//	//	PT_BR	Portugisisch-Brasilien
//	//	//	SV_SV	Schwedisch
ISO8859_2	22	1	ISO8859_2	Latin 2 — Zentraleuropa (Kroatisch, tschechisch, ungarisch, polnisch, romanisch, serbisch, slovakisch, slowenisch)
//	//	//	CS_CZ	Tschechisch

Zeichensatz	ID	Bytes pro Zeichen	Sortierung	Sprache
//	//	//	ISO_HUN	Hungarian
//	//	//	ISO_PLK	Polnisch
ISO8859_3	23	1	ISO8859_3	Latin 3 — Südeuropa (Malta, Esperanto)
ISO8859_4	34	1	ISO8859_4	Latin 4 — Nordeuropa (Estnisch, lettisch, litauisch, grönländisch, lappisch)
ISO8859_5	35	1	ISO8859_5	Kyrillisch (Russisch)
ISO8859_6	36	1	ISO8859_6	Arabisch
ISO8859_7	37	1	ISO8859_7	Griechisch
ISO8859_8	38	1	ISO8859_8	Hebräisch
ISO8859_9	39	1	ISO8859_9	Latin 5
ISO8859_13	40	1	ISO8859_13	Latin 7 — Baltikum
//	//	//	LT_LT	Litauisch
KOI8R	63	1	KOI8R	Russisch — Wörterbuchsortierung
//	//	//	KOI8R_RU	Russisch
KOI8U	64	1	KOI8U	Ukrainisch — Sortierung nach Wörterbuch
//	//	//	KOI8U_UA	Ukrainisch
KSC_5601	44	2	KSC_5601	Koreanisch
//	//	//	KSC_DICTIONARY	Koreanisch — Sortierung nach Wörterbuch
NEXT	19	1	NEXT	Coding NeXTSTEP
//	//	//	NXT_DEU	Deutsch
//	//	//	NXT_ESP	Spanisch
//	//	//	NXT_FRA	Französisch
//	//	//	NXT_ITA	Italienisch
NEXT	19	1	NXT_US	U.S. Englisch
NONE	0	1	NONE	Neutrale code page. Umwandlung in Großschreibung wird nur für ASCII-Codes 97-122 durchgeführt. Empfehlung: Zeichensatz vermeiden.`
OCTETS	1	1	OCTETS	Binäre Zeichenkodierung

Zeichensatz	ID	Bytes pro Zeichen	Sortierung	Sprache
SJIS_0208	5	2	SJIS_0208	Japanisch
TIS620	66	1	TIS620	Thailändisch
//	//	//	TIS620_UNICODE	Thailändisch
UNICODE_FSS	3	3	UNICODE_FSS	Alle englischen
UTF8	4	4	UTF8	Alle durch Unicode 4.0 unterstützte Sprachen
//	//	//	UCS_BASIC	Alle durch Unicode 4.0 unterstützte Sprachen
//	//	//	UNICODE	Alle durch Unicode 4.0 unterstützte Sprachen
//	//	//	UNICODE_CI	Alle durch Unicode 4.0 unterstützte Sprachen — groß- und kleinschreibunabhängig
//	//	//	UNICODE_CI_AI	Alle durch Unicode 4.0 unterstützte Sprachen — groß- und kleinschreibunabhängig
WIN1250	51	1	WIN1250	ANSI — Zentraleuropa
//	//	//	BS_BA	Bosnisch
//	//	//	PXW_CSY	Tschechisch
//	//	//	PXW_HUN	Ungarisch
//	//	//	PXW_HUNDC	Ungarisch — Sortierung nach Wörterbuch
//	//	//	PXW_PLK	Polnisch
//	//	//	PXW_SLOV	Slowenisch
//	//	//	WIN_CZ	Tschechisch
//	//	//	WIN_CZ_CI	Tschechisch — groß- und kleinschreibungsunabhängig
//	//	//	WIN_CZ_CI_AI	Tschechisch — groß- und kleinschreibungsunabhängig und akzentunabhängig
WIN1251	52	1	WIN1251	ANSI Kyrillisch
//	//	//	WIN1251_UA	Ukrainisch
//	//	//	PXW_CYRL	Paradox Kyrillisch (Russisch)
WIN1252	53	1	WIN1252	ANSI — Latin I

Zeichensatz	ID	Bytes pro Zeichen	Sortierung	Sprache
//	//	//	PXW_INTL	Englisch international
//	//	//	PXW_INTL850	Paradox mehrsprachig Latin I
//	//	//	PXW_NORDAN4	Norwegisch und Dänisch
//	//	//	PXW_SPAN	Paradox Spanisch
//	//	//	PXW_SWEDFIN	Schwedisch und Finnisch
//	//	//	WIN_PTBR	Portugisisch — Brasilianisch
WIN1253	54	1	WIN1253	ANSI Griechisch
//	//	//	PXW_GREEK	Paradox Griechisch
WIN1254	55	1	WIN1254	ANSI Türkisch
//	//	//	PXW_TURK	Paradox Türkisch
WIN1255	58	1	WIN1255	ANSI Hebräisch
WIN1256	59	1	WIN1256	ANSI Arabisch
WIN1257	60	1	WIN1257	ANSI Baltisch
//	//	//	WIN1257_EE	Estnisch — Sortierung nach Wörterbuch
//	//	//	WIN1257_LT	Litauisch — Sortierung nach Wörterbuch
//	//	//	WIN1257_LV	Lettisch — Sortierung nach Wörterbuch
WIN1258	65	1	WIN1258	Vietnamesisch

Anhang H: Lizenzhinweise

Die originale Dokumentation wurde unter dem Titel *Firebird 3.0 Language Reference* erfasst. Dieses Dokument basiert auf der *Firebird 2.5 Language Reference*.

Die ursprünglichen Schreiber der Originaldokumentation sind: Paul Vinkenoog, Dmitry Yemanov, Thomas Woinke und Mark Rotteveel. Verfasser der ursprünglichen Texte in Russisch sind Denis Simonov, Dmitry Filippov, Alexander Karpeykin, Alexey Kovyazin und Dmitry Kuzmenko.

Anhang I: Dokumenthistorie

Die exakte Dateihistorie ist im *Git*-Repository des `firebird-documentation`-Repository zu finden; siehe https://github.com/FirebirdSQL/firebird-documentation

Historie

1.13.1.de	2. April 2024	MR	Protokollnamen werden in Kleinbuchstaben geschrieben (#205)
1.13.de	30. Januar 2023	MR	Tippfehler im Sortierungsnamen `UCS_BASIC` behoben
1.12.de	30. Januar 2023	MR	• Falsche Leerzeichen in Tabellennamen behoben • Fehlenden Tabellennamen in `FROM` in Deterministische Funktionen hinzugefügt (#177)
1.11-de	31. Juli 2022	MR	Die Beschreibung `RDB$TRIGGER_TYPE` wurde in einen separaten Abschnitt verschoben, um das Abschneiden von Tabellenzellen in PDF zu verhindern
1.10-de	18. Juli 2022	MR	Dokumentation für `RDB$INDICES.RDB$INDEX_TYPE` korrigiert (#174)
1.9-de	13. July 2022	MR	• `DATEDIFF` Unit `MILLISECOND` gibt seit Firebird 3.0.8 `NUMERIC(18,1)` zurück (#173) • Werte für `RDB$RELATION_FIELDS.RDB$IDENTITY_TYPE` wurden vertauscht (#168)
1.7-de	21. November 2021	MK	Übersetzung der englischen Sprachreferenz für Firebird 3.0 ins Deutsche.
1.7	16. Oktober 2021	MR	`EXECUTE STATEMENT`-benannte Parameter sind reguläre Bezeichner (#164)
1.6	29. September 2021	MR	Explizite Dokumentation der Transaktions-Isolationsstufen für `ON CONNECT`/`ON DISCONNECT`-Trigger (#163)
1.5	31. Juli 2021	MR	Behebung des Verhaltens dokumentiert für `SNAPSHOT TABLE STABILITY` (#158)
1.4	23. Juli 2021	MR	Extra `SELECT` in Select-Syntax entfernt
1.3	13. Juni 2021	MR	• Falschen Linktitel geändert: `NUMERIC` → `DECIMAL` • Falschen Linktitel geändert: `DATEADD` → `DATEDIFF`
1.2	27. April 2021	MR	• Fehlende } in regulären Ausdruck für Sonderzeichen hinzugefügt (siehe issue 124) • Problem beim Rendern mit unsichtbarem _ im regulären Ausdruck für Sonderzeichen behoben • Verbesserungen des Ausdrucks von `CURRENT_CONNECTION` und `CURRENT_TRANSACTION` (siehe issue 96)

Historie

1.1	05. April 2021	MR	Syntaxfehler in der Dokumentation zu `SUBSTRING(⋯ SIMILAR ⋯)` korrigiert.
1.0	20 Februar 2021	MR.	Nutzung von *Firebird 2.5 Language Reference* als Ausgangspunkt. Alle Änderungen von Firebird 3.0 eingearbeitet. Dazu wurden die Firebird 3 Release Notes und die russische Firebird 3.0-Sprachreferenz verwendet. Einige Neustrukturierung für bessere Wart- und Lesbarkeit vorgenommen.