Guida sull'uso di NULL nel linguaggio SQL di Firebird

Uso e malintesi su NULL in Firebird

Paul Vinkenoog

31 marzo 2007 - Versione italiana 1.0.1-it

Traduzione in italiano: Umberto Masotti

Sommario

Che cosa è NULL?

Un giorno sì e l'altro pure, nella mailing list di supporto di Firebird, vengono fatte domande relative a «strane cose» che succedono con i NULL in Firebird SQL. Il concetto sembra difficile da comprendere, in parte a causa del nome che suggerisce «nulla» o «niente», e che quindi apparentemente non dovrebbe creare problemi se aggiunto ad un numero o concatenato ad una stringa. In realtà, fare quelle cose da' come risultato dell'espressione `NULL`.

Questo articolo approfondisce il funzionamento di `NULL` in Firebird SQL, evidenziando alcuni errori comuni e mostra come si possono gestire correttamente le espressioni che contengono `NULL` o che possono valere `NULL`.

Per avere un riferimento veloce, giusto per rinfrescarsi la memoria, si può saltare a piè pari direttamente al riassunto (che è davvero sintetico e stringato).

Allora, che cosa è?

In SQL, NULL non è un valore. È uno *stato* che indica che il valore di un oggetto è sconosciuto o inesistente. Non è zero, nè il carattere spazio e neppure una stringa vuota e, sopratutto, non si comporta come nessuno dei tre. Poche cose creano altrettanta confusione in SQL come il `NULL`, eppure il suo funzionamento non dovrebbe essere così difficile da capire dal momento che basta fissarsi su questa semplice definizione: `NULL` significa *sconosciuto*.

Lo ripeto:

> **`NULL` significa SCONOSCIUTO**

Questo fatto va tenuto bene in mente leggendo il resto dell'articolo, così molti dei comportamenti apparentemente illogici che si hanno con `NULL` immediatamente si spiegheranno in modo automatico.

> **Nota**
>
> Alcune parti ed esempi di questa guida sono state prese dalla *Firebird Quick Start Guide*, pubblicata originariamente dalla IBPhoenix, ora parte del progetto Firebird.

Come Firebird supporta il NULL nel linguaggio SQL

Solo pochi elementi del linguaggio sono stati appositamente progettati per determinare un risultato non ambiguo con il NULL; si intende «non ambiguo» nel senso che vengono prese particolari azioni specifiche o risulta un valore che non è NULL. Questi casi sono mostrati nei seguenti paragrafi.

Impedire il `NULL`

Nella definizione di una colonna o di un dominio, si può specificare che possono essere ammessi solo valori diversi dal NULL, aggiungendo la clausola NOT NULL alla definizione:

```
create table Tabella ( i int not null )
```

```
create domain DComune as varchar( 32 ) not null
```

Particlare attenzione andrebbe presa aggiungendo un campo NOT NULL ad una tabella già esistente che contiene delle registrazioni. Questa operazione verrà discussa in dettaglio nella sezione *Modificare le tabelle piene*.

Controllare un campo per `NULL`

Per sapere se una variabile, un campo, o un'ntera espressione è NULL, è necessario usare la sintassi seguente:

<expression> IS [NOT] NULL

Esempi:

```
if ( QuestoCampo is null ) then QuestaStringa = 'Nonso'
```

```
select * from Alunni where NumeroTelefono is not null
```

```
select * from Alunni where not ( NumeroTelefono is null )
/* fa la stessa cosa dell'esempio precedente */
```

```
update NumeriVari set Totale = A + B + C where A + B + C is not null
```

```
delete from Agendina where NumTelefono is null
```

NON usare assolutamente «... = `NULL`» per verificare se l'espressione è NULL. Questa sintassi è illegale nelle versioni di Firebird fino alla 1.5.n, e dà risultato sbagliato (o meglio inatteso) in Firebird 2 e successivi: riporta NULL qualsiasi sia l'espressione da confrontare. Questo è per progetto ed in tal senso non è *proprio* sbagliato – non dà semplicemente il risultato sperato. Ovviamente lo stesso vale per «... <> `NULL`», pertanto è meglio non usare neanche questa espressione: va usato invece IS NOT NULL.

IS NULL e IS NOT NULL riportano sempre `true` oppure `false`; non riportano mai `NULL`.

Assegnare `NULL`

Per assegnare ad un campo o ad una variabile il `NULL` si usa l'operatore «=», come per tutti gli altri valori. Si può utilizzare il `NULL` anche nelle liste della clausola INSERT:

```
if ( Stringaccia = 'Nonso' ) then Campo = null
```

```
update Patate set Tuberi = null where Importo < 0
```

```
insert into TabellaX values ( 3, '8-5-2004', NULL, 'Che cosa?' )
```

Ricordarsi:

- Non si può e non si deve usare *l'operatore di comparazione* «=» per *controllare* se qualcosa *è* `NULL`...
- ...ma si può – e spesso si deve – usare *l'operatore di assegnazione* «=» per *impostare* qualcosa *a* `NULL`.

Controllare la diversità (Firebird 2+)

Solo da Firebird 2 e successivi, su può paragonare la diversità anche nulla di due espressioni qualsiasi con «IS [NOT] DISTINCT FROM»:

```
if ( A is distinct from B ) then...
```

```
if ( Cliente1 is not distinct from Cliente2 ) then...
```

I campi, le variabili ed altre espressioni sono considerate:

- distinte, usando DISTINCT, se hanno valori diversi o uno dei valori è `NULL` ma non l'altro;
- uguali, usando NOT DISTINCT, se hanno lo stesso valore oppure se sono entrambe `NULL`.

[NOT] DISTINCT riporta sempre `true` oppure `false`, mai NULL o qualsiasi altra cosa.

Con le versioni precedenti di Firebird bisognava scrivere codice più complesso per ottenere lo stesso risultato. Lo vedremo in seguito.

La costante letterale `NULL`

La possibilità di usare costanti letterali `NULL` dipende dalla versione di Firebird utilizzata.

Fino a Firebird 1.5 compreso

In Firebird 1.5 e precedenti si può usare la parola letterale «`NULL`» solo in alcune situazioni, in particolare quelle descritte nei precedenti paragrafi più poche altre come «cast(`NULL` as *`<datatype>`*)» e «select `NULL` from Tabella».

In tutte le altre circostanze, Firebird dirà che `NULL` è un oggetto sconosciuto (`unknown token`). Se si *deve* usare `NULL` in tali contesti, bisogna ricorrere a trucchetti del tipo «cast(`NULL` as int)», oppure usare un campo o una variabile che è notoriamente `NULL`.

Firebird 2.0 and up

Firebird 2 permette l'uso di costanti letterali `NULL` in ogni contesto in cui può essere ammesso un normale valore. Si può, ad esempio, includere `NULL` in una lista della clausola IN(), scrivere espressioni come «if (Campo = `NULL`) then...», e così via. Tuttavia, come regola generale **non si dovrebbe** fare uso di queste nuove possibilità! In quasi tutte le situazioni pensabili, un tale uso del `NULL` è segno di una progettazione SQL qualitativamente scarsa literals e porta a risultati `NULL` dove invece si desidererebbe `true` oppure `false`. In questo senso la precedente condotta, più restrittiva, è migliore, sebbene si possa sempre aggirarla con forzature tipo `cast` ecc. ma questo almeno comporta che sia necessario fare certi passi deliberatamente.

NULL nelle espressioni

Come molti di noi hanno scoperto con sconforto, NULL è contagioso: usato in una espressione numerica, di stringhe, o data/ora, il risultato sarà sempre e comunque NULL. Usandolo invece in una operazione logica, il risultato dipende dal tipo dell'operazione e degli altri valori coinvolti.

Notare, inoltre, che nelle versioni di Firebird precedenti alla 2.0 è abbastanza illegale utilizzare NULL direttamente nelle operazioni o nei confronti. Dovunque si vede NULL nelle seguenti espressioni, va letto come «un campo, una variabile o altra espressione che vale NULL». In Firebird 2 e successivi questa espressione può anche essere un NULL letterale.

Operazioni matematiche e con stringhe

Le espressioni nella seguente lista riportano *sempre e comunque* tutte NULL:

- `1 + 2 + 3 + NULL`
- `5 * NULL - 7`
- `'Casa ' || 'dolce ' || NULL`
- `IlMioCampo = NULL`
- `IlMioCampo <> NULL`
- `NULL = NULL`

Se si ha proprio difficoltà a capire perchè, basta rammentare che NULL significa «sconosciuto». Guardiamo ora nella seguente tabella, dove sono spiegate caso per caso tutte le precedenti espressioni. In questa tabella non usiamo NULL nelle espressioni (come già accennato, questo è spesso illegale); invece usiamo due entità, diciamo A e B, che sono entrambe NULL. A e B possono essere campi, variabili, od intere sottoespressioni, le quali, fintantochè sono di valore NULL, si comportano sempre allo stesso modo nelle espressioni mostrate.

Tabella 1. Operazioni sulle entità null A and B

Se A e B sono NULL, allora:	vale	per questo motivo
`1 + 2 + 3 + A`	NULL	Se A è sconosciuto, allora anche 6 + A è sconosciuto.
`5 * A - 7`	NULL	Se A è sconosciuto, allora lo è anche 5 * A. Sottrargli 7 finisce per dare un altra quantità sconosciuta.
`'Casa ' || 'dolce ' || A`	NULL	Se A è sconosciuto, 'Casa dolce ' || A è sconosciuto.
`MioCampo = A`	NULL	Se A è sconosciuto, non si può dire se MioCampo ha lo stesso valore...
`MioCampo <> A`	NULL	...ma non si può neanche dire se MioCampo ha un valore *differente*!
`A = B`	NULL	Se A e B sono sconosciuti, è impossibile sapere se sono uguali.

Questa è la lista completa degli operatori matematici e su stringhe che riportano `NULL` se almeno uno degli operandi è `NULL`:

- `+`, `-`, `*`, `/`, e `%`
- `!=`, `~=`, e `^=` (sinonimo di `<>`)
- `<`, `<=`, `>`, e `>=`
- `!<`, `~<`, e `^<` (bassa precedenza, sinonimo di `>=`)
- `!>`, `~>`, e `^>` (bassa precedenza, sinonimo di `<=`)
- `||`
- [NOT] BETWEEN
- [NOT] STARTING WITH
- [NOT] LIKE
- [NOT] CONTAINING

Ogni spiegazione segue lo stesso schema: se A è sconosciuto, non si può dire che sia maggiore di B; se la stringa S1 è sconosciuta, non si può dire che contiene la stringa S2; e così via.

Usare LIKE con carattere di escape `NULL` fa in modo da mandare in crash il server nelle versioni Firebird fino alla 1.5 inclusa. Questo problema è stato risolto nella versione 1.5.1. Da quella versione in poi un tale comando riporta un insieme di record vuoto.

`NULL` nelle operazioni logiche

> **Nota**
>
> Nel seguito indicheremo spesso con la coppia di termini inglesi `true` e `false`, usate abitualmente in informatica, le relativa coppia di condizioni logiche «vero» e «falso».

Tutti gli operatori fin qui visti riportano `NULL` se qualche operando è `NULL`. Con gli operatori logici booleani, le cose sono un po' più complicate:

- `not NULL = NULL`
- `NULL or false = NULL`
- `NULL or true = true`
- `NULL or NULL = NULL`
- `NULL and false = false`
- `NULL and true = NULL`
- `NULL and NULL = NULL`

Firebird SQL non ha un tipo di dato boolean; nè `true` o `false` sono costanti definite. Nella colonna a sinistra della tabella successiva, «`true`» e «`false`» rappresentano delle espressioni con campi, variabili, ecc.., che valgono `true` oppure `false` (ad esempio: 1=1 è sempre «vero», 1=0 è sempre «falso»).

Tabella 2. Operazioni logiche sull'entità di valore null A

Se A è `NULL`, allora	vale	per questo motivo
`not (A)`	`NULL`	Se A è sconosciuto, è sconosciuto anche il suo inverso (o la sua negazione).
`A or (false)`	`NULL`	«`A or false`» ha sempre il valore di A - che è sconosciuto.
`A or (true)`	`true`	«`A or true`» è sempre `true`, perchè il valore di A non ha importanza.
`A or A`	`NULL`	«`A or A`» è sempre A, che è sconosciuto, cioè `NULL`.
`A and (false)`	`false`	«`A and false`» è sempre falso, perchè il valore di A non conta.
`A and (true)`	`NULL`	«`A and true`» ha sempre il valore di A - che è sconosciuto.
`A and A`	`NULL`	«`A and A`» è sempre A, che è sconosciuto, cioè `NULL`.

Tutti questi risultati sono in accordo con la logica booleana. Il fatto che non si ha la necessità di conoscere il valore di X per calcolare «`X or true`» e «`X and false`» è alla base di una caratteristica nota in molti linguaggi di programmazione col nome di `short-circuit boolean evaluation` (valutazione logica cortocircuitata).

I risultati precedenti possono essere generalizzati come segue per le espressioni con un tipo di operatore binario booleano (`and` | `or`) e qualsiasi numero di operandi:

Disgiunzioni («A or B or C or D or ...»)

1. Se almeno un operando è `true`, il risultato è `true`.
2. Altrimenti, se almeno un operando è `NULL`, il risultato è `NULL`.
3. Altrimenti (cioè se tutti gli operandi sono `false`) il risultato è `false`.

Congiunzioni («A and B and C and D and ...»)

1. Se almeno un operando è `false`, il risultato è `false`.
2. Altrimenti, se almeno un operando è `NULL`, il resto è `NULL`.
3. Altrimenti (cioè se tutti gli operandi sono `true`) il risultato è `true`.

O, più brevemente:

- `TRUE` sovrasta `NULL` nelle disgiunzioni (operazioni OR);
- `FALSE` sovrasta `NULL` nelle congiunzioni (operazioni AND);
- In tutti gli altri casi, vince `NULL`.

Se si fatica nel ricordare chi sovrasta che cosa in che operazione, meglio memorizzare la regoletta della seconda lettera: t**R**ue vince con o**R** — f**A**lse con **A**nd.

Ancora logica (anche se non sembra)

I risultati determinati dai cortocircuiti logici booleani di cui sopra possono suggerire le seguenti idee:

- 0 per `x` è uguale a 0 per qualsiasi `x`. Pertanto, anche se `x` è sconosciuto, `0 * x` vale 0. (N.B.: questo solo se il tipo di dato di `x` può contenere solo numeri, non `NaN` o infiniti.)

- La stringa vuota è in ordine lessicografico prima di ogni altra stringa. Pertanto, `S >= ''` è vero per qualsiasi valore di S.

- Ogni valore è uguale a sé stesso, sia conosciuto che incognito. Così, sebbene sia giustificato che `A = B` sia `NULL` quando A e B sono entità diverse di valore `NULL`, invece `A = A` dovrebbe sempre riportare `true`, anche quando A è `NULL`. Lo stesso vale per `A <= A` e `A >= A`.

  Analogicamente, `A <> A` dovrebbe sempre essere `false` così come `A < A` e `A > A`.

- Ogni stringa *contiene* sé stessa, *comincia con* sé stessa ed è *come* sé stessa. Così, «`S CONTAINING S`», «`S STARTING WITH S`» e «`S LIKE S`» dovrebbero sempre essere, o meglio valere, `true`.

Ebbene, come viene implementato questo in Firebird SQL? È mio dovere informare che, contrariamente al buon senso e all'analogia con i risultati dell'algebra booleana mostrati sopra, tutte le seguenti espressioni valgono `NULL`:

- `0 * NULL`
- `NULL >= ''` e `'' <= NULL`
- `A = A`, `A <= A` e `A >= A`
- `A <> A`, `A < A` e `A > A`
- `S CONTAINING S`, `S STARTING WITH S` e `S LIKE S`

Il fatto è che non vanno confusi gli operatori logici (quali `OR` e `AND`) con gli operatori quali la moltiplicazione ed il confronto che sono operatori aritmetici che hanno risultati rispettivamente numerici o booleani.

Funzioni interne ed altre direttive

Funzioni interne

Le seguenti funzioni integrate riportano `NULL` se almeno uno degli argomenti è `NULL`:

- CAST()
- EXTRACT()
- GEN_ID()
- SUBSTRING()
- UPPER()
- LOWER()
- BIT_LENGTH()
- CHAR[ACTER]_LENGTH()
- OCTET_LENGTH()
- TRIM()

Nota bene

- In Firebird 1.0.0, EXTRACT da un dato `NULL` fa crollare il server the server. Risolto nella 1.0.2.
- Se il primo argomento di GEN_ID è un nome valido di generatore ed il secondo argomento è `NULL`, il generatore mantiene il valore corrente.
- In versioni fino alla 2.0 inclusa, i risultati di SUBSTRING sono talvolta riportati come «false stringhe vuote». Queste stringhe di fatto sono `NULL`, ma sono descritte dal server come non annullabili. Pertanto molti programmi le mostrano come stringhe vuote. Vedere la lista dei problemi per una descrizione più approfondita.

FIRST, SKIP e ROWS

Le due direttive seguent **«mandano in crash»** un server Firebird 1.5.n o precedente se gli viene passato un argomento a `NULL`. In Firebird 2, invece, trattano il `NULL` come se fosse il valore 0:

- FIRST
- SKIP

Questa nuova direttiva di Firebird 2 non riporta righe (cioè il result set è vuoto) se un qualsiasi argomento è `NULL`:

- ROWS

Nota a latere: ROWS è conforme allo standard SQL. Nel codice è consigliato usare ROWS, e non FIRST e SKIP.

I predicati

I predicati sono frasi con oggetti che riportano un valore booleano: `true`, `false` oppure `sconosciuto` (= `NULL`). Nella programmazione si possono trovare i predicati nelle posizioni in cui bisogna prendere delle decisioni o fare delle scelte. Nel SQL di Firebird, ciò significa nelle frasi contenenti le clausole WHERE, HAVING, CHECK, CASE WHEN, IF e WHILE.

Anche i confronti come «x > y» danno come risultato valori booleani, ma non sono generalmente chiamati predicati, sebbene sia questione di forma. Un'espressione come Maggiore(x, y) che esprime la stessa cosa verrebbe immediatamente riconosciuta come predicato. I matematici preferiscono dare un *nome* ai predicati – come «Maggiore» o solo «M» – ed una coppia di *parentesi* per raccogliere gli argomenti.

Firebird supporta i seguenti predicati SQL: IN, ANY, SOME, ALL, EXISTS e SINGULAR.

Nota

È accettabile dire che «IS [NOT] NULL» e «IS [NOT] DISTINCT FROM» sono predicati, nonostante l'assenza di parentesi. Ma che siano predicati o no, ne abbiamo già parlato e non ne parleremo ancora in questa sezione.

Il predicato IN

Il predicato IN confronta l'espressione alla sua sinistra con le espressioni passate in una lista come argomenti e riporta `true` se trova una corrispondenza. NOT IN riporta sempre l'opposto di IN. Alcuni esempi del suo uso sono:

```
select NumAula, Piano from Classi where Piano in (3, 4, 5)
```

```
delete from Clienti where upper(NomeCliente) in ('IGNOTO', 'NN', 'BOH', '' )
```

```
if ( A not in (Var1, Var1 + 1, Var2, Var3 )) then ...
```

La lista può essere anche generata da una subquery ad un solo campo:

```
select Num, Nome, Classe from Studenti
  where Num in (select distinct Assegnatario from PrestitoLibri)
```

Con una lista vuota

Se la lista è vuota (ciò è possibile solo con una subquery), IN riporta sempre `false` e NOT IN sempre `true`, anche se l'espressione da valutare è `NULL`. Il senso è questo: anche se un valore è ignoto, sicuramente non c'è in una lista vuota.

Quando è `NULL` l'espressione di confronto

Se la lista non è vuota e l'espressione di test, che chiameremo «A» nei successivi esempi, è invece `NULL`, i predicati seguenti riporteranno sempre `NULL`, indipendentemente dalle espressioni nella lista:

- A IN (Expr1, Expr2, ..., Expr*N*)
- A NOT IN (Expr1, Expr2, ..., Expr*N*)

Il primo esempio può essere compreso pensando l'intera espressione come una catena di disgiunzioni (funzioni OR) di test sull'uguaglianza:

A=Expr1 or A=Expr2 or ... or A=Expr*N*

che pertanto, se A è `NULL`, diventa

`NULL` or `NULL` or ... or `NULL`

cioè `NULL`.

Il secondo predicato viene determinato dal fatto che «not (`NULL`)» è uguale a `NULL`.

Quando `NULL` è nella lista

Se A ha un valore proprio ma la lista contiene una o più espressioni `NULL`, le cose diventano un po' più complicate:

- Se almeno una delle espressioni della lista ha lo stesso valore di A:

  - «A IN(Expr1, Expr2, ..., Expr*N*)» vale `true`
  - «A NOT IN(Expr1, Expr2, ..., Expr*N*)» vale `false`

  Questo lo si deve al fatto che «`true` or `NULL`» vale `true` (vedi sopra). Più in generale: una disgiunzione dove almeno uno degli elementi è `true`, riporta `true` perfino se altri elementi sono `NULL`. Ovviamente tutti i `false` eventualmente presenti non interessano. In una disgiunzione interessa se c'è almeno un `true`

- Se nessuna delle espressioni in lista ha lo stesso valore di A:

  - «A IN(Expr1, Expr2, ..., Expr*N*)» vale `NULL`
  - «A NOT IN(Expr1, Expr2, ..., Expr*N*)» vale `NULL`

  Questo perchè «`false` or `NULL`» riporta `NULL`. Generalizzando si può dire che una disgiunzione che ha solo elementi `false` e `NULL`, vale `NULL`.

Inutile dire che se né A né ciascuna espressione in lista è `NULL`, il risultato è quello atteso e può essere solo o `true` o `false`.

I risultati di IN()

La tabella seguente mostra tutti i possibilirisultati per IN e NOT IN. Per usarla nel modo giusto, si parte dalla prima domanda a sinistra. Se la risposta è No, passare alla linea successiva. Se la risposta è Si, leggere il risultato dalla seconda o terza colonna, come appropriato, ed è tutto.

Tabella 3. Results for «A [NOT] IN (<list>)»

Conditions	Results	
	IN()	**NOT IN()**
La lista è vuota?	`false`	`true`
Altrimenti, A è `NULL`?	`NULL`	`NULL`
Altrimenti, esiste almeno un elemento della lista uguale ad A?	`true`	`false`
Altrimenti, almeno un elemento della lista è `NULL`?	`NULL`	`NULL`
Altrimenti tutti gli elementi della lista sono diversi da A e non `NULL`, `quindi`	`false`	`true`

In alcuni contesti (ad esempio nelle clausole IF e WHERE), un risultato `NULL` si comporta come se ci fosse `false` nel senso che la condizione non è soddisfatta quando l'espressione di controllo è `NULL`. Ciò conviene nei casi dove ci si aspetterebbe `false` ma invece c'è `NULL`: semplicemente non si nota la differenza. Il rovescio della medaglia è che ci si aspetterebbe `true` quando l'espressione viene invertita (usando NOT) ed invece si casca dall'asino nel senso che l'espressione più «pericolosa» fra i casi elencati nella tabella qui sopra è del tipo «A NOT IN (<lista>)», dove A non è presente nella lista (cioè ci si aspetterebbe `true`) ma nella lista succede che ci siano uno o più `NULL`.

Attenzione

Bisogna stare particolarmente attenti ad usare NOT IN in una subselect invece che in una lista esplicita, cioè:

```
A not in ( select Numeri from Tabella )
```

Se A non è presente nella colonna `Numeri`, il risultato è `true` se nessuno dei `Numeri` è `NULL`, ma è `NULL` se c'è un `NULL` fra i `Numeri`. Attenzione inoltre che perfino in situazioni dove A è costante ed il suo valore non è mai contenuto nella colonna `Numeri`, il risultato dell'espressione (e di conseguenza il flusso del programma) può cambiare nel tempo con la presenza o l'assenza di `NULL` nella colonna. Divertitevi a beccare l'errore in debug! Naturalmente il problema si evita facilmente con l'aggiunta di «where Numeri is not `NULL`» alla subselect.

Problema

Tutte le versioni precedenti alla 2.0 contengono un problema per cui [NOT] IN riporta il risultato errato se è attivo un indice nella subselect e risulta vera una delle seguenti condizioni:

- A è `NULL` e la subselect non riporta nessun `NULL`
- A non è `NULL` e la subselect non contiene A ma contiene dei `NULL`.

Notare che un indice può essere attivo anche se non è stato creato esplicitamente, ad esempio se è stata definita una chiave con il campo A.

Esempio: la tabella TA ha una colonna A con valori { 3, 8 }. La tabella TB ha una colonna B con i valori { 2, 8, 1, `NULL` }. Entrambe le espressioni

```
A [not] in ( select B from TB )
```

dovrebbero riportare `NULL` quando A = 3, a causa del `NULL` in B. Ma se B è indicizzato, IN riporta `false` e NOT IN riporta `true`. Come conseguenza, la query

```
select A from TA where A not in ( select B from TB )
```

riporta un dataset con un record di un campo contenente il valore 3, mentre avrebbe dovuto riportare un dataset vuoto (nessun record). Altri errori possono esser dietro l'angolo, cioè usando «NOT IN» in una frase o una clausola tipo IF, CASE o WHILE.

Come alternativa a NOT IN, si può usare «<> ALL». Il predicato ALL lo vedremo tra breve.

IN() nei controlli di CHECK

Il predicato IN() è usato spesso nei controlli di CHECK. In tali contesti, le espressioni `NULL` hanno un effetto sorprendentemente differente nelle versioni di Firebird 2.0 e successive. Questo verrà dettagliato nella sezione dedicata ai *controlli di CHECK*.

I quantificatori ANY, SOME e ALL

Firebird ha due quantificatori che permettono di comparare un valore al risultato di una subselect:

- ALL riporta `true` se il confronto è vero (cioè `true`) per *ogni* elemento della subselect.
- ANY e SOME (completamente sinonimi) sono veri se il confronto è `true` per *almeno uno* degli elementi della subselect.

Con ANY, SOME e ALL si può gestire meglio l'operatore di confronto. Questo rende il tutto più flessibile che con IN, che supporta solo l'operatore (implicito) «=». D'altra parte, ANY, SOME e ALL accettano solo subselect come argomento e non è possibile dare una lista di elementi esplicita come per IN.

Gli operatori validi sono `=`, `!=`, `<`, `>`, `=<`, `=>` e tutti i loro sinonimi. Non si possono usare LIKE, CONTAINING, IS DISTINCT FROM, ed altri operatorior.

Alcuni esempi d'uso tipico:

```
select nome, stipendio from operai
  where stipendio > any( select stipendio from dirigenti )
```

(riporta la lista degli operai che guadagnano più di almeno un dirigente)

```
select nome, provincia from operai
  where provincia != all( select distinct provincia from dirigenti )
```

(riporta la lista degli operai che non vivono in nessuna città in cui vive un dirigente)

```
if ( StipendioDirigente !> some( select stipendio from operai ) )
  then Dirigentaccio = 1;
  else Dirigentaccio = 0;
```

(mette Dirigentaccio a 1 se lo stipendio di almeno un operaio è non minore del valore di Stipendio-Dirigente)

I risultati di ANY, SOME e ALL

Se la subselect non riporta record, ALL vale `true` e ANY|SOME valgono `false`, anche se la parte sinistra dell'espressione è `NULL`. Questo segue naturalmente dalle definizioni e dalle regole della logica formale. Le menti matematiche avranno già notato che ALL è equivalente al quantificatore universale «#» e che ANY|SOME lo sono a quello esistenziale «#».

Per gli insiemi non vuoti, si può scrivere «A <*op*> ANY|SOME (<*subselect*>)» come

A <*op*> E1 **or** A <*op*> E2 **or** ... **or** A <*op*> E*n*

dove <*op*> è l'operatore usato e E1, E2 ecc. sono i valori riportati dalla subquery.

Allo stesso modo, «A <*op*> ALL (<*subselect*>)» è lo stesso di

A <*op*> E1 **and** A <*op*> E2 **and** ... **and** A <*op*> E*n*

Tutto questo ha l'aria familiare: la prima riscrittura è uguale a quella del predicato IN, cambia l'operatore che può essere ora diverso dal solito «=». La seconda è diversa ma ne mantiene la forma generale. Adesso si può determinare come in A e/o nel risultato della subselect, il `NULL` modifica il comportamento di ANY|SOME e ALL. Questo verrà mostrato allo stesso moso di come abbiamo visto prima per IN, solo che adesso invece di mostrare il procedimento passo passo, andiamo direttamente a presentare la tabella dei risultati. Al solito, leggere le domande nella colonna di sinistra dall'alto inbasso e, non appena la risposta è «si», ricavare il risultato dalla colonna di destra.

Tabella 4. Risultati per «A <op> ANY|SOME (<subselect>)»

Conditizioni	Risultati
	ANY | SOME
La subselect non ha riportato niente (cioè nessuna riga)?	
altrimenti, A è `NULL`?	`NULL`
altrimenti, uno dei confronti almento è vero, cioè riporta `true`?	`true`
altrimenti, uno dei confronti almeno riporta `NULL`?	`NULL`
altrimenti (tutti i confronti sono falsi, e riportano `false`)	`false`

A pensare che questi risultati siano proprio simili a quelli visti con IN(), non ci si sbaglia molto: con l'operatore «=», ANY è si comporta come IN. Allo stesso modo, «<> ALL» è equivalente a NOT IN.

Problemi rivisitati

Nelle versioni precedenti alla 2.0, «= ANY» aveva lo stesso problema di IN: in «certe» circostanze, si hanno risultati sbagliati con espressioni del tipo «NOT A = ANY(...)».

Il lato buono è che , «<> ALL» non ha quel problema e da' sempre i risultati giusti.

Tabella 5. Risultati per «A <op> ALL (<subselect>)»

Condizioni	Risultati
	ALL
La subselect non ha riportato niente (cioè nessuna riga)?	`true`
altrimenti, A è `NULL`?	`NULL`
altrimenti, uno dei confronti almento è falso, cioè riporta `false`?	`false`
altrimenti, uno dei confronti almeno riporta `NULL`?	`NULL`
altrimenti (tutti i confronti sono veri, e riportano `true`)	`true`

Problemi di ALL

Per quanto «<> ALL» faccia sempre il suo dovere, nondimeno ALL, in tutte le versioni di Firebird precedenti alla 2.0, non è scevro da problemi: qualsiasi altro operatore che non sia «<>», può dare risultati errati se è attivo un indice nella subselect, indipendentemente dalla presenza di `NULL`.

> **Nota**
>
> Parlando papale papale, il secondo punto in entrambe le tabelle («A è NULL?») è rindondante e potrebbe essere eliminato. Se A è NULL, tutti i confronti riportano NULL, pertanto la situazione potrebbe essere rinviata alle domande successive. E già che ci siamo, si potrebbe eliminare pure la prima domanda: il caso "senza righe" è un caso particolare dell'ultima. Il tutto ancora una volta diventa «true batte il NULL che batte il false» nelle disgiunzioni (ANY|SOME) e «false batte il NULL che batte il true» nelle congiunzioni (ALL).
>
> La ragione per includere tutti i dettagli è la praticità: si riesce a vedere a colpo d'occhio se non ci sono record, ed è ancor più facile verificare che la parte sinistra di un'espressione sia NULL piuttosto che valutare ogni singolo confronto dei risultati. Si può fare come si crede, saltare entrambi i punti o solo il secondo. Ad ogni modo, *non* va ignorata la prima domanda iniziando con la seconda: questo porta a conclusioni errate se la subselect è vuota!

EXISTS e SINGULAR

I predicati EXISTS e SINGULAR danno informazioni su una subquery, di solito correlata. Si possono usare nelle clausole WHERE, HAVING, CHECK, CASE, IF e WHILE (le ultime due sono disponibili solo in PSQL, il linguaggio di Firebird dedicato alle stored procedure ed ai trigger).

EXISTS

EXISTS dice se una subquery riporta almeno una riga di dati. Per ottenere ad esempio una lista di lavoratori che sono anche proprietari, si potrebbe avere una cosa del genere:

```
SELECT Lavoratore FROM Aziende WHERE EXISTS
  (SELECT * FROM Proprietari
   WHERE Proprietari.Nome = Aziende.Lavoratore)
```

Questa query riporta i nomi di tutti i lavoratori che sono anche nella tabella proprietari. Il predicato EXISTS riporta true se l'insieme di righe risultanti dalla subselect ne contiene almento una, se invece è vuoto, EXISTS riporta false. EXISTS non riporta mai NULL, in quanto un result set o c'ha righe o non ce n'ha. Può succedere che le condizioni di ricerca della subselect possano dare tutto a NULL per certe righe, ma non crea incertezze perchè tali righe non vengono incluse nel risultato.

> **Nota**
>
> La subselect in realtà non riporta nessun result set. Il sistema semplicemente scorre la subselect (la tabella Proprietari dell'esempio) record per record applicando la condizione di ricerca. Se trova che vale true, EXISTS riporta true immediatamente e i record rimanenti non vengono guardati. Se è false o NULL, prosegue con il successivo record. Se sono stati controllati tutti i record e non è stato trovato neanche l'ultimo controllo ha dato true, EXISTS riporta false.

NOT EXISTS da' sempre l'opposto di EXISTS: false oppure true, mai NULL. NOT EXISTS riporta false immediatamente non appena ha un risultato true nella condizione di ricerca. Per riportare true deve scorrersi tutti i record.

SINGULAR

SINGULAR è una estensione allo standard SQL di InterBase/Firebird. Si può dire che riporta true se nella subquery esattamente un record soddisfa la condizione. Per analogia con EXISTS ci si aspetterebbe che anche

SINGULAR possa riportare solo o `true` o `false`. Dopo tutto un result set o ha esattamente un record oppure ne ha un numero differente. Purtroppo tutte le versioni di Firebird fino alla 2.0 compresa hanno un problema che provoca dei risultati `NULL` in un certo numero di casi anomali. Il comportamento è abbastanza inconsistente, ma allo stesso tempo perfettamente riproducibile: per esempio, in una colonna contenente (1, `NULL`, 1), un test per SINGULAR con la condizione «`A=1`» riporta `NULL`, ma lo stesso test con i dati (1, 1, `NULL`) riporta `false`. Notare che cambia solo l'ordine di inserimento dei dati!

A render le cose peggiori, in tutte le versioni precedenti la 2.0 talvolta viene riportato `NULL` da NOT SINGULAR quando invece SINGULAR da' `false` o `true`. Nella 2.0, almeno questo non succede più: o è `false` invece di `true` o entrambi `NULL`.

Il codice è stato corretto nella versione 2.1 di Firebird; da quella versione in poi SINGULAR riporterà:

- `false` se la condizione di ricerca non è mai `true` (includendo il caso dell'insieme vuoto, cioè nessun record);
- `true` se la condizione di ricerca è vera per esattamente 1 record;
- `false` se la condizione di ricerca è vera per più di 1 record.

Se altri record sono `false`, `NULL` o una combinazione di questi, non ha più importanza.

NOT SINGULAR riporterà sempre l'esatto opposto di SINGULAR (come già succede nella 2.0, solo che il valore NULL è adesso impossibile).

Nel frattempo che la 2.1 diventi definitiva, se c'è anche una *qualsiasi* piccola possibilità che la ricerca possa riportare `NULL` per una o più righe, bisogna sempre aggiungere una condiziione di IS NOT NULL alla clausola [NOT] SINGULAR come, ad esempio

```
... SINGULAR( SELECT * from Tabella
              WHERE Campo > 38
              AND Campo IS NOT NULL )
```

Ricerche

Se la condizione di ricerca di una SELECT, UPDATE o DELETE risolve a `NULL` per certe righe, l'effetto è come se fosse `false`. Mettendola in altro modo, se l'espressione è `NULL`, la condizione non è raggiunta, e di conseguenza la riga non è inclusa fra quelle selezionate per la ricerca (o l'aggiornamento o la cancellazione).

> **Nota**
>
> Si dice *condizione di ricerca* (o *search condition* o *search expression*) la clausola WHERE senza la parola iniziale WHERE.

Alcuni esempi, con la condizione di ricerca evidenziata in grassetto:

```
SELECT Fattori, Mucche FROM Fattorie WHERE Mucche > 0 ORDER BY Mucche
```

La frase sopra riporta i Fattori che hanno registrata almeno una mucca. Quelli con un numero sconosciuto (`NULL`) non vengono inclusi, perche l'espressione «`NULL > 0`» riporta `NULL`.

```
SELECT Fattori, Micche FROM Fattorie WHERE NOT (Mucche > 0) ORDER BY Mucche
```

Adesso si sarebbe tentati di dire che questa riporti «tutti gli altri record» della tabella Fattorie, giusto? No, sbagliato. Almeno se la colonna Mucche contiene qualche `NULL`. Ci siamo scordati che `not(NULL)` è ancora

NULL? Pertanto per ogni reiga per la quale Mucche è NULL, «Cows > 0» sarà NULL, e «NOT (Cows > 0)» sarà pure NULL e quindi non sarà nel record set risultante.

```
SELECT Fattori, Mucche, Pecore FROM Fattorie WHERE Mucche + Pecore > 0
```

A prima vista sembra che questa sia una query che genera l'elenco di tutti i fattori che hanno almeno una nucca o una pecora (assumendo di non avere mucche o pecore negative...). Tuttavia se il fattore Antonio ha 30 mucche ed un numero sconosciuto di pecore, la somma Mucche + Pecore è NULL, e quindi l'tera condizione di ricerca risolve a «NULL > 0», che è... capito? Così, nonostante le sue 30 mucche, il caro amico Antonio non sarà mai nell'elenco risultante perchè non soddisfa la condizione di ricerca.

Per ultimo, riscriviamo l'esempio precedente in modo che riporti comunque tutti i fattori che hanno *almeno* un animale di un qualsiasi tipo, anche se la quantità dell'altro tipo è ignota, cioè NULL. Per farlo, approfittiamo del fatto che «NULL or true» riporta true – uno dei rari casi in cui un operando NULL non trasforma l'intera espressione in NULL:

```
SELECT Fattori, Mucche, Pecore FROM Fattorie WHERE Mucche > 0 OR Pecore > 0
```

In questo caso, le 30 mucche di Antonio rendono il primo confronto true, mentre quello con le pecore rimane a NULL. Così c'è «true or NULL», che è true, e la riga verrà inclusa nell'insieme risultante.

> **Attenzione**
>
> Se la condizione di ricerca contiene uno o più predicati IN, c'è la complicazione che alcuni degli elementi in lista (o risultati di una subselect) potrebbe essere NULL. Le imlicazioni di questo le abbiamo viste ne *Il predicato IN()*.

Gli ordinamenti

In Firebird 2.0, i NULL sono considerati «precedenti» a qualsiasi altra cosa nell'ordinamento. Di conseguenza vengono prima negli ordinamente ascendenti e ultimi negli ordinamenti discendenti. Questo può essere modificato aggiungendo la direttiva NULLS FIRST oppure NULLS LAST alla clausola ORDER BY.

Nelle versioni precedenti, i NULL erano sempre messi alla fine dell'insieme ordinato, indipendentemente dal fatto che fosse ascendente o discendente. In Firebird 1.0 la storia finiva qui: i NULL sono ultimi in qualsiasi ordinamento, punto. In Firebird 1.5 è stata introdotta la sintassi NULLS FIRST/LAST, per forzarli all'inizio o alla fine.

Per riassumere il tutto:

Tabella 6. Posizionamento dei NULL negli ordinamenti

Ordinamento	Posizione dei NULL		
	Firebird 1	Firebird 1.5	Firebird 2
order by ... [asc]	alla fine	alla fine	all'inizio
order by ... desc	alla fine	alla fine	alla fine
order by ... [asc | desc] nulls first	—	all'inizio	all'inizio
order by ... [asc | desc] nulls last	—	alla fine	alla fine

Naturalmente è inutile, per quanto corretto, specificare NULLS FIRST nel caso ascendente oppure NULLS LAST in quello discendente con Firebird 2. Lo stesso dicasi per NULLS LAST in qualsiasi ordinamento in Firebird 1.5.

Note

- Se si richiede un ordinamento dei `NULL` non default, non verrà utillizzato nessun indice. In Firebird 1.5, questo è il caso dei NULLS FIRST. In 2.0 e successivi, succede con NULLS LAST negli ordinamenti ascendenti e con NULLS FIRST in quelli discendenti.
- Aprendo un database pre-2.0 con Firebird 2, mostrerà il *vecchio* comportamento sull'ordinamento dei `NULL`, cioè alla fine, a meno di esplicito NULLS FIRST. Un ciclo di backup/restore rimette le cose a posto, se la restore viene effettuata con la gbak di Firebird!
- Firebird 2.0 ha un problema che provoca il fallimento della direttiva NULLS FIRST|LAST in certe circostanze insieme a SELECT DISTINCT. Vedere la lista dei problemi per maggiori dettagli.

Avvertimento

Non si deve essere tentati dal fatto che siccome `NULL` è la «cosa più piccola» negli ordinamenti prima di Firebird 2, si possa pensare che espressioni come «`NULL` < 3» possano ora riportare `true`. No! Usare `NULL` in questo tipo di espressioni darà sempre risultato `NULL`.

Le funzioni di aggregazione

Le funzioni di aggregazione – `COUNT`, `SUM`, `AVG`, `MAX`, `MIN` e `LIST` – non gestiscono `NULL` allo stesso modo delle funzioni ordinarie e degli operatori. Invece di riportare `NULL` non appena viene trovato un operando a `NULL`, queste considerano solo i campi non `NULL` per calcolare il risultato. Cioè se avete questa tabella:

Tabella		
ID	**Nome**	**Importo**
1	John	37
2	Jack	NULL
3	Jim	5
4	Joe	12
5	Josh	NULL

...eseguendo **`select sum(Importo) from Tabella`** riporta 54, che è appunto 37 + 5 + 12. Se fossero stati sommati tutti i cinque campi, il risultato sarebbe stato `NULL`. Per la funzione `AVG`, i campi non-`NULL` sono sommati tra loro, e la somma divisa per il numero dei campi non-`NULL`.

C'è una eccezione a questa regola: `COUNT(*)` riporta il numero di tutte le righe, perfino di quelle in cui tutti i campi sono `NULL`. Ma `COUNT(`*`NomeCampo`*`)` si comporta come le altre funzioni aggregate, nel senso che conta solo le righe in cui quello specifico campo non è `NULL`.

Un'altra cosa che vale la pena di sapere è che sia `COUNT(*)` sia `COUNT(FieldName)` non riportano mai `NULL`: se non ci sono righe nell'insieme risultato della **`SELECT...`**, entrambe valgono 0. Inoltre, anche `COUNT(NomeCampo)` vale 0 se tutti i campi *`NomeCampo`* nel risultato sono `NULL`. In questi casi tutte le altre funzioni di aggregazione riportano `NULL`. Bisogna avvertire che anche `SUM` riporta `NULL` se utilizzata su un insieme risultato vuoto, che è contrario alla logica comune: se non ci sono righe la media, il minimo ed il massimo non sono definiti, ma la somma è zero.

Ora mettiamo il tutto insieme in una tabella riassuntiva:

Tabella 7. I risultati delle funzioni aggregate in diverse situazioni

Funzione	**Risultati**		
	Insieme vuoto	**Insieme o colonna tutta a null**	**Altri casi**
COUNT(*)	0	Numero totale delle righe	Numero totale delle righe
COUNT(Campo)	0	0	Numero di righe dove Campo non è `NULL`
MAX, MIN	`NULL`	`NULL`	Il valore minimo o massimo trovato nella colonna
SUM	`NULL`	`NULL`	Somma dei valori non `NULL` della colonna
AVG	`NULL`	`NULL`	Media dei valori non `NULL` della colonna. Questo vale SUM(Campo) / COUNT(Campo).[a]
LIST[b]	`NULL`	`NULL`	Stringa ottenuta concatenando i valori non `NULL` della colonna separati da virgola

[a] Se Campo è un campo di tipo intero, AVG è sempre arrotondato verso 0. Esempio: 6 INT con somma -11 danno media -1, non -2.
[b] LIST è stata aggiunta in Firebird 2.1

La clausola GROUP BY

Una clausola GROUP BY non cambia la logica delle funzioni di aggregazione descritta sopra, eccetto che adesso è applicata a ciascun gruppo individualmente invece che all'insieme di righe risultante nel suo insieme. Supponiamo d'avere una tabella Dipendenti con dei campi Dipartimento e Salario che permettono entrambi `NULL`, e di lanciare questa query:

```
SELECT Dipartimento AS DIP, SUM(Salario) AS SOMMA FROM Dipendenti GROUP BY Dipartimento
```

Il risultato potrebbe essere una cosa del genere (Dipartimento a `<null>` può essere in cima o in fondo, dipende dalla versione di Firebird):

```
DIP                     SOMMA
====== ====================
<null>             219465.19
000                266643.00
100                155262.50
110                130442.81
```

```
115              13480000.00
120                   <null>
121                 110000.00
123                 390500.00
```

Notare che chi ha il dipartimento sconosciuto (NULL) è raggruppata inscieme, sebbene non si possa dire che abbiamo davvero lo *stesso valore* del campo Dipartimento. L'alternativa sarebbe stata dare a ciascuno di essi un «gruppo» a sé. Non solo questo avrebbe generato un grandissimo numero di linee al risultato, ma avrebbe degenerato lo scopo del *raggruppare*: quelle linee non sarebbero un raggruppamento, ma semplicemente «SELECT Dipartimento, Salario». Pertanto ha senso raggruppare i dipartimenti a NULL secondo il loro stato e gli altri secondo il loro valore.

Comunque, non è questo che interessa maggiormente. Che cosa ci dice la colonna SUM? Forse che tutti i salari sono diversi da NULL, eccetto nel dipartimento 120? No. Quel che si può dire è che tranne nel dipartimento 120, c'è nel database almeno un dipendente con salario noto. Ogni dipartimento *può* avere salari a NULL; nel 120 *tutti* i salari sono NULL.

Per saperne di più, aggiungendo una o più colonne COUNT(). Per sapere ad esempio il numero dei salari a NULL in ciascun gruppo si aggiunge una colonna «COUNT(*)» oppure «COUNT(Salary)».

Countare le frequenze

Una clausola GROUP BY può essere usata per trovare le frequenze con cui i valori ricorrono in una tabella. In questo caso si usa lo stesso nome campo più volte nella query. Con una tabella TT che ha una colonna A i cui valori sono { 3, 8, NULL, 6, 8, -1, NULL, 3, 1 }. Per avere la lista delle frequenze, si può fare:

```
SELECT A, COUNT(A) FROM TT GROUP BY A
```

che dà il seguente risultato:

```
           A        COUNT
============ ============
          -1            1
           1            1
           3            2
           6            1
           8            2
      <null>            0
```

Oh! Qualcosa non va col conteggio dei NULL, ma perchè? Rivediamo: COUNT(*Nomecampo*) salta tutti i campi a NULL, pertanto con COUNT(*A*) il conteggio del gruppo <null> può essere solo 0! Riformulando la query in questo modo:

```
SELECT A, COUNT(*) FROM TT GROUP BY A
```

e si otterrà il valore corretto (nel caso 2).

La clausola HAVING

La clausola HAVING può aggiungere restrizioni ad una query con aggregati, così come la WHERE fa nelle query record per record. La clausola HAVING impone condizioni su ogni colonna del risultato o combinazioni di colonne, che siano aggregate o meno.

Per quanto riguarda i NULL, vanno tenuti ben presenti i seguenti due fatti (e credo di non meravigliare nessuno ormai):

- Le righe per le quali la condizione di HAVING risolve a NULL non sono incluse nel risultato finale. («Va bene solo il true.»)
- «HAVING <col> IS [NOT] NULL» è una condizione lecita e spesso utile, indipendentemente dal fatto che <col> sia raggruppata o meno. (Se <col> non è aggregata, si può risparmiare un po' di lavoro al server cambiando HAVING in WHERE e mettendolo nelle condizioni prima del «GROUP BY». Questo vale per tutte le condizioni su colonne non di aggregazione.)

Aggiungendo ad esempio la seguente clausola alla query d'esempio del paragrafo «GROUP BY»:

```
...HAVING Dipartimento IS NOT NULL
```

si impedisce alla prima riga di essere stampata, mentre con quest'altra:

```
...HAVING SUM(Salario) IS NOT NULL
```

sopprima la sesta riga (quella del Dipartimento = 120).

Frasi condizionali e cicliche

NULL negli statement di IF

Se l'espressione di test di uno statement IF risolve a NULL, la clausola THEN viene saltata e la parte ELSE (se presente) viene eseguita. In altre parole, NULL e false di comportano allo stesso modo in questo contesto, pertanto nelle situazioni in cui si attende false ma arriva NULL, non ci sono problemi. Però abbiamo già visto esempi di NULL arrivati al posto di true e quindi possono succedere cose strane perchè questo modificherebbe *completamente* il funzionamento del programma!

Gli esempi seguenti mostrano alcuni fra i diabolici (e quindi perfettamente logici) effetti del NULL negli statement di IF.

> **Suggerimento**
>
> Usando Firebird 2 o successivi, si possono raggirare tutte le trappole di cui qui stiamo parlando, semplicemente usando l'opratore [NOT] DISTINCT invece di «=» e «<>» !

- ```
  if (a = b) then
    Variabile = 'uguali';
  else
    Variabile = 'diversi';
  ```

  Se a e b sono entrambi NULL, la Variabile sarà «diversi» dopo aver eseguito il codice. La ragione è che l'espressione «a = b», come abbiamo visto precedentemente, vale NULL se almeno uno dei termini è NULL. Con l'espressione di test che vale NULL, il blocco then non viene eseguito, ed invece viene eseguito il blocco else.

- ```
  if (a <> b) then
  ```

```
  Variabile = 'diversi';
else
  Variabile = 'uguali';
```

In questo caso, `Variabile` diventerà «`uguali`» se `a` vale `NULL` ed invece `b` no, oppure viceversa. La spiegazione è analoga a quella dell'esempio precedente.

Allora come si può mettere in piedi un test di egualglianza che dia *effettivamente* il risultato aspettato in tutti i casi, anche con operandi a `NULL`? In Firebird 2 abbiamo già visto che si può usare DISTINCT (in *Controllare la diversità (Firebird 2+)*). Nelle precedenti versioni bisogna scrivere un po' di codice. Questo è mostrato nella sezione *Controlli di eguaglianza*, più avanti. Per ora è sufficiente ricordare che bisogna andare con i piedi di piombo con gli IF nei condizionali se possono trasformarsi in `NULL`.

Altro aspetto da non scordare: un'espressione di controllo a `NULL` può *comportarsi* come un `false` in una IF, ma *non vale* `false`. È sempre ed ancora `NULL`, ciò significa che la sua negazione è ancora (oibò!) `NULL` e non «`true`». Come conseguenza, negare l'espressione di controllo e contemporaneamente scambiare le parti THEN ed ELSE fra loro può cambiare il comportamento dello statement IF. Nella logica binaria, dove esistono solo `true` e `false`, una cosa del genere non potrebbe mai accadere.

Per mostrare questo fatto, riformuliamo l'ultimo esempio in questo modo:

- ```
  if (not (a <> b)) then
    Variabile = 'uguali';
  else
    Variabile = 'diversi';
  ```

  Nella versione originale, se un operando fosse stato `NULL` e l'altro no (quindi intuitivamente diversi) il risultato sarebbe stato «`uguali`». Qui invece è «`diversi`». Spiegazione: un operando è `NULL`, pertanto «`a <> b`» è `NULL`, pertanto «`not(a <> b)`» è ancora `NULL`, e quindi viene eseguito l'ELSE. Ma non c'è modo di gioire del fatto che questo risultato è corretto mentre il precedente non lo era: in questa versione riformulata, il risultato è «`diversi`» se sono entrambi `NULL`, mentre la versione originale almeno questo lo «imbroccava» giusto.

Naturalmente, finchè nessuno dei due operandi nell'espressione di test rischia di essere `NULL`, si possono fare gli IF come sopra. Inoltre, negando il test e contemporaneamente scambiando le parti THEN e ELSE continua a funzionare giusto, per quanto complessa sia la frase, finchè restano diversi da `NULL` gli operandi. È tremendamente perfido quel che succede quando gli operandi sono *quasi sempre* diversi da `NULL`, cosicchè nella stragran maggioranza dei casi fila tutto liscio, In queste situazioni, qualche raro `NULL` vagante può rimanere nascosto per molto tempo, rovinando i dati silenziosamente.

Le frasi con CASE

Firebird ha introdotto il costrutto CASE nella versione 1.5, con due varianti sintattiche. La prima, detta *sintassi semplice*;

```
case <espressione>
  when <espr1> then <valore1>
  when <espr2> then <valore2>
  ...
  [else <valoredefault>]
end
```

Questa funziona più o meno come il `case` del Pascal o lo `switch` del C: l'*<espressione>* viene confrontata con *<espr1>*, *<espr2>* ecc., finchè non viene trovata una corrispondenza, nel qual caso viene riportato il
```

valore associato. Se non c'è corrispondenza e c'è la clausola ELSE, si riporta il *<valoredefault>*. Se manca anche la clausola ELSE, il valore è un bel NULL.

Importante è sapere che i confronti sono fatti proprio con l'operatore «=», per cui se *<espressione>* è NULL, ignora tutte le *<esprN>*. In questo caso, per avere un risultato non nullo, bisogna adoperare la clausola ELSE.

Ad ogni modo, è corretto specificare NULL (o una qualsiasi espressione valida che possa valere NULL) per il valore risultante.

La seconda sintassi, o *sintassi analitica*:

```
case
  when <condizione1> then <valore1>
  when <condizione2> then <valore2>
  ...
  [else <valoredefault>]
end
```

Qui, le varie *<condizioneN>* sono confronti che possono dare uno dei tre possibili risultati: true, false, oppure NULL. Ancora una volta, solo true va bene, per cui una condizione come «A = 3» o perfino «A = null» non viene soddisfatta quando A è NULL. Ricordando che «IS [NOT] NULL» non riporta mai NULL, se A è NULL, la condizione «A is null» riporta true ed il corrispondente *<valoreN>* viene riportato. In Firebird 2+ si può usare anche «IS [NOT] DISTINCT FROM» nelle condizioni, in quanto anche questo operatore non riporta mai NULL.

Le cicliche WHILE

Quando si valuta la condizione di una ciclica WHILE, NULL si comporta come in una frase IF: se la condizione risolve a NULL, non si rientra nel ciclo, come se fosse false. Ancora una volta, vediamo cosa succede con l'inverso, cioè usando NOT. In una condizione come

```
while ( Contatore > 12 ) do
```

che salterebbe tutto il blocco del ciclo se Contatore è NULL, negandola con

```
while ( not Contatore > 12 ) do
```

farà lo stesso. Può essere che entrambe le situazioni vadano nel verso desiderato, solo che è neccessario essere informati del fatto che questi controlli apparentemente complementari in realtà si comportano allo stesso modo.

Le cicliche FOR

Per evitare ogni possibile confusione, bisogna evidenziare che i cicli di FOR nel PSQL di Firebird hanno una funzione completamente diversa dai cicli di WHILE, o dai cicli di **for** nei linguaggi di programmazione in generale. Il ciclo di FOR in Firebird ha la forma:

```
for <select-statement> into <var-list> do <code-block>
```

ed esegue il blocco di codice tante volte quante sono le righe lette attraverso la SELECT. Continua finchè non vengono lette tutte le righe, a meno che non intervenga una eccezione oppure uno statement tipo BREAK, LEAVE or EXIT. Leggere un NULL, o una riga di campi tutti a NULL, *non* ferma il ciclo!

Chiavi ed indici univoci

Le chiavi primarie

I NULL non sono mai permessi nelle chiavi primarie. Una colonna che sia (parte di) una chiave primaria (*primary key* o PK) deve essere definita come NOT NULL o nella colonna stessa o nella definizione del dominio. Notare che un vincolo «CHECK (XXX IS NOT NULL)» non funziona: è proprio necessaria la specifica NOT NULL subito dopo il tipo di dato.

> **Avvertimento**
>
> Firebird 1.5 ha un problema che permette la definizione di chiavi primarie su colonne NOT NULL che possono contenere valori NULL. Come questo possa succedere, e ci possano essere NULL in colonne del genere lo spieghiamo più avanti.

Chiavi ed indici univoci

Firebird 1.0

In Firebird 1.0, le *chiavi* univoche sono soggette alle stesse restrizioni delle chiavi primarie: la colonna o le colonne che ne fanno parte devono essere definite come NOT NULL. Per gli *indici* univoci, questo non è necessario. Tuttavia, quando viene creato un indice univoco, la tabella non deve contenere nè NULL o valori duplicati nei campi coinvolti, altrimenti la creazione dell'indice fallisce. Una volta creato, risulta impossibile inserire NULL o valori duplicati.

Firebird 1.5 e successivi

Da Firebird 1.5, lke chiavi univoche e gli indici univoci non solo permettono i NULL, ma perfino permettono NULL molteplici. Con una chiave o un indice di una sola colonna, si possono iserire quanti NULL si vuole in quella colonna, ovviamente ogni valore non NULL si può inserirer solo una volta.

Se la chiave o l'indice risulta definita su più di una colonna in Firebird 1.5 e successivi:

- Si possono inserire più righe dove tutte le colonne della chiave o indice sono NULL;
- Non appena uno o più colonne della chiave non sono NULL, ogni combinazioni di valori diversi da NULL deve essere univoca nella tabella. Ovviamente va compreso che (1, NULL) non è la stessa cosa di (NULL, 1).

Le chiavi esterne

Le chiavi esterne in quanto tali non impongono restrizioni ai NULL. Le colonne delle chiavi esterne devono sempre riferirsi a colonne (o insiemi di colonne) che sono chiavi primarie o univoche. Un indice univoco con la colonna o le colonne riferite non è sufficiente.

> **Nota**
>
> Nelle versioni fino alla 2.0 inclusa, cercando di creare una chiave esterna su una destinazione che non è chiave primaria nè una chiave univoca, Firebird lamenta il fatto che non ci riesce a trovare nessun *indice* univoco anche se un tale indice esiste. In Firebird 2.1, il messaggio correttamente informa che non vi esiste nessuna *chiave* univoca o primaria.

Ad ogni modo, anche se i `NULL` sono assolutamente proibiti nella chiave di destinazione (per esempio quando sono verso chiavi primarie), una qualsiasi colonna appartenente ad una chiave esterna può acora contenere `NULL`, a meno che non sia impedito da vincoli addizionali.

Vincoli di controllo (CHECK constraints)

È stato spesso detto in questa guida che quando un'espressione di confronto riporta `NULL`, ha lo stesso effetto di `false`: la condizione non è cioè soddisfatta. A partire da Firebird 2, questo **noè più vero** per i vincoli di controllo, cioè i «CHECK constraint». Per conformità allo standard SQL, un vincolo di CHECK adesso **è soddisfatto** se la condizione vale `NULL`. Solo un vero e proprio `false` provoca un rigetto del dato immesso.

In pratica, questo signica che controlli come

```
check ( value > 10000 )
```

```
check ( upper( value ) in ( 'A', 'B', 'X' ) )
```

```
check ( value between 30 and 36 )
```

```
check ( ColA <> ColB )
```

```
check ( Comune not like 'Mila%' )
```

...non accettano un dato `NULL` in Firebird 1.5, ma lo fanno passare in Firebird 2. Gli script di creazione per database già esistenti hanno bisogno di essere attentamente riesaminati prima di riusarli in Firebird 2. Bisogna riadattarli nel caso in cui un dominio o una colonna non ha un vincolo NOT NULL, e il relativo controllo di CHECK può diventare `NULL`, il che succede spesso ma non esclusivamente quando il dato in ingresso è `NULL`. Il vincolo di controllo può essere ad esempio esteso così:

```
check ( value > 10000 and value is not null )
```

```
check ( Comune not like 'Mila%' and Comune is not null )
```

Tuttavia, è più semplice e più chiaro aggiungere un NOT NULL alla definizione del dominio o della colonna:

```
create domain DSTIPENDIO int not null check ( value > 10000 )
```

```
create table Luoghi
(
  Comune varchar(24) not null check ( Comune not like 'Mila%' ),
  ...
)
```

Se gli script o i database devono funzionare in modo consistente in tutti i server vecchi e nuovi, bisogna fare in modo da evitare che qualsiasi vincolo di CHECK possa trasformarsi in `NULL`. Si può aggiungere «`or ...`

`is null`» per permettere un `NULL` in input nelle versioni più vecchie. Si può aggiungere un vincolo di NOT NULL oppure restrizioni come «`and ... is not null`» per impedirne l'accettazione esplicitamente nelle versioni di Firebird più recenti.

SELECT DISTINCT

Una frase del tipo SELECT DISTINCT considera tutti i `NULL` uguali fra loro (vale a dire che sono NOT DISTINCT FROM l'uno dall'altro), cosicchè se la selezione è su una sola colonna, al massimo può riportare solo un `NULL`.

Come già detto, Firebird 2.0 ha un problema legato alla direttiva NULLS FIRST|LAST che non funziona in certi casi con il SELECT DISTINCT. Per maggiori dettagli si può vedere nella lista dei problemi.

Funzioni definite dall'utente (UDF)

Le *funzioni definite dell'utente*, o *UDF* (*User Defined Functions*) sono funzioni che non sono interne al motore, ma definite in moduli separati. Firebird arriva con due librerie UDF: `ib_udf` (ampiamente usata dai tempi di InterBase) e `fbudf`. Si possono aggiungere altre librerie, comprandole o scaricandole da internet, oppure scrivendosele in un qualche linguaggio di programmazione idoneo. Le UDF non possono essere usate così come sono; devono essere prima «dichiarate» al database. Questo è vero anche per le UDF che vengono con Firebird.

Conversioni `NULL` <–> non-`NULL` non richieste

Non fa parte degli scopi di questa guida insegnare, usare o scrivere le UDF. Tuttavia è necessario avvertire che le UDF possono occasionalmente effettuare inaspettatamente delle conversioni da e verso `NULL`. Questo comporta che alle volte un `NULL` in input possa essere convertito in un valore regolare particolare, ed altre volte che un valore valido come una stringa vuota venga `null`ificato.

La causa principale di questo problema è che il modo di chiamare le UDF di «vecchio tipo» (ereditata da Interbase) non è in grado di passare un `NULL` in input ad una funzione. Quando una funzione UDF quale ad esempio `LTRIM` (elimina gli spazi eccedenti a sinistra) viene chiamata con un argomento `NULL`, l'argomento viene passato alla funzione come una stringa vuota. (Nota: in Firebird 2 e successivi, *può* essere passata come un puntatore nullo; ma ci ritorniamo più oltre). Dall'interno della funzione *non esiste nessun modo* per determinare se l'argomento è veramente una stringa vuota oppure un `NULL`. Pertanto che dovrebe fare chi scrive la routine? Deve fare una scelta: o prendere il valore come gli arriva o assumere che fosse originariamente un `NULL` e trattarlo di conseguenza.

Se il tipo risultato di una funzione è un puntatore, è possibile ottenere `NULL` da una funzione anche se non è possibile passarglielo. A tal punto che potrebbero accadere i seguenti fenomeni inattesi:

- Si chiama una UDF con un argomento a `NULL`. Viene passato alla funzione con un valore, cioè 0 oppure `''`. All'interno della funzione questo valore non è ricambiato in `NULL`; pertanto viene riportato un risultato non-`NULL`.
- Si chiama una UDF con un argomento valido come 0 oppure `''`, questo viene passato come è, ovviamente. Ma il codice della funzione suppone che sia invece la rappresentazione di un `NULL`, lo tratta come un buco nero, e riporta al chiamante un `NULL`.

Entrambe le conversioni sono normalmente indesiderate, ma la seconda probabilmente lo è di più della prima (è meglio validare qualcosa che è NULL piuttosto che NULLificare qualcosa di valido). Per tornare al nostro esempio della `LTRIM`: in Firebird 1.0 questa funzione riporta `NULL` per una stringa vuota, che è sbagliato; nella versione 1.5, non riporta mai `NULL`: in questa versione, anche le stringhe `NULL` sono cambiate in stringhe vuote. Intediamoci, pure questo è sbagliato, ma è considerato il minore dei due mali. In Firebird 2 finalmente funziona bene: una stringa NULL dà un risultato NULL, una stringa vuota viene ridotta ad una stringa vuota, semprechè si dichiari la funzione nella giusta maniera.

Descrittori

Fin da Firebird 1.0, è stato introdotto un nuovo metodo di passare gli argomenti ed i risultati con le UDF: «per descrittore». I descrittori permettono di segnalare la presenza di un `NULL` indipendentemente dal tipo di dato. La libreria `fbudf` fa ampio uso di questa tecnica. Sfortunatamente, usare i descrittori è un po' complicato; è necessario più lavoro ed è men o piacevole per lo sviluppatore dell'UDF. D'altra parte si è in grado di risolvere tutti i tradizionali problemi legati al `NULL`, e per il chiamante è semplice come per le UDF standard di vecchio stile.

Miglioramenti di Firebird 2

Firebird 2 ha migliorato il sistema di chiamare le UDF di vecchio tipo. Il sistema adesso passa un `NULL` di input alla funzione come un puntatore nullo, **se** la funzione è stata dichiarata al database con una parola chiave NULL dopo l'argomento in questione:

```
declare external function ltrim
  cstring(255) null
  returns cstring(255) free_it
  entry_point 'IB_UDF_ltrim' module_name 'ib_udf';
```

Questa richiesta assicura che i database esistenti e le loro relative applicazioni continuino a funzionare come prima. Omettendo la parola chiave NULL la funzione si comporterà esattamente come avrebbe fatto in Firebird 1.5.

Notare che non è sufficiente aggiungere il NULL alla dichiarazione e pensare che una qualsiasi funzione gestisca correttamente il `NULL` in ingresso. Ogni funzione deve essere anche riscritta in modo tale da gestire correttamente il `NULL` nel nuovo modo. Bisogna sempre osservare le dichiarazioni date dallo sviluppatore della funzione. Per le funzioni della libreria `ib_udf`, consultare il file `ib_udf2.sql` che è nel sottodirettorio `UDF` di Firebird. Notare il `2` nel nome del file; le dichiarazioni vecchio stile sono in `ib_udf.sql`.

Le seguenti funzioni della libreria `ib_udf` sono state aggiornate per riconoscere il `NULL` in input e gestirlo in modo proprio:

- `ascii_char`
- `lower`
- `lpad` e `rpad`
- `ltrim` e `rtrim`
- `substr` e `substrlen`

Molte funzioni `ib_udf` rimangono tali e quali; in ogni caso, passare `NULL` ad una UDF di vecchio stile è impossibile se l'argomento non è di tipo per riferimento.

Una nota a latere: è sconsigliato usare le funzioni UDF `lower`, `.trim` e `substr*` nel nuovo codice; è preferibile invece usare le funzioni interne LOWER, TRIM e SUBSTRING.

«Aggiornare» le funzioni della libreria `ib_udf` in un database esistente

Usando sotto Filrebird 2 una o più funzioni listate sopra con un database preesistente, e volendo beneficiare della nuova gestione dei `NULL`, bisogna eseguire nel proprio database i comandi preparati in `ib_udf_upgrade.sql`. Lo si può trovare nella directory di Firebird `misc\upgrade\ib_udf`.

Prepararsi alle conversioni non desiderate

Le conversioni non richieste `NULL` <-> non-`NULL` descritte precedentemente normalmente accadono con le UDF compatibili, ma ce ne sono molte in giro (sopratutto nella `ib_udf`). Inoltre nulla può fermare un programmatore superficiale dal ricadere nello stesso errore in una funzione nel nuovo stile. Pertanto il comportamento più sicuro da tenere se si usa una UDF e non si conosce come si comporta nei confronti del `NULL`:

1. guardare alla sua dichiarazione per controllare come sono passati e riportati i suoi valori. Se dice «by descriptor», si comporta in modo corretto (sebbene non fa mai male assicurarsene). Idem se sono seguiti dalla parola chiave NULL. In tutti gli altri casi, verificate i passi seguenti.

2. Avendone i sorgenti, e sapendo leggere il linguaggio in cui è scritto (C, C++, Delphi,...) controllare cosa fa il codice della funzione.

3. Valutare la funzione con valori `NULL` e con valori come 0 (per argomenti numerici) e/o `''` (per stringhe).

4. Se la funzione effettua una conversione indesiderata `NULL` <-> non-`NULL`, bisogna girarci intorno nel proprio codice prima di chiamare la UDF (vedere anche *Come controllare se ci sono `NULL`* , altrove in questa guida).

Le dichiarazioni per le UDF rilasciate possono essere trovate nel sottodirettorio di Firebird `examples` (per la versione 1.0) oppure `UDF` (dalla versione 1.5 in poi): si trovano nei files con estensione `.sql`

Altre informazioni sulle UDF

Per ottenere informazioni più approfondite sulle UDF, si possono consultare i documenti (al momento tutti in inglese) *InterBase 6.0 Developer's Guide* (gratuito, scaricabile da http://www.ibphoenix.com/downloads/60DevGuide.zip), *Using Firebird* e la *Firebird Reference Guide* (entrambi su CD), oppure il *Firebird Book*. I CD ed il libro possono essere acquistati attraverso http://www.ibphoenix.com.

Convertire da e verso `NULL`

Sostituire `NULL` con un valore

La funzione `COALESCE`

In Firebird 1.5 e successivi c'è una funzione che è in grado di convertire `NULL` a quasi qualsiasi altra cosa. Questo permette di fare una conversione al volo e di usare il risultato per ulteriori elaborazioni, senza l'uso del

costrutto «`if (Espressione is null) then`». Questa particolare funzione è la `COALESCE` e si usa in questo modo:

```
COALESCE( Espr1, Espr2, Espr3, ... )
```

`COALESCE` riporta il valore della prima espressione diversa da `NULL` della lista degli argomenti. Se tutte le espressioni sono `NULL`, la `COALESCE` riporta `NULL`.

Questo è il modo in cui per esempio si può usare `COALESCE` per ricostruire il nome completo di una persona a partire da nome, cognome e titolo, assumendo che talvolta il campo Titolo possa essere `NULL`:

```
select coalesce ( Titolo || ' ', '' )
       || Cognome
       || ' ' || Nome
from Persone
```

Oppure, per creare un nominativo il più informale possibile a partire da una tabella che contenga anche i soprannomi, e assumendo che possano essere `NULL` sia i soprannomi che i nomi:

```
select coalesce (Soprannome, Nome, 'Sig./Sig.ra')
       || ' ' || Cognome
from AltrePersone
```

`COALESCE` può venire in aiuto solo in quelle situazioni in cui il valore `NULL` può essere gestito allo stesso modo degli altri valori permessi per il tipo di dato. Se `NULL` necessità di una gestione particolare, diversa da ogni altro valore, l'unica opzione è usare un costrutto con IF o con CASE.

Firebird 1.0: le funzioni `*NVL`

In Firebird 1.0 non c'è la funzione `COALESCE`. Tuttavia si possono usare quattro funzioni UDF che permettono una buona parte della sua funzionalità. Queste UDF risiedono nella libreria `fbudf` e sono precisamente:

- `iNVL`, per argomenti interi
- `i64NVL`, per argomenti bigint
- `dNVL`, per argomenti in duplice precisione
- `sNVL`, per le stringhe

Le funzioni `*NVL` hanno esattamente due argomenti. Come `COALESCE`, riportano il primo argomento se non è `NULL`; altrimenti, riportano il secondo. Notare che in Firebird 1.0 la libreria `fbudf` e pertanto le funzioni `*NVL` sono solo disponibili in ambiente Windows.

Convertire valori a `NULL`

Talvolta è utile avere certi valori a `NULL` in uscita, o cime valori intermedi. Non capita spesso, ma potrebe essere utile, per esempio, per escludere certi valori da una sommatoria o da una media. Le funzioni `NULLIF` possono fare questo, anche se per un valore alla volta.

Firebird 1.5 e successivi: la funzione `NULLIF`

La funzione interna NULLIF ha due argomenti: se sono uguali, la funzione riporta `NULL`. Altrimenti riporta il valore del primo argomento.

Un tipico esempio è

```
select avg( nullif( Peso, -1 ) ) from Ciccioni
```

che darà il peso medio della popolazione dei Ciccioni, senza contare quelli con peso -1, ricordando che le funzioni di aggregazione come AVG escludono dal conteggio tutti i campi a `NULL`.

Lavorando un po' su questo esempio, supponiamo che finora si fosse usato -1 per indicare «peso sconosciuto» perchè non sicuri sull'uso di `NULL`. Dopo aver letto questa guida, si sarà pratici abbastanza da dare il comando:

```
update Ciccioni set Peso = nullif( Peso, -1 )
```

A questo punto, finalmente i pesi sconosciuti saranno realmente *sconosciuti*.

Firebird 1.0: Le UDF `*nullif` UDF

Firebird 1.0.x non ha la funzione interna `NULLIF`. Al suo posto ha quattro funzioni UDF nella libreria `fbudf` per ottenere il medesimo obiettivo:

- `inullif`, per argomenti interi
- `i64nullif`, per argomenti bigint
- `dnullif`, per argomenti in duplice precisione (double precision)
- `snullif`, per argomenti stringhe

Notare che la libreria di Firebird 1.0 `fbudf` e quindi anche l'insieme di funzioni`*nullif` è disponibile solo in Windows.

> **Avvertimento**
>
> Le note di rilascio di Firebird 1 dicono che, a causa di una limitazione del sistema, queste UDF riportano un valore equivalente a zero se gli argomenti sono identici. Ciò è sbagliato: se gli argomenti hanno lo stesso valore, le funzioni riportano un vero e proprio `NULL`.
>
> Notare che riportano `NULL` anche quando il primo valore è un valore vero e proprio ed il secondo argomento è `NULL`. Questo è un risoltato sbagliato: invece le funzioni interne `NULLIF` di Firebird 1.5 correttamente riportano il primo argomento.

Modifica delle tabelle piene di dati

Se le tabelle contengono già dati, e si desidera aggiungere una colonna non null o modificare la annullabilità di una colonna preesistente, ci sono alcune conseguenze di cui bisogna tenere conto. Nelle prossime sezioni vedremo le varie possibilità in dettaglio.

Aggiungere un campo NOT NULL ad una tabella con dati preesistenti

Supponendo di avere questa tabella:

Tabella 8. Tabella avventure

Nome	Data_acquisto	Prezzo
Goffredo di Buglione	12-06-1995	€ 49,00
Giuseppe Garibaldi	19-10-1995	€ 54,95

In questa tabella di improbabili giochi ci sono già alcune registrazioni quando si decide di dover aggiungere un campo ID sempre diverso da `NULL`. Ci sono due diverse soluzioni, ma ciascuna si porta problemi specifici da valutare.

Aggiungere un campo NOT NULL

Questo è il metodo preferito, forse perchè più immediato, ma provoca tutta una serie di problemi se usato in una tabella con dati già presenti come vedremo. Innazitutto, si supponga di aggiungere un campo con il seguente comando:

```
alter table Avventure add id int not null
```

Dopo la conferma (commit), il nuovo campo ID in tutte le righe già esistenti, avrà valore `NULL`. In questo caso speciale, Firebird permette pertanto la presenza di dati non validi in una colonna NOT NULL. Non solo, è in grado di farne il backup senza problemi apparenti, ma si rifiuterà di recuperare i dati salvati, precisamente perchè essi violano il vincolo NOT NULL.

> **Nota**
>
> Firebird 1.5 (ma non 1.0 o 2.0) permettono perfino di rendere tale colonna una chiave primaria!

Valorizzazione errata di `NULL` come se fosse zero

A rendere le cose anche peggiori, Firebird mente quando si leggono i dati dalla tabella. Con isql e molti altri programmi, «SELECT * FROM AVVENTURE» riporta questo insieme di dati:

Tabella 9. Dati riportati dopo aver aggiunto una colonna NOT NULL

Nome	Data_acquisto	Prezzo	ID
Goffredo di Buglione	12-06-1995	€ 49,00	0
Giuseppe Garibaldi	19-10-1995	€ 54,95	0

Naturalmente questo farà pensare a molta gente «Ma che bello! Firebird usa un default valore di 0 per i nuovi campi: non c'è problema allora». Ma si riesce a verificare che il campo ID è veramente tutto a `NULL` con queste query:

- SELECT * FROM AVVENTURE WHERE ID = 0 (non riporta nulla)
- SELECT * FROM AVVENTURE WHERE ID IS NULL (riporta il set visto qui sopra con tutti gli 0 fasulli)

- SELECT * FROM AVVENTURE WHERE ID IS NOT NULL (non riporta nulla)

Un altro tipo di query che dimostra che qualcosa di strano sta' succedendo è la seguente:

- SELECT NOME, ID, ID+3 FROM AVVENTURE

Questa query riporta 0 nella colonna «ID+3». Se ci fosse stato un vero e proprio 0 in ID avrebbe dovuto esserci 3. Il risultato *corretto* dovrebbe essere `NULL`, naturalmente!

Con un campo di tipo (VAR)CHAR, si avrebbero delle stringhe vuote fasulle ('). Con colonne di tipo DATE, ci sarebbero delle «date zero» fasulle pure quelle, che indicano il giorno 17 Novembre 1858. In tutti i casi, il vero valore della data è `NULL`.

Spiegazione

Che cosa succede?

Quando una applicazione come isql interroga il server, la conversazione avviene attraverso un certo numero di passi. Durante una di queste, la fase di descrizione – «describe» – il sistema informa del tipo e della annullabilità per ogni colonna che appare nell'insieme del risultato. Fa questo in una struttura che viene succesivamente utilizzata per recuperare il valore attuale del dato. Per le colonne che sono marchiate con NOT NULL dal server, non c'è alcun modo per riportare il `NULL` al client — a meno che il client reimposti l'informazione prima di iniziare la fase di recupero dei dati. Molte applicazioni client non lo fanno. D'altra parte, se il server assicura che una colonna non può contenere `NULL`, per quale motivo si dovrebbe pensare di saperne di più, e soprassedere alla sua decisione controllando comunque i `NULL`? Ma purtroppo è esattamente quello che andrebbe fatto se si vuole evitare il rischio di riportare valori fasulli agli utenti.

FSQL

L'esperto di Firebird Ivan Prenosil ha scritto un programma libero a linea di comando che funziona più o meno come isql, che, tra gli altri perfezionamenti, riporta i `NULL` correttamente anche nelle colonne NOT NULL. Si chiama FSQL e lo si può scaricare da:

http://www.volny.cz/iprenosil/interbase/fsql.htm

Assicurarsi della validità dei dati

Per essere sicuri di avere dei dati validi quando si aggiunge una colonna NOT NULL ad una tabella con dati preesistenti bisogna fare una delle operazioni seguenti:

- per impedire il problema delle colonne non null invece che null, si può aggiungere un valore di default aggiungendo la nuova colonna:

  ```
  alter table Avventure add id int default -1 not null
  ```

  I valori di default non sono applicati di solito quando si aggiungono campi a righe esistenti, ma lo sono con campi NOT NULL.

- Altrimenti, si possono assegnare ai nuovi campi i valori che dovrebbero avere in modo esplicito, subito dopo aver aggiunto la relativa colonna. Si può verificare che sono tutti validi con una query «SELECT ... WHERE ... IS NULL», che dovrebbe riportare un insieme di righe vuoto.

- Se il danno è stato fatto e ci si ritrova con una copia di backup impossibile da recuperare, si può usare la variante *-n* del programma gbak per ignorare i vincoli di validità nel restore. Dopodichè è meglio aggiustare i dati ed i vincoli manualmente. Di nuovo, si verifica il tutto con una query «WHERE ... IS NULL».

> **Importante**
>
> Le versioni di Firebird fino alla 1.5.0 inclusa hanno un problema che fa rimettere alla gbak i vincoli NOT NULL anche specificando il *-n*. Con tali versioni, se si è fatto un backup del database con dati NULL in campi NOT NULL, si è veramente nei guai. Soluzione: installare la versione 1.5.1 o successiva, recuperare i dati con gbak *-n* e poi aggiustarli.

Aggiungere un campo CHECK

Usare un vincolo CHECK è un modo diverso per impedire valori NULL in una colonna:

```
alter table Avventure add id int check (id is not null)
```

Se lo fai in questo modo, una successiva SELECT darà:

Tabella 10. Result set dopo aver aggiunto un campo CHECK

Nome	Data_acquisto	Prezzo	ID
Goffredo di Buglione	12-06-1995	€ 49,00	<null>
Giuseppe Garibaldi	19-10-1995	€ 54,95	<null>

Be', almeno adesso si riesce a vedere che i campi sono NULL! Firebird non attua i vincoli di CHECK sulle righe preesistenti aggiungendo nuovi campi. Questo è vero anche aggiungendo controlli a campi esistenti con ADD CONSTRAINT o con ADD CHECK.

In questo caso, Firebird non solo tollera la presenza ed il backup dei valori NULL, ma ne permette pure il recupero. Lo strumento gbak di Firebird recupera i vincoli di CHECK, ma non li applica ai dati esistenti nel backup.

> **Nota**
>
> La gbak rimette i vincoli di CHECK anche con lo switch *-n*. Ma siccome non vengono utilizzati ver convalidare i dati di backup, questo non comporterà mai errori in fase di recupero.

La recuperabilità dei dati NULL nonostante la presenza di vincoli CHECK è consistente col fatto che Firebird permette loro di essere presenti inizialmente e di farne il backup. Da un punto di vista pratico, c'è il rovescio della medaglia: si può fare tutto un ciclo di backup e restore con i dati «illegali» che sopravvivono senza che neanche se abbia la minima notizia o avvertimento. Pertanto: bisogna essere sempre sicuri che i dati esistenti siano consistenti con le regole aggiornate subito dopo aver aggiunto la nuova colonna con vincoli. Notare che il trucco del «default» in questo caso non funzionerebbe; bisogna ricordarsi di mettere i valori giusti subito. Se ce lo si scorda, ci sono serie possibilità che sopravvivano per lungo tempo dei NULL fuorilegge, in quanto non ci saranno poi più sveglie inserite.

Aggiungere un campo non annullabile usando i domini

Al posto di specificare il tipo di dato ed il vincolo direttamente, si possono usare anche i domini, ad esempio così:

```
create domain icnn as int check (value is not null);
alter table Avventure add id icnn;
```

Per quanto riguarda la presenza di campi NULL, il riportare valori fasulli a 0, l'effetto dei valori default, ecc., *non fa alcuna differenza* fra scegliere il metodo dei domini o l'approccio diretto. Tuttavia, un vincolo NOT NULL che viene posto attraverso un dominio può essere sempre rimosso, un vincolo NOT NULL diretto su una colonna ci starà per sempre.

Rendere le colonne esistenti non annullabili

Rendere una colonna esistente NOT NULL

Non si può aggiungere il vincolo NOT NULL ad una colonna esistente, ma si può aggirare il problema semplicemente; supponendo che il tipo sia intero, allora con

```
create domain intnn as int not null;
alter table Tabella alter Colonna type intnn;
```

cambierà il tipo della colonna in «int not null».

Se la tabella ha già delle registrazioni, ogni NULLs già presente nella colonna rimarrà NULL, e come sopra Firebird lo riporterà come 0 all'utente nelle interrogazioni. La situazione è abbastanza simile a (vedi) *aggiungere un campo NOT NULL*. La sola differenza è che dando ad un dominio (e pertanto alla colonna) un valore default, in questo caso non si può essere sicuri che esso verrà applicato ai preesistenti valori NULL. Le prove mostrano che talvolta il default viene applicato a tutti i NULLs, qialche volta a nessuno, ed in qualche raro caso ad *alcuni* dei valori esistenti ma non a tutti! Ultima cosa: cambiando il tipo di dato della colonna, qualora il nuovo tipo includa un default, bisogna controllare tutti i valori già esistenti, specialmente se «sembrano contenere» 0 o valori equivalenti a zero.

Aggiungere un vincolo di CHECK ad una colonna esistente

Ci sono due modi per aggiungere un vincolo di CHECK ad una colonna:

```
alter table Articoli add check (Valore is not null)
```

```
alter table Articoli add constraint ValoreNonNull check (Valore is not null)
```

La seconda forma è la preferita, in quanto permette in modo semplice di eliminare il controllo, per quanto funzionino esattamente allo stesso modo. Come ci si può facilmente aspettare, i campi con NULL già esistenti nella colonna rimangonon, può esserne fatto il backup ed il restore, ecc. ecc. Vedi *aggiungere un campo CHECK*.

Rendere le colonne non annullabili di nuovo annullabili

Se una colonna era stata resa non annullabile con un vincolo CHECK, quest'ultimo può essere eliminato:

```
alter table Articoli drop constraint ValoreNonNull
```

Avendo dato un nome al vincolo ma con il CHECK aggiunto direttamente alla colonna o alla tabella, bisogna trovare il suo nome prima di poterlo rimuovere. Questo si può fare con il comando isql «SHOW TABLE» (in questo caso: SHOW TABLE ARTICOLI).

In caso di un vincolo NOT NULL, conoscendo il suo nome si può semplicemente rimuoverlo:

```
alter table Articoli drop constraint NN_Valore
```

Se non si conosce il nome si può provare con isql ed usare al solito «SHOW TABLE», ma questa volta mostrerà il nome del vincolo *solo* se è stato ridefinito dall'utente. Il nome esatto si ha in ogni caso con:

```
select rc.rdb$constraint_name
from   rdb$relation_constraints rc
       join rdb$check_constraints cc
       on rc.rdb$constraint_name = cc.rdb$constraint_name
where  rc.rdb$constraint_type   = 'NOT NULL'
       and rc.rdb$relation_name = '<TableName>'
       and cc.rdb$trigger_name  = '<FieldName>'
```

Non c'è da spaccarsi la testa sui nomi dei campi e delle tabelle di questa query: non c'è dubbio che siano abbastanza strani, ma sono corretti. Basta solo assicurarsi che il nome della tabella e del campo siano in maiuscolo se sono stati definiti in modo indipendente dal caso. Altrimenti devono essere scritti in modo esatto.

Se il vincolo NOT NULL arriva attraverso un dominio, si può rimuovere modificando il tipo di colonna ad un dominio annullabile o ad un tipo di dato predefinito:

```
alter table Articoli alter Importo type int
```

Ogni `NULL` nascosto, se presente, sarà di nuovo visibile.

In qualsiasi modo si rimuova il vincolo NOT NULL, bisogna confermare il lavoro (**commit**) e *chiudere tutte le connessioni al database*. Dopodichè ci si può riconnettere ed inserire `NULL` nelle colonne.

Controllare per `NULL` e per l'eguaglianza nella pratica

Questa sezione contiene alcuni suggerimenti pratici ed esempi che possono essere utili avendo a che fare con i `NULL`. Riguarda i casi in cui si vuol verificare lo stato di `NULL` di un campo o l'(in)uguaglianza fra due cose qualora possano essere implicati dei `NULL`.

Come controllare se ci sono `NULL`

Frequentemente non si ha bisogno di prendere misure particolari per i campi o le variabili che potrebbero essere `NULL`. Ad esempio, facendo come segue:

```
select * from Clienti where Comune = 'Verona'
```

molto probabilmente non si vogliono elencati i clienti per i quali la città non risulta specificata. Allo stesso modo:

```
if (Eta >= 18) then PotestVotare = 'Si'
```

non include le persone di età sconosciuta, quindi è difendibile, ed include le persone di età sconosciuta, ed è pure questo difendibile. Ma:

```
if (Eta >= 18) then PotestVotare = 'Si';
else PotestVotare = 'No';
```

sembra meno giustificabile: se non si conosce l'età di una persona, non si può esplicitamente negargli il diritto di voto. Ancora peggio, è far questo:

```
if (Eta < 18) then PotestVotare = 'No';
else PotestVotare = 'Si';
```

che non può avere lo stesso effetto del precedente esempio. Se alcuni di cui non si sa l'età sono in realtà minorenni (Eta < 18), gli si permette di votare!

Il miglior metodo in questo caso è controllare esplicitamente se vale `NULL`:

```
if (Eta is null) then PotestVotare = '??';
else
  if (Eta >= 18) then PotestVotare = 'Si';
  else PotestVotare = 'No';
```

Poichè si hanno più di due possibilità, è più elegante usare la sintassi dello statement CASE, che è disponibile a partire da Firebird 1.5 e successivi:

```
PotestVotare = case
            when Eta is null then '??'
            when Eta >= 18   then 'Si'
            else 'No'
          end;
```

O, ancora meglio:

```
PotestVotare = case
            when Eta >= 18 then 'Si'
            when Eta <  18 then 'No'
            else '??'
          end;
```

Test di uguaglianza e confronti

Quando si vuole verificare se due campi o variabili sono identici e li si vuol considerare uguali anche se sono entrambi `NULL`, il modo per farlo dipende dalla versione di Firebird che si sta' usando.

Firebird 2.0 e successivi

In Firebird 2 e succesivi, si confronta per la non uguaglianza dei valori con DISTINCT. Questo è già stato visto in precedenza, ma qui lo rivediamo brevemente. Due espressioni sono considerate:

- DISTINCT se hanno valori diversi oppure una delle due è `NULL` e l'altra no;

- NOT DISTINCT se hanno lo stesso valore oppure se entrambe sono `NULL`.

[NOT] DISTINCT riporta sempre o `true` o `false`, mai `NULL` o altri valori. Examples:

```
if (A is distinct from B) then...
```

```
if (Cliente1 is not distinct from Cliente2) then...
```

Chi non è interessato ai metodi pre-Firebird 2, può saltare le seguenti sezioni.

Versioni precedenti alla 2

Queste versioni non supportano l'uso di DISTINCT. Di conseguenza i test sono un po' più complicati e ci sono alcuni trabocchetti da evitare.

Il test di uguaglianza corretto per le versioni in esame è:

```
if (A = B or A is null and B is null) then...
```

oppure, esplicitando le precedenze degli operatori:

```
if ((A = B) or (A is null and B is null)) then...
```

È necessario avvertire di una cosa: se esattamente uno solo fra A e B è proprio `NULL`, l'espressione diventa `NULL`, non falsa! Questo è giusto, per quanto abbiamo visto prima, in uno statement di `if`, e si potrebbe anche aggiungere perfino un clausola `else` che può essere eseguita nel caso in cui A e B non sono uguali (incluso il caso in cui uno dei due è `NULL` e l'altro no):

```
if (A = B or A is null and B is null)
  then ...cose da fare se A è uguale a B...
  else ...cose da fare se A è diverso da B...
```

Ma bisogna evitare come la peste la brillante idea di invertire le espressioni e di usarle come un test di ineguaglianza:

```
/* Da evitare! */
if (not(A = B or A is null and B is null))
  then ...cose da fare se A non è uguale a B...
```

Il codice qui sopra funziona correttamente se A e B sono entrambi `NULL` o entrambi diversi da `NULL`. Ma nel caso in cui uno solo dei due sia `NULL` non entra nella parte `then` perchè il test vale NULL, invece che essere valutato a vero come si potrebbe essere indotti erroneamente a pensare.

Volendo eseguire un codice se e solo se A e B sono differenti, si può usare una delle espressioni corrette viste sopra mettendo uno statement innocuo nella clausola `then`. A partire dalla 1.5 si possono usare anche blocchi `begin..end` vuoti. In alternativa si può usare una espressione più lunga come questa:

```
/* Questo è un corretto test per diseguaglianza: */
if (A <> B
    or A is null and B is not null
    or A is not null and B is null) then...
```

Ricordare che questo è necessario dsolo nelle versioni precedenti alla 2.0 di Firebird. A partire dalla 2 in poi, il test di disuguaglianza si fa semplicemente con «`if (A is distinct from B)`».

Riassunto sui controlli di (dis)uguaglianza

Tabella 11. Verificare la (dis)uguaglianza di A e B in versioni differenti di Firebird

Tipo di controllo	Versione di Firebird	
	<= 1.5.x	>= 2.0
Uguaglianza	`A = B or A is null and B is`	`A is not distinct from B`

Disuguaglianza	`A <> B`
	`or A is null and B is not null`
	`or A is not null and B is null`
		`A is distinct from B`

Tenere ben presente che in Firebird 1.5.x e precedenti:

- il test d'uguaglianza riporta `NULL` se uno solo degli operandi è `NULL`;
- il test di disuguaglianza riporta `NULL` se entrambi gli operandi sono `NULL`.

In un contesto di tipo IF o WHERE, questi risultati `NULL` si comprtano come `false` – che andrebbe bene in generale. Ma bisogna fare attenzione che invertendo con NOT() da' ilmedesimo risultato `NULL`, e non «`true`». Inoltre se si usano i confronti della versione 1.5 e precedenti con un vincolo di CHECK in Firebird 2 o successivi, leggere attentamente la sezione *Vincoli di controllo (CHECK constraints)*, se non l'avete già fatto.

> **Suggerimento**
>
> Molte operazioni di JOIN sono costituite da uguaglianze su campi di diverse tabelle, ed usano l'operatore «=». Questo elimina tutte le coppie `NULL`-`NULL`. Per far corrispondere fra loro due `NULL`, selezionare il test di uguaglianza adatto per la versione di Firebird usata dalla tabella sopra.

Determinare se un campo è cambiato

Nei trigger è spesso utile sapere se un certo campo è stato modificato (compresa la trasformazione da `NULL` a non-`NULL` o viceversa) oppure è rimasto identico (compreso il mantenere lo stato di `NULL`). Questo non è altro che un caso speciale del controllo di (dis)uguaglianza di due campi.

In Firebird 2 e successivi si usa questo codice:

```
if (New.Valore is not distinct from Old.Valore)
  then ...il Valore non è cambiato...
  else ...il Valore è cambiato...
```

E nelle precedenti versioni:

```
if (New.Valore = Old.Valore
    or New.Valore is null and Old.Valore is null)
  then ...il campo Valore è rimasto uguale...
  else ...il campo Valore è cambiato...
```

Sommario

NULL in sintesi:

- NULL significa *sconosciuto*.
- Per escludere NULL da un dominio o da una colonna, aggiungere «NOT NULL» dopo il nome del tipo.
- Per determinare se A è NULL, si usa «A IS [NOT] NULL».
- Assegnare NULL è come assegnare un qualsiasi altro valore: con «A = NULL» o con una lista di insert.
- Per determinare se A e B sono identici, sapendo che tutti i NULL sono identici fra loro e differenti da qualsiasi altra cosa, si usa «A IS [NOT] DISTINCT FROM B» in Firebird 2 e successivi. Nelle versioni precedenti i controlli sono:

```
// uguaglianza:
A = B or A is null and B is null
```

```
// disuguaglianza:
A <> B
or A is null and B is not null
or A is not null and B is null
```

- In Firebird 2 e succesivi si può usare il valore letterale NULL in quasi tutte le situazioni dove sarebbe permesso un qualsiasi altro valore normale. In pratica questo dà però solo più spago con cui appendersi.
- Quasi sempre, gli operandi NULL fanno sì che l'intera operazione riporti NULL. Le eccezioni degne di nota sono:
  - «NULL or true» vale true;
  - «NULL and false» vale false.
- I predicati IN, ANY|SOME ed ALL possono (ma non sempre) riportare NULL o quando l'espressione a sinistra o quando un elemento della lista/subselect è NULL.
- Il predicato [NOT] EXISTS nonriporta mai NULL. Il predicato [NOT] SINGULAR non riporta mai NULL in Firebird 2.1 e successivi, è invece bucato in tutte le versioni precedenti.
- Nelle funzioni di aggregazione, solo i campi non-NULL sono utilizzati per il calcolo. L'eccezione è COUNT(*).
- Nei risultati ordinati, i NULLs sono messi...
  - 1.0: alla fine;
  - 1.5: alla fine, a meno che non sia specificato NULLS FIRST;
  - 2.0: all'inizio del risultato (inizio se ascendente, alla fine se discendente), a meno che non venga specificato esplicitamente NULLS FIRST/LAST.
- Se una clausola WHERE o HAVING vale NULL, la riga non è inclusa nel risultato.
- Se l'espressione di test di una frase IF è NULL, allora si salta il blocco THEN e viene eseguito il blocco ELSE.

- Una frase CASE riporta `NULL`:
  - se il risultato selezionato è `NULL`.
  - se non sono trovati abbinamenti validi (CASE semplice) o nessuna delle condizioni è `true` (CASE complesso) e non c'è la clausola ELSE.
- In una frase CASE semplice, «CASE *<null_expr>*» *non* si abbina ad un «WHEN *<null_expr>*».
- Le l'espressione di controllo di una ciclica WHILE vale `NULL`, non si (ri)entra nel ciclo.
- Una frase FOR continua anche se si ricevono `NULL`. Essa continuerà a ciclare finchè non verranno processate tutte le righe, o finchè non sarà interrotta da un'eccezione o da una direttiva PSQL di interruzione del ciclo.
- Nelle chiavi primarie, i `NULL` non sono permessi.
- Nelle chiavi univoche e negli indici univoci, i `NULL` sono
  - *non permessi* in Firebird 1.0;
  - *permessi* (anche multipli) in Firebird 1.5 e successivi.
- Nelle colonne delle chiavi esterne, sono permessi `NULL` multipli.
- Se un vincolo di CHECK vale `NULL`, allora il valore viene
  - *rifiutato* sotto Firebird 1.5 e precedenti;
  - *accettato* sotto Firebird 2.0 e successivi.
- SELECT DISTINCT considera eguali tutti i `NULL`, pertanto ne riporta al massimo uno.
- Le UDF talvolta convertono `NULL` <–> non-`NULL` in un modo apparentemente casuale.
- Le funzioni `COALESCE` e `*NVL` possono convertire un `NULL` in un valore.
- La famiglia di funzioni `NULLIF` è in grado di convertire valori in `NULL`.
- Aggiungendo una colonna NOT NULL senza valore di default ad una tabella con dati preeesistenti, tutti i valori in quella colonna saranno `NULL` dopo la creazione. Molti programmi tuttavia - incluso lo strumnento isql di Firebird – riportano erroneamente come zeri (0 per i campi numerici, " per i campi stringa, ecc.)
- Cambiando il tipo di dato di una colonna ad un dominio NOT NULL, tutti gli eventuali `NULL` esistenti nella colonna rimarranno tali. Di nuovo molti programmi, inclusa isql, li mostreranno come zeri.

Ricordare: così funziona il `NULL` *in Firebird SQL*. Ci possono essere più o meno sottili differenze con altri RDBMS.

Appendice A: Problemi relativi a NULL in Firebird

Attenzione: nelle sezioni seguenti sono elencati sia i problemi attuali sia quelli superati. Controllare se e quando un particolare problema è stato risolto prima di determinare se esiste nella vostra particolare versione di Firebird.

Problemi che fanno cadere il server

EXECUTE STATEMENT con argomento NULL

EXECUTE STATEMENT con un argomento a NULL fanno cadere i server Firebird 1.5 e 1.5.1. Risolto nella versione 1.5.2.

EXTRACT da una data NULL

In 1.0.0, EXTRACT da una data NULL potrebbe far cadere il server. Risolto nella versione 1.0.2.

FIRST e SKIP con argomento NULL

FIRST e SKIP fanno cadere i server Firebird 1.5.n o precedenti se gli viene passato un argomento NULL. Risolto nella versione 2.0.

LIKE con carattere escape NULL

Usare LIKE con un carattere di escape a NULL potrebbe far cadere il server. Risolto nella versione 1.5.1.

Altri problemi

NULL in colonne NOT NULL

I NULL possono esistere nelle colonne NOT NULL nelle seguenti situazioni:

- Se si aggiunge una colonna NOT NULL ad una tabella già piena, i campi della colonna aggiunta saranno tutti NULL.
- Se si rende una colonna esistente NOT NULL, ogni NULL già presente nella colonna rimane in quello stato.

Firebird permette a quei NULL di restare, e ne permette il backup, ma si rifiuta di recuperare la copia con gbak. Vedere *Aggiungere una colonna NOT NULL* and *Rendere una colonna esistente NOT NULL*.

NULL illegali riportati come 0, '', ecc.

Se una colonna NOT NULL contiene NULL (vedere il problema precedente), il server lo descrive come non annullabile al programma client. Poichè molti programmi si fidano ciecamente del server, interpretano quei NULL come se fossero 0 (o equivalente) e come tale lo presentano all'utente. Vedere *Valorizzazione errata di NULL come se fosse zero*.

Chiavi primarie con valori a NULL

Il problema seguente è comparso in Firebird 1.5: avendo una tabella con dati, ed aggiungendo una colonna NOT NULL (che automaticamente crea dei NULL nelle righe esistenti– vedi sopra), è possibile rendere quella colonna la chiave primaria anche se ha già dei valori NULL. In 1.0 questo non sarebbe potuto funzionare a causa delle regole più restrittive per gli indici UNIQUE. Risolto nella versione 2.0.

SUBSTRING descritta come non annullabile

Il sistema descrive le colonne risultato di una SUBSTRING come non annullabili nei seguenti due casi:

- Se il primo argomento è una costante stringa, come in «SUBSTRING('Paperino' FROM 5 FOR 2)».
- Se il primo argomento è una colonna NOT NULL

Questo è sbagliato perchè perfino di una stringa nota, la sottostringa può essere NULL quando il primo degli altri argomenti è NULL. Nella versione 1.* qusto non era un problema: gli argomenti del FROM e del FOR dovevano essere per forza costanti, pertanto non avrebbero mai potuto essere NULL. Ma a partire da Firebird 2, è permessa ogni espressione idonea a rappresentare il tipo di dato richiesto. E sebbene il sistema correttamente riporti NULL se un argomento è NULL, *descrive* la colonna risultato come non annullabile, cosicchè molti programmi mostrano il risultato come stringa vuota.

Questo problema sembra risolto a partire dalla versione 2.1.

Gbak -n recupera i NOT NULL

Gbak *-n[o_validity]* recuperava i vincoli NOT NULL nelle prime versioni di Firebird. Risolto nella versione 1.5.1.

IN, =ANY e =SOME con subselect indicizzate

Sia *A* be l'espressione sinistra e *S* il risultato di una subselect. Nelle versioni precedenti la 2.0, «IN», «=ANY» e «=SOME» riportavano false invece di NULL se un indice era attivo nella colonna della subselect e:

- o *A* è NULL e *S* non contiene NULL;
- o *A* non è NULL, *A* non esiste in *S*, e *S* contiene almeno un NULL.

Vedere gli avvertimenti nelle sezioni IN e ANY. Alternativa: usare invece «<> ALL» . Risolta nella versione 2.0.

ALL con subselect indicizzate

Con qualsiasi operatore, eccetto «<>», ALL può dare risultati sbagliati se un indice è attivo sulla colonna della subselect. Questo può capitare con o senza NULL. Vedere Problemi di ALL. Risolto nella versione 2.0.

SELECT DISTINCT con l'ordinamento sbagliato di NULLS FIRST|LAST

Firebird 2.0 ha il seguente problema (difficile da spiegare senza esempi): se una SELECT DISTINCT è ordinata con [ASC] NULLS LAST o DESC NULLS FIRST, ed i campi di ordinamento (nella ORDER BY) formano solo l'inizio ma non sono tutti i campi della lista nella SELECT, ogni campo nella clausola ORDER BY che è seguito nella SELECT da un campo con un ordinamento differente (o nessun ordinamento) si ritrova i NULL posizionati in modo default, ignorando la direttiva NULLS XXX. Risolto nella versione 2.0.1 e 2.1.

UDF che riportano valori invece di riportare NULL

Questo dev'essere certamente considerato un problema. Se un'angolo è sconosciuto, *non si può* affermare che il suo coseno è 1! Sebbene tutta la storia di queste funzioni sia ben nota ed è comprensibile il perchè si comportino così (vedere *Funzioni definite dall'utente (UDF)*), è comunque sbagliato. Sono riportati valori errati e questo non dovrebbe accadere. La maggior parte delle funzioni matematiche nella ib_udf, oltre a qualche altra, hanno questo problema.

UDF che riportano NULL quando invece dovrebbero riportare un valore

Questo è un problema complementare al precedente. LPAD ad esempio riporta NULL se si vuol riempire una stringa vuota con, ad esempio, 10 punti. Questa ed altre funzioni sono state corrette nella versione 2.0, con l'avvertenza che bisogna esplicitamente dichiararle con la parola chiave NULL, altrimenti si comporteranno nel modo sbagliato di una volta. LTRIM e RTRIM trasformano stringhe vuote in NULL in Firebird 1.0.n. Questo è stato risolto nella versione 1.5 al prezzo di riportare stringa vuota (cioè '') se l'argomento è una stringa NULL, e completamente risolto solo nella versione 2.0 (se dichiarata con la parola chiave NULL).

SINGULAR inconsistente con risultati a NULL

NOT SINGULAR talvolta riporta NULL quando SINGULAR riporta true o false. Risolto nella versione 2.0.

SINGULAR può erroneamente riportare NULL, in un modo riproducibile ma inconsistente. Risolto nella versione 2.1.

Vedere la sezione su SINGULAR.

Appendice B: Cronologia

La cronologia completa è registrata nel modulo del manuale (`manual` module) nell'albero del CVS; vedi http://sourceforge.net/cvs/?group_id=9028

Diario delle Revisioni

0.1	8 Aprile 2005	PV	Prima edizione.
0.2	15 April 2005	PV	Menzionato il fatto che Fb 2.0 ha legalizzato i confronti del tipo «`A = NULL`». Cambiato il testo in «Come controllare se ci sono `NULL`». Riaggiustato il testo nella sezione «Lavorare con i `NULL`».
0.2-it	6 dicembre 2006	UM	Prima versione in italiano. Piccoli aggiustamenti al testo e aggiunta una tabella riassuntiva degli esempi con le funzioni di aggregazione
1.0	24 Jan 2007	PV	Questo è un grosso aggiornamento, con molto materiale aggiunto ex-novo per cui il documento è quattro volte la versione precedente. Inoltre molta parte del testo esistente è stato riorganizzato e rifatto in toto. Difficile dare un riassunto di tutto il lavoro fatto: meglio considerarla una guida completamente nuova con un 15%-25% di vecchio materiale. Le aggiunte principali sono: • `NULL` come costanti • IS [NOT] DISTINCT FROM • funzioni interne e direttive • Predicati: IN, ANY, SOME, ALL, EXISTS, SINGULAR • Filtri o ricerche (WHERE) • Ordinamenti (ORDER BY) • GROUP BY e HAVING • CASE, WHILE e FOR • Chiavi ed indici • vincoli CHECK • SELECT DISTINCT • Convertire valori a `NULL` con `NULLIF` • Modificare tabelle piene • Liste dei problemi • Indice alfabetico
1.0.1	26 Jan 2007	PV	*Rendere le colonne NOT NULL di nuovo annullabili*: correzione provvisoria di un errore riguardante la rimozione di vincolo NOT NULL.
1.0.1-it	Mar 2007	UM	Versione italiana della 1.0.1

Appendice C: Licenza d'uso

Il titolo del documento originale è *Firebird Null Guide*.

L'autore iniziale del documento originale è: Paul Vinkenoog.

L'autore della versione italiana del documento è: Umberto Masotti.

Indice alfabetico

T

U

V

W