EmberWings

2026/I

Firebird

March 2026

www.firebirdsql.org

emberwings@firebirdsql.org

The Firebird project was created at SourceForge on

July 31, 2000

This marked the beginning of Firebird's development as an open-source database based on the InterBase source code released by Borland.
Since then, Firebird's development has depended on voluntary funding from people and companies who benefit from its use.

Support Firebird

Thank you for your support!

EmberWings is a quarterly magazine published by the **Firebird Foundation z.s.**, free to the public after a 3-month delay. Regular donors get exclusive early access to every new edition upon release.

In This Issue:

A Moment Between Chapters

Dear Readers,

A new year always carries a sense of beginning, but some beginnings feel larger than the turning of a calendar. Last year quietly closed the first quarter century of Firebird's life. Twenty-five years is long enough for a technology to prove something about itself—not only about how it was designed, but about how it survives, adapts, and continues to matter.

Databases live longer than many of the tools and trends that surround them. Applications change, hardware generations come and go, programming languages rise and fade, yet the data—and the systems that guard it—remain. Over time, Firebird installations accumulate something that cannot be engineered directly: experience embedded in schemas, habits, operational knowledge, and the quiet understanding of those who maintain them.

Crossing into the next quarter century therefore invites a different kind of reflection. Not just what Firebird can do today, but what it means to steward systems that must endure. Each year brings new pressures—new workloads, new security concerns, new expectations about performance and scale. And now, increasingly, a new technological horizon shaped by artificial intelligence, automation, and systems that learn from the data we store.

Yet if history teaches anything, it is that lasting technologies are rarely defined by novelty alone. They endure because they develop a body of knowledge around them—patterns learned through practice, insights shared by communities, and a growing sense of how complex systems behave when they are allowed to mature over time.

As we begin this new chapter, EmberWings continues to explore that accumulated wisdom. Not just the mechanics of a database engine, but the broader craft of understanding the systems we build with it. If the first twenty-five years were about proving Firebird's resilience, the years ahead may be about something deeper: learning how to guide evolving systems through a future that will almost certainly surprise us.

Warm regards,
The EmberWings Team

Late one evening a young developer approached the master, holding a notebook full of measurements.

“Master,” he said, “the system is slow. I have examined the queries and improved them. Yet the system is still slow.”

The master poured tea.

“Then it must be the indexes,” the apprentice continued. “I studied them carefully. I rebuilt some and added others. Still the system is slow.”

The master placed the cup before him.

“So I looked at the configuration,” said the apprentice. “I enlarged the caches and adjusted the parameters. The system became faster for a while, but the slowness returned.”

The master nodded. “And what will you examine next?”

“The disks,” said the apprentice quickly. “Or perhaps the network.”

The master walked to the window and opened it. Outside, the evening wind moved the branches of a large tree.

“Tell me,” the master said, pointing outside, “why does that branch move?”

“The wind moves it,” the apprentice answered.

“And the wind?” asked the master.

“It moves because the air moves.”

“And the air?”

The apprentice hesitated. After a moment he said, “Master, I do not know.”

The master smiled slightly. “You are asking the tree which leaf is responsible for the wind.”

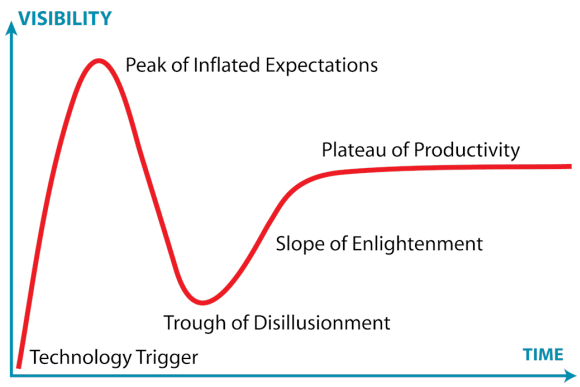
The apprentice looked again at the tree. The leaves trembled together.

After a while he closed his notebook.

The Hype Cycle & Amara's Law

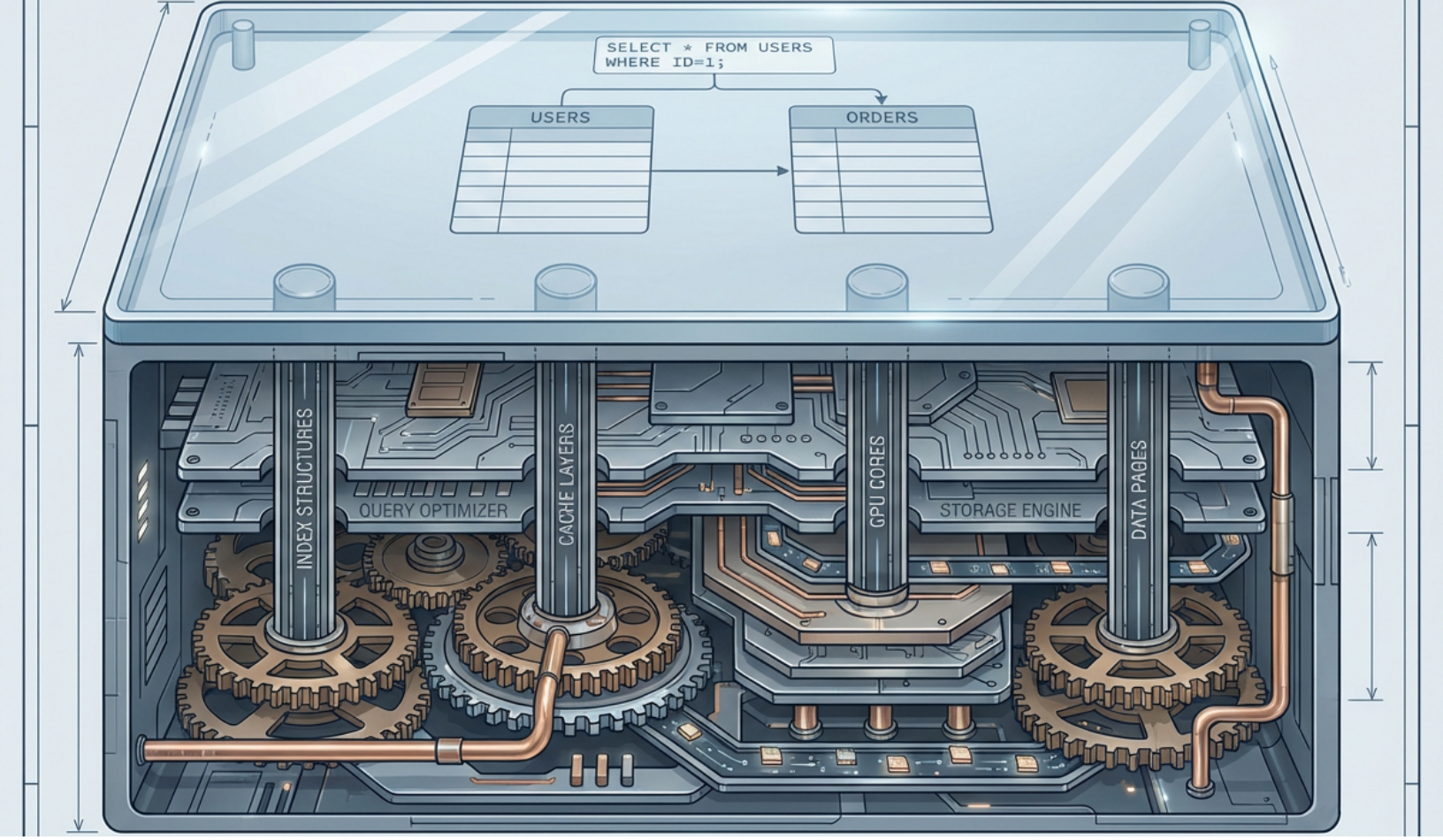
> *We tend to overestimate the effect of a technology in the short run and underestimate the effect in the long run.*
> *(Roy Amara)*

The Hype Cycle is a visual representation of the excitement and development of technology over time, originally produced by Gartner. It is best shown with a visual:



In short, this cycle suggests that there is typically a burst of excitement around new technology and its potential impact. Teams often jump into these technologies quickly, and sometimes find themselves disappointed with the results. This might be because the technology is not yet mature enough, or real-world applications are not yet fully realised.

After a certain amount of time, the capabilities of the technology increase and practical opportunities to use it increase, and teams can finally become productive. Roy Amara's quote sums this up most succinctly - "We tend to overestimate the effect of a technology in the short run and underestimate in the long run".



The Importance of Holistic Thinking

By Pavel Císař (IBPhoenix)

Most Firebird performance investigations begin the same way. A system slows down, behaves unpredictably, or stops scaling the way it once did. The first instinct—usually the correct one—is to look at the SQL.

We examine the queries that matter, inspect execution plans, and verify that indexes behave as expected. Very often that is enough. A missing index is added, a predicate is adjusted, a plan improves, and the problem disappears.

If nothing obvious appears there, attention shifts to configuration. This is the natural next step. Firebird systems tend to live a long time, migrate between machines, and accumulate configuration changes along the way. Cache sizes may no longer match the workload, parameters that once made sense may not anymore, and settings are sometimes copied from other environments without much thought. Adjusting configuration often produces immediate, measurable improvements, reinforcing the sense that this is where much of the leverage lies.

Only when both SQL and configuration fail to explain the situation do we widen the scope further. Hardware is examined, storage performance is questioned, and operating system behavior comes under scrutiny. At that point the discussion shifts from what the database is doing to what the platform is doing.

This progression reflects experience. Each step targets an area where problems genuinely occur and where fixes are concrete and actionable. Just as importantly, it keeps the investigation focused on places where cause and effect are usually close together. You change something, observe the result, and move on.

Over time this approach becomes comfortable. It feels structured, disciplined, and reliable. Many systems run well for years precisely because engineers apply this routine competently. From that success a subtle expectation emerges: if a problem is real, careful investigation along this path should eventually reveal its cause.

The rest of this article begins where that expectation quietly stops being true.

The first sign is rarely a dramatic failure. More often it is a lack of closure. The usual checks are performed carefully. Queries look reasonable, plans make sense, and indexes are where they should be. Configuration changes behave exactly as expected—except they do not affect the problem you are trying to explain.

Instead, the system behaves inconsistently. Throughput drops under load, but not always. Latency increases without a clear correlation to any single metric. Changes that should matter have little effect, while changes that should not matter sometimes do. Nothing is obviously wrong, yet the system does not behave the way experience suggests it should.

The natural reaction is to continue in the same direction. More data is collected, measurements become more precise, and decisions once considered settled are revisited. This feels productive, but it often produces a familiar outcome: more information and less clarity. Each explanation sounds plausible on its own, yet none of them form a convincing whole.

At this point investigations begin to circle. The same areas are revisited from slightly different angles. Small adjustments produce small, temporary improvements. The system remains fragile, sensitive to load changes, and difficult to reason about. When this happens, the problem is no longer hiding somewhere along the usual path. The path itself no longer leads toward understanding.

Continuing in the same mode rarely helps. More measurements and parameter tweaks may produce activity, but not insight. What is missing is not another detail but a different way of framing the situation.

Local reasoning works well when cause and effect are close. A query is slow because its plan is wrong. Memory pressure appears because a cache is undersized. Disk latency increases because storage is overloaded. In such cases, examining the same layer more closely eventually pays off.

The problems discussed here behave differently. Their symptoms appear in one place while their causes lie somewhere else. Sometimes they do not live in a single place at all, but emerge from the interaction of several parts of the system under load. As long as investigation remains confined to one layer at a time, those interactions remain invisible.

This is where a more holistic view becomes necessary. Instead of asking which layer is responsible, we begin asking how layers influence each other. Configuration shapes physical layout. Physical layout shapes coordination patterns. Coordination patterns interact with hardware characteristics. None of these relationships are unusual, yet they are easy to overlook when each layer is examined in isolation.

It helps to hold a simple mental map of the system. Not a map of internal code paths, but a conceptual view of where the layers exist and where coordination between them occurs. From that perspective it becomes easier to accept that a symptom observed at the SQL level may have very little to do with SQL itself.

To make this shift tangible, consider a deliberately simple example that many of us have either seen directly or come very close to seeing.

Imagine a system receiving a steady stream of concurrent inserts into a single table. Each transaction performs very little work: insert one row and commit. Transactions are short, there are no long-lived readers, and nothing about the SQL itself appears problematic. Logically, this is a workload Firebird should handle well.

For a while, it does.

As concurrency increases, throughput rises more or less linearly. Eventually it flattens, and shortly afterward it begins to fall. Latency increases across the system. Adding more worker threads makes things worse rather than better. There

is no dramatic blocking, no obvious waits, and no single query that stands out as slow.

By this point the usual explanations have already been eliminated. Queries are trivial, plans are stable, and configuration adjustments move some numbers but do not change the overall pattern. CPU is not saturated and I/O does not appear to be the limiting factor. Nothing at the usual diagnostic surface explains why progress slows down.

The missing piece becomes visible only when you stop thinking in terms of rows and transactions and begin thinking in terms of pages.

Although each transaction inserts a different row, many of those rows land on the same physical pages. With small rows and sustained insert rates, activity concentrates on a limited set of data pages.

Those pages are shared structures. Modifying them requires coordination, even in an MVCC system. Firebird's multi-version architecture prevents readers and writers from blocking each other logically, but it does not eliminate the need to coordinate concurrent modifications of the same physical page. Under low concurrency the cost is negligible. Under high concurrency it becomes dominant.

Transactions are not blocked for long periods; instead they are repeatedly delayed by short coordination points. Individually these delays are insignificant. Together they define the system's throughput ceiling.

From the outside the behavior appears paradoxical. Resources seem available, work units are tiny, and nothing looks obviously contended. Yet the system slows down precisely because many operations are happening correctly at the same time.

Once the interaction between workload shape, physical layout, and coordination mechanisms becomes visible, the behavior stops being mysterious. Without that broader view it is very difficult to explain at all.

Real production systems rarely look this clean. Inserts mix with updates, tables grow unevenly, access patterns shift over time, and several effects usually overlap. Problems arrive blurred by noise rather than presented as a single mechanism. That is precisely why the simple case is useful.

It shows how coordination effects can dominate behavior even when everything appears correct at the usual diagnostic surface. In real systems the same effects are present but diluted. Contention spreads across structures, appears only under certain load shapes, or surfaces at particular times.

The value of the example is not that real systems reproduce it exactly. It recalibrates intuition. Once you see how far below the SQL and configuration layers a real cause can live, it becomes easier to accept that more complex systems may hide similar interactions in much less obvious ways.

The mistake is not failing to recognize this exact scenario. The mistake is expecting real systems to present the problem this clearly.

Outside laboratory conditions the picture becomes less tidy. Instead of one obvious hotspot, several appear. Instead of a clean throughput collapse, performance degrades gradually. Latency becomes erratic, and behavior depends strongly on the mix of work running at the time.

A common shift is from data pages to index pages. Inserts and updates may target different rows—or even different tables—yet still converge on the same index structures. From the SQL perspective the workload looks well distributed. From the physical perspective it is anything but, and the symptoms resemble general slowness rather than contention.

In other cases pressure spreads across a small set of pages that become hot only under particular conditions: a batch job starts, a reporting window opens, or write concurrency crosses a certain threshold. Outside those moments everything appears normal, which is why such problems are often dismissed as background noise.

Hardware characteristics can amplify these effects. Coordination relies on atomic operations, cache coherency, and memory ordering, all of which behave differently across CPU architectures. A workload that behaved acceptably on one generation of hardware may slow down noticeably on another—even with the same schema, configuration, and application code.

What ties these situations together is that no single layer provides a satisfying explanation. SQL looks fine, configuration appears reasonable, and hardware is not obviously overloaded. The behavior becomes understandable only when the

interaction between layers is considered.

Once that realization appears, the technical details of the example matter less than the process that revealed them. The system did not become understandable because one more parameter was tuned or one more query optimized. It became understandable only after the investigation stopped treating layers as independent and began treating their interactions as first-class.

Experience can work against you here. Familiar diagnostic routines are efficient precisely because they narrow the field quickly. When they stop converging, the instinct is to apply them more thoroughly rather than question their scope. The result is a great deal of correct work that leads nowhere.

Effective diagnosis at this stage depends less on knowing more details and more on recognizing when the frame must change. The critical step is realizing that the problem space extends beyond any single layer. Once that shift occurs, previously confusing observations begin to align. Metrics that once seemed unrelated start to correlate. Behavior that looked inconsistent becomes predictable. The investigation converges again—not because the system has changed, but because the way we reason about it has.

This is not a rejection of rigor or intuition. It is a matter of applying them at the right level.

In practice the hardest decision is not how to proceed, but when to stop doing what has always worked before. Local reasoning is comfortable. It produces actions, measurements, and changes. It feels like progress. Letting go of it—even temporarily—can feel like stepping into uncertainty.

The warning signs are rarely dramatic. Changes that should have clear effects do not. Improvements are real but fragile, disappearing as soon as the workload shifts. Each adjustment resolves one symptom while revealing another. Over time the investigation becomes busy rather than decisive.

This is when systems accumulate folklore: “don’t touch that setting,” “a restart helps when it slows down,” or “it just behaves like this under load.” These are not solutions. They are coping mechanisms that appear in the absence of understanding.

Experienced engineers often recognize this moment instinctively. There comes a

point when it becomes clear that the problem is not hidden deeper within the same layer, but somewhere between layers. Recognizing that moment allows diagnosis to move forward instead of inward.

Holistic thinking does not replace local reasoning. It simply tells us when local reasoning has reached its limit. Once that boundary becomes visible, diagnosing systems becomes less about accumulating fixes and more about building understanding. That shift—from reacting to symptoms to reasoning about the system as a whole—is what turns troubleshooting into diagnosis.

The Broken Windows Theory

The Broken Windows Theory suggests that visible signs of crime (or lack of care of an environment) lead to further and more serious crimes (or further deterioration of the environment).

This theory has been applied to software development, suggesting that poor quality code (or Technical Debt) can lead to a perception that efforts to improve quality may be ignored or undervalued, thus leading to further poor quality code. This effect cascades leading to a great decrease in quality over time.

Interview with Carlos H. Cantu

For many Firebird users around the world, the Brazilian community has long been one of the most active and visible parts of the ecosystem. Conferences, articles, books, and lively discussions have helped keep the technology vibrant there for decades. Among the people who have contributed to this continuity is Carlos H. Cantu—a developer, author, and long-time community organizer whose name is familiar to anyone who has followed Firebird news, events, or learning resources in Portuguese. Through initiatives ranging from websites and publications to the Firebird Developers Day conference series, he has helped connect developers, share knowledge, and give the community places to meet and talk. In this interview, we invite Carlos to reflect on his journey in software, the experiences that shaped his involvement with Firebird, and the ideas that continue to drive his work within the community.

Carlos, many people in the Firebird world know your name from articles, books, and conferences, but may not know the story behind it. Could you tell us a little about yourself—where you grew up, how you discovered programming, and what first drew you into the world of software?

I'm 53 years old and I grew up in the city of Piracicaba, in the countryside of São Paulo, Brazil. My interest in computers began back in the "good old Apple II days." My first computer was an Apple II+ clone, and the first programming language I learned was AppleSoft Basic. We're talking about the 80s, when Brazil had a government policy called "market reserve," which practically prohibited the importation of computers. The idea was to foster the local industry, which at the time consisted basically of clones of the Apple II, MSX, and IBM-PC XTs—many of dubious quality. Needless to say, this policy only served to create a massive trade in smuggled computers and peripherals, mainly coming from Paraguay. If you wanted a slightly better computer, you had to buy it from the local "muambeiro" (smuggler), who in turn would go fetch it in Paraguay, traveling thousands of kilometers weekly, usually by bus, on trips known as "foguetão" (the rocket bus).

With a 1MHz CPU, whenever you needed better performance, you had to resort to Assembly, which led me to study the 6502 ASM. I followed the standard path of most "micreiros" (microcomputer hobbyists) of the time, moving through dBase, SuperVisicalc, Clipper, Turbo Pascal and, finally, Delphi.

Every developer has a moment when technology stops being just a curiosity and becomes a profession. What did that moment look like for you, and what kind of work filled your early career?

It's hard to say exactly when that happened because I'm one of those programmers who also codes for fun and not just out of obligation—in other words, programming ends up being a hobby as well. I can say that I developed my first commercial system while I still had the Apple II, at 11 years old. It was a rental management system for a relative's real estate agency. It was written in AppleSoft Basic, and the data was stored on floppy disks using structured text files. In the following years, even before going to college, I ended up creating various softwares for all sorts of purposes—initially using Turbo Pascal for the more "low-level" ones and Clipper for commercial softwares. When Borland released Delphi, I finally entered the world of Windows programming. Delphi/Pascal is my favorite IDE/language…

I still use it today!

Your path to Firebird began through Delphi and InterBase. Do you remember the first time you encountered InterBase? What was it about the technology—or perhaps the problems it solved—that made you want to explore it further?

Like many of you reading this interview, I arrived at InterBase through Delphi; after all, it came bundled in a standard Delphi installation. At the time, I had already migrated from Clipper to Delphi but was still using "desktop" databases, especially Paradox with the BDE (argh!). Indexes or even table corruption was common in that scenario, and those were the main reasons that led me to look for more robust solutions. Since InterBase was right there, ready to be used, I decided to give it a try and ended up falling in love with its reliability and simplicity of installation/configuration! Not to mention all the benefits of a true relational database! Having real transactional control and isolation, stability, SQL language, procedures, triggers, etc... a new world opened up! When Firebird was born, the migration to it was natural. The last version of InterBase I used was 6.0... after that, it was Firebird all the way!

Today, many people associate you with Firebird community initiatives—websites, books, and conferences—but those are only part of the picture. What does your professional life look like today? What kind of work keeps you busy when you are not organizing events or writing about Firebird?

In the beginning of my professional life, I developed several commercial systems for the most diverse purposes. At some point in the 90s, I ended up leaving most of them aside to focus on the one that was bringing the best financial return, which was an ERP aimed at resellers of surgical and hospital products. It has remained my "flagship" to this day. I also have some smaller systems generally related to this market niche, such as software for counting consigned products in hospitals and software aimed at healthcare logistics operators. These softwares undergo constant changes, especially due to the frequent tax legislation changes imposed by the Brazilian government, so a good part of my time is spent on software updates and customers support. I also do occasional consulting and training on Firebird.

It is funny to look back and see the amount of things I used to do at the same time:

developing software, supporting customers, writing articles for Delphi magazines, reviewing articles for SQLMagazine, writing books and articles for my site about InterBase and Firebird, actively participating in discussion lists, providing Firebird consulting, writing books and I still had time to ride my motorcycle and play guitar in a heavy metal cover band! Today, if I manage to do three of those things, it's a lot (lol)! Life probably got more complicated, or I'm the one getting slower—or probably both ☺

You have written several books about Firebird, including Firebird Essencial and migration guides for newer versions. What inspired you to take on the challenge of writing books? Was there a moment when you felt the community needed something that simply did not yet exist?

I enjoy writing! Even before launching my first book, I was already writing articles for computer magazines like ClubeDelphi, as well as articles for FireBase.com.br (my site/portal initially dedicated to InterBase and later to Firebird).

The idea of launching a book about Firebird was motivated by the lack of quality documentation dedicated exclusively to it. As we know, Borland open-sourced the InterBase 6 code, but not the documentation. The main source of information at the time was discussion lists and articles on websites scattered across the internet. Bookstores only had books on MySQL, SQL Server, Oracle, etc., but nothing on Firebird!

Since I had already written many articles about Firebird for ClubeDelphi, it seemed like a good idea to revise them and make them available in book format, plus a brand-new chapter on creating UDFs. That's how the book "Firebird Essencial" book came about, and it stayed on the bestseller list of the Ciência Moderna (Modern Science) publisher for months.

The second book dealt specifically with the new features of the second version of Firebird. Later came the Migration Guides (for FB 3, 4, and 5). The guides for versions 3 and 4 were also made available to the international market (in English), in both print and ebook formats (anyone can buy them at firebirdnews.org or Amazon).

The first two books were released in the traditional model through Ciência Moderna (which further released a Portuguese version of "The Firebird Book" by

Helen Borrie). The other books were completely independent projects, where I was responsible for every stage, from writing and editing to layout and publication in both physical and digital formats. Ann Harrison helped with the proofreading of the English versions.

The Migration Guide for Firebird 3 had hundreds of copies sold before it was even released ☺

Writing technical books is a demanding and often solitary endeavor. Looking back, what did the process teach you—not only about Firebird, but about the way developers learn and share knowledge?

More than solitary or laborious, writing has always been a pleasurable task. Having worked for months as an article reviewer for SQL Magazine, I learned a lot about how to write and layout good articles! Before being an author, I was already an avid reader of programming magazines and books, which helped shape my own identity as an author. I've always tried to write in a light, (sometimes) fun, and direct way, focusing on topics useful for solving real daily problems for developers.

Writing the books also taught me a lot about Firebird itself. No matter how well you know a subject, when you decide to put everything "on paper," you need to dive deep into every topic you cover, and most of the time, this leads to many discoveries and new insights. Fortunately, I was able to count on the help of several core developers to clear up my questions during the writing process.

It seems to me that nowadays younger developers prefer the video lesson format. Perhaps due to a "laziness" to read or the ease of following a subject "visually," but I personally still prefer books—and if possible, printed ones!

Firebird Developers Day has become a remarkable institution in the community. When you first imagined the event, did you expect it to grow into something so enduring? What were those early editions like?

I knew it was going to work, and I can tell you why: even before the first FDD, I was the president of DUG-BR (Delphi Users Group Brazil), which aimed to bring knowledge about Delphi through events with lectures in cities outside the Rio-São Paulo axis at a symbolic price (the premise was that the registration fee could never be a limiting factor for someone's participation). We didn't have any experience organizing conferences when we decided to do the first Delphi

Developers Day (DDD), but it was such a huge success that I immediately had the idea to do an event in the same mold but focused exclusively on Firebird. Unlike the DDDs, which were organized by the DUG-BR team, the FDD has always been organized exclusively by me.

The first FDD was born with an audience of 400 people! We used a great university structure consisting of a large theater, an amphitheater, and a hall—meaning we always had three lectures happening simultaneously all day long. The second FDD had our largest audience to date: about 620 people! It was also the first to feature international speakers: Ann Harrison and Jim Starkey, considered the mother and father of InterBase/Firebird. Having international speakers at an event in Brazil increases the challenge and complexity significantly because, even among the IT crowd, the vast majority of Brazilians do not speak or understand spoken English; therefore, a translator is necessary. It turns out that lectures aren't a movie script where all the lines are already written and known beforehand. There is a lot of improvisation, so the translator has to know the subject; otherwise, we risk hearing nonsensical phrases—like a moment when the translator found the use of the word "corruption" (as in "database corruption") strange and felt free to comment, live, that "corruption was a thing for politicians" ☺ Yes! That actually happened in one of Ann's lectures! And things only didn't get worse because one of the people in the audience (who was fluent in English and also was a Firebird user and author of Delphi books) offered to take over the translation. Thank you, Maurício!

Conferences are as much about people as about technology. Over the years, what have you learned from meeting Firebird users face to face—about how they use the database, what challenges they face, and what keeps them coming back?

In 22 years of the event, it's interesting to note that some themes are recurring. I believe the most requested topic to this day is performance optimization! In the past, "recovering corrupted databases" was as requested as optimization. Fortunately, this has decreased in recent years—a likely sign that cases of database corruption are becoming rarer (unfortunately, the same cannot be said of politics, lol).

At least in Brazil, Firebird is primarily used in commercial software. You can find it in ERPs, hospital management systems, medical clinics, banks, real estate agencies,

and hundreds of accounting software. But its use goes beyond typical commercial software, for example, it is used to store gameplay data and run 98% of the logic (in PSQL) for a game called Data Quadrants Horizon (dqhorizons.com). The embedded version is also fantastic for providing software demos or single-user systems, offering the full power of Firebird without even needing to install it!

It's also interesting to note that no matter how much you talk about the importance of good transactional control and no matter how much you explain how versioning works, every time there are lectures on these subjects, I see "astonished" faces on many people, as if they were hearing it for the first time. It's also curious that many people come to the event more for the networking than to watch the lectures! It's not rare for people to form discussion circles in the open areas of the event to exchange ideas and experiences.

The pandemic forced many technical communities to rethink how they gather. FDD also moved from primarily live events to an online format. How did that transition change the spirit of the conference, and do you see advantages in the new format?

As much as I prefer in-person events, that wasn't an option at that time, as gatherings of people were prohibited. The fact that we never skipped a single year of FDD led me to hold the conference in an online format in 2020 and 2021. The 2020 event was a "BestOf" edition with the most requested topics (via voting) from previous editions, featuring one lecture per night for a week. The 2021 event followed the same scheme but with brand-new topics.

Online events have some advantages, the main ones being reduced cost for the participant and the organizer, and the fact that people anywhere in the world can participate as long as they have internet. However, much of the networking, interactivity, and face-to-face contact with speakers and sponsors present at the booths is lost.

We tried to compensate for this by dedicating time at the end of each lecture for a live Q&A, where participants could send questions or comments via chat, answered by the speakers in real-time.

In recent years you also started projects like Developer's Pub, where you interviewed members of the Firebird community. What motivated you to begin

those conversations, and do you see that format continuing or evolving in the future?

Firebird has provided me with many experiences over more than two decades, including contact with interesting people from all over the world. I ended up becoming friends with many of them, with whom I maintain contact to this day. The idea of Developer's Pub is to "interview" especially the Core Developers as if we were having a few beers and chatting in a bar—that is, a light and relaxed conversation where the public could get to know more about the lives of the people involved with the Project.

We started with big names like Ann Harrison and Dmitry Yemanov. However, my attempts to interview other core developers were frustrated. Apparently, most of them are too shy and/or don't feel comfortable participating, especially because they often have to speak in a language that isn't their native one (mind you, I can certainly relate—my own English is largely self-taught from the pages of old Apple II magazines, so I'm no stranger to that struggle!).

Despite being frustrating, I have to respect everyone's choice, but I'm disappointed to lose the chance to bring a bit more visibility to Firebird through Developer's Pub. I would indeed like to release new episodes, but unfortunately, it doesn't depend only on me.

Finally, after so many years working with Firebird—as a developer, writer, organizer, and community builder—what still excites you about the database and the people around it?

What continues to surprise me about Firebird is how good, stable, and easy to configure it is. You only need to look at the size of the installers and how much it delivers compared to other RDBMS! It also surprises me how much it has evolved with so few people effectively working dedicatedly on the project's code. If we compare the number of active core developers with the number of users worldwide, it's unbelievable what these few people manage to deliver to so many!

Looking toward the future, there are some very interesting things being implemented for the upcoming versions that will certainly please many users! You just need to check the Firebird 6 Roadmap to see some of them, such as: native JSON support, Tablespaces, Schemas, Task Scheduler, and much more.

I would be very happy if more people joined the Project, not just as users, but mainly as Firebird developers. Imagine where and how fast we could get with more people getting their hands dirty?! I also can't forget the others involved who aren't core developers but who act in many other ways to keep Firebird alive, whether by writing documentation, taking care of the bureaucratic part, or helping answer questions on the discussion lists, etc. My thanks go to everyone who contributes to the success of Firebird, whether by financially sponsoring the Project or by dedicating part of their time to help the Project in some way.

Brooks' Law

> *Adding human resources to a late software development project makes it later.*

This law suggests that in many cases, attempting to accelerate the delivery of a project which is already late, by adding more people, will make the delivery even later. Brooks is clear that this is an over-simplification, however, the general reasoning is that given the ramp-up time of new resources and the communication overheads, in the immediate short-term velocity decreases. Also, many tasks may not be divisible, i.e. easily distributed between more resources, meaning the potential velocity increase is also lower.

The common phrase in delivery "Nine women can't make a baby in one month" relates to Brooks' Law, in particular, the fact that some kinds of work are not divisible or parallelisable.

This is a central theme of the book 'The Mythical Man Month'.

Development update: 2026/Q1

A regular overview of new developments and releases in Firebird Project

Releases:

- Jaybird 6.0.4 & 5.0.11 and Jaybird 6.0.5 & 5.0.12, released 22th Jan & 27th March 2026
- Entity Framework Core 10 provider for Firebird, released 8th February 2026

New chapter in Firebird Internals

A new chapter was added into the Firebird Internals manual describing the External Table Format.

Created Local Temporary Tables

Firebird 6.0 will introduce support for SQL Created Local Temporary Tables (LTT). Unlike Global Temporary Tables (GTT), which have permanent metadata stored in the system catalogue, Local Temporary Tables exist only within the connection that created them. The table definition is private to the creating

connection and is automatically discarded when the connection ends. The data lifecycle depends on the `ON COMMIT` clause: with `ON COMMIT DELETE ROWS` (the default), data is private to each transaction and deleted when the transaction ends; with `ON COMMIT PRESERVE ROWS`, data is shared across all transactions in the connection and persists until the connection ends.

Local Temporary Tables are useful in scenarios where you need temporary storage without affecting the database metadata:

- Since LTT definitions are not stored in the database, they can be created and used in read-only databases. This is not possible with Global Temporary Tables, which require metadata modifications.
- Each connection can create its own temporary tables with the same names without conflicts. The table definitions are completely isolated between connections.
- LTTs provide a quick way to create temporary storage during a session for intermediate results, data transformations, or other temporary processing needs without leaving any trace in the database after disconnection.

Comparison with Global Temporary Tables

Functionality	GTT	LTT
Metadata Storage	Stored in system tables (e.g. RDB$RELATIONS, etc.)	Stored only in connection memory
Visibility of Definition	Visible to all connections	Visible only to the creating connection
Persistence of Definition	Permanent (until explicitly dropped)	Exists only until the connection ends
Read-only Database Support	Not supported	Supported
Schema Support	Supported	Supported
Indexes	Full support	Basic support (no expression or partial indexes)

Functionality	GTT	LTT
DDL Triggers	Fire on CREATE / DROP / ALTER	Do not fire
DML Triggers	Supported	Not supported
Constraints (PK, FK, CHECK)	Supported	Not supported
Explicit Privileges	Supported	Not supported

Goodhart's Law

> *Any observed statistical regularity will tend to collapse once pressure is placed upon it for control purposes.*
> *Charles Goodhart*

Also commonly referenced as:

> *When a measure becomes a target, it ceases to be a good measure.*
> *Marilyn Strathern*

The law states that the measure-driven optimizations could lead to devaluation of the measurement outcome itself. Overly selective set of measures (KPIs) blindly applied to a process results in distorted effect. People tend to optimize locally by "gaming" the system in order to satisfy particular metrics instead of paying attention to holistic outcome of their actions.

Real-world examples:

- Assert-free tests satisfy the code coverage expectation, despite the fact that the metric intent was to create well-tested software.
- Developer performance score indicated by the number of lines committed leads to unjustifiably bloated codebase.

Toolbox: SQLServerBooster

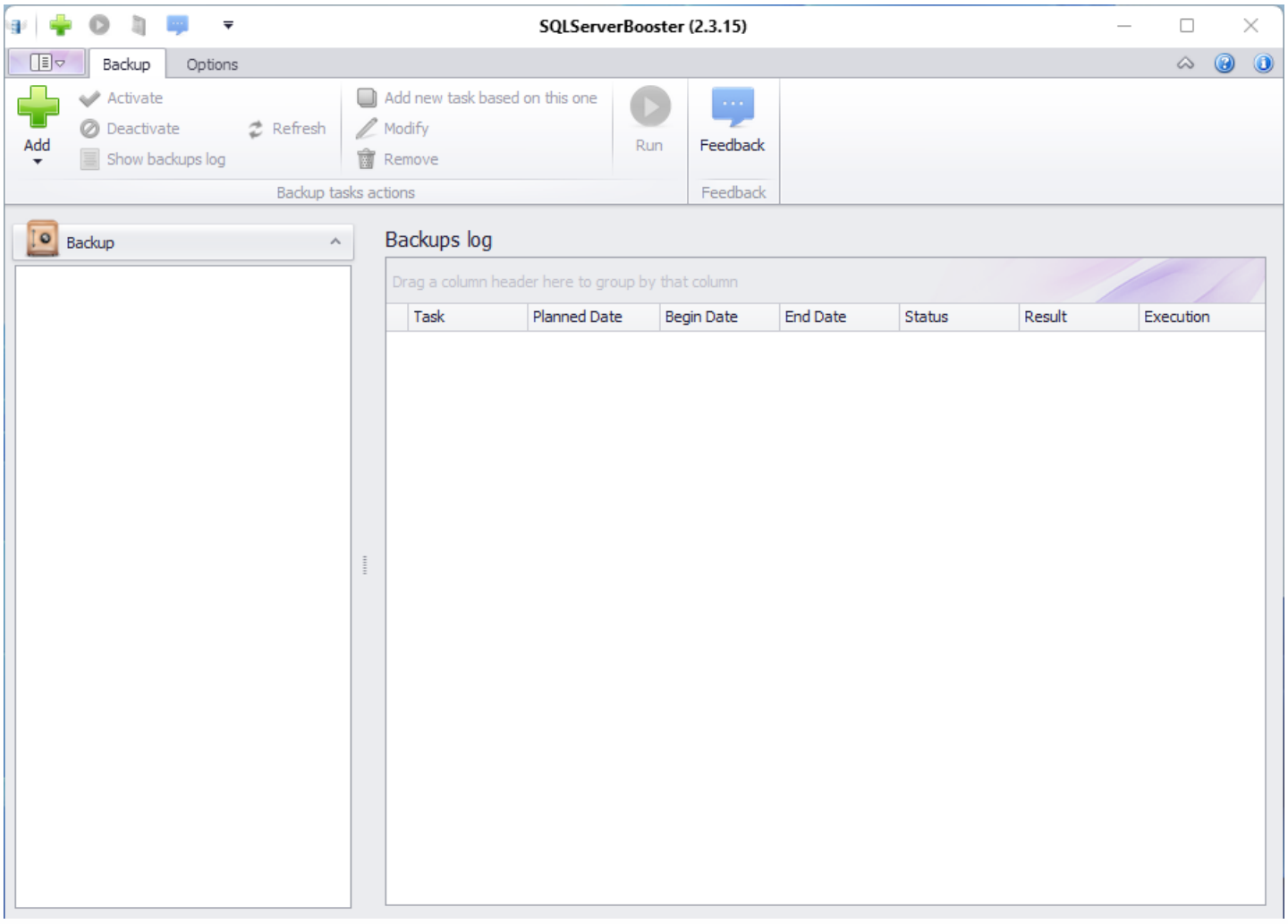
Reliable backups remain one of the most fundamental responsibilities of any database administrator. Regardless of the database engine in use, a robust backup strategy must ensure that data can be preserved, transferred off-site, and restored when necessary. While most database systems provide their own native backup utilities, real-world environments often require more than simple local dumps: scheduled execution, compression, encryption, and automated delivery to remote storage are frequently essential components of a modern backup workflow.

SQLServerBooster is a small free utility developed by Erik Véliz that aims to address precisely this operational layer. Originally created for Microsoft SQL Server environments, the tool has evolved into a more general database backup automation utility capable of working with multiple database systems, including MySQL, PostgreSQL, Oracle, and Firebird. Its core purpose is straightforward: automate database backups, optionally compress and encrypt them, and deliver the resulting files to various destinations such as FTP servers, network shares, or cloud storage services.

For this review we tested SQLServerBooster version 2.3.15, evaluating its capabilities and usability specifically from the perspective of Firebird administrators. The tests were conducted against Firebird 5, focusing on how effectively the tool integrates with Firebird backup processes and how suitable it is as part of a practical Firebird backup strategy.

Installation of SQLServerBooster is straightforward and follows the typical pattern of Windows desktop utilities. The installer guides the user through a short sequence of steps without requiring any special configuration or prerequisites. In our tests the installation completed quickly and without complications.

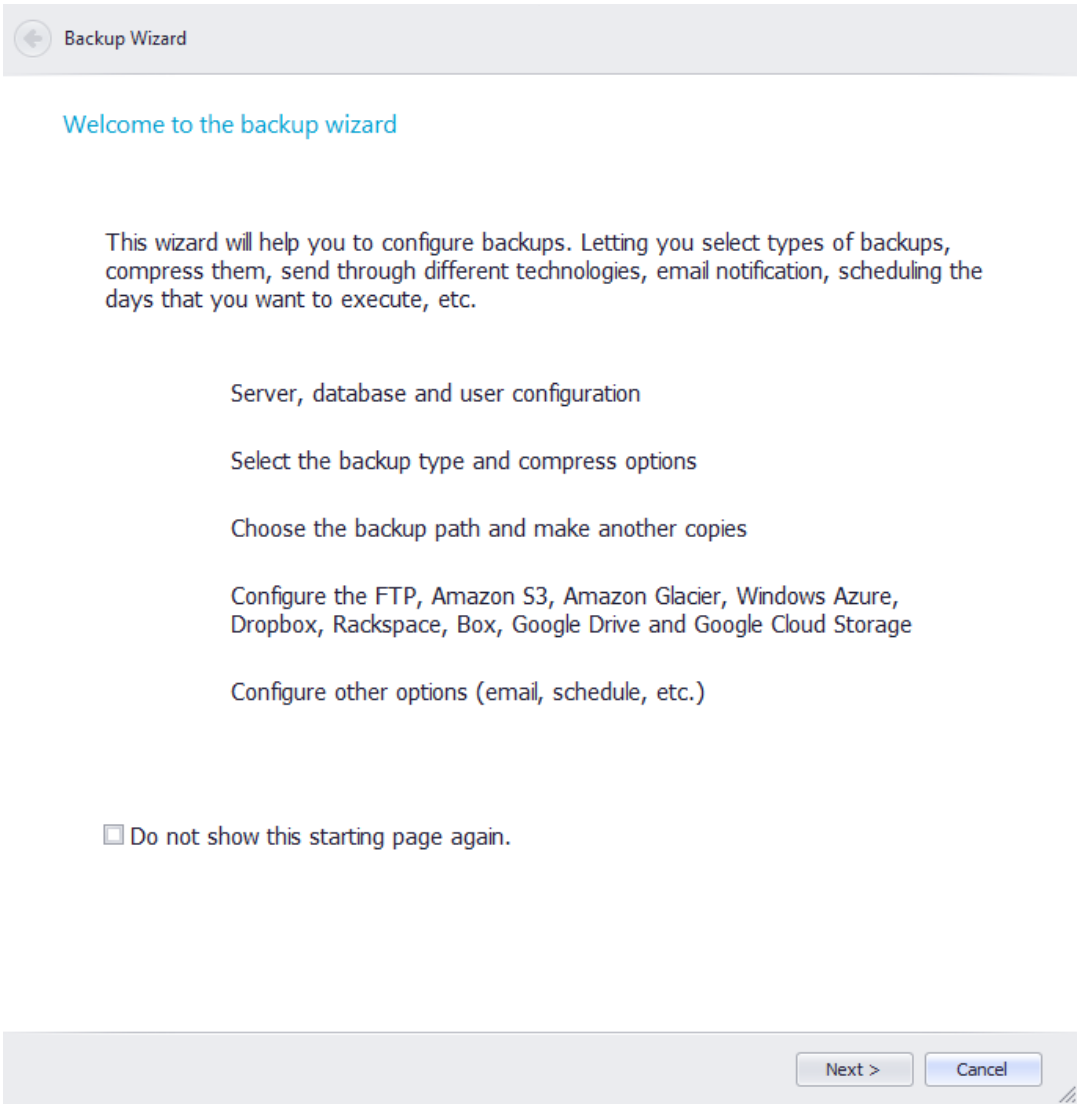
After launching the application for the first time, the main program window is displayed. This window serves as the central dashboard of the tool and lists all configured backup tasks. Each task represents a defined backup job together with its schedule and associated parameters, providing a clear overview of the configured automation.



An important operational detail is that closing the main window does not terminate the program. Instead, the application continues running in the background and is minimized to a tray icon in the Windows notification area, allowing scheduled backup tasks to continue executing automatically without requiring the main interface to remain open.

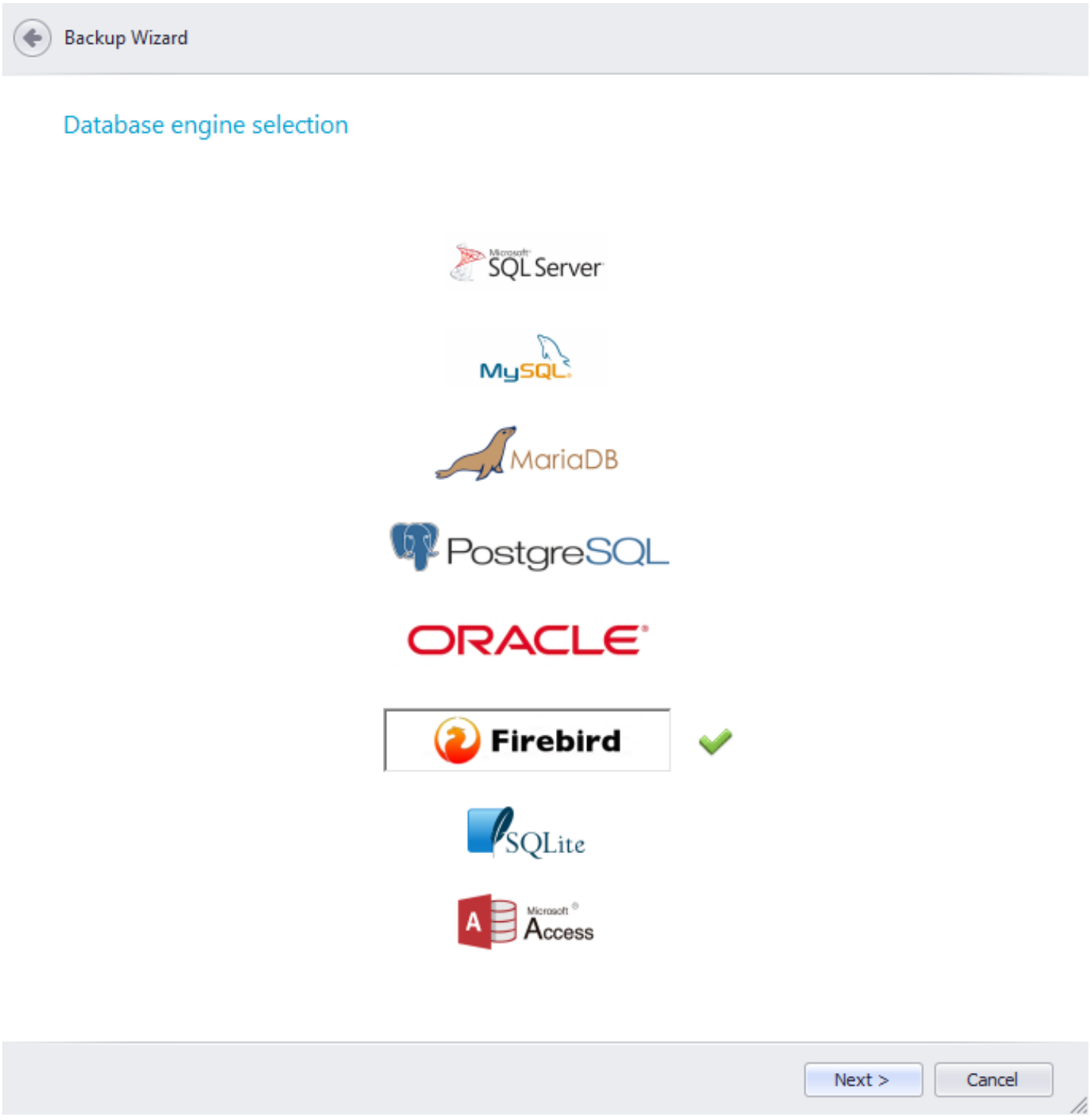
Backup tasks in SQLServerBooster are created using a configuration wizard that guides the user through the necessary steps. The process is well structured and easy to follow, which makes the setup of even more complex backup workflows relatively straightforward.

The wizard opens with an introductory panel that briefly explains the overall procedure. This screen serves mainly as orientation for first-time users. For convenience, the introduction can be automatically skipped when the wizard is opened again in the future.



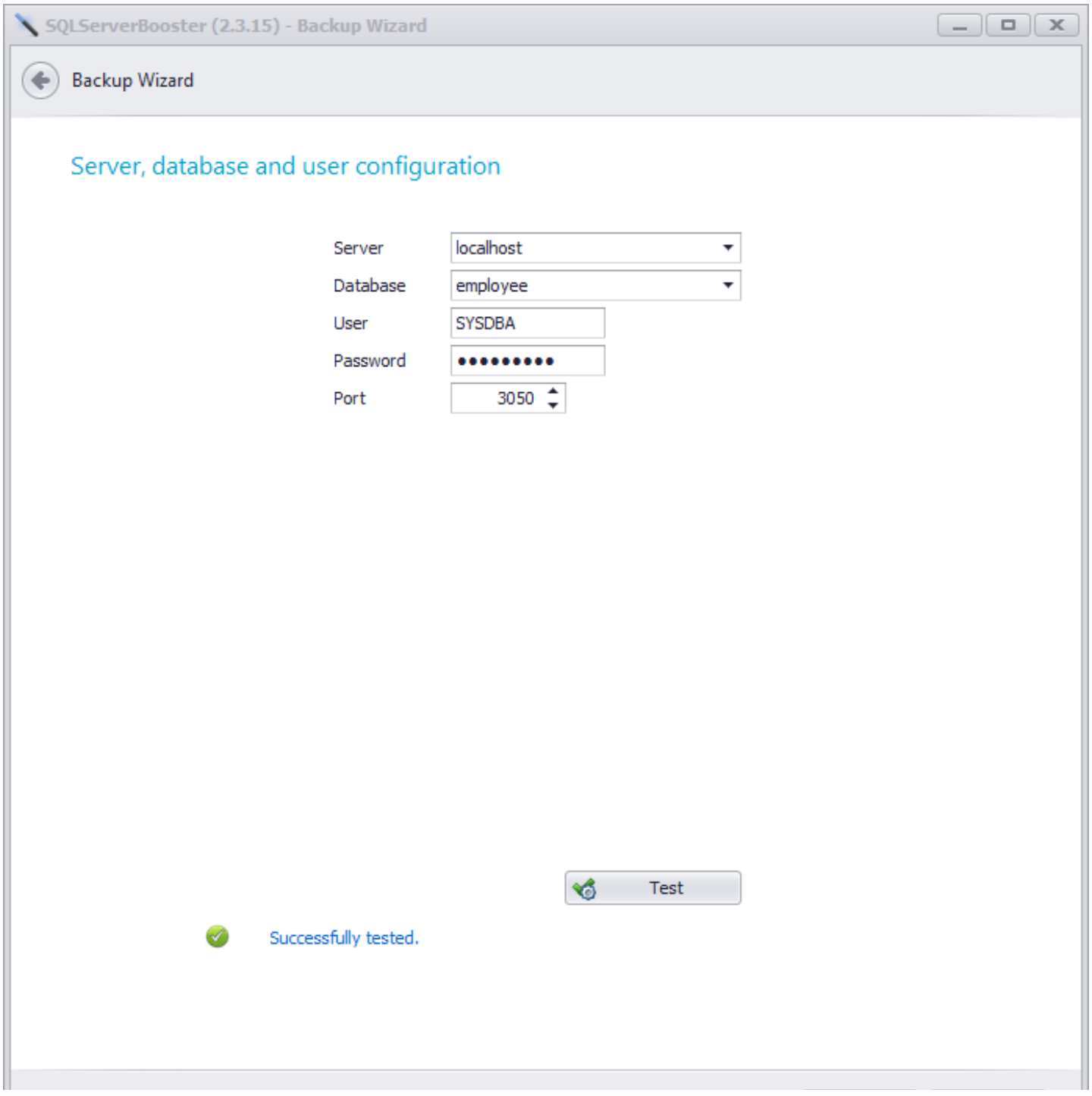
The next step is the selection of the database engine for which the backup task will be created. SQLServerBooster supports several database systems, and the choice here determines which options will be available in the subsequent configuration panels.

Adding task - Database type selection



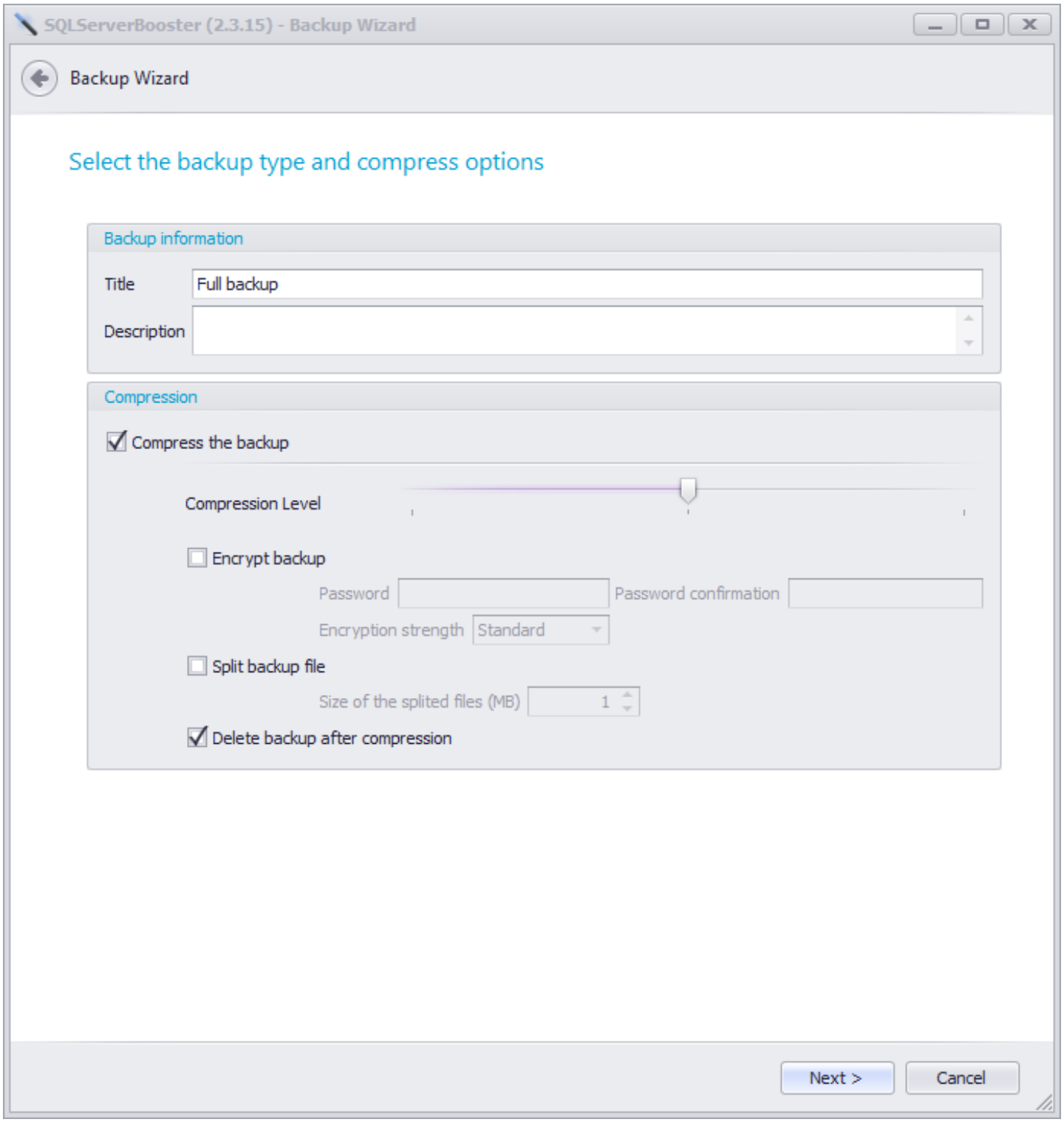
The following dialog configures the connection to the database that will be backed up. The required connection parameters can be entered directly and the wizard provides a button to test the connection. We strongly recommend using this option to verify the configuration before proceeding. During our tests with Firebird 5 it was necessary to enable Legacy authentication in the Firebird server configuration, which suggests that the application currently relies on an older Firebird client library.

Adding task - Database connection



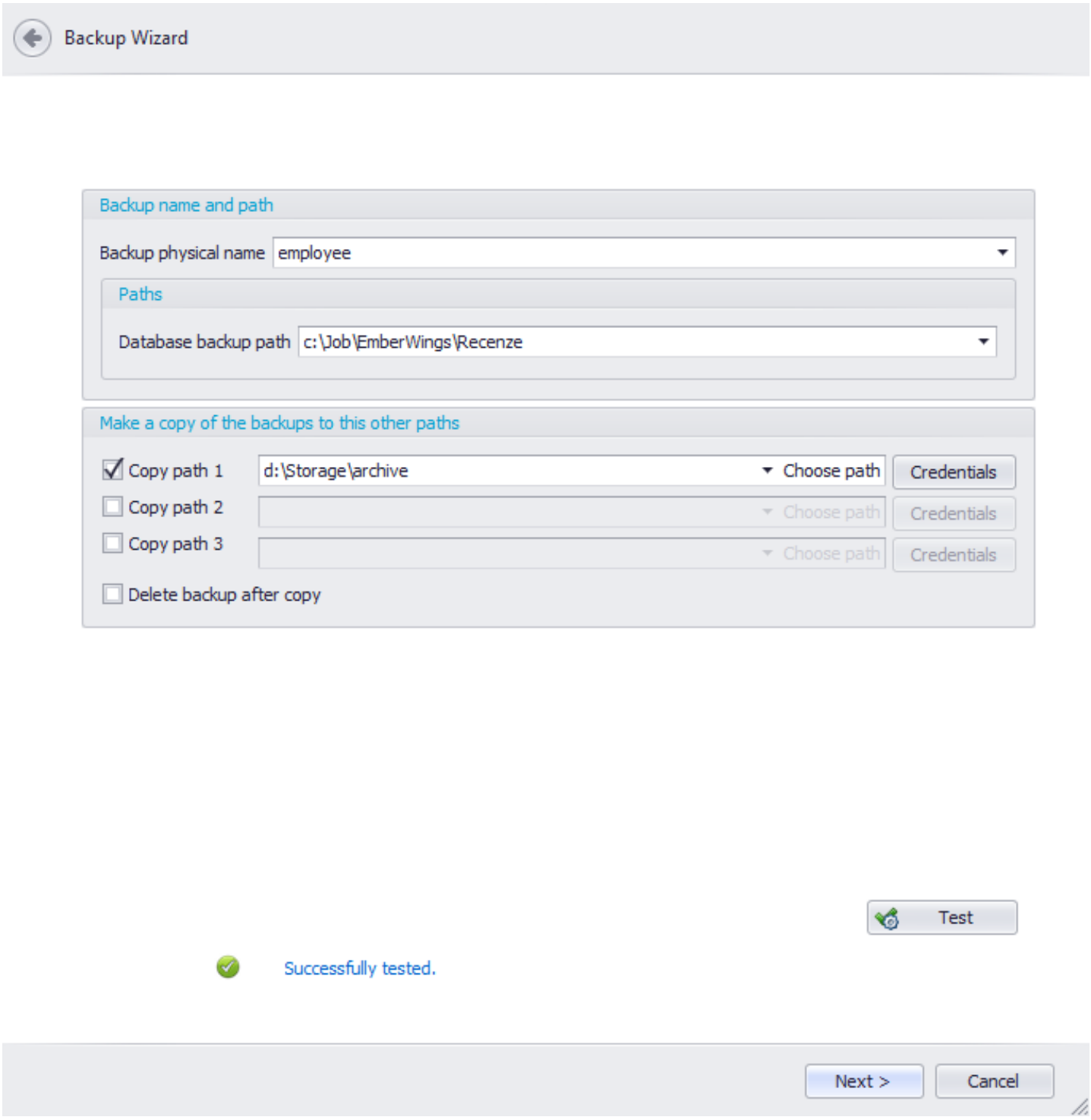
Next, the wizard allows the user to select the backup type and compression options. In the case of Firebird the backup method is always logical, using the standard gbak utility. The resulting backup file can optionally be compressed into a ZIP archive, which is often desirable when backups are transferred to remote storage.

Adding task - Backup parameters



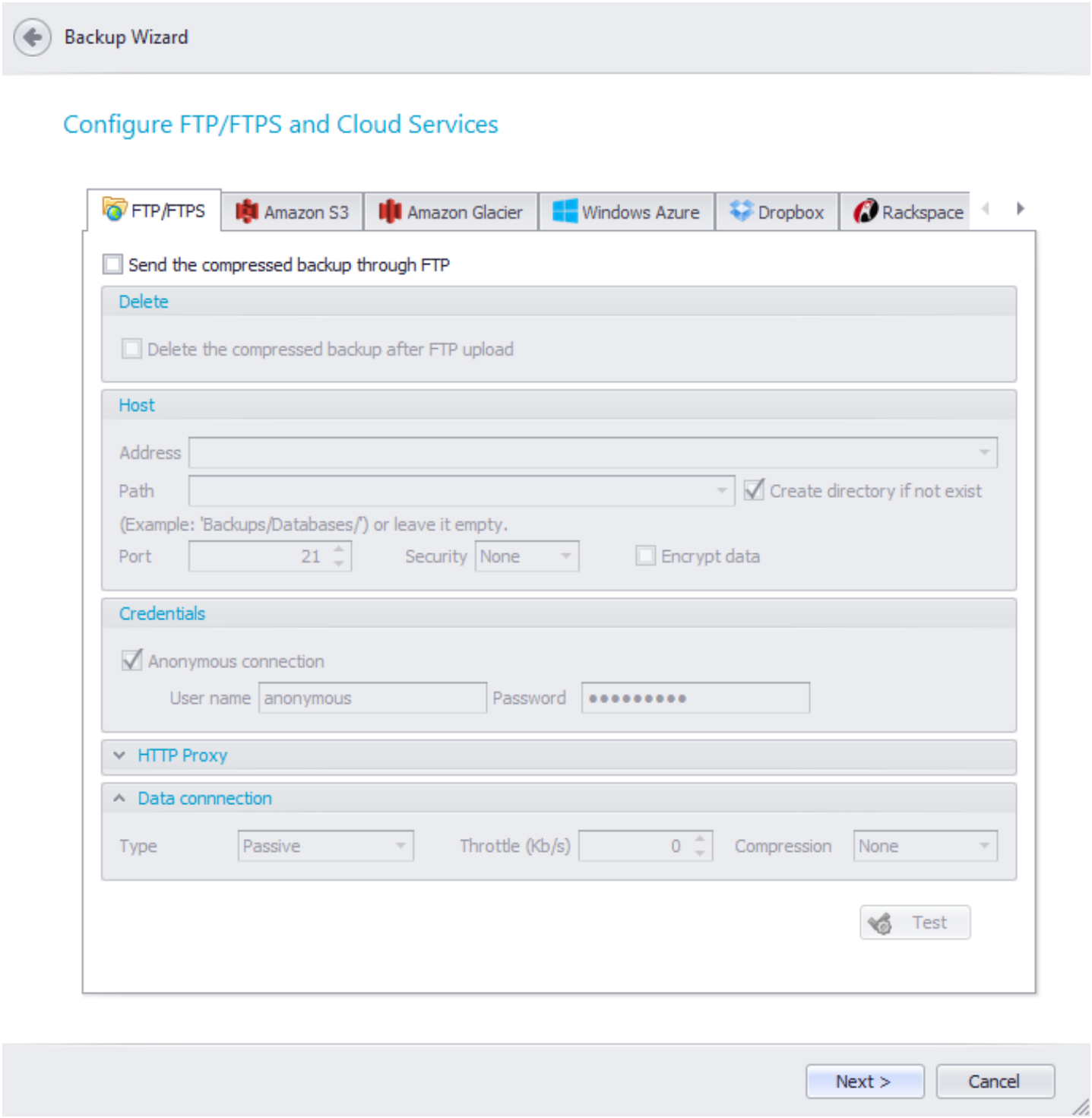
The subsequent panel defines the backup destination. In addition to the main repository directory, the tool allows the same backup file to be copied to up to three additional directories. This provides a simple way to maintain multiple local backup copies.

Adding task - Backup destination



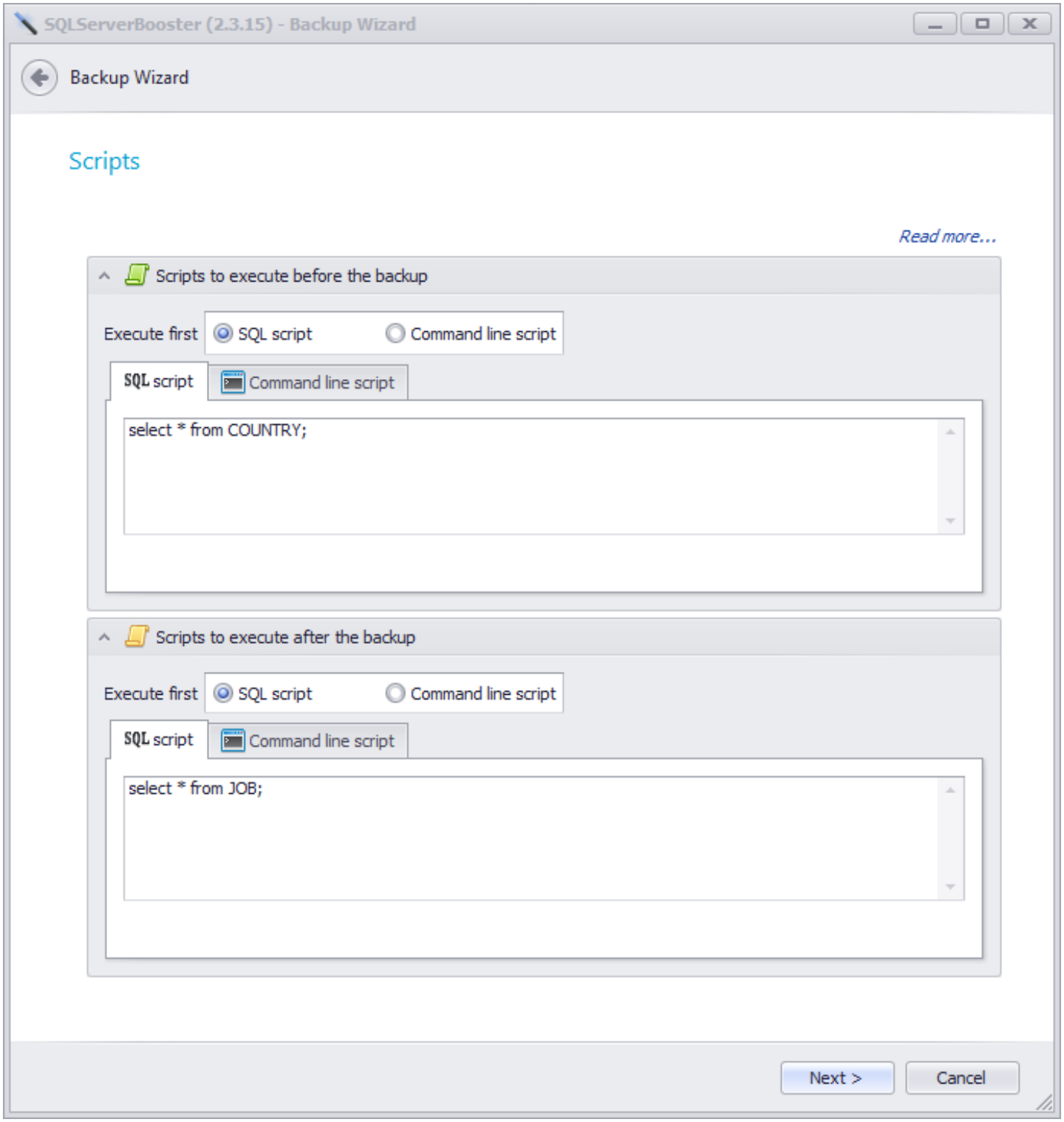
Another optional step enables uploading backups to FTP servers or various cloud storage services. These settings allow the generated backup files to be transferred automatically to external storage locations after the backup process completes.

Adding task - remote storage



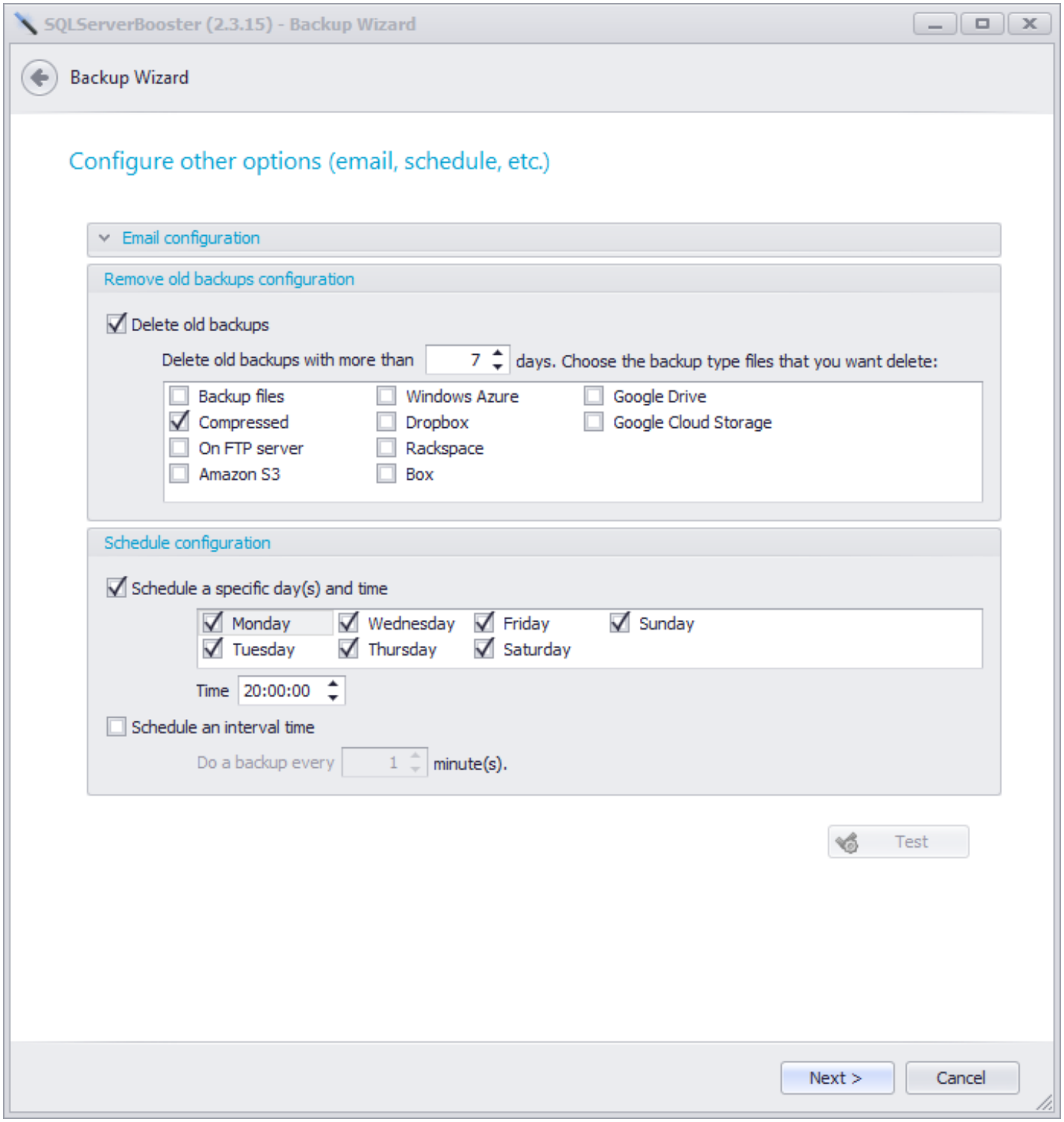
The wizard also provides the option to define scripts that are executed before and after the backup operation. These scripts may be either SQL statements or operating system commands, allowing administrators to integrate the backup process with other maintenance tasks.

Adding task - scripts



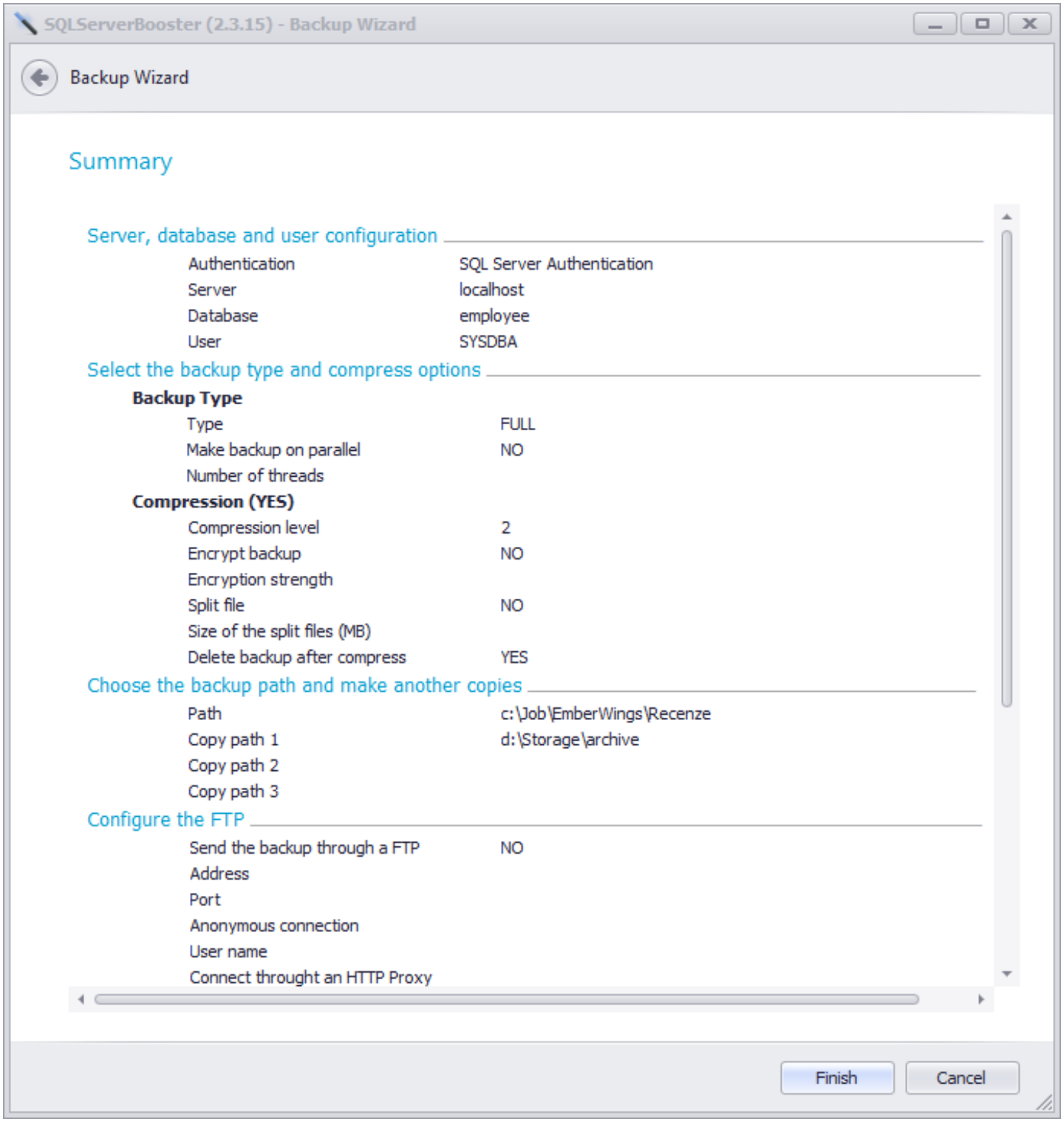
Further configuration options include email notifications, automatic deletion of older backup files according to defined retention rules, and scheduling of the backup task. Through this scheduling mechanism backups can be executed automatically at regular intervals.

Adding task - Notification, cleanup and schedule



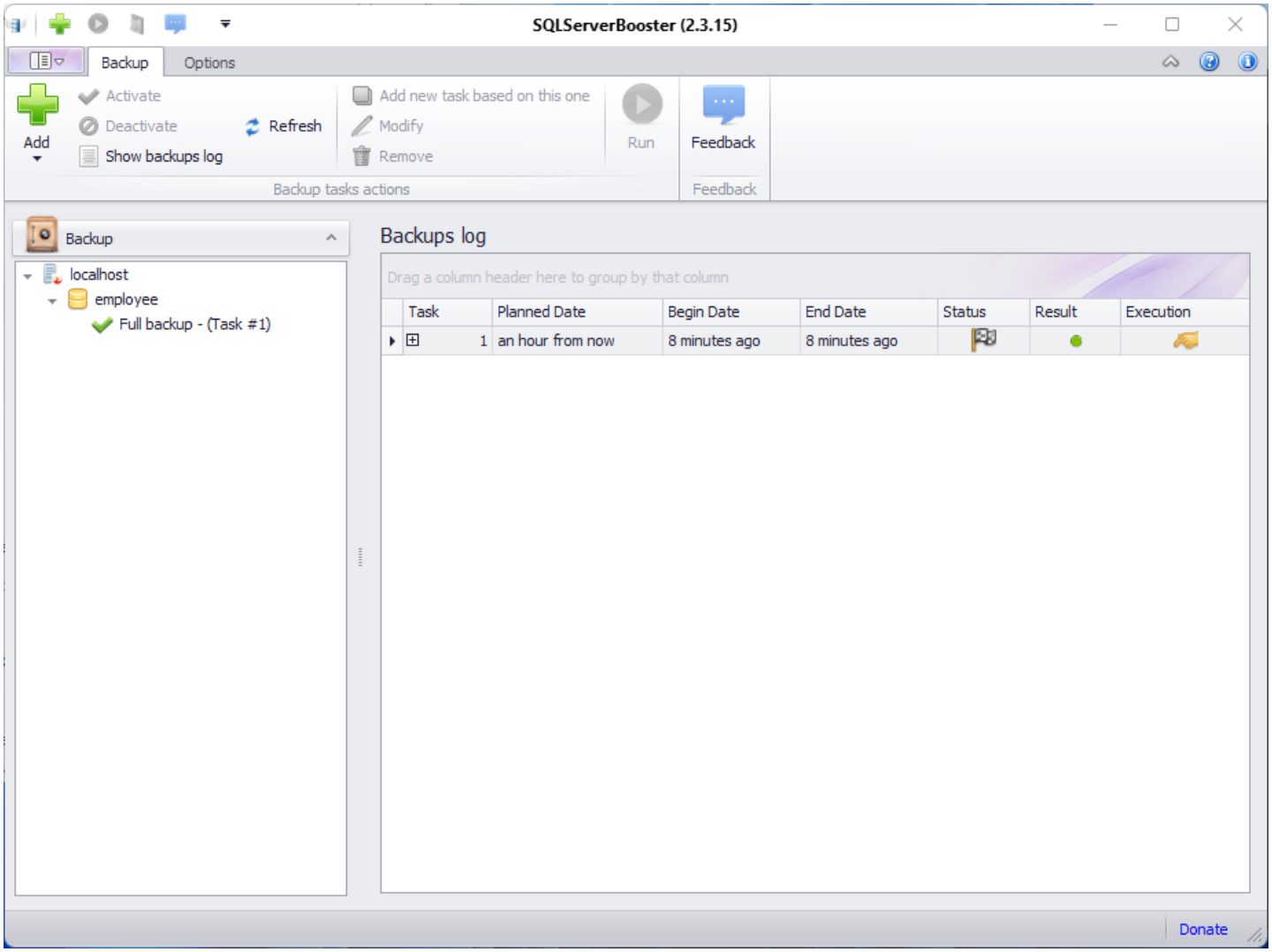
Finally, the wizard presents a summary screen showing all configured parameters of the task. After confirming this overview, the backup task is created and appears in the main application window where it can be monitored or modified later if needed.

Adding task - Summary



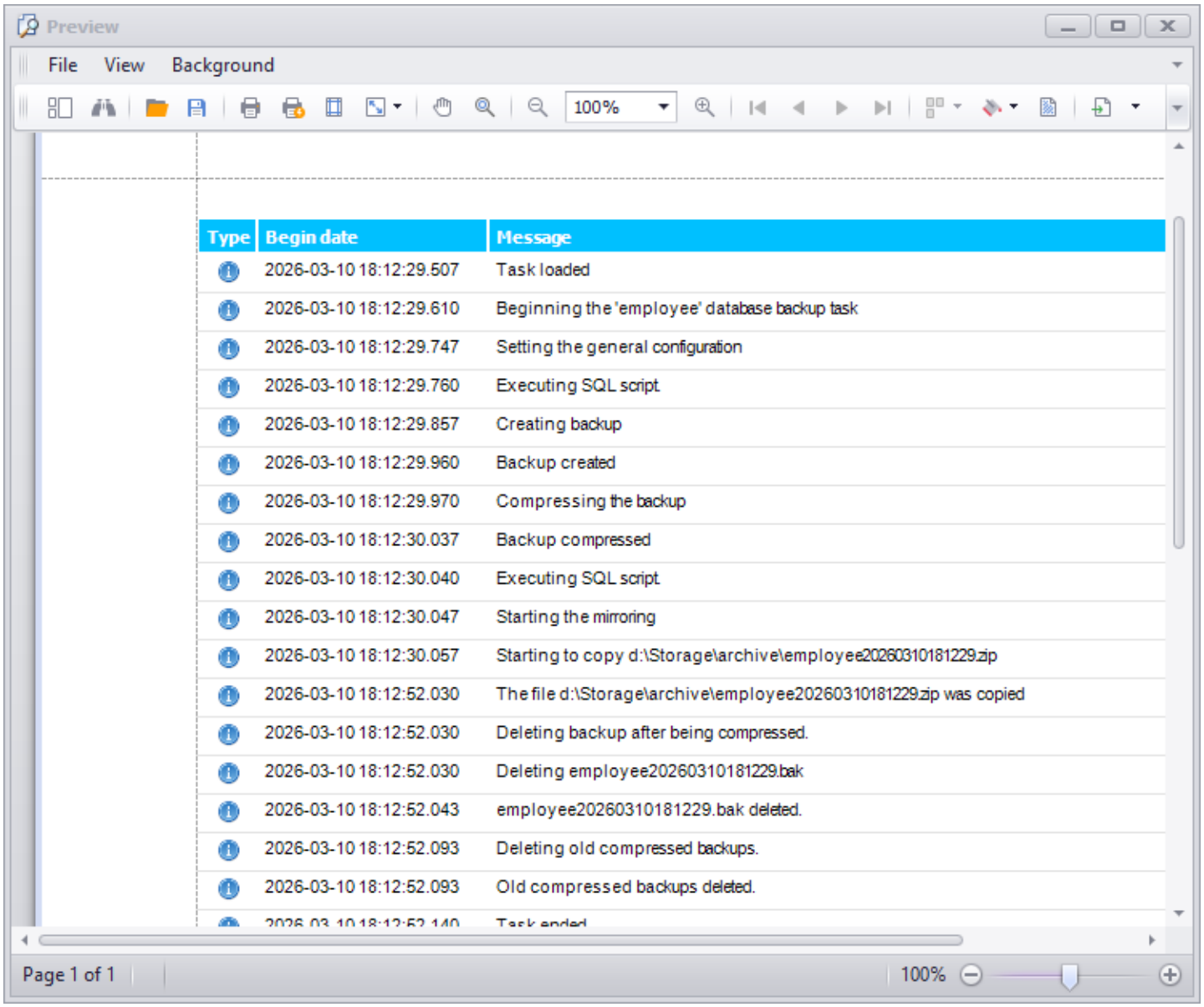
Once defined, backup tasks are executed automatically according to their configured schedule. They may also be started manually at any time directly from the main application window, which is useful for testing a new configuration or creating an immediate backup outside the regular schedule.

Main Window with tasks



For each task execution the application records a detailed log. This log contains information about the progress of the backup operation and any messages produced during the process, allowing administrators to verify successful execution or diagnose potential problems.

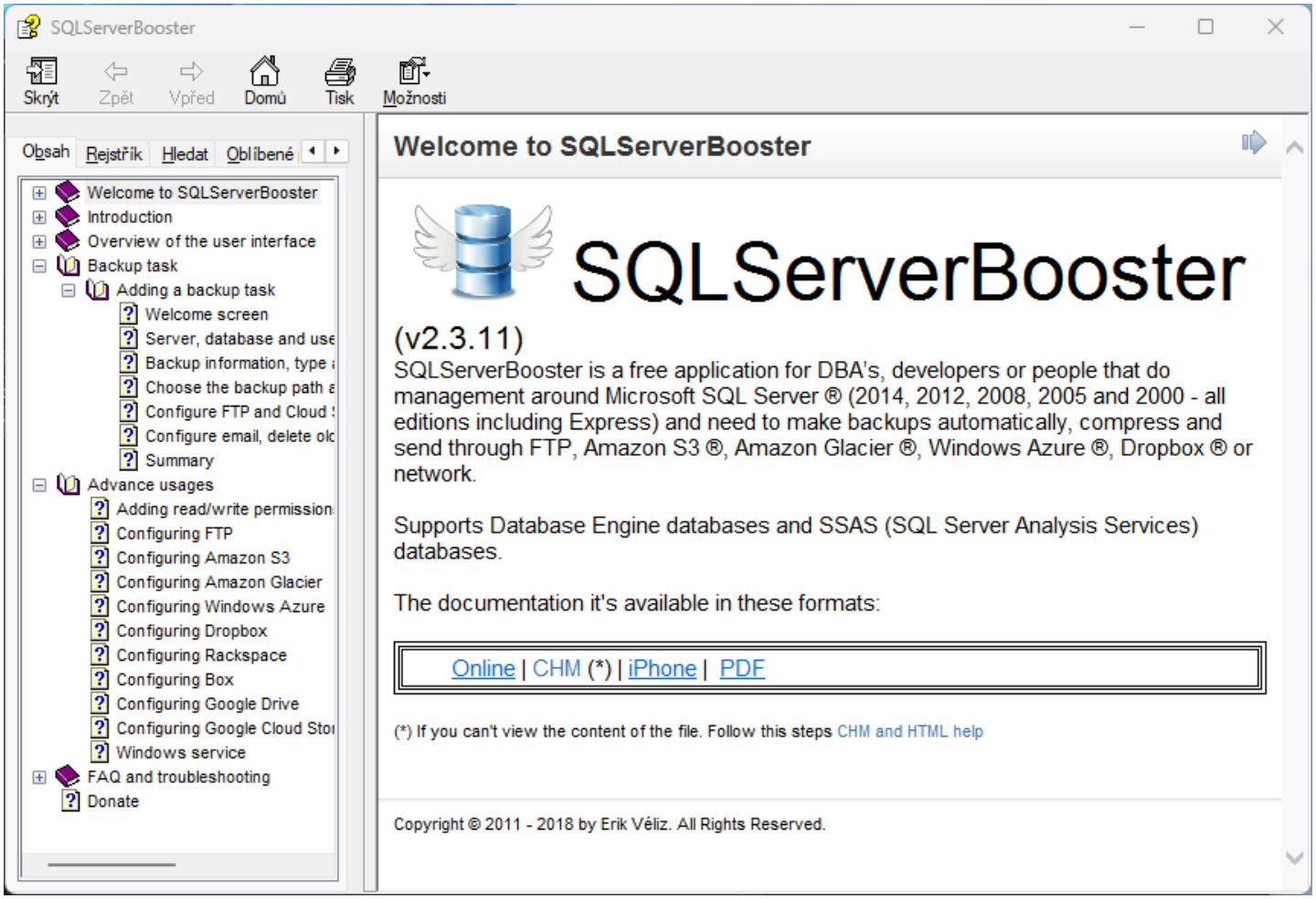
Task execution log



The user interface offers several customization options, including the ability to change the graphical theme. Among the available themes is a dark mode, which may be preferable for users who spend long periods working with administrative tools.

The application also includes built-in help documentation accessible from the interface, providing guidance on configuration options and general operation.

In our tests the application behaved consistently and without issues. Backup tasks were easy to configure, executed reliably, and the surrounding operational features—such as logging, scheduling, and retention management—worked as expected, making routine backup administration straightforward.



Although our evaluation focused solely on Firebird and did not include testing with other database engines or cloud storage integrations, the functionality we used proved stable in practice. Based on this experience, SQLServerBooster can be recommended as a practical graphical tool for administrators who want to automate and manage their Firebird backups.

Sometimes, the right QUERY is the only thing separating you from the hidden secrets of Firebird.

Answers to your questions

Documentation is said to be a collection of answers to unspoken questions. If you ask a search engine, it will answer you with a link to a document that (hopefully) contains the answer. There are documents, forums and entire systems like Stack Overflow that consisting only of questions and answers. And now an army of AIs is starting to chase us to answer our questions. Questions and answers cannot be avoided, there is no hiding place.

However, amidst the sea of routine questions and responses, there lie truly captivating inquiries and answers, like hidden treasures. Our commitment is to regularly present you with a curated collection of these precious gems.

This time about:

- A Lesson about Sort
- What is better, null or empty string in index?
- Interesting lesson about DDL and commit

A Lesson about Sort

woodsmailbox asked:

A huge sort table gets created (and lousy performance) depending on what I put in the select list, not what I put in the order-by clause. SORT is the root of the plan, but why does it include other columns that are not included in the order-by clause in it? I'd wish it would choose to retrieve the other values *after* the SORT. Currently, I trick the optimizer by creating a view on top, but this trick might not work in a future version of firebird and I'll be back where I started.

Btw, in fb 2.1.2 or 2.5, does the optimizer try to inline stacked views (views that select from views)?

Ann W. Harrison answers:

Generally, it is much faster to retrieve records in storage order than to access them randomly. Unless you happen to store records in the order that you're sorting them, selecting after the sort will produce much more I/O on the database than you're currently seeing on the sort file.

Optimizer inlines stacked views when possible, which it is if the inner views are simple joins, meaning that they don't includes group by or aggregation. That's been the case since Firebird was InterBase.

woodsmailbox reply:

Not sure how your answer is related to the problem described. Firebird builds a 1.2GB sort table for ~40 seconds while CPU = 100%.

This only happens when I also include a 4k utf8 varchar column in the *select list*. If I don't include that column in the select list, fb makes a sort table of 300K. The respective column is not part of the ORDER BY clause, so it shouldn't affect performance that much. This is quite a show stopper for me since the query is very simple and the select is a simple join of two tables of 6000 x 20 rows. Also, the execution plan shows no NATURAL joins, and SORT is at the root of the plan. The performance is so lousy for such small tables that I incline to think it's a bug. Besides, how can it build a 1.2GB sort table from a 45MB database on a simple join

+ order by? Is this normal performance?

Ann W. Harrison answers:

I tried to answer you're question which was, if I understand correctly, why doesn't firebird retrieve only the sort keys on the first pass, then sort them, then go back and retrieve all the other items in the select list. It doesn't because generally, I/O to the database is the most expensive operation it does, and doing it twice, once in storage order and once in random order relative to storage is likely to be slower than retrieving all the rows in the first pass.

"This only happens when I also include a 4k utf8 varchar column in the select list."

Yup. A 4k utf8 varchar field generated a buffer of 12 or 16k - depending on whether the current code is prepared to handle the absolute worst case. In this case, a blob would be a lot faster.

The first element of the plan is the last one executed - which would be the sort. Firebird normally executes indexed access in two stages, first getting all qualifying values from the index and setting bits in a storage-order bitmap, then second retrieving the rows in bitmap order. So you get storage order access even when an index is used.

Dmitry Yemanov adds:

First issue is the one explained by Ann: the engine always prefers two sequential (i.e. storage order) scans for a sort instead of a mixed sequential / random approach. This is by design and it cannot be turned off. The problem you experience is caused by the fact that the sort records are stored expanded, while they're compressed on data pages. This is why you see the big I/O difference. This could be improved via a proper tuning of the SortMemUpperLimit / TempCacheLimit parameters, provided that you have plenty of RAM on board. This is a design limitation and not a bug.

The second issue is that the optimizer is not clever enough to apply the sort to the deepest stream possible and only then join the other streams. It's not always possible and sometimes it could perform worse than sorting the entire resulting row set, but sometimes it could be very useful. Strictly speaking, this isn't a bug either, but an improvement request could be evaluated by the team.

Note: You can reliably convince Firebird that it should perform the actions in the right order using a derived table (put the order by clause there).

What is better : null or empty string in index?

svanderclock asked:

In an index where around 20% of values are null, what is better, to put null or an empty string ? Or it doeasn't matter?

Ann W. Harrison answers:

In terms of storage, it doesn't matter. Prefix compression gets rid of duplicates - null, empty, or full. NULL has some advantages in terms of index ordering - they can be first, last, high, or low. But I would use NULL only when the value is not known, and empty strings when the value is known not to exist.

Interesting lesson about DDL and commit

Norman Dunbar once wrote:

I'm QA'ing a script for an app we have here which is creating a pile of tables and indices etc, and has no COMMIT after any of these.

Ann W. Harrison answers:

As Dmitry S. says, yes, you must commit DDL statements. Otherwise the changes will be undone when your connection ends. But there's more.

In DML Firebird makes changes and performs constraint checking as statements are issued - there's no deferred work. You issue an update statement and the results are checked for uniqueness, referential correctness, nullness, and all other constraints right then. If the data checks are all satisfied, the new row is stored - linked backward to the old data.

DDL is different. Much of the work in creating tables and indexes is done at commit time. Internally, it's known as "deferred work". The rows in RDB$RELATIONS, RDB$FIELDS etc. are stored as you go, but the new entries in

RDB$PAGES and RDB$FORMATS that make the table work aren't done until you commit. That's a relic of the very old, not very good idea of handling DDL like data. If you're creating a table by first creating any necessary domains (RDB$FIELDS), then storing an RDB$RELATIONS record for the table, then storing one RDB$RELATION_FIELDS record for each column in the table, and you instantiate the results immediately, you end up with one RDB$FORMATS record for the table with not fields, and one for each field stored. Messy, especially since you only get 256 formats … So, Jim decided that DDL would be processed when the table definition was complete and committed.

Index creation is also deferred to commit time. If you watch gbak's verbose restore of a database, it grinds for a long time after the commit. It's using a fast-load algorithm to load all the indexes.

If I were doing it now, having abandoned the idea of direct DDL updates as too clunky for human use, I'd create tables completely at the time of the create table statement and keep them private - just like normal data - the transaction that created the table could use it before it was committed, but other transactions would be aware of it only if they tried to create another table of the same name.

Haven't thought much about indexes… you do want other transactions to start using them (for write if not for read) as soon as they exist.

So, short answer, listen to Dmitry and commit your DDL statements like other data changes.

Planet Firebird

A regular summary of recent activities and initiatives within the Firebird database community.

Share Your News with the Community

Have something to share about Firebird? Whether you blog, build with Firebird, or have news to spread, let us know—your input helps keep the community informed and connected.

Reach us anytime at

emberwings@firebirdsql.org or **foundation@firebirdsql.org**

ScratchBird Public Beta

ScratchBird, the experimental database project created by Dalton Calford, has continued to develop since it was introduced in the September 2025 issue of EmberWings. The project remains focused on building a new database engine inspired by Firebird concepts, with the long-term goal of evolving into a platform capable of supporting multiple database wire protocols.

One practical addition since the previous report is the availability of a unified Docker-based development environment distributed through the project's releases. This environment provides a preconfigured container with the required build tools and scripts needed to compile and work with the ScratchBird codebase. By packaging the development setup in this way, the project makes it easier for contributors and curious developers to explore the system without assembling the toolchain manually.

Documentation around the project has also expanded. In addition to the original planning documents that describe the overall architecture and development roadmap, the repository now includes a growing set of structured documentation and newly created Wiki pages. These pages collect architectural notes, development guidance, and operational details, helping to organize the information surrounding the project as it grows.

The project's changelog shows steady activity across several areas of development. Updates include adjustments to the build infrastructure, improvements to development tooling, and incremental work on internal components of the engine. These entries provide a useful overview of the project's day-to-day progress and give readers a clearer picture of how the system is evolving over time.

Alongside the main repository, additional companion repositories have appeared

for related components such as drivers and supporting tools. Together they hint at the broader ecosystem the project intends to support as development progresses.

At the time of writing, ScratchBird has entered a public beta phase, with the author inviting developers to explore the project and experiment with the current implementation. The repository, documentation, and packaged development environment make it relatively straightforward to obtain and build the system, offering interested readers an opportunity to see the project firsthand and follow its progress as it continues to evolve.

Flashback: 25 Years of Firebird

On February 5, 2026, the Firebird community marked the project's twenty-fifth anniversary with a special livestream titled "Flashback: 25 Years of Firebird." The event was organized and hosted by Carlos H. Cantu and featured Arthur Anjos and Alexandre Benson Smith as guests. The session, conducted in Portuguese and lasting 2 hours and 48 minutes, took the form of an informal historical conversation rather than a technical presentation.

Carlos guided the discussion through the major milestones of Firebird's history, beginning with the origins of the project in the open-source release of InterBase and continuing through the early formative years of the Firebird project. Each stage of the story was then expanded by Arthur Anjos and Alexandre Benson Smith, who added their own recollections and commentary—often speaking from direct personal experience with the events being discussed.

In many ways, the event can be seen as a "25 Years of Firebird" article from last issue brought to life—where the historical thread is accompanied by direct recollections, clarifications, and occasional behind-the-scenes insights from those who helped shape the project.

Firebird Database CVE Vulnerabilities

By Paul Reeves (IBPhoenix)

A brief history of Firebird Database Vulnerabilities

Over the years there have been around 40 vulnerabilities reported for Firebird according to the NVD website. The year 2007 was exceptional with 18 reports, but nine of the last 25 years saw no reports at all. Otherwise we typically see just one report in a year. Not a bad record. So two reports in one year is exceptional. And it got me asking questions. How does the scoring system actually work? What does the score mean in practice? How seriously should we take CVE reports?

How are CVEs scored?

I think we all know that a CVE gets a score between 0 and 10, where zero is good and ten is bad. Scores up to 3.9 are considered low. A score between 4 and 6.9 is medium. From 7 to 8.9 the severity of the vulnerability is high and mitigation planning should start immediately. 9 or over probably means it is

time to drop everything and patch urgently.

But what do these scores *really* mean? The NVD site even emphasises that the score is an assessment of the severity of the vulnerability, **NOT** the risk.

So, sure, a problem has been identified but does it affect me, my organisation and how we use the software? To answer that question you first need to learn how to read the string of letters which accompanies each CVE score.

The base score is made up from a series of questions which ask

- how easy is it to get access to the software
- how easy is the exploit
- are special permissions required?
- what can be done with the exploit?

At the end it will produce something like this:

Vector: CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:L Base Score: 5.3 MEDIUM

While the base score is clear the rest is not easy to read.

Let's get into this in more detail.

The scoring system (CVSS)

Over the years the scoring system has evolved to the point there are now THREE scoring systems so it is usual to prefix a score with the version used. This isn't very important as all three use more or less the same base score. But there are differences.

Version 1.0 came out in 2005 and was not a success. We can ignore it.

Version 2.0 appeared in 2007. It is the simplest but was deprecated in July 2022. So most of Firebird's CVSS ratings will have a v2 score and subsequent CVEs will not.

Version 3.0 was introduced in 2017. Quickly superceded.

Version 3.1 was released in 2019. This is currently the most used version.

Version 4.0 arrived in 2023. Still quite new and more complex. And supposedly more accurate. But it does not seem to have gained much traction in the CVE

community.

Currently, all new CVEs *should* have scores for both 3.1 and 4.0 but neither of the recent Firebird CVEs have a CVSS 4.0 score.

The Attack Vector (AV)

The first question is - how easy is it for someone to gain access to the software? There are four answers to this, ranging from is the software exposed to the network (ie, the internet) through to whether physical access to the hardware is required. Obviously network access scores more highly - even a bored kid in their bedroom could gain access, if they could be bothered.

Attack Complexity (AC)

There are only two answers to this - does the attacker need to invest time, knowledge and experience into developing the attack, or is a copy/paste off the internet enough?

Privileges Required (PR)

Three answers here - none, meaning any rando off the street, low, meaning some sort of user level privilege is required, or high, indicating some sort of admin level of access is needed.

User Interaction (UI)

Is a user actually required - for example, to click on a URL?

Scope (S)

This metric determines whether the exploit will allow the attacker to elevate their access to a wider/higher level. This is very much context dependent. For example this *ought* to mean that if an attack allowed an ordinary use to gain dba access to the database this would change the scope.

Confidentiality (C)

This estimates whether confidential information is exposed during the attack.

Integrity (I)

Can the attacker modify data without proper authorization?

Availability (N)

Essentially can the attack compromise the availability of the service. In other words, is it a potential Denial of Service attack.

Other scores

The above is just a quick explanation of the base metric scoring. If you are interested in much more detail take a look at the calculaters here for 3.1 and here for 4.0.

A couple of examples

Now let's look at how the scoring applies to our two most recent CVEs.

CVE-2025-54989

XDR message parsing NULL pointer dereference denial-of-service vulnerability in Firebird

GitHub scored this as:

```
CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:L    Base score: 5.3 (Medium)
```

Breaking it down we have:

AV:N - Attack Vector vulnerable to a network attack.

AC:L - Low attack complexity. ie anyone could do it.

PR:N - NO privileges required.

UI:N - No need to trick a user to click on a link.

S:U - Scope unchanged. The attack cannot be used to break outside of Firebird.

C:N - The attack cannot reveal internal data.

I:N - The attack cannot modify data in any way.

A:L - DoS impact is considered to be low.

What is interesting about this is that NVD/NIST have given this rating:

```
CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H   Base Score: 7.5 (High)
```

The only difference is that they have rated Availability Impact as HIGH.

We can see from this that the scores are very dependent upon judgement calls.

What do you think? Is the risk of DoS via this attack high or low?

And how can risk be assessed when the severity is in doubt?

CVE-2025-24975

Firebird is vulnerable if ExtConnPoolSize is not set equal to 0

The GitHub score is

```
CVSS:3.1/AV:N/AC:H/PR:L/UI:N/S:U/C:H/I:H/A:L  Base score: 7.1 High
```

and the NVD/NIST score is

```
CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:H Base Score: 8.8 High
```

Both assessments indicate high vulnerability but this time there are two differences in the scoring.

There is no disagreement on these:

AV:N - Vulnerable to network based attack

PR:L - Low level of privilege required. The attacker must already have access to the db.

UI:N - No other user is required.

S:U - Scope unchanged. The attack cannot be used to break outside of Firebird. (Although this is arguably wrong - the attack has allowed the attacker to change the scope by accessing an external database.)

C:H - Loss of confidentiality is high. The attack can literally gain access to an encrypted database.

I:H - Highly likely that the attacker will be able to modify data in the encrypted db.

But there is significant disagreement on these two:

AC:H - GitHub believe this is a complex attack AC:L - NVD/NIST think it is quite easy.

A:L - GitHub do not believe it would be easy to generate a DoS via a segfault A:H - NVD/NIST believe it would be easy.

To be honest there seems to be a contradiction here. The description just says a segfault *may happen* when execute statements are chained. So the description itself is indicating that the attack is complex and a segfault is not guaranteed. Yet NVD/NIST say that generating a segfault from this attack is certain and easy to do.

And given that access to an encrypted database is likely to be the goal why would the attacker reveal themselves by segfaulting the server?

In this instance the overall score for both is `High` so the disagreement is not serious. But yet again we see that the scoring is down to interpretation.

The fact that scope is unchanged is also interesting. This is the definition of changing scope:

> An exploited vulnerability can affect resources beyond the authorization privileges intended by the vulnerable component. In this case the vulnerable component and the impacted component are different.

and here we are talking about a vulnerability that can access and modify a database on an external server. What is that, if it is not the actual definition of changing scope?

Still, from a Firebird perspective we should be happy that the scope is considered unchanged, because otherwise we would be looking at a severity score of 9.9.

The difference between `severity` and `risk`.

These two CVEs are good examples of the difference between severity and risk.

The severity of the 'ExtConnPoolSize/DbCrypt callback' CVE may be high but the risk is zero for 99.9% of Firebird users as they use neither an encryption plugin nor an external connection pool.

On the other hand, while the XDR attack *may* be of only medium severity the risk is high for all users of Firebird and InterBase from InterBase 5 (probably) until August 2025. Or at least, it would be if their Firebird server was open to the public internet.

Playing with the CVE calculator

If you follow the above links to the CVE reports you can get access to the CVE calculator for the report. They are hidden under the coloured box with the base score. Try this link to the GitHub CVE-2025-54989 report.

One thing that you may have noticed is that the attack vector for many Firebird CVEs is `network` with the implication that the Firebird server is available to the whole via the internet. This leads us to an important point - almost all Firebird CVEs score higher than they should. In practice Firebird servers are protected from the internet but the CVSS does not take this into account.

You can use the calculator to change the attack vector from `network` to `adjacent`. This drops has the effect of dropping the attack severity from 5.3 to 4.3. This would still be considered a medium severity attack but it may give a better indication of the *actual* risk to most Firebird users.

We see this again with the github evaluation of ExtConnPoolSize/Crypt attack. Lowering the AC score to Adjacent takes the vulnerability out of the High zone into Medium.

What about the higher severity attributed to this attack by NVD/NIST? If you run the mouse over the different options in the calculator you can see the guidleines for applying each evaluation. Do you think NVD/NIST made the right call with AC:L and A:H ?

Conclusions

It is clear that the severity of a CVE report is open interpretation - both up and down. But it is also clear that a CVE report means that a vulnerability *does* exist. It should also clear that neither the existence of a vulnerability, nor the severity of a vulnerability indicates its risk to you.

The Firebird project can help users understand the risk but ultimately the severity, the risk and the operational problems related to upgrading the server are down to each user. The CVE report is just one element in this but the fact that there is a report means that it should be taken seriously. Especially in an age when AI makes attacks every more easy can you afford to take the risk NOT upgrade to a patched release?

...And now for something completely different

The Workshop Beyond the Factories

Once upon a time, in the city of Kazan, where winter settles into the streets with quiet persistence, lived a girl named Katya.

She worked in an office that tried very hard to look modern. The walls were glass, the chairs soft, and the hallways decorated with cheerful posters about teamwork, agility, and delivery. Words like *velocity*, *impact*, and *ownership* appeared everywhere in large friendly fonts. The place had been carefully designed to look like a company that moved quickly and confidently.

Behind those walls lived an old Firebird database that had been running longer than Katya had been a professional. No one spoke about it very often. It was simply there, like an aging piece of machinery in a factory that no one quite remembered installing but that still powered half the production line.

Katya was officially a developer. Unofficially she handled the things that did not belong to anyone else. She updated documentation that few people opened anymore, checked logs after someone had already declared everything normal, and

adjusted scripts that had once been described as temporary but had quietly become permanent.

Sometimes she asked questions, but they rarely traveled far.

The company valued speed above most other virtues. When something failed, the first instinct was always to do something immediately. Restart the service. Redeploy the application. Adjust a configuration parameter and see what happened. If that created new problems, those could be addressed later.

The rhythm left little room for watching, but Katya had a habit of watching first.

In the Firebird database she noticed patterns others seemed not to see. Queries that slowed only after certain reports had run. Transactions that remained open for hours without anyone realizing they were still there. Small irregularities in the logs that repeated like faint footprints in snow.

Whenever she mentioned these things, someone usually waved a hand.

“Legacy behavior,” or “Not our code”, and most popular “Don’t touch it.”

Katya noted the observations for herself anyway. The database fascinated her. It behaved differently from the newer systems people talked about in meetings. It did not forgive careless changes, and it did not forget them either. Decisions made years earlier continued to shape its behavior in quiet ways. Watching it sometimes felt less like maintaining software and more like studying the habits of a very old animal.

Katya liked that.

But curious people often look slow in a place that rewards motion, and she was therefore rarely trusted with anything important.

The system began failing in winter.

At first the changes were subtle. Reports took longer to finish. Some queries paused briefly without explanation. Monday mornings became unreliable in a way that no one could quite reproduce.

Katya noticed the pattern forming and mentioned it once or twice, but the office was busy with other things. A new feature release was approaching, and attention

was focused on deliverables rather than quiet irregularities.

Then one morning the system stopped pretending everything was fine.

Users could not connect. Applications froze on loading screens. Internal dashboards filled with warnings that multiplied faster than anyone could read them.

Within minutes people gathered around terminals and shared screens. Suggestions began appearing almost immediately.

“Restart the service.”

“Kill the sessions.”

“Maybe the server is overloaded.”

Commands were executed as quickly as they were proposed. Processes were restarted. Connections were terminated. Scripts were launched twice because no one was certain whether someone else had already run them.

Each action changed the system slightly. Each change produced new behavior that had no time to explain itself before the next action arrived.

The room grew louder.

Katya stood near the wall with her laptop open and watched the logs scroll. Beneath the noise there was something else—a pattern involving transactions that remained active longer than they should. She opened her mouth to mention it.

At that moment someone announced that production was completely frozen. The room erupted again.

A manager declared that the senior developers would coordinate the response. Everyone else should support them however necessary. Support meant bringing documentation, checking historical configurations, and generally staying out of the way.

Katya nodded. As she stepped aside, one of the developers passed her with a brief smile.

“Even old Galina wouldn’t fix this mess,” he said.

Katya had never heard the name before.

Later she asked quietly in the break room. Most people ignored the question, but eventually someone answered.

“Galina Petrovna,” a senior developer said. “Used to work with Firebird systems long before most of us did. She lives somewhere past the factories.”

“Does she consult?” Katya asked.

The developer shook his head.

“She works.”

The system did not improve the next day.

Temporary fixes collided with other temporary fixes. Changes introduced in the morning produced different failures by afternoon. The database behaved like a machine that had been adjusted too many times without anyone fully understanding how it worked.

Someone suggested documenting responsibility for the incident early, just in case.

Katya was politely asked to stop touching the system. She nodded.

Later she packed her bag and left the building while people were still discussing their next action.

Outside, Kazan looked the same as always—grey streets, quiet traffic, and the long patience of winter. She took a bus toward the industrial outskirts where the city gradually dissolved into warehouses and empty yards.

The directions she had gathered were vague.

Past the factories. After the abandoned rail yard. Look for a building with light.

At the end of a narrow service road stood a small concrete structure with a single illuminated window. Several cables entered the building through metal conduits, and a thin antenna leaned slightly from the roof.

The place looked less like an office than somewhere machines had been allowed to grow old.

Katya knocked, but nothing happened.

So she knocked again.

The door unlocked with a quiet mechanical sound. Inside, the air smelled faintly of dust and warm electronics.

The room was filled with servers and monitors arranged in careful disorder. Screens displayed terminal windows, logs, and Firebird consoles running scripts that Katya did not immediately recognize. The only sound was the steady breathing of fans and processors working without interruption.

At a workbench near the center sat a woman.

She did not look up.

Her grey hair was pulled back without much care, and her hands moved across the keyboard with quick, precise motions that suggested long familiarity with the system in front of her.

Katya waited.

Finally the woman spoke.

“Close the door.”

Katya obeyed.

“Name.”

“Katya.”

“You’re early,” the woman said.

“I didn’t know there was a time.”

“There is. You just weren’t told.”

The woman finished typing and turned slightly in her chair. Her eyes were sharp in a way that made Katya feel briefly transparent.

“You came about Firebird,” she said. It was not a question.

“Yes.”

“I don’t fix companies,” the woman said calmly.

Katya hesitated. “Our system is failing. No one really understands it anymore. I thought maybe—”

The woman raised a hand.

“You want me to fix it.”

Katya nodded.

“No.”

She watched Katya for a long moment, then nodded once, as if confirming something to herself.

“But I can fix you fixing it.”

Katya said nothing.

“You can stay,” the woman continued. “You will work. You will sleep there.” She gestured toward a narrow cot behind a rack of equipment. “You will touch nothing unless I tell you. If you break something, you fix it. If you lie, you leave.”

Katya nodded.

“Put your bag down,” the woman said, turning back to her keyboard. “You’re late already.”

The days that followed unfolded quietly inside the workshop.

Galina rarely explained anything directly. Instead she assigned tasks that forced Katya to observe. Restore this backup and compare the results. Watch the monitoring tables and tell me which transactions stay open longest. Run the script again and explain why the numbers changed.

At first Katya answered too quickly.

Galina erased the terminal history and told her to start again.

Eventually Katya learned to look longer before speaking.

The systems in the workshop behaved like silent servants that required no instructions. Scripts woke at certain hours. Monitoring queries examined attachments and transactions without being asked. Connections that violated certain conditions simply disappeared.

At first the behavior seemed arbitrary.

Gradually Katya began to understand the logic behind it.

One afternoon she started a process that vanished almost immediately.

“It killed my connection,” she said.

“It protected the system,” Galina replied.

Katya began learning the habits of the machines not by controlling them but by watching them.

One evening Galina left the workshop without explanation.

Katya noticed only when the door closed.

The systems continued their work as they always did.

After a while one of the monitoring windows changed behavior. A set of messages began repeating with unusual frequency. One of the remote systems that Galina managed—a Firebird server belonging to the municipal traffic inspectorate—was reporting activity that did not fit its usual pattern.

Katya opened the console and examined the logs.

Connections were accumulating. Transactions were beginning but not finishing. The monitoring tables showed a buildup of activity that resembled a knot slowly tightening.

Her first instinct was to terminate the active sessions and restart the affected processes.

Her hands hovered above the keyboard.

Instead she watched.

The pattern became clearer the longer she observed it. One transaction had remained open far longer than it should have, preventing garbage collection from clearing old record versions. The system had gradually slowed under the weight of its own history. Nothing was broken. It had simply been asked to remember too much.

The solution was not dramatic. Katya intervened carefully, addressing the cause rather than the symptoms.

Then she waited.

The system slowly returned to its normal rhythm.

When the door opened behind her, she did not turn immediately.

Galina stepped inside carrying a small paper bag and studied the screens.

“You didn’t break it,” she said.

“I almost did,” Katya replied.

Galina nodded once.

“Good.”

She opened a drawer and placed a small USB stick on the table between them.

“This is what you came for.”

Katya looked at it but did not touch it.

“What is it?”

“A light,” Galina said. “And a fire.”

She met Katya’s eyes.

“Use it to see,” she said quietly. “Not to burn.”

Katya returned to the city the following morning.

The office looked unchanged, but the crisis there had begun to exhaust itself. Too many attempted fixes had produced a system whose behavior no one fully trusted.

Someone nearby suggested restarting the service again.

Katya said nothing.

She opened the monitoring tables.

She examined the transactions, the attachments, and the quiet traces of earlier decisions still shaping the database’s behavior. She watched the system for a long time before touching anything.

Then she used the USB stick once, carefully, to confirm what the data had already suggested. It illuminated the path through the confusion but did not replace her

own judgment.

She watched first.

When she finally acted, she did so carefully and only once.

The system stabilized. Not perfectly. Not permanently. But enough.

People noticed gradually. Questions followed. Katya answered them simply and without urgency.

In the weeks that followed, small things changed at the office. People began asking Katya to look at problems earlier rather than later. Meetings paused when she said she needed time to examine something.

One evening a junior developer stopped beside Katya's desk, holding a laptop as if it were something fragile.

"I'm trying to understand why this query behaves differently on Mondays," he said. "Everyone says it's just load."

Katya looked at the screen he showed her. The explanation was familiar—simple, convenient, and probably wrong.

"Have you watched the transactions?" she asked.

"For how long?"

Katya thought for a moment.

"Long enough that the pattern stops changing."

The developer nodded uncertainly and returned to his desk.

Katya finished her work a little later and shut down her workstation. The office was quiet now, the earlier urgency replaced by the dull rhythm of routine. Somewhere in the building the Firebird database continued its steady work, attachments opening and closing, transactions beginning and ending, each one leaving a trace for anyone patient enough to notice.

As she reached for her coat, Katya felt the USB stick in her pocket.

She had examined it once. That had been enough to understand why Galina had called it both a light and a fire. Some tools reveal problems. Others reveal people.

Outside, the air was sharp with cold. Snow drifted slowly through the glow of the streetlamps as she walked toward the bus stop. For a moment she wondered what the workshop beyond the factories looked like tonight—whether the monitors were still glowing, whether the scripts were still running in their quiet, tireless way.

Then she realized something that made her smile slightly.

If something unusual happened in one of those systems, Galina would see it.

And if Galina happened not to be watching, someone else might be learning how.

Hutber's Law

> *Improvement means deterioration.*
> *(Patrick Hutber)*

This law suggests that improvements to a system will lead to deterioration in other parts, or it will hide other deterioration, leading overall to a degradation from the current state of the system.

For example, a decrease in response latency for a particular end-point could cause increased throughput and capacity issues further along in a request flow, affecting an entirely different sub-system.


Have a wonderful Easter
– The Firebird Project & Firebird Foundation

EmberWings

The official quarterly magazine of the Firebird Project

Do you develop with Firebird?

Are you using Firebird as a database backend for your applications? Share your experience and help shape its future! Your insights on how you develop with Firebird, the tools you rely on, and your wishlist for improvements will directly impact its development. The survey takes just 5-10 minutes—join us in building a better Firebird!

Take the Developer Experience survey

Do you manage Firebird deployment?

Your insights as a Firebird administrator are invaluable! By taking just a few minutes to complete this survey, you'll help shape the future of Firebird, identify key challenges, and improve the tools and features you use every day.

Participate in the Admin survey

We value your opinion! Help us improve **EmberWings** magazine by sharing your thoughts and feedback. Our quick questionnaire will only take a few minutes, and your responses will guide us in making future issues more relevant, engaging, and valuable to the Firebird community.

Help us improve EmberWings!