EmberWings

2025/3

Firebird

September 2025

www.firebirdsql.org

emberwings@firebirdsql.org

Firebird

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:

Back to Fire and Shadows

Dear Readers,

As summer winds down and the golden hues of autumn take over, many of us return to our desks, greeted by the familiar hum of servers, the soft glow of monitors, and the tasks that waited patiently during the holidays. There's always a certain thrill in this seasonal shift: a little crispness in the air, a hint of pumpkin spice, and the feeling that new challenges—and maybe a few surprises—are just around the corner.

This September, we continue the celebration of Firebird's 25th Anniversary, a milestone we marked on 31st July. While the official day has passed, the party continues through the year, giving the Firebird community ample time to honor the decades of innovation, collaboration, and dedication that have made this project remarkable. Every release, every contribution, every discussion across the forums and mailing lists reminds us that Firebird's strength has always been its people.

As the calendar turns toward Halloween, there's an air of playful spookiness in the Firebird world. "Office hauntings" aren't ghosts in the machine—they're the quirky issues, tricky scenarios, and performance quirks that can scare newcomers or unprepared users. But seasoned Firebird developers know these are just part of the landscape: challenges to learn from, puzzles to solve, and opportunities to master the engine's full potential. They are reminders that even in the face of scary-looking problems, skill and knowledge are the ultimate protective charms.

Autumn is a season of reflection as much as it is of preparation. As the leaves turn and the air grows cooler, it's a good time to revisit old projects, polish your code, or try new ideas with Firebird. The community's shared wisdom and collaborative spirit make every challenge easier to face, turning potential spooks into moments of growth.

So, as you sip your coffee and watch the shadows lengthen outside, remember that the Firebird flame burns steadily. The 25th Anniversary celebration continues, the community continues to thrive, and the thrill of discovery—be it new features, clever solutions, or even a well-handled "spook"—remains alive in every corner of the Firebird world. Here's to a season of learning, a dash of Halloween fun, and another exciting chapter in Firebird's story.

Warm regards,
The EmberWings Team

The junior developer came to the master, shoulders heavy with worry.

“Master,” he said, “I’m unsure about the database. I write my queries, I build my tables, but I don’t know if they’ll hold up. What if more data changes how they behave? What if performance drops over time? What if a backup doesn’t restore when it’s needed most?”

The master listened, then spoke softly: “You see shadows where others see walls. That is because you are still learning the shape of this house.”

The junior frowned. “Shadows?”

The master nodded. “Yes. In this house of Firebird, spooks dwell. They are not monsters of their own, but echoes of our choices. Leave transactions open, and a spook of creeping slowness will follow you. Forget to test your backups, and you may one day open the coffin to find it empty. Ignore how data grows, and your swift queries may turn to stone before your eyes.”

The junior’s face paled. “Then we are doomed to hauntings?”

The master poured tea into two cups, and slid one across the table. “Not doomed. Spooks thrive where there is neglect. They fear knowledge, discipline, and care. To see them for what they are is to rob them of their terror."

He leaned closer, voice calm. "Remember: a spook is only a reminder—of a habit unkept, a test not run, a pattern left unchecked. Learn their signs, and they will serve as teachers, not tormentors."

The junior lifted the cup, still trembling, but with a spark of courage. "Then I must study these spooks, so they no longer frighten me."

The master smiled. "Good. Walk with me through the house. I will show you where they hide."

The Little Shop of Horrors (Firebird Edition)

By Pavel Císař (IBPhoenix)

Octoberber always brings a chill in the air, and not just from the turning leaves. For Firebird users, Halloween doesn't need costumes or haunted houses—our ghosts lurk in SQL plans, backups, and connection pools. Whether you're a database administrator standing guard over production or a developer weaving Firebird into your applications, you've likely felt those shivers when something strange goes bump in your logs.

In this special Halloween-themed feature, we're shining a lantern on ten of the most common (and most unnerving) troubles that stalk Firebird users. Each one appears in the form of a Spook—specters with their own "Spook level," from mildly unsettling nuisances to bone-chilling catastrophes that can keep you up all night. Think of this as a guided tour through Firebird's very own little shop of horrors.

Dare to step inside?

👻 Courtyard of the Vanishing Index

Spook Level: 😐 Mild shiver

In this courtyard, you'll find indexes that appear solid but offer no real help. Queries stumble into full table scans, and performance quietly withers, leaving you wondering what went wrong. The optimizer hasn't turned hostile—it's simply learned that some indexes aren't worth using.

The main culprit is low selectivity. As your tables grow and data distribution shifts, indexes that once gave quick access may lose their edge. Even when statistics are fresh and accurate, an index might still be ignored because it can't effectively narrow down results.

On rare occasions, an index may remain inactive after a restore—but this usually comes with clear warnings in gbak's output or the process return value. As long as you review those carefully, this ghost rarely escapes into production.

Countermeasures to banish the Vanishing Index:

- Monitor index selectivity by querying RDB$STATISTICS.
- Refresh selectivity values with SET STATISTICS to keep the optimizer well-informed.
- If an index proves consistently useless due to data distribution, consider redesigning it—or replacing it with a more effective one (for example, a compound index or an alternative approach to querying).
- Always check gbak restore results to ensure no index silently failed to rebuild.

With careful watching, you can keep your indexes alive and useful—rather than letting them fade into ghostly decorations that haunt your queries.

👻 Hall of "It Works on My Machine"

Spook Level: 😐 Mild shiver

Step into this hall and you'll hear developers swear their code runs flawlessly—until it dares to leave their machine. In production, the same application falters, tripping over errors that never showed themselves in the safety of the dev lair.

Logs groan with mysterious failures, and suddenly the once-reliable build feels cursed.

This haunting thrives on differences between environments. A slight mismatch in Firebird versions, OS settings, or hardware quirks is enough to awaken it. What seemed harmless in development becomes a showstopper in production, and debugging feels like chasing shadows.

Countermeasures to escape the Hall:

- Align your environments: use the same Firebird version and configuration across development, staging, and production.
- Don't test in a vacuum—use production-like data volumes and hardware to surface problems before they strike live systems.
- Automate environment setup with scripts or containers, so each new machine is conjured consistently.

With discipline, the Hall loses its power. Ignore it, and you'll be forever haunted by the refrain: "But it worked on my machine…"

⚡ Crypt of the Forgotten Triggers

Spook Level: 😐 Mild shiver

In this crypt, code awakens at night. Updates ripple through tables, and suddenly rows change in ways no one expected. Developers scratch their heads—nothing in the application explains it. The truth lies buried in a trigger written long ago, now forgotten, but still firing faithfully in the dark.

The spook shows itself when business logic is buried inside triggers and procedures without proper documentation. Over time, as applications evolve, these hidden rules clash with new code paths. The result: mysterious side effects, duplicated logic, and bugs that seem to come from nowhere.

Countermeasures to silence the forgotten voices:

- Keep an inventory of triggers and procedures, and review them whenever schema changes are planned.

- Document business logic in a shared knowledge base, not just in database objects.
- Favor clear application-level logic when appropriate, so triggers aren't the only guardians of rules.
- Use commenting and naming conventions to make triggers visible and less likely to be overlooked.

Tended carefully, triggers are powerful allies. Left forgotten, they become restless spirits that haunt every change.

🕯️ Ritual Room of the ODS Upgrade

Spook Level: 😨 Heart racing

The Ritual Room awaits those who seek to cross into a new Firebird age. The price of passage is the sacred backup/restore rite, required when moving to a database with a new ODS (On-Disk Structure). Many pass through without trouble, but others find themselves facing unexpected errors, their applications stumbling as unseen differences surface.

The true danger here isn't corruption, but compatibility issues. Major ODS upgrades can reveal hidden dependencies in metadata or expose assumptions in applications—especially when new data types or behavioral changes come into play. Firebird's developers work hard to keep the path smooth, often providing compatibility configuration options, but no one can anticipate every combination of features in use.

Countermeasures to survive the ritual:

- Always rehearse first: perform the backup/restore on a staging server before attempting production.
- Read the Release Notes thoroughly, especially the sections on upgrades and compatibility.
- Keep multiple backup generations so you can roll back if the new version refuses your database or your application.

Approach the ritual with caution and preparation, and you'll emerge stronger. Skip the warnings, and the ghosts of incompatibility may follow you into production.

⛓️ Hall of Invisible Shackles

Spook Level: 😨 Heart racing

You step into the hall and suddenly your transaction freezes. No locks appear in sight, yet progress halts as if unseen chains have bound your query. The session waits, caught in silence, while the true blocker lurks somewhere in the shadows.

This spook emerges when you enter with a WAIT transaction option while another connection holds uncommitted changes. From your view, nothing looks wrong—yet Firebird faithfully waits for the other transaction to finish. Sometimes the chains come from less obvious dependencies buried in triggers or cascades, leaving you staring at a screen that won't move.

Countermeasures to break the shackles:

- Shine a light with the MON$ tables to track down which transaction is blocking yours.
- Keep transactions short and commit often—the longer they stay open, the more ghosts they can spawn.
- Train developers on Firebird's concurrency model, so they know how WAIT behavior and dependencies can ensnare their code.

Handled well, the hall is only a momentary scare. Ignore it, and your system may find itself bound in invisible chains for much longer than you'd like.

😈 Dungeon of the Sleep Paralysis Query

Spook Level: 😨 Heart racing

Deep in the dungeon lurks a query that looks harmless in development—fast, efficient, obedient. But once released into production, it transforms. Instead of returning results, it consumes CPU endlessly, holding the system hostage as if frozen in a waking nightmare.

The source of this terror is often a bad execution plan chosen by the optimizer. What flies with small datasets can collapse under the weight of millions of rows. Missing indexes or poorly chosen join strategies add to the paralysis. And even when queries are tested on realistic data, production environments may tell a different story: one system has many categories with a few items each, while another has only a few categories but thousands of items per category. These differences in cardinality can completely change which plan is efficient.

Countermeasures to escape the Dungeon:

- Examine the PLAN output of your queries to see how the optimizer intends to execute them.
- Test with realistic—and varied—datasets to uncover how queries behave under different distributions, not just average cases.
- Keep indexes aligned with query patterns, ensuring the optimizer has the tools it needs to choose wisely.

With careful testing and attention to indexing, the Sleep Paralysis Query can be contained. Neglect it, and you risk a production system that lies awake, consumed by one query that never ends.

🌀 Labyrinth of the Shifting Schema

Spook Level: 😨 Heart racing

The schema looks familiar at first glance—but venture deeper, and walls begin to move. Columns vanish, constraints appear where none were before, triggers spring traps you didn't expect. What worked yesterday breaks today, leaving developers and DBAs wandering the labyrinth of migration.

This spook emerges whenever a database schema evolves without careful planning. Column type changes can cause data loss or unexpected conversion errors. Constraint updates may reject rows that once were valid. Trigger or procedure changes can alter application behavior in subtle ways. And when different environments drift out of sync, testing loses its reliability, and production becomes the true labyrinth.

Countermeasures to survive the shifting walls:

- Manage schema changes with versioned migration scripts (and apply them consistently across dev, test, and production).

- Test migrations on a copy of production data to uncover issues with real-world values, not just empty structures.
- Keep thorough documentation of schema evolution, so the path through the labyrinth remains visible to all who enter.
- When possible, use tools or containers to standardize deployment, ensuring the same migration path is followed everywhere.
- After extensive schema changes, remember that Firebird stores existing data in the old format. These records are converted on every read, which can degrade performance after multiple upgrades. To refresh all data into the new storage format, perform a gbak backup and restore.

With discipline and foresight, the schema evolves safely. Without it, you may wander endlessly in the labyrinth—chased by ghosts of past structures.

👻 Catacombs of the Undead Data

Spook Level: 😱 Cold sweats

In the deepest catacombs, the worst nightmare lurks: corrupted data that refuses to rise again. A server crash, a dying disk, or an unsafe shutdown can leave your database whispering in dread with “internal gds consistency check” errors. Once here, queries stumble, tables groan, and you realize parts of your data may never return.

This spook is born not from Firebird itself, but from its surroundings—failing hardware, unreliable storage, and power cuts that strike at the worst possible time. And if you’ve trusted backups without ever testing a restore, the path out of the catacombs may be closed.

Countermeasures to escape the catacombs:

- Protect your database home with reliable storage, ECC memory, and UPS power backup.
- Run gfix -v periodically to check database health and spot trouble before it grows.
- Keep recent, tested backups, so you can restore cleanly if corruption appears.

With vigilance and tested defenses, the undead data stays buried. Ignore the warnings, and the catacombs will eventually open beneath your feet.

⚫ Pit of Creeping Slowness

Spook Level: 😨 Heart racing

In older times, this pit was bottomless. A single forgotten transaction could trap record versions forever, choking the life from performance until the database lay still. Developers whispered of it as the most dreadful of haunts.

But with Firebird 4, the pit has lost some of its darkness. The engine now sweeps away intermediate ghosts in the version chains—record remnants no longer visible to anyone. Background garbage collection, sweeps, table scans during index builds, and even regular attachments all help scatter these shades. The slow rot is harder to summon, and easier to clean.

Still, the pit is not gone. At its core lies the same truth: an uncommitted transaction can stall the advance of the oldest snapshot. Record versions pile up, and though Firebird is stronger at sweeping them aside, it cannot erase what must remain visible to the laggard.

Countermeasures to keep the pit shallow:

- Monitor the oldest active transaction using info calls or MON$ tables, and react if it stops moving.
- Educate developers to commit promptly—no design change can save you from poor habits.
- Run sweeps when appropriate; even with intermediate cleanup, a neglected database can still accumulate debris.

The pit is no longer endless—but it is still a place where carelessness takes root.

⚰ Vault of the Empty Coffin

Spook Level: 😱💀 Ultimate fear

Few discoveries are as chilling as opening the vault, only to find your backup reduced to dust. The database you thought was safe is gone, and the restore you desperately need yields nothing but errors. This is the Vault of the Empty Coffin—the place where untested backups betray their keepers.

The dangers here are many: a corrupt backup silently written to disk, the wrong database file captured by mistake, or a restore that fails halfway because it was never rehearsed. Trusting a backup without verification is walking blind into the vault, hoping the coffin holds more than air.

Both gbak and nbackup can guard your data, but each carries different risks. gbak performs logical backups and restores, exposing metadata or compatibility issues, while nbackup captures physical states that can be more fragile if not handled carefully. Knowing their strengths—and their weak points—helps keep your coffins from going empty.

Countermeasures to keep the coffin full:

- Regularly test restores to confirm your backups can actually bring a database back to life.
- Keep offsite copies to protect against local disasters.
- Automate backup validation in your maintenance scripts, so you don't have to rely on memory or luck.

Respect the vault, and it will safeguard your data. Neglect it, and you may one day open the coffin to find nothing but dust.

Interview with Fabio Augusto Dal Castel

The Firebird community has always thrived on the quiet dedication of developers who shape its ecosystem behind the scenes. Some contributions arrive not with fanfare but with steady impact, changing how we deploy, connect, and build with Firebird every day. In this edition of EmberWings, we sit down with one of those developers — F. D. Castel — whose work has touched multiple layers of the Firebird experience, from tooling to integration.

In this conversation, we explore his journey into the Firebird world, the motivations that drive his open-source contributions, and his thoughts on where things are headed next. His perspective offers both practical insight and a personal glimpse into the spirit of collaboration that keeps Firebird moving forward.

Could you start by telling our readers a bit about yourself? Where you grew up, your current city, and a brief overview of your professional background before getting involved with Firebird.

I grew up, live and work in Porto Alegre, the capital of Brazil's southernmost state, Rio Grande do Sul — a vibrant hub for tech companies.

I'm one of the founders and CTO of Dataweb Tecnologia, a company dedicated to developing solutions for the optical market.

Before working with Firebird I had some experiences with Oracle, Sybase, and Microsoft SQL Server, though mostly in smaller roles.

Computer programming was always a passion since childhood. But, in many ways, I can say that my professional journey truly began about 25 years ago, alongside both Firebird and Dataweb.

How did you first encounter Firebird, and what drew you to working with it? Was it a work project, a personal interest, or something else that started your journey?

Our company, Dataweb, was established in 2000. At the time, we were looking for an RDBMS for our applications, and the then-nascent Firebird Project proved to be an ideal fit. It was lightweight (its binaries were only about 6~10 MB, if I recall correctly), fast, reliable, and required no dedicated DBA to manage — a role that many of our initial customers neither had nor wanted.

Your name is closely linked with Firebird Docker images. Can you walk us through how that project started, why you felt it was needed, and what challenges you faced in making it "official" for the Firebird project?

Like many open-source efforts, it started with an "itch that needed scratching".

At the time, the best Docker images available were those created by the excellent work of Jacob Alberty. But it had become clear that the images were not being updated as regularly as they should have been.

So, during what I like to call "a dark and stormy weekend", I began building a new set of images. I researched best practices from other Docker projects, added comprehensive test coverage to ensure robustness, and fully automated the build process. Within a few days, I had a set of images that I considered reliable enough to use in production projects.

I kept them for in-house use for a few months (to address any quirks). From there, it was simply a matter of reaching out to the Firebird team and expressing my intention to publish the images. I also explained the idea to Jacob, who was very supportive and graciously passed the mantle.

The Firebird team welcomed the idea enthusiastically, and I can say the transition involved virtually no challenges. Thanks to the outstanding work of Adriano dos Santos Fernandes, the build scripts were quickly adapted to run within Firebird's CI/CD system and the rest, as they say, is history.

Beyond Docker, you've contributed to Firebird's Python ecosystem, particularly to the firebird-driver and sqlalchemy-firebird packages. What led you to get involved with these projects, what needs were you trying to address, and are there any interesting challenges or stories from working on them that you'd like to share?

This story has several similarities to the Docker images project. It also began with a specific need that had to be addressed.

SQLAlchemy is a fantastic library and greatly used in the Python ecosystem. At Dataweb, we rely heavily on Python for a number of internal tools, many of which interact with Firebird.

Interestingly, Firebird was one of the very first databases natively supported by SQLAlchemy. However, due to a lack of maintainers at the time, the SQLAlchemy team eventually decided to move it out of the core project and to keep it as an external dialect.

Over time, it became clear that the Firebird dialect was no longer keeping pace with Firebird's ongoing development.

That's when I stepped in: About two years ago — on yet another rainy weekend — I began working on the dialect and diving into SQLAlchemy's extensive test suite. It took me around two months to refactor outdated code, adopt modern project practices, and troubleshoot tricky test failures. The effort paid off: within some more weeks, we released a completely revamped SQLAlchemy Firebird dialect with full support for the latest Firebird features, up to Firebird 5.

Throughout the process, the SQLAlchemy team was highly supportive and collaborative. In particular, my chats with Michael Bayer and Federico Caselli provided an invaluable learning experience, making the entire journey even more rewarding.

Can you tell us about your PSFirebird PowerShell toolkit — what inspired its creation, what key functionalities it offers, and what needs or gaps in the Firebird ecosystem it addresses?

I was actually planning to mention this in the "what I’m working on now" section (smile) — it’s a very recent project, so I’m glad you brought it up.

Automation has always been one of my core principles in both software development and deployment. I have headaches when I see people clicking through GUIs to perform repetitive tasks manually. If you look at my GitHub profile, you’ll notice other projects centered on automation.

Whenever I need to set up a new system, I make it a point to document every step in an executable form. If the process can’t be reproduced by simply running a script — or by pasting a sequence of commands into a terminal — then it’s not good enough. I’ll go the extra mile to make sure it’s fully automated, and it always pays off in the long run.

While working on Firebird tools, I relied on several ad hoc scripts I had developed over the years to manage multiple Firebird environments on a single machine for development and testing purposes.

The need for something like this became apparent when I first started working with the Python and C# Firebird drivers (way before I started working on Docker images).

PSFirebird brings together these scripts and best practices into a single toolkit designed to automate the setup and use of multiple Firebird environments. Its primary audience would be Firebird tools developers, rather than its end users (for example, those who need to run tests across Firebird 3, 4, and 5).

Looking back over your Firebird contributions, is there a moment or milestone you are especially proud of? This could be a technical breakthrough, a community response, or even a fun “battle story” from development.

I could surely say that was when I got an e-mail from Alexey Kovyazin, then president of the Firebird Foundation, telling me that the team was agreed to accept the Docker images project as an official project and move it to Firebird Foundation repositories.

For the battle stories, "this margin is too narrow to contain" (smile)

What are you working on right now, and what are your plans for the rest of this year and next? Are there any upcoming releases, experiments, or new tools you can give us a sneak peek at?

I’ve just returned from a long — and much-needed — vacation. Before that, my most recent focus was the PSFirebird project, which I plan to refine further before making a more formal announcement to the community.

I also have a few PRs and issues to wrap up in both the Python and DotNet Firebird driver projects. Hopefully, the maintainers of these projects haven’t forgotten about them. (wink)

From your perspective, how is the Firebird community doing in Brazil? Is it growing, stable, or facing challenges? Any trends you’ve noticed over the past few years?

Brazil is definitely a land of challenges. People here often joke that "living in Brazil is like playing on HARD mode" (as in videogames).

That said, I’m particularly proud of the organization of Firebird Developers Day (which occurs every year) and the work of my fellow countrymen like Adriano dos Santos Fernandes and Carlos Cantu.

Do you participate in local or national Firebird events, meetups, or user groups? If yes, what role do you play, and if not, is it something you’d like to see more of?

I must admit that, unfortunately, I don’t participate in these events as much as I would like. I have a strong desire to be more active in the community, but professional commitments often limit the time I can dedicate.

Finally, what advice would you give to developers who are just discovering Firebird today? Any tips for contributing to the ecosystem or making the most of Firebird in their projects?

Learn about Firebird MVCC and keep an eye on your transactions, kids.

Thanks for you time!

Resources:

1. F.D.Castel on GitHub
2. PowerShell toolkit for Firebird databases
3. Official Firebird Docker images
4. Dataweb Tecnologia website

The Vampire Index

Too many unused indexes can slowly drain the lifeblood of your storage and write performance. Indexes are powerful, but like vampires, they need to be invited wisely—only where they truly belong.

The Mirror View

Views can be elegant mirrors of your data—but if they're built without careful indexing or with too many joins, they can summon performance monsters. Always test your views' query plans before trusting the reflection.

The Howling Alias

Queries without clear aliases can turn into beasts under a full moon—ambiguous column names, hard-to-read SQL, and maintenance nightmares. Tame them early with proper aliasing before they start howling.

Development update: 2025/Q3

A regular overview of new developments and releases in Firebird Project

Releases:

- Firebird 5.0.3, 4.0.6 and 3.0.13, released 15.7.2025
- firebird-qa 0.21.0, released 20.7.2025
- ODBC driver 3.0.1, released 21.7.2025
- fdb 2.0.4, released 22.7.2025
- Beta version of ODBC for ARM, released 23.7.2025
- firebird-testcontainers-java 1.6.0, released 12.8.2025
- Jaybird 6.0.3 and 5.0.9, released 22.8.2025
- jaybird-fbclient 5.0.3.0, released 23.8.2025
- Firebird 2.5.9 with CVE-2025-54989 fix, released 25.8.2025

Firebird 6 Status

The alpha release was planned for Q3, but had to be pushed to the next quarter. Although many of the planned minor new features and improvements have already been completed, the major blocker is the completion of the major new features:

- SQL schema support is already completed.
- Shared metadata cache is also ready in principle. It works for SuperServer and support for Classic is being finalized (mainly testing).
- Support for SQL data type ROW, JSON functions and tablespaces is also ready, and PR review is underway.

Support for Multiple Trace Plugins

Firebird 6 now supports selecting which trace plugin(s) to use for a trace session. Previously, you could specify multiple trace plugins in `firebird.conf`, but there was no way to choose between them. Artyom Abakumov proposed and implemented this improvement to the trace API and `tracemngr`. The pull request has been accepted and merged into the Firebird 6 codebase.

Firebird Docker Images

The Firebird Docker repository has steadily evolved to support a wide range of Firebird versions and operating system platforms, keeping pace with both new Firebird releases and modern Linux distributions. Currently, the images cover Firebird versions 3, 4, and 5 (including point releases like 5.0.0, 5.0.1, 5.0.2, 5.0.3), each built for multiple Debian/Ubuntu base distributions such as Bullseye, Jammy, and others (e.g. “noble”). Over time, the project has refined its build scripts, updated version assets automatically from Firebird releases, and ensured that new distributions are supported or noted when a dependency issue prevents immediate support. Users can spin up containers with custom configurations via environment variables for root and user accounts, database creation, and can also mount configuration files or initialize databases via scripts.

Introducing Awesome Firebird

The Firebird Project is pleased to announce the launch of Awesome Firebird, a curated collection of resources, tools, and libraries for the Firebird SQL ecosystem. This list is still in its early stages, but it already brings together drivers, bindings, utilities, and administrative tools that showcase the richness and versatility of Firebird in practice. Our goal is to make this a central, community-driven hub where both new and experienced users can discover what is available and where to get started. To achieve that, we warmly invite readers to contribute their own favorite projects, bindings, or utilities that deserve a place on the list. Contributions can be made directly as pull requests or by opening an issue on the GitHub repository, helping to shape this resource into a living catalog of the best that Firebird has to offer.

The Hollow Hostname

Relying on DNS lookups instead of static IPs can summon hollow delays when connections resolve slowly. Bind your database to something solid, not a phantom.

The Swamp of Subselects

Stacking too many subselects can bog down queries in a swamp of slowness. Flatten them with joins or CTEs before they drag you under.

The Specter of Savepoints

Savepoints feel safe, but nesting too many can summon specters—locking resources and cluttering transaction flow. Use them sparingly, or be haunted by their overhead.

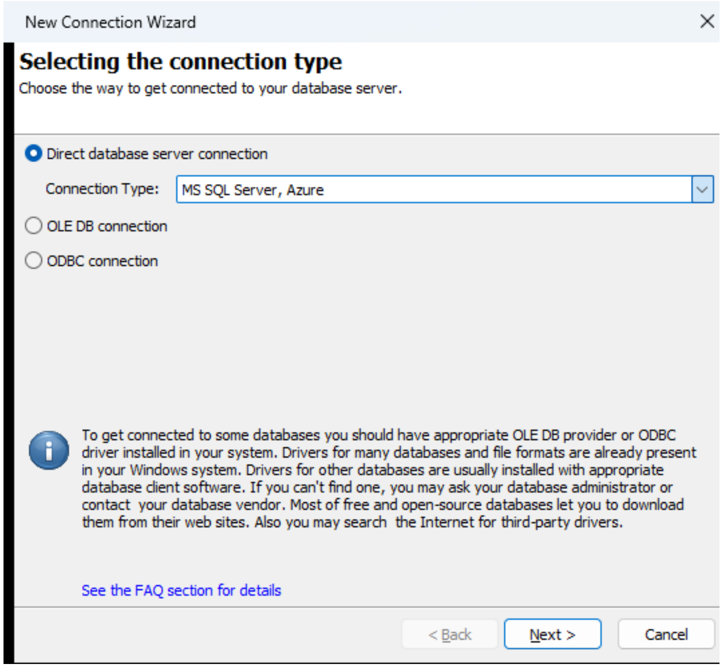
Toolbox: FlySpeed SQL Query

FlySpeed SQL Query, developed by Active Database Software, is a database query and data analysis tool designed to streamline working with multiple relational database systems. Its primary purpose is to provide both developers and analysts with a convenient environment for writing SQL, visually designing queries, and exploring data interactively. While not intended to replace database administration utilities, it positions itself as a bridge between productivity and flexibility, offering features for both casual inspection and serious development work.

The Free edition version 4.10.7.0 was used for this review.

The tool is distributed as a 64-bit or 32-bit Windows desktop application. The installation process is straightforward, packaged as a standard installer with minimal prerequisites. Once installed, it launches as a standalone client without requiring additional runtimes. In addition to English, it offers localization in German, Italian, French, Polish, Russian, and Japanese. It includes native support for Firebird (via FireDAC), making setup both quick and flexible.

Beside Firebird and InterBase, FlySpeed SQL Query supports wide range of database servers, including: Microsoft SQL Server/Azure, Oracle, MySQL/MariaDB, PostgreSQL, SQLLite, IBM DB2, Teradata, Sybase ASA/ASE, Advantage DB, MS Jet 4 Engine, Informix, DBF files and others via ODBC.



New Connection Wizard

Selecting the connection type

Choose the way to get connected to your database server.

● Direct database server connection

Connection Type: MS SQL Server, Azure

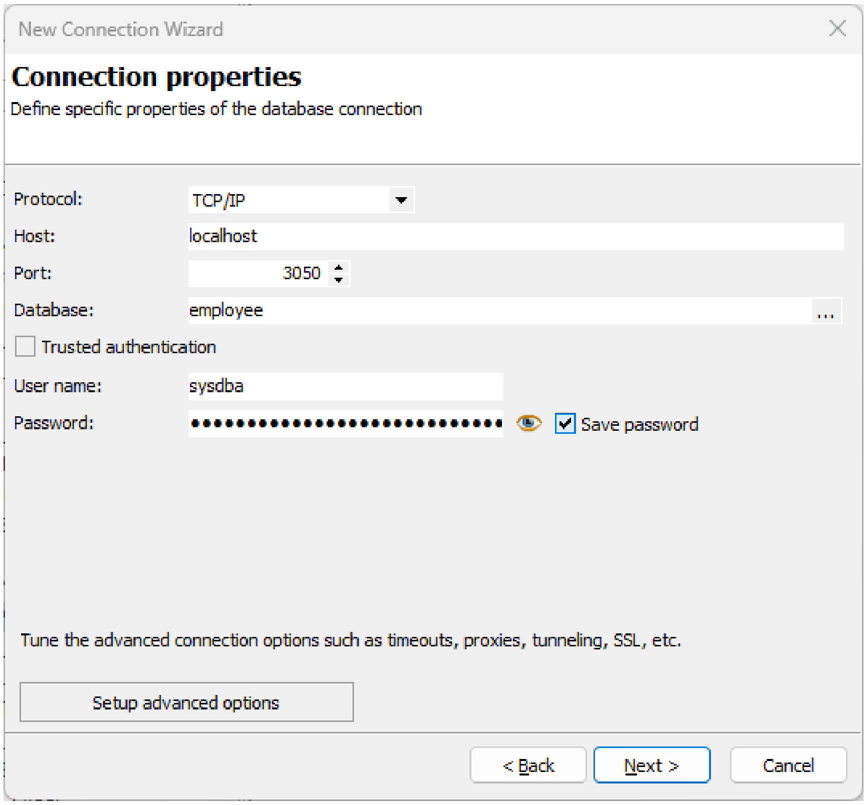
○ OLE DB connection

○ ODBC connection

To get connected to some databases you should have appropriate OLE DB provider or ODBC driver installed in your system. Drivers for many databases and file formats are already present in your Windows system. Drivers for other databases are usually installed with appropriate database client software. If you can't find one, you may ask your database administrator or contact your database vendor. Most of free and open-source databases let you to download them from their web sites. Also you may search the Internet for third-party drivers.

See the FAQ section for details

The connection wizard allows users to specify database file paths or remote host addresses, along with user credentials.



New Connection Wizard

Connection properties

Define specific properties of the database connection

Protocol: TCP/IP

Host: localhost

Port: 3050

Database: employee

□ Trusted authentication

User name: sysdba

Password: ••••••••••••••••••••••••••••• ☑ Save password

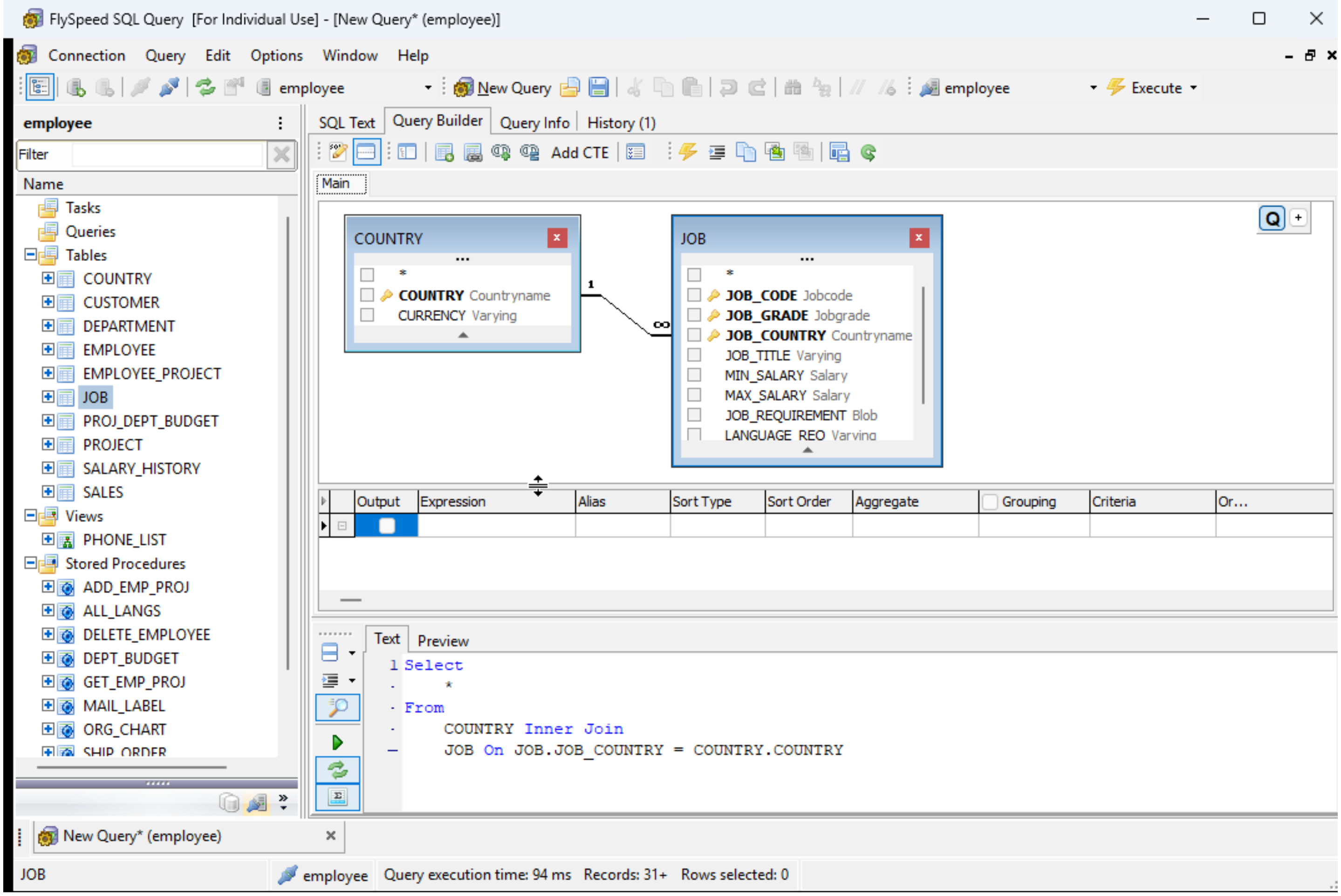
Tune the advanced connection options such as timeouts, proxies, tunneling, SSL, etc.

Advanced options are available for configuring character sets, dialects, role or custom client library location. However you can't fine-tune connection parameters beyond provided base set.

Multiple connection profiles can be stored, enabling quick switching between environments.

The interface is organized into a clean layout designed for both SQL editing and visual interaction. At its core, FlySpeed SQL Query combines two key spaces:

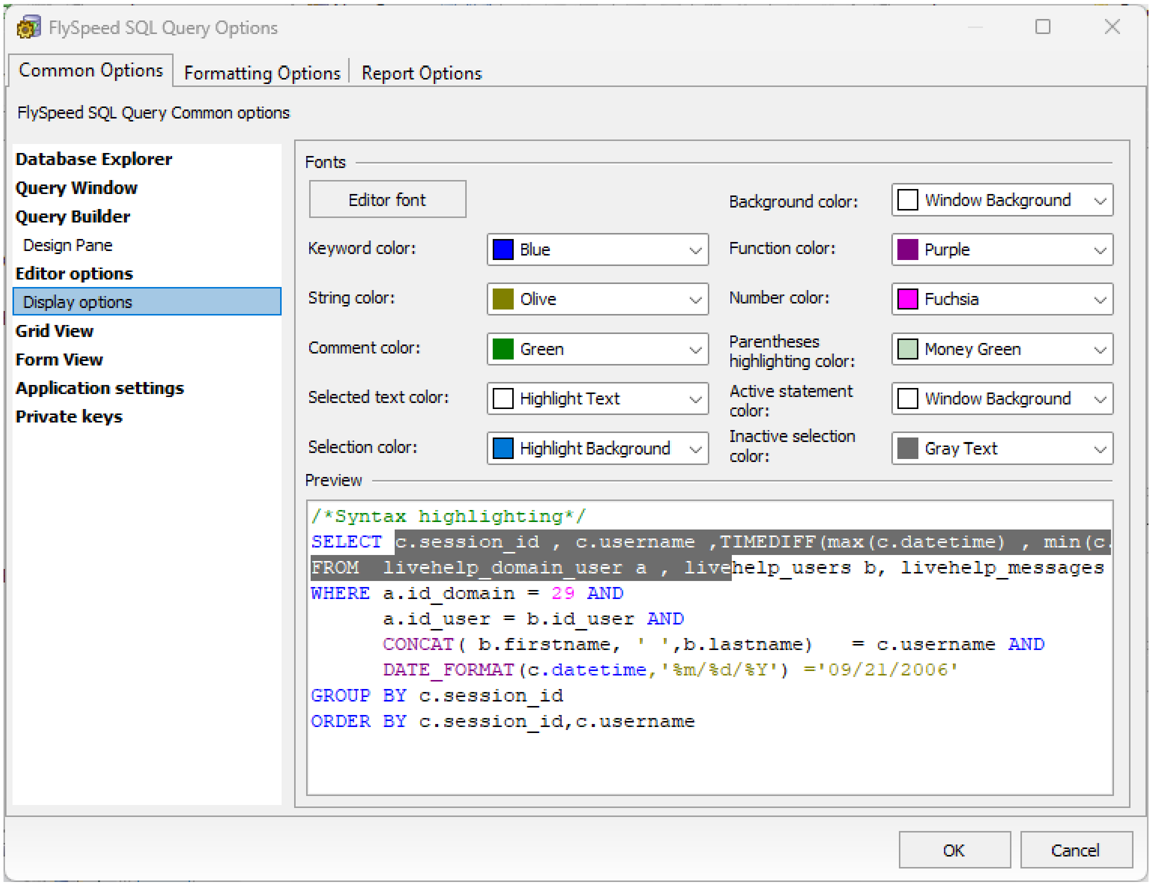
- Database Explorer panel – a tree-based view of tables, views and stored procedures.
- Space for SQL Editors – A window that includes tabs with a text editor with syntax highlighting and auto-completion, visual query builder, results panel, query information and history. Each query has its own window.



```
Select
    *
From
    COUNTRY Inner Join
    JOB On JOB.JOB_COUNTRY = COUNTRY.COUNTRY
```

Query execution time: 94 ms Records: 31+ Rows selected: 0

The workspace is highly customizable. Visual themes, fonts, colors, and syntax themes are adjustable, making long query sessions easier on the eyes.



Background color: Window Background

Keyword color: Blue

Function color: Purple

String color: Olive

Number color: Fuchsia

Comment color: Green

Parentheses highlighting color: Money Green

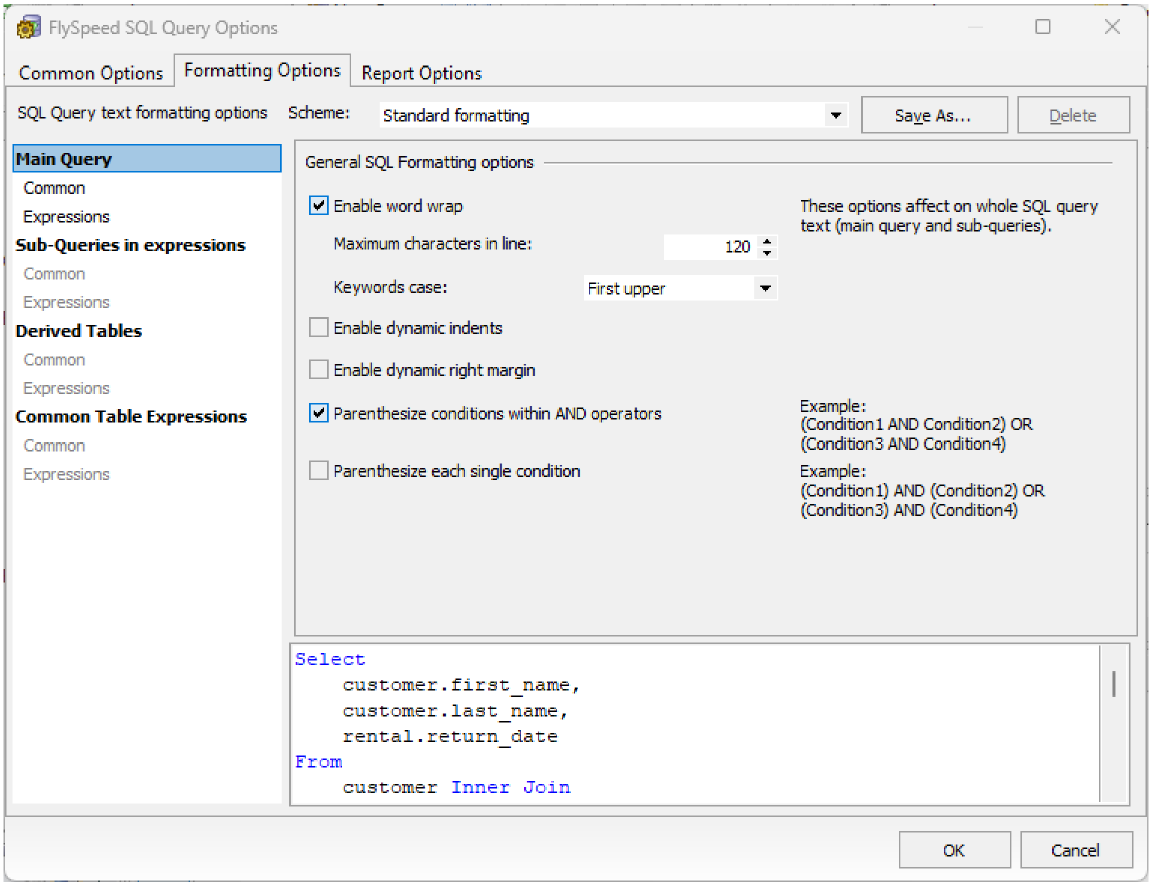
Selected text color: Highlight Text

Active statement color: Window Background

Selection color: Highlight Background

Inactive selection color: Gray Text

```
/*Syntax highlighting*/
SELECT c.session_id , c.username ,TIMEDIFF(max(c.datetime) , min(c.
FROM  livehelp_domain_user a , livehelp_users b, livehelp_messages
WHERE a.id_domain = 29 AND
      a.id_user = b.id_user AND
      CONCAT( b.firstname, ' ',b.lastname)   = c.username AND
      DATE_FORMAT(c.datetime,'%m/%d/%Y') ='09/21/2006'
GROUP BY c.session_id
ORDER BY c.session_id,c.username
```



Scheme: Standard formatting

☑ Enable word wrap

Maximum characters in line: 120

Keywords case: First upper

□ Enable dynamic indents

□ Enable dynamic right margin

☑ Parenthesize conditions within AND operators

□ Parenthesize each single condition

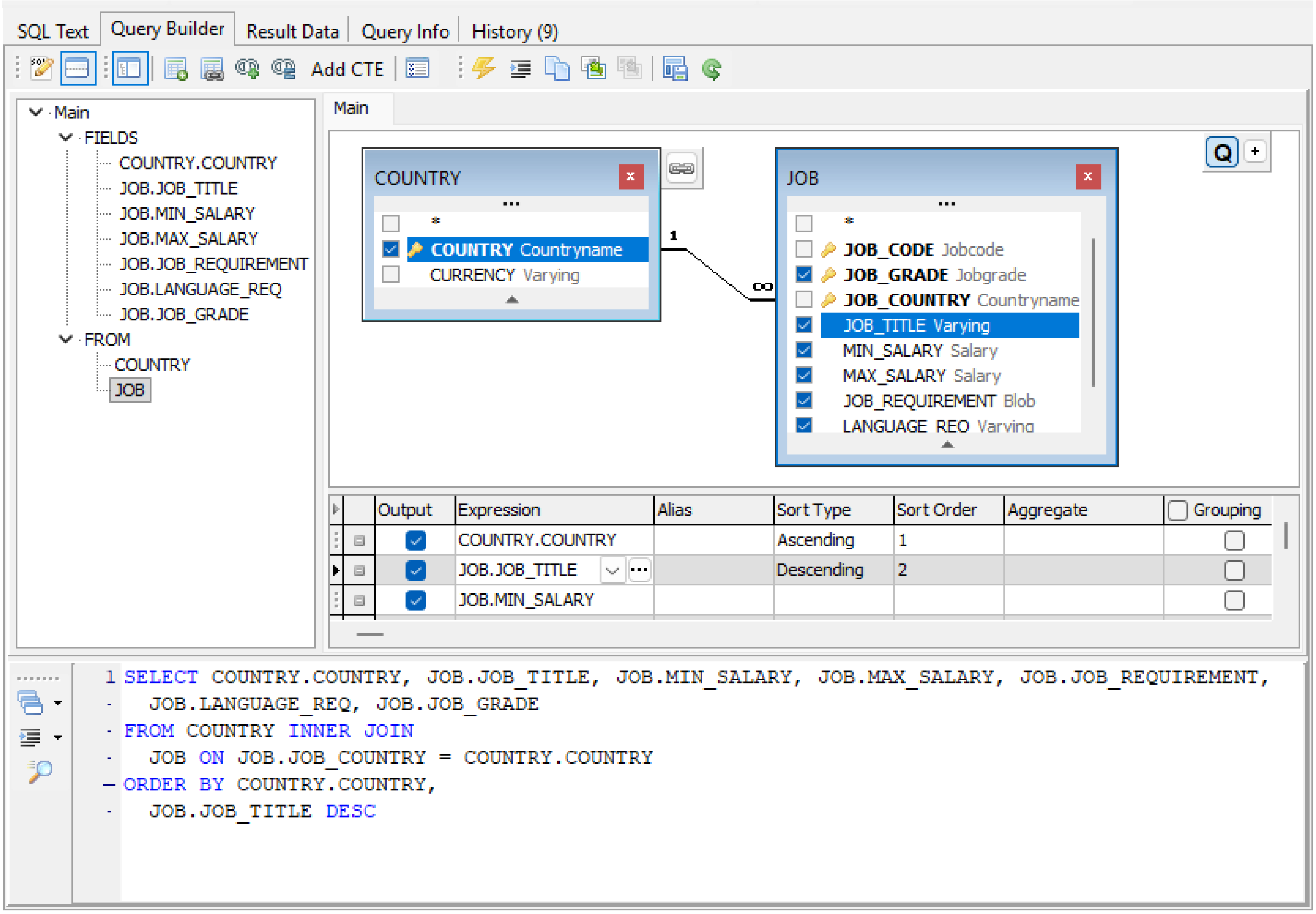
These options affect on whole SQL query text (main query and sub-queries).

Example:
(Condition1 AND Condition2) OR
(Condition3 AND Condition4)

Example:
(Condition1) AND (Condition2) OR
(Condition3) AND (Condition4)

```
Select
    customer.first_name,
    customer.last_name,
    rental.return_date
From
    customer Inner Join
```

FlySpeed SQL Query provides a visual query designer, allowing users to drag and drop tables, define joins, and build queries without typing SQL manually. The visual builder generates Firebird-compatible SQL, which can then be edited in raw text mode if needed. Common Table Expressions (CTEs), window functions, and derived tables are supported, ensuring modern Firebird syntax is preserved.

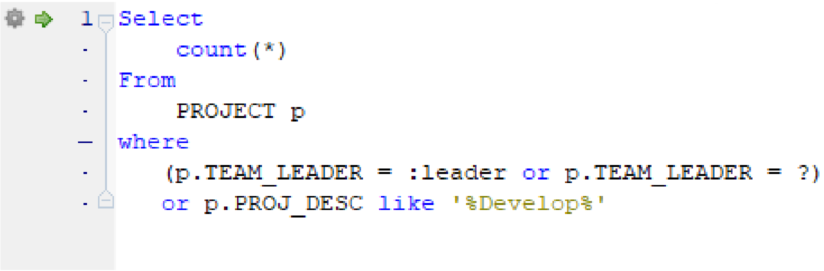


- Main
  - FIELDS
    - COUNTRY.COUNTRY
    - JOB.JOB_TITLE
    - JOB.MIN_SALARY
    - JOB.MAX_SALARY
    - JOB.JOB_REQUIREMENT
    - JOB.LANGUAGE_REQ
    - JOB.JOB_GRADE
  - FROM
    - COUNTRY
    - JOB

Output	Expression	Alias	Sort Type	Sort Order	Aggregate	☐ Grouping
☑	COUNTRY.COUNTRY		Ascending	1		☐
☑	JOB.JOB_TITLE		Descending	2		☐
☑	JOB.MIN_SALARY					☐

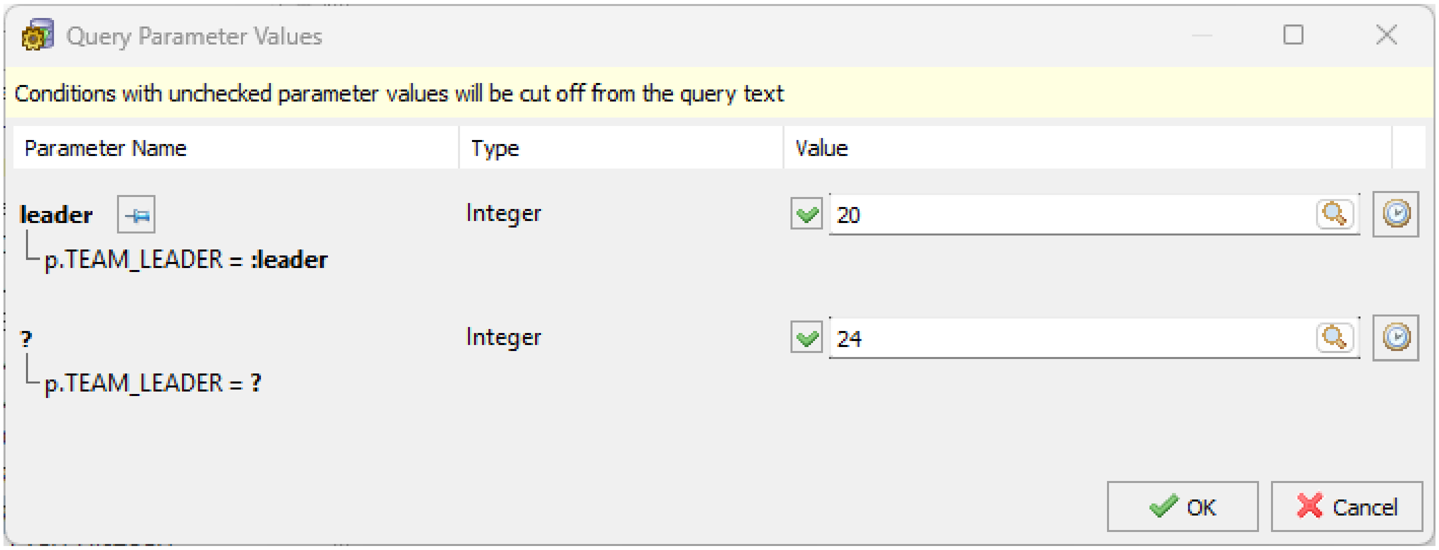
```
SELECT COUNTRY.COUNTRY, JOB.JOB_TITLE, JOB.MIN_SALARY, JOB.MAX_SALARY, JOB.JOB_REQUIREMENT,
  JOB.LANGUAGE_REQ, JOB.JOB_GRADE
FROM COUNTRY INNER JOIN
  JOB ON JOB.JOB_COUNTRY = COUNTRY.COUNTRY
ORDER BY COUNTRY.COUNTRY,
  JOB.JOB_TITLE DESC
```

What is particularly neat is support for query parameters. You can use either named or classic (?) parameters.



```
Select
    count(*)
From
    PROJECT p
where
   (p.TEAM_LEADER = :leader or p.TEAM_LEADER = ?)
   or p.PROJ_DESC like '%Develop%'
```

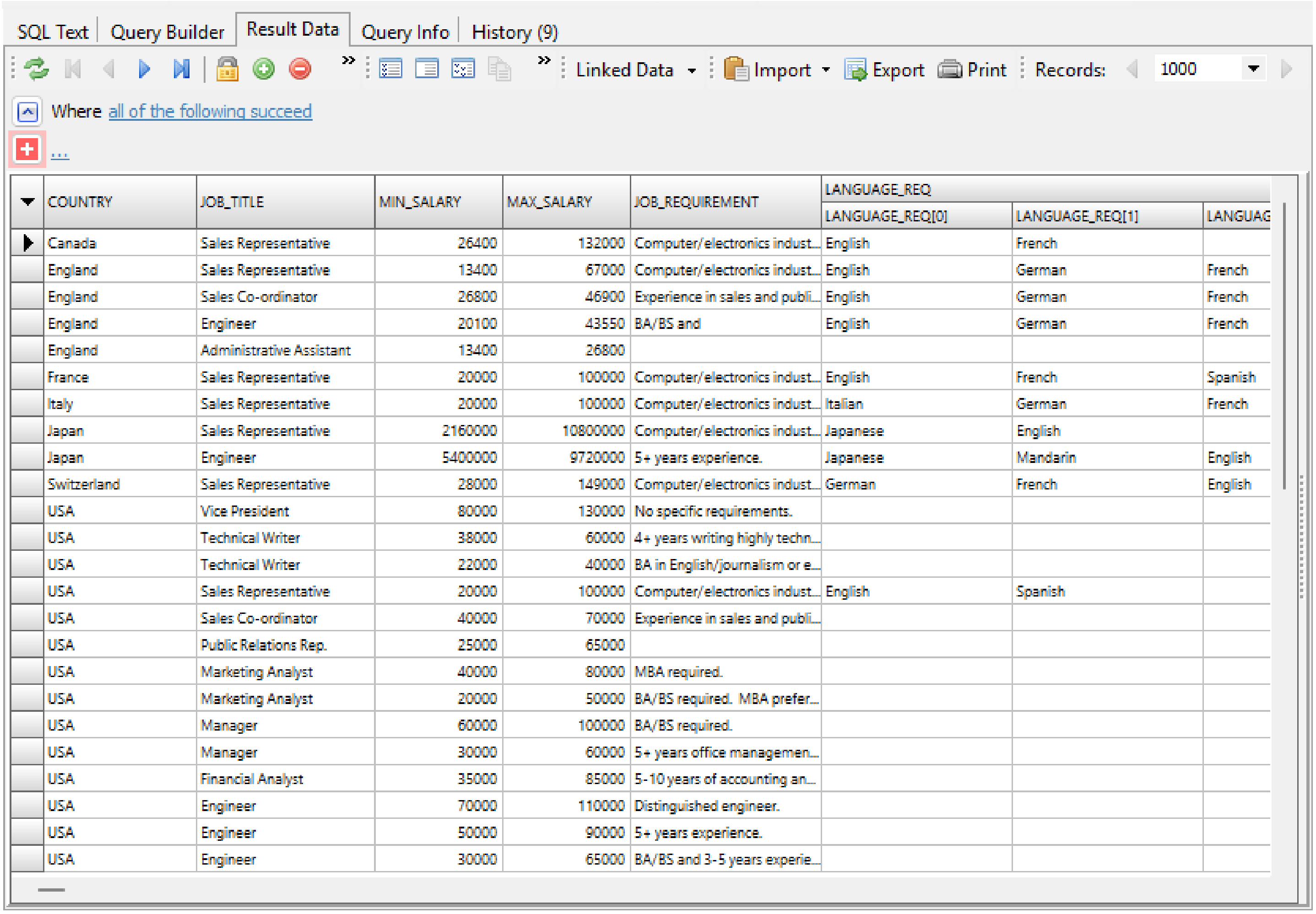
Paremeter values are entered on query execution.



Conditions with unchecked parameter values will be cut off from the query text

Parameter Name	Type	Value
leader p.TEAM_LEADER = **:leader**	Integer	☑ 20
? p.TEAM_LEADER = **?**	Integer	☑ 24

The data grid is central to FlySpeed's appeal. It allows interactive filtering, grouping, and sorting of result sets. Users may apply quick filters without altering SQL statements. Note that ARRAY columns are fully supported.



Where all of the following succeed

...

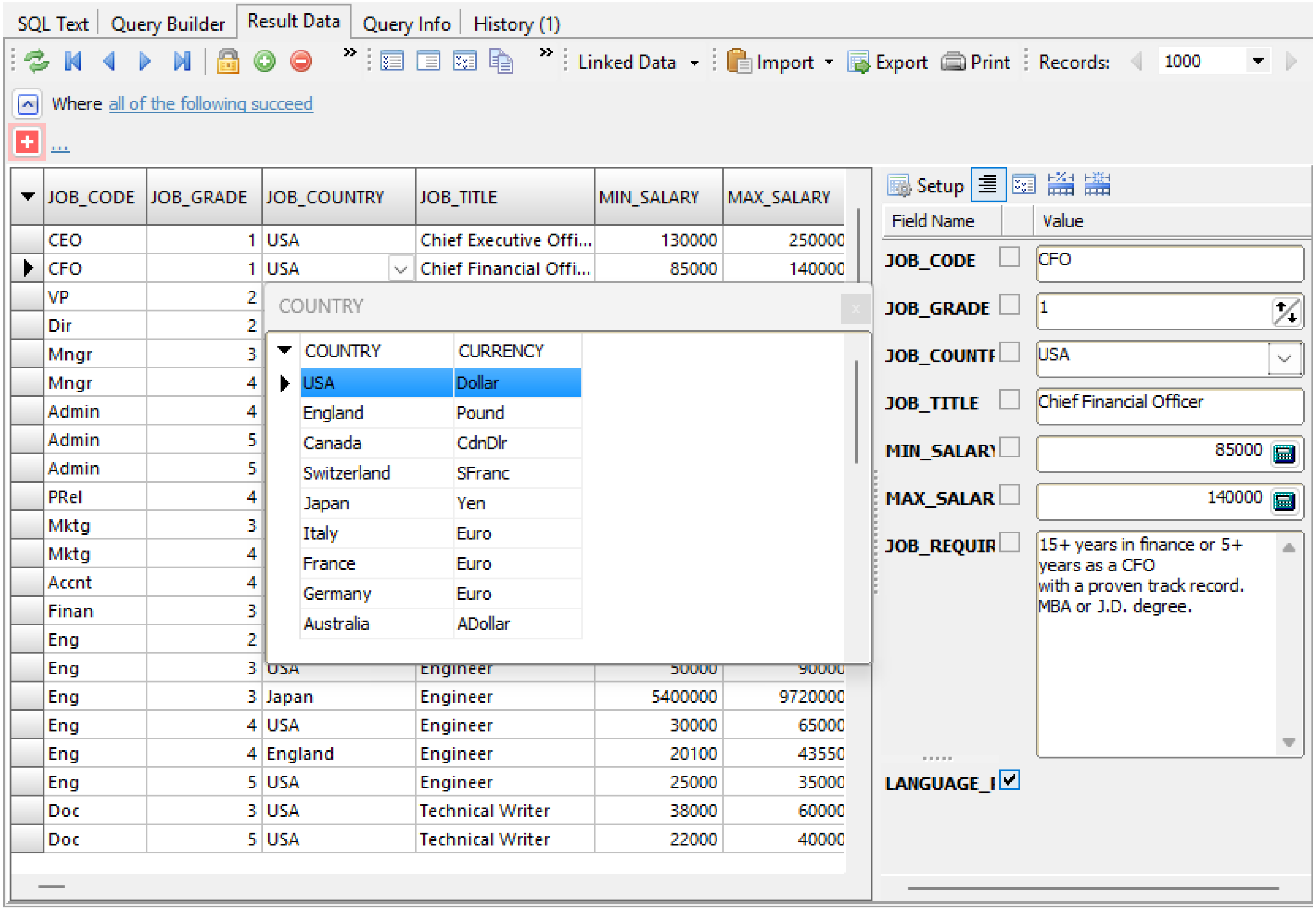
COUNTRY	JOB_TITLE	MIN_SALARY	MAX_SALARY	JOB_REQUIREMENT	LANGUAGE_REQ		
					LANGUAGE_REQ[0]	LANGUAGE_REQ[1]	LANGUAG
Canada	Sales Representative	26400	132000	Computer/electronics indust...	English	French	
England	Sales Representative	13400	67000	Computer/electronics indust...	English	German	French
England	Sales Co-ordinator	26800	46900	Experience in sales and publi...	English	German	French
England	Engineer	20100	43550	BA/BS and	English	German	French
England	Administrative Assistant	13400	26800				
France	Sales Representative	20000	100000	Computer/electronics indust...	English	French	Spanish
Italy	Sales Representative	20000	100000	Computer/electronics indust...	Italian	German	French
Japan	Sales Representative	2160000	10800000	Computer/electronics indust...	Japanese	English	
Japan	Engineer	5400000	9720000	5+ years experience.	Japanese	Mandarin	English
Switzerland	Sales Representative	28000	149000	Computer/electronics indust...	German	French	English
USA	Vice President	80000	130000	No specific requirements.			
USA	Technical Writer	38000	60000	4+ years writing highly techn...			
USA	Technical Writer	22000	40000	BA in English/journalism or e...			
USA	Sales Representative	20000	100000	Computer/electronics indust...	English	Spanish	
USA	Sales Co-ordinator	40000	70000	Experience in sales and publi...			
USA	Public Relations Rep.	25000	65000				
USA	Marketing Analyst	40000	80000	MBA required.			
USA	Marketing Analyst	20000	50000	BA/BS required. MBA prefer...			
USA	Manager	60000	100000	BA/BS required.			
USA	Manager	30000	60000	5+ years office managemen...			
USA	Financial Analyst	35000	85000	5-10 years of accounting an...			
USA	Engineer	70000	110000	Distinguished engineer.			
USA	Engineer	50000	90000	5+ years experience.			
USA	Engineer	30000	65000	BA/BS and 3-5 years experie...			

After running the SQL query, all records are retrieved, up to the specified limit. The default limit is 1000 records, the maximum is 100k.

Transaction handling is implicit. Each Query Editor opens its own database attachment, and uses single transaction (with default parameters) to execute the query. Transactions are closed immediately after loading the data.

In addition to additional filtering, you can edit the data (if relevant), either directly in the data grid or in a more convenient form view for the current record.

You can also view data from linked tables either directly in the foreign key column...



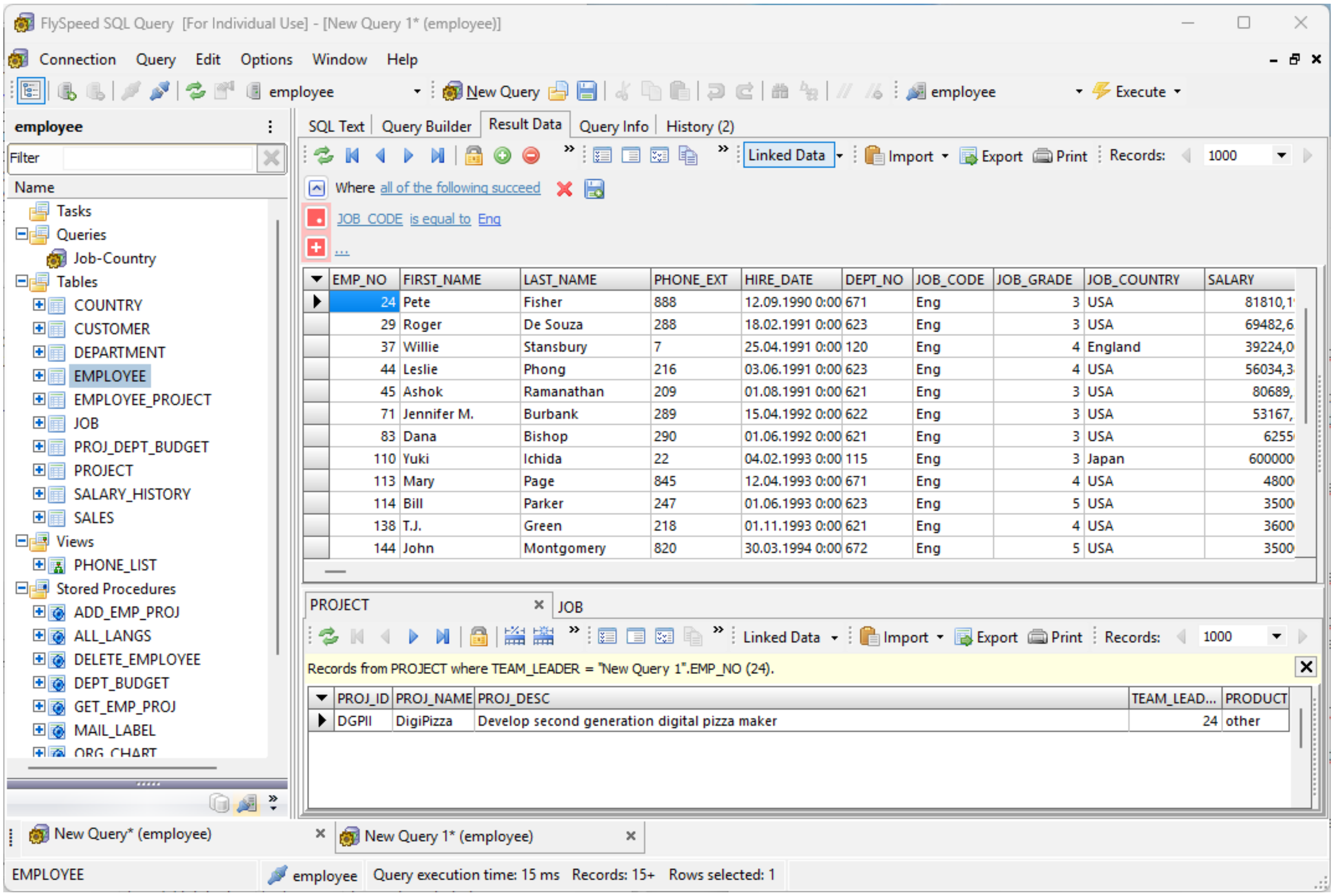
JOB_CODE	JOB_GRADE	JOB_COUNTRY	JOB_TITLE	MIN_SALARY	MAX_SALARY
CEO	1	USA	Chief Executive Offi...	130000	250000
CFO	1	USA	Chief Financial Offi...	85000	140000
VP	2				
Dir	2				
Mngr	3				
Mngr	4				
Admin	4				
Admin	5				
Admin	5				
PRel	4				
Mktg	3				
Mktg	4				
Accnt	4				
Finan	3				
Eng	2				
Eng	3	USA	Engineer	50000	90000
Eng	3	Japan	Engineer	5400000	9720000
Eng	4	USA	Engineer	30000	65000
Eng	4	England	Engineer	20100	43550
Eng	5	USA	Engineer	25000	35000
Doc	3	USA	Technical Writer	38000	60000
Doc	5	USA	Technical Writer	22000	40000

COUNTRY

COUNTRY	CURRENCY
USA	Dollar
England	Pound
Canada	CdnDlr
Switzerland	SFranc
Japan	Yen
Italy	Euro
France	Euro
Germany	Euro
Australia	ADollar

Field Name		Value
JOB_CODE	□	CFO
JOB_GRADE	□	1
JOB_COUNTR	□	USA
JOB_TITLE	□	Chief Financial Officer
MIN_SALARY	□	85000
MAX_SALAR	□	140000
JOB_REQUIR	□	15+ years in finance or 5+ years as a CFO with a proven track record. MBA or J.D. degree.
LANGUAGE_I	☑	

or in a separate linked results pane that only displays records that match the current master record in the table. These child tables can be viewed very easily using the “Linked Data” feature. You can open multiple child tables, either for the main query result or for any child table to create a cascade.



Where all of the following succeed

JOB_CODE is equal to Eng

EMP_NO	FIRST_NAME	LAST_NAME	PHONE_EXT	HIRE_DATE	DEPT_NO	JOB_CODE	JOB_GRADE	JOB_COUNTRY	SALARY
24	Pete	Fisher	888	12.09.1990 0:00	671	Eng	3	USA	81810,1
29	Roger	De Souza	288	18.02.1991 0:00	623	Eng	3	USA	69482,6
37	Willie	Stansbury	7	25.04.1991 0:00	120	Eng	4	England	39224,0
44	Leslie	Phong	216	03.06.1991 0:00	623	Eng	4	USA	56034,3
45	Ashok	Ramanathan	209	01.08.1991 0:00	621	Eng	3	USA	80689,
71	Jennifer M.	Burbank	289	15.04.1992 0:00	622	Eng	3	USA	53167,
83	Dana	Bishop	290	01.06.1992 0:00	621	Eng	3	USA	6255
110	Yuki	Ichida	22	04.02.1993 0:00	115	Eng	3	Japan	600000
113	Mary	Page	845	12.04.1993 0:00	671	Eng	4	USA	4800
114	Bill	Parker	247	01.06.1993 0:00	623	Eng	5	USA	3500
138	T.J.	Green	218	01.11.1993 0:00	621	Eng	4	USA	3600
144	John	Montgomery	820	30.03.1994 0:00	672	Eng	5	USA	3500

Records from PROJECT where TEAM_LEADER = "New Query 1".EMP_NO (24).

PROJ_ID	PROJ_NAME	PROJ_DESC	TEAM_LEAD...	PRODUCT
DGPII	DigiPizza	Develop second generation digital pizza maker	24	other

Query execution time: 15 ms Records: 15+ Rows selected: 1

FlySpeed SQL Query is available in both free and commercial editions. The paid version enables functionality for:

- Data export to most popular office formats: MS Excel, CSV, plain text, JSON, SQL script, HTML and XML.
- Printing and exporting data to PDF.
- Import data from the clipboard, CSV and Excel files.

Conclusion

For Firebird users, FlySpeed SQL Query fills a useful niche: a dedicated query and analysis tool that combines visual design with raw SQL power. It is not a substitute for Firebird-specific administration utilities, but it complements them by making query design, data inspection, and everyday exploration more efficient.

The free edition is suitable for casual use, but professionals working with Firebird on a daily basis will benefit from the extended features of the paid version. Its balance of usability, performance, and Firebird compatibility makes it a valuable addition to the Firebird user's toolkit.

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:

- Using one prepared statement for updating a different field
- Best way to update index statistics on new database
- Difference between * vs fieldname in aggregate
- External file structure

Using one prepared statement for updating a different field

Ariel Sakin asked:

I have the following prepared statement:

```
UPDATE TABLE requests SET f1=?, f2=?, f3=?
```

Where `f1`, `f2` and `f3` are columns that may contain large string values (some of them may be BLOBs). In reality I have 20+ columns.

I am looking for a way of using that statement for updating my table without having to provide it with all the values all the time. for example, if I need to update only `f1` and `f3` I don't want to be forced to pass the value for `f2` that do not need to be changed. Writing a different statement for every column is not a good option since sometimes I need to update 15 out of 20 fields and the overhead of running 15 queries is too large.

Is there a way to do it?

Roger Vellacott answers:

You can write the update query in the following form:

```
UPDATE MY_TABLE SET
MYFIELD1 = COALESCE(:MYPARAM1,MYFIELD1),
MYFIELD2 = COALESCE(:MYPARAM2,MYFIELD2)
WHERE ...etc
```

Make sure that the parameter values are `NULL` for those fields you do not want to update.

Best way to update index statistics on new database

Bill Oliver asked:

I have a user who creates his databases, creates tables, inserts initial data, and then wants to set the statistics on the indexes he just created. In PostgreSQL, he would issue the `VACUUM DATABASE;` command.

Here is the what he came up with

```
EXECUTE BLOCK AS
DECLARE VARIABLE L_INDEX VARCHAR(100);
DECLARE VARIABLE L_SQL VARCHAR(200);
BEGIN
  FOR SELECT RDB$INDEX_NAME FROM RDB$INDICES INTO :L_INDEX DO
  BEGIN
    L_SQL = 'SET statistics INDEX ' || L_INDEX || ';';
    EXECUTE STATEMENT :L_SQL;
  END
END
```

That sequence reduces load time, creates dense indexes, and sets the statistics correctly.

Ann W. Harrison answers:

The cleanest way to do it is to:

1. Create databases
2. Create tables
3. Load data & commit
4. Create indexes

That sequence reduces load time, creates dense indexes, and sets the statistics correctly.

Bill Oliver follows:

I suppose the user finds it more logical to submit all of the DDL first and create the data model up front.

If you used a data modeling tool like Erwin, you would get an output SQL file that would have the tables and indexes created in one place.

Ann W. Harrison answers:

That's nice and logical (I guess) but loading first then indexing is much more efficient.

Pavel Cíař added:

Another way is to create indices together with other database objects, but as `INACTIVE`. Then activate them after data insert. You can use the same `EXECUTE BLOCK`, but with `ALTER INDEX` … `ACTIVE` instead `SET STATISTICS` command.

Difference between * vs fieldname in aggregate

Is there any time where there is an effective difference between:

```
select count(`*`) from ...
```

and:

```
select count(myfieldname) from ... ?
```

Helen Borrie answers:

Sure is. `count(*)` counts all rows, whereas `count(fieldname)` counts all rows that have a non-null value in that field.

External file structure

Johnson Zhu asked:

How are data stored in external file for the data type like smallint, integer, char, varchar, datetime? If we want to generate external file ourselves, what shall we do? Our goal is to generate the file related to the external table file. Any suggestion?

Ann W. Harrison answers:

External tables use the same data types as internal tables. The actual representation of data other than char depends on the endianness and alignment rules of the processor, and on the ODS version of Firebird. These are the rules (I think) for ODS 11. Earlier ODS's used different alignment rules for dates (I think) but the general rules are still true.

`Char(n)` turns in to characters in the specific character set.

`Varchar(n)` turns into a two byte integer followed by characters. The two bytes (when viewed in the correct order) indicate the number of significant characters. Varchar fields always start on two-byte boundaries. Thus `table t1 (f2 char (5), f2 varchar(10))` will be stored as `'abcde 0x30x0abc'` on a little-endian machine.

The various numeric types (decimal, int, etc.) turn into the appropriate sized binary number for the precision of the number. (<5 two bytes, <10 four bytes, <19 eight bytes). The scale of decimal and numeric values is carried in the field definition, not in the store value, so `decimal (9,0)` value `55`, `decimal (9,1)` value `5.5`, and `decimal (9,2)` value `0.55` are all stored as the same value. Two byte integers are stored on two byte boundaries. Four byte integers are stored on four byte boundaries. Eight byte integers are stored on eight byte boundaries. To make the boundaries work, Firebird will put in pad bytes which must be ignored.

Floating point numbers are stored in ISO format and aligned on four (for float) or eight (for double) byte boundaries. In earlier ODS's, on some machines, doubles were stored on four byte boundaries.

Remember that date, floating point, and integer values use the processor rules for endianness.

Blobs don’t work in external files. When written to an external file a blob turns into a meaningless eight byte value.

Unlike internal tables, external tables do not have a built-in representation of null. Internal table include a bit array for each record indicating the null state for each field.

As Alexandre said, the easiest way to deal with external files is to turn everything into character strings. They’re transportable, you can read and edit them, and they don’t have pad characters. Remember to add a two character field at the end of each record for an end of line (CRLF characters).

The Pumpkin Query

A poorly written query might look harmless on the outside, but inside it’s stuffed with unnecessary full table scans. Don’t let your SQL turn into a jack-o’-lantern full of hot air—use indexes wisely, learn to read the query plan, and carve performance into shape.

The Cobweb of Cascades

Foreign keys with cascading actions are powerful, but too many tangled together can create a sticky web of unexpected deletes and updates. Always trace the strands carefully—you don’t want your data caught in a trap of unintended consequences.

The Batty Generator

Generators (sequences) are handy, but letting one fly out of control can leave you with surprising gaps or values far larger than expected. Keep an eye on your sequences, and reseed them if they start to flap into the night.

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

Firebird 25th Anniversary

This year Firebird marks a momentous milestone—25 years since the project began on 31 July 2000. In lieu of a physical celebration, we're celebrating in a very special way. Starting from 31th July, we'll be releasing a new Firebird-themed song every Thursday on our official Firebird-songs YouTube channel. Each track reflects a different aspect of the Firebird journey—innovation, community, and legacy. Tune in weekly, stream the music, and help us keep the rhythm of twenty-five years alive.

The Firebird Book — A Gift for Everyone

In honour of this milestone and the memory of Helen Borrie, The Firebird Book, Second Edition was made publicly available by IBPhoenix via their digital store. Everyone can download the complete edition **free of charge** — no strings attached.

If you'd like to support Helen's legacy and the Firebird community, there is an optional pay-what-you-like contribution. Funds will support the organization of the **Helen Borrie Memorial Award**, recognizing individuals with sustained contributions to the Firebird project.

Official Firebird Webshop

The Firebird Project has taken an important new step to strengthen its future by opening an official Webshop on the project's website. The idea is simple but powerful: every purchase made through this shop generates a commission that goes directly back into funding Firebird's open-source development. In other words, supporting the project can now be as easy as choosing the right tool, book, or service you already need.

The shop brings together a range of products connected with the Firebird ecosystem. Readers will find books, reporting tools, recovery utilities, monitoring add-ons, encryption modules, and services for optimization, migration, and database auditing. By placing these offerings under one roof, the Firebird team not only provides a trusted point of access but also creates a steady way to channel commercial interest back into the project.

This initiative is meant to serve both sides of the community. For everyday users, it is an easy way to discover resources that enhance Firebird and, at the same time, contribute to its growth. For companies that provide Firebird-related products or services, the Webshop offers a unique opportunity to reach the project's global audience. Vendors who would like to join are warmly invited to get in touch with the team by email to discuss how their products could be listed.

ScratchBird

ScratchBird is an ambitious experimental fork of the Firebird 6 codebase, created by Dalton Calford. Its guiding vision is to transform Firebird's legacy into a modern embedded database engine, and then extend it into a multi-protocol platform capable of speaking PostgreSQL and MySQL wire languages. Unlike most database forks, ScratchBird is also experimenting with development itself: multiple AI agents are being employed to generate tests and new features, adding an unusual layer of automation to the project.

At present, the project remains in its alpha stage. The current milestone, Alpha Stage 1, concentrates entirely on the embedded core without any networking features. The emphasis is on building a stable foundation: storage management, transactions, logging, error handling, and configuration. Only when this base is solid will the focus shift outward toward network capabilities and wire protocol compatibility. This cautious, phased approach is reinforced by a pair of detailed guiding documents—the Master Plan and the Authoritative Implementation Plan—which map out each stage of development.

The roadmap beyond the alpha stage is ambitious. Once the embedded engine is proven, ScratchBird will gradually introduce networking and client-server interaction, with the Y-Valve becoming the central dispatcher for multiple parsers. These parsers will allow the same engine to present itself as a Firebird, PostgreSQL, or MySQL server, potentially opening the door to cross-ecosystem compatibility. Supporting this evolution is a strong emphasis on testing, a necessity given the reliance on AI agents, and a safeguard against architectural drift as the project grows in complexity.

ScratchBird is still in its early days, but it stands out for the discipline of its planning and the boldness of its goals. It is neither a quick fork nor a simple experiment; rather, it is an attempt to build an embedded database core for the future, one that can adapt to multiple worlds of database connectivity. If its roadmap holds, the next phases will see it move from an intriguing concept into a platform that combines Firebird's proven strengths with new architectural flexibility and multi-protocol reach.

New Book: Secrets of Firebird Query Performance

Denis Simonov, a major contributor to the official Firebird SQL documentation, has written a comprehensive guide for database professionals seeking to master query optimization. Secrets of Firebird Query Performance will be published on November 12, 2025, as a 320-page printable PDF in A4 format. The book will be available at Firebird SQL Shop and through partners in four languages: English, German, Portuguese, and Russian, making it accessible to database developers and administrators worldwide.

New Articles

Comparison of data filling of Firebird 5 and PostgreSQL 17 databases

This database comparison study by Igor Valchenko analyzes storage efficiency between Firebird 5 and PostgreSQL 17 by creating identical tables with various data types (boolean, char, varchar, date, integer, numeric, float, time, timestamp) and populating them with one million rows each.

The research compares space utilization across different data types, revealing that PostgreSQL stores integer types more efficiently (81 MB versus 99.63 MB in Firebird), while Firebird demonstrates better efficiency with character data types, particularly for larger field sizes like CHAR(200) (2604.25 MB versus 2728 MB in PostgreSQL), and floating-point numbers.

The study's key findings emerge during update operations, where after updating

25% of all rows, PostgreSQL's database size increased to 9862 MB (a 25% growth), while Firebird showed more conservative growth to 8504 MB (only 8% increase), and after updating 50% of data, PostgreSQL required 26% additional space compared to Firebird's 21%, indicating that Firebird's multi-version concurrency control (MVCC) implementation is more space-efficient during frequent data modifications.

Writing custom providers for Firebird to access external databases

Ever wondered how to make Firebird talk to MySQL, PostgreSQL, or any other database through ODBC? Denis Simonov has crafted an excellent deep-dive into creating custom Provider plugins for Firebird that does exactly that. In this article you will find the coverage of implementation challenges encountered during the development of HQbird's MySQLEngine and ODBCEngine plugins, including architectural solutions for DBMS limitations.

Migrating from Firebird to PostgreSQL. What can go wrong?

We would like to draw your attention to this in-depth article, written on the basis of real-world experience, and devoted to a rarely discussed topic: migration from Firebird to PostgreSQL.

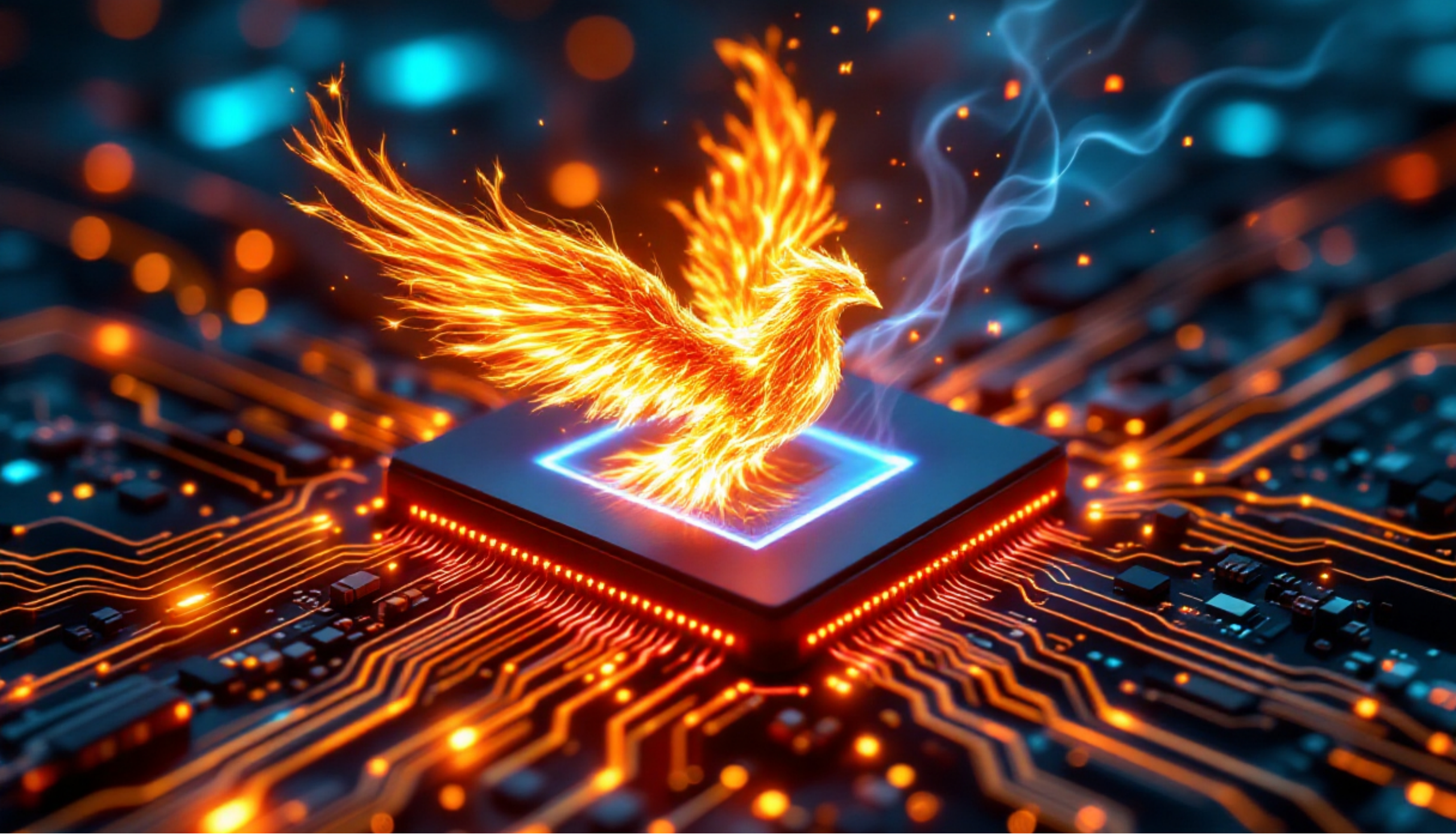
The article breaks down 23 real-world problems that catch teams off guard - like how PostgreSQL can make simple updates run 17 times slower than Firebird, or why your backup strategy will need a complete overhaul, and why you will need to rewrite all your triggers. It's not about bashing PostgreSQL (which is an excellent database), but rather giving DBAs, CTOs, and developers a clear picture of what they're signing up for.

The article is available in English, German, and Portuguese.

We would like to thank all who helped to gather materials and prepare translations: Dmitry Yemanov, Sergey Volkov, Denis Simonov, Alexey Kovyazin, Martin Köditz, and Edson Gregorio.

If you are considering database migration from Firebird to PostgreSQL or another database, please share with us your concerns and questions.

Please feel free to contact Alexey Kovyazin with all questions.



The Power of embedded Firebird

By Pavel Císař (IBPhoenix)

One of the most significant advantages of Firebird is its architecture, which allows it to be operated in a total of three modes that provide completely identical functionality with different operating characteristics. What is completely unique is the ability to operate Firebird both as a shared server and as a subsystem that is a direct part of an application built on top of it.

While much has been written about the characteristics of the Classic, SuperClassic and SuperServer architectures, there is only a little information about the possibilities of using Firebird as a direct part of an application, and even that information focuses mainly on the distribution and configuration of embedded Firebird. It is generally assumed that Firebird embedded is only good for small applications intended for one user. Therefore, in this issue we will take a closer look at the full range of possibilities that Firebird embedded offers to application developers. We believe that its capabilities will surprise you, and the outlined possibilities will inspire you to make the embedded engine a real ace up your sleeve.

First, the key facts

First of all, it is necessary to state that from the application's perspective, working with the shared and embedded versions of Firebird is completely identical and transparent. Therefore, any experimentation with embedded Firebird is completely safe and problem-free. If your existing application corresponds to one of the scenarios discussed in this article, you don't have to be afraid to try it with the embedded version. It is quite possible that the result will pleasantly surprise you.

Even with the embedded version, the application can work with multiple databases, create multiple connections to a single database, and create multiple transactions simultaneously within a single connection. Working with threads is also safe if each thread uses its own database connection (and its associated transactions and queries).

In fact, it's very likely that you've already worked with the embedded version of Firebird, or are currently working with it without even realizing it. Since Firebird 3, embedded has been a standard part of every server installation on all platforms, and is automatically used whenever you create a connection to a database without specifying a target computer or communication protocol. That's because the heart of Firebird - the Firebird engine - is a separate component, which can be found in the `plugins` subdirectory of the Firebird installation as a shared library `engineXX` (where XX is the engine version number). Only by "wrapping" it with an outer layer does it become a shared or embedded server running under the chosen Classic, SuperClassic or SuperServer architecture.

Firebird's operating mode is controlled by two parameters in the `firebird.conf` configuration file: `Providers` and `ServerMode`.

ServerMode Parameter

This parameter controls the method Firebird engine uses to work with databases and related Firebird server startup parameters.

The values are:

- **Super** or **ThreadedDedicated**: databases are opened exclusive by a single server

- process, attachments share single DB pages cache. This is the default mode.
- **SuperClassic** or **ThreadedShared**: databases are opened by a single server process, but it does not prevent opening them in other processes (embedded access), each attachment has its own DB pages cache.
- **Classic** or **MultiProcess**: for each attachment to the server, a separate process is started, each database may be opened by multiple processes (including local ones for embedded access), each attachment (process) has its own DB pages cache.

> *For embedded connections, the Classic setting is treated as SuperClassic.*

Providers Parameter

This parameter determines what providers will be used by Firebird. Format is the same as for the list of plugins because internally, a provider is just a kind of plugin.

What is important here is that this parameter is configurable also per database (via `databases.conf` file) or per connection (using `isc_dpb_config` connection parameter).

The default value for Firebird 5 is:

```
Providers = Remote, Engine13, Loopback
```

When creating a new connection, providers are tried sequentially with the specified connection string, and the one that reports success first is active.

Perhaps the most famous example that demonstrates the work of providers is the configuration:

```
Providers = Remote, Engine13, Engine12, Loopback
```

This configuration allows Firebird 5.0 to work with both native ODS 13.1 databases and ODS 12.0 databases from Firebird 3.0.

The `Remote` provider is responsible for interaction over the network, and `Loopback` provider is responsible for TCP/IP connections via `localhost`, as well as (on Windows) XNET connections to databases on the local machine.

> *Because providers are plugins, it is possible to create providers that can use other means to access data. These are particularly useful in conjunction with SQL EXECUTE ON EXTERNAL statements. For example, commercial providers for ODBC connections are available from IBPhoenix (FireLink) or IBSurgeon (as part of HQBird).*

So if you want to test how your existing application behaves when using embedded Firebird, you just need to change the connection string. However, it is necessary to take into account that the embedded engine works in the context of your application process, and therefore it may be necessary to adjust some configuration settings, e.g. database cache size, etc.

Use cases for the embedded engine

Using an embedded engine is certainly suitable for all single-user applications. You can also create a personal version of an application that is otherwise intended for simultaneous work by multiple users, essentially "for free". But there are other possible scenarios in which an embedded engine can be invaluable.

Web-based multiuser applications

Web technologies are no longer just the domain of the Internet or corporate networks, but also of personal computers. You can easily create and deploy a web application on a network, as well as on a workstation, and it can be used by multiple users or just one.

Using an embedded engine within a web application has the fundamental advantage of very low latency and overhead for data transfer between the database and the application. The safety of multithreading, and the ability to create multiple connections and transactions within a single connection make embedded Firebird a very attractive solution for web application data storage.

Of course, the scalability of such a solution has its limits. In this case, for the suitability of deploying embedded Firebird is more significant the amount of processed data and the size of the database than a large number of concurrent users. For vast amount of concurrent users you can leverage the possibility to run multiple application instances with embedded Firebird in SuperClassic mode as they can work with the same database simultaneously. But if your database is really huge, it might be better to use a remote server with enough memory to allocate a large page cache.

In any case, the ease of deploying web applications with an embedded engine (especially if you use containers like Docker) is very attractive for a wide range of applications.

Local-first applications

Local-first applications treat the data stored on a user's device as the primary copy, enabling them to work offline and experience faster responses due to the absence of network delays. While cloud-based servers act as secondary copies for synchronization and backup, the core operations occur locally, making the applications resilient to outages and giving users more control and ownership over their data. This architecture provides a superior user experience through speed, seamless offline functionality, and built-in collaboration, while also benefiting developers by simplifying network state management.

Using embedded Firebird as a local data store has a number of advantages, regardless of whether Firebird is also used as a central data store or not. The main problem (and not only) with this type of application is the synchronization of changes between the local and central storage. Firebird since version 3 offers an elegant solution to this problem, which we will introduce in more detail later.

Embedded as local data cache

The article "Challenges with Primary Keys" published in issue 2024/03 highlighted the fundamental problem of artificial primary keys - the need to retrieve data from linked tables, because the value of the foreign key cannot be displayed to the user (or used for selection). The consequence is increased query complexity (number of additional table joins) with all the consequences for performance.

With the help of Firebird embedded, it is possible to relatively easily implement a local cache for data from these tables, which allows to:

1. Load the necessary data for display from the referenced tables from the local database, which eliminates the need to join this table in the query.
2. To filter data using values from these tables, a local query is used to obtain the primary keys of records with the required values, which can then be used directly in the query to the master table, which eliminates the need for a join in this case as well.

Similar to the previous case, the key problem is (in this case, one-way) updating of data in the local database from the central storage.

Field workers

This is a hybrid case of the previous two scenarios. Many enterprise applications that are in principle designed for simultaneous work of multiple users also need to cover a special category of users who require access to all or a significant part of the data available in the central database, but at the same time they usually move in an environment that does not allow reliable and efficient work with the central application. Typical examples are, for example, field inspectors (pipelines, buildings, electrical wiring, etc.), business travelers, insurance agents resolving insurance claims on site, doctors on the go, etc.

As you can see, Firebird embedded finds its application in a wide range of practical solutions. Some tasks cannot even be easily implemented without its use.

Data synchronization between local and central database

For many use cases, it is important to efficiently handle data synchronization between a local and remote Firebird database (or other data store). Although Firebird does not provide direct support for this task (built-in replication cannot be effectively used), it does provide the means to create your own system for efficient data synchronization.

Basic concept of data synchronization

It is a fairly well-known fact that every Firebird table contains a pseudo-column RDB$DB_KEY that uniquely addresses a given row within the database. Less well-known is the fact that since Firebird version 3, every table also contains a pseudo-column RDB$RECORD_VERSION. This BIGINT column contains the ID of the transaction that created or updated the given row (or more precisely its version visible to the querying transaction).

```
SELECT t.RDB$RECORD_VERSION, t.* FROM TABLE_A t
```

So, using this column, you can easily identify rows that were inserted or changed after a certain transaction by a query:

```
SELECT * FROM TABLE_A WHERE RDB$RECORD_VERSION >= ?
```

With this knowledge, it is relatively easy to design a mechanism that selects the rows of the source database table needed to update the target database table:

1. We need to centrally store the source database transaction number that we will use to search for changes. The initial value will be set to zero. For this example, we will call this variable `LAST_SOURCE_TRANS`.
2. After connecting to the source database, we will start a SNAPSHOT transaction. For this example, we will call it `SRC_SYNC_TRANS`.
3. We will query the changed data in the source table (see above), using `LAST_SOURCE_TRANS` as the value of the search parameter.
4. In a loop, we will update the data in the target database table using the SQL statement `UPDATE OR INSERT INTO`.
5. We will perform points 3. and 4. for all synchronized tables.
6. We will update and save the value of `LAST_SOURCE_TRANS` for use in the next synchronization.
7. We will commit the transaction started in step 2.

The fundamental question is what value of `LAST_SOURCE_TRANS` to keep for the next synchronization. The first thing you might think of is to keep the `SRC_SYNC_TRANS` transaction ID. This is certainly the right choice if at the moment of its start there are no other transactions active in the source database that can modify the synchronized tables (e.g. when synchronizing from a local database to a shared one, this is easy to ensure). If such active transactions can exist, their possible changes would not be included in the subsequent synchronization.

For this purpose, it is necessary to keep the ID of the oldest active transaction (OAT) at the time of SRC_SYNC_TRANS start. Although this information can be obtained from monitoring tables, this method cannot be recommended at the moment due to the difficulty of avoiding increased server load in case of accumulation of such requests. Unfortunately, the value of OAT cannot yet be determined even with the help of Firebird context variables, so the only option is to use info calls for transactions (see the article "Firebird Info Calls" published in issue 2024/04).

However, when using OAT, there may be a situation where some changes are transferred to the target database twice, because between OAT and SRC_SYNC_TRANS there may be changes from already committed intermediate transactions. However, when using UPDATE OR INSERT INTO, this does not create any problem other than redundant data transfer.

Dealing with deleted rows

With RDB$RECORD_VERSION you can easily detect inserted or updated rows, but not deleted rows. Therefore, a different solution is needed for this case.

The simplest way is to store a record of each deletion in a separate log table in the AFTER DELETE trigger. If you use uniform artificial primary keys in replicated tables, you can create just one log table with two columns: the table identifier (preferably of type SMALLINT) and the primary key. If you use natural primary keys, the most optimal method is to create a separate log table (with only the primary key) for each replicated table. Querying deleted rows is then done in a similar way using RDB$RECORD_VERSION.

The logging table (or tables) needs to be cleaned from time to time of unnecessary records. When replicating from a client to a central database, these records can be deleted as part of the replication itself. The same applies if replication from the central database is only performed to a single client. If there are more clients, it is necessary to keep records of the transaction numbers used during replication by each client (e.g. in a dedicated table where each client has its own record that it updates). All records with RDB$RECORD_VERSION smaller than the smallest transaction number used by the clients can then be deleted.

Optimizations

Using `RDB$RECORD_VERSION` is very simple and straightforward, but at the time of writing it has one major drawback when processing very large tables. Queries cannot currently be optimized using an index, and the entire table is traversed. Therefore, if you are going to synchronize larger tables, you should first test the performance on realistic data volumes.

However, this limitation should be purely temporary. Currently, you can create a `COMPUTED BY` index on `RDB$RECORD_VERSION`, but queries using it return an empty result. If you were to try to work around this bug by creating an additional indexing column (e.g. `TRA_ID`) of type BIGINT, and try to populate it in a `BEFORE INSERT OR UPDATE` trigger with the value `NEW.RDB$RECORD_VERSION`, Firebird will allow you to do so, but will always store the value zero in the entry. Both are considered bugs (see issue 8749) that should be fixed.

Even without this optimization, synchronization using `RDB$RECORD_VERSION` should be fast enough for most usage scenarios. You can easily simulate the time to retrieve changes for individual tables. The `SELECT COUNT(*) FROM table` command will give you the basic time to process the table (X). The `SELECT FIRST 100 * FROM table` command will give you the rough time to retrieve 100 changed rows (Y). The formula (X + Y) * 0.95 will then give you a fairly accurate estimate of the time required to retrieve 100 changed rows.

Security aspects

Storing data on a local computer has major security aspects. While access permissions to data on a remote server can be handled fairly well, the only way to prevent unauthorized access to local databases is to encrypt the database. Fortunately, Firebird offers the option of encrypting databases since version 3. Although Firebird does not include any encryption plugin by default, several commercial solutions are available:

- Encryption Plugin Framework for Firebird from IBSurgeon
- DBEncryption Plugin for Firebird 3-5 from IBExpert
- Encryption plugin from IBPhoenix
- PhoenixVault from IBPhoenix is Firebird Embedded with built-in encryption

If none of the offered solutions suits you, you can create your own encryption plugin. An example can be found in the `examples/dbcrypt` directory of your Firebird installation.

Conclusion

Embedded Firebird is often underestimated, yet it represents one of the most versatile and practical deployment options available in the Firebird ecosystem. Far from being limited to small, single-user applications, it can serve as a powerful tool for developers who need the full strength of a relational database tightly integrated with their software.

From transparent compatibility with shared server deployments, through performance optimizations and synchronization techniques, to security features such as encryption, embedded Firebird offers a complete package. It allows applications to remain lightweight, portable, and independent, while still benefiting from all the robustness and features of the Firebird engine.

Whether you are building desktop software, specialized utilities, or even complex systems that later need to scale to a shared server, embedded Firebird ensures that you don't have to compromise. It is a hidden ace that many developers overlook—yet once discovered, it can transform the way you design and deliver your applications.

The Oracle of Statistics

Indexes keep their own internal statistics, but like a crystal ball, those numbers can grow cloudy over time. If queries act strangely, consider recomputing index statistics with SET STATISTICS—it's like clearing the glass so you can see the future again.

The Spider Join

Complex joins without proper conditions can spin a massive web, pulling in rows you never expected. Always double-check join criteria —or risk being caught in sticky, slow-running queries.

...And now for something completely different

The Solstice Enigma

The Solstice Code had become more than a curiosity—it was now the most sought-after secret in the Firebird world. After Renata's discovery and the Firebird Foundation's announcement of a reward for anyone who could stabilize the code beyond its fleeting solstice window, the community split into factions. Forums buzzed with speculation, corporations quietly spun up research teams, and whispers suggested that governments, too, were taking an interest. What had once been a hidden spark in the codebase had ignited a worldwide hunt.

But Renata was not part of the noisy crowds. She belonged to a smaller circle—just six people—who trusted each other enough to share what they dared not say publicly. Phoenix42 was there, the elusive veteran who had once guided her through the hazards of Turbo Mode. Two young Firebird developers, sharp and steady beyond their years, had earned their place through both skill and discretion. The group also included two seasoned IT professionals: one a longtime friend of Renata's, the other a quiet specialist introduced by Phoenix42.

They were equals. No one led. Each brought a different strength: technical precision, historical insight, intuition, or sheer persistence. Together, they built a picture that no single one of them could see alone.

Their discussions always circled back to the same question: who was Ignis, really? His code showed more than brilliance in programming—it hinted at someone who thought like a physicist, maybe even a mathematician with rare imagination. To tie a software optimization to the exact geometry of the solstice required not just coding, but a grasp of fields, waves, and celestial mechanics. Had Ignis been dabbling far outside the usual bounds of database engineering? Or had he been part of something larger—a research project, perhaps, where physics and computation collided?

The group tried to reason it through. Ignis could not have tested his idea only twice a year in a short solstice window. He must have had access to a lab, simulation power, and equipment most developers could never dream of. That raised disturbing implications. If he had worked alone, he would have needed a covert place to build a lab and tap immense amounts of electricity without drawing attention. If he had worked for an employer, then who? A corporation with deep pockets? A government agency? Something more shadowed?

The further they pushed the logic, the more uneasy they grew. Ignis had guarded his identity with the same care as Satoshi Nakamoto, only more so. He had vanished without trace, leaving behind code but no trail. People that careful usually had a reason. And if those reasons were tied to corporate secret projects, or military research, then anyone poking too deeply might find more than they bargained for.

Still, they pressed on. Step by step, their conversations turned toward the most practical lead: power. Physics experiments and simulations of this scale consumed vast amounts of it. And if Ignis hadn't paid for it, then he must have stolen it—from somewhere designed to mask such drains. Data centers rose to the top of their list: backup facilities, large enough to conceal a private lab, modestly secured, often overlooked.

After weeks of piecing together traces from Ignis's old communications, they narrowed down several candidates—centers that existed during his active years, that had never been thoroughly modernized, and that were big enough to hide something extraordinary.

Renata’s group investigated several sites, but found nothing, until now.

Halloween night cloaked the industrial district in a brittle silence. Once, this part of the city had been alive with furnaces and machines, but now it was mostly a skeleton of rust and shadows. The old data center sat at its heart, a windowless block of concrete with only the dull hum of cooling units to remind passersby that it still breathed. Officially, it served as a backup facility of low importance, but Renata’s group suspected otherwise. If Ignis had ever hidden a laboratory, this was the kind of place it might be.

The plan was simple: three members—Renata’s longtime friend Marta and the two young developers, Julian and Kira—would scout the site in person. Renata herself, Phoenix42, and his quiet associate stayed connected from their own locations, monitoring the security feeds and providing remote assistance. Trust and nerves bound them together.

“Gate ahead,” Marta whispered, her voice hushed but steady over the comm. She raised her phone just enough for the others to see the faint glow of the guard booth. Inside sat a lone security guard, more interested in his thermos than the perimeter.

“Only one?” Kira asked.

“Yeah. But the real trouble’s electronic,” Julian replied, already tapping on his laptop balanced awkwardly against his knees. “The system’s old, but not that old. Probably patched together over decades.”

Renata’s voice came through softly from the remote channel. “Careful with your probing. These systems sometimes throw false alarms. Just slip in quietly.”

They huddled near a patch of fence shadow, the autumn air crisp, carrying the faint smell of rotting leaves. Julian cracked into the security interface. For long minutes, only the sound of his typing filled the channel. Then—

“Uh, guys?” Renata’s voice broke the silence. “Pause a second. Do you see the lower right corner of that control panel?”

Julian frowned. “What corner? …oh. Huh. That’s not standard.”

A small icon glowed faintly on the old monitor—barely visible unless you knew where to look. It was a flame, stylized as a wing, the very symbol Ignis had once used as his avatar in the community.

Kira leaned in closer, squinting. “That… that can’t be a coincidence.”

“Click it,” Renata urged.

Julian hesitated, then tapped the icon. The screen flickered, resolving into stark white text on a black background:

```
To fly the bird at its peak,
Secrets lie where codes don’t speak.
But beware the flame that burns too bright,
For power gained may end the flight.
Who am I?
```

“That’s him,” Phoenix42 said quietly over the line. His voice, usually even, carried a weight now. “That’s Ignis.”

The three on-site exchanged glances.

“It’s ‘turbo,’ right?” Kira said. “Has to be.”

Julian typed the answer. A soft beep confirmed it, and the maintenance gate mechanism clicked. The side entrance light turned green.

“Access granted,” Marta muttered with a half-smile. “Let’s move.”

Inside, the data center was a cathedral of silence. Endless rows of server racks blinked like watchful eyes. Cooling fans whispered in the dark. Their footsteps echoed faintly off concrete as they moved along maintenance corridors, guided by Renata and Phoenix42 watching through tapped camera feeds.

“He wouldn’t hide it out in the open,” Marta reasoned. “Needs to be off the main path. Quiet. Accessible, but… private.”

“Check service corridors,” Phoenix42 suggested. “Back routes. That was always his style—indirection.”

They turned down a dimly lit passage. Most doors were plain, marked with dull stenciled numbers. But then Kira froze.

“There,” she whispered.

Half-concealed behind a stack of unused equipment was a narrow door. At eye level, faintly etched into the metal, was the familiar flame-wing icon.

Julian tried the handle. It was locked, secured by a simple numeric keypad glowing with five empty squares.

“Five digits,” he muttered. “And no indication what the code is.”

“Try the obvious,” Kira suggested. “What about ‘turbo’ again? On a phone keypad, that’s **88726**.”

Julian entered it. The keypad buzzed angrily, red light flashing. A counter blinked: 2 attempts left.

“Damn,” Marta whispered.

“Don’t burn them,” Phoenix42 warned. “Ignis was clever. He’d punish brute force.”

Silence stretched. All eyes turned toward Renata’s face on their screen, her expression tight with concentration. She knew Ignis better than any of them—at least, as much as anyone could from the trail he left in code and comments. Her thoughts churned: his patterns, his pride, his obsession with fire, flight, and hidden meaning. At the threshold of his inner sanctum, what would he choose?

Renata exhaled slowly. “Not turbo. Not some abstract reference. Here, at this door, it has to be personal. He would use his own name.”

Julian nodded, entered the digits: **44647**.

For a moment, nothing. Then the keypad chimed, soft and final. The lock clicked open. Marta pushed the door, and the dark room beyond seemed to draw them in.

The door creaked open on stale, heavy air. Dust drifted in their flashlight beams as the three stepped inside, their remote companions watching through grainy camera relays.

“Smells like it hasn’t been opened in decades,” Marta muttered, pulling her scarf tighter around her face.

The room stretched deeper than they expected. Against one wall stood a hulking computer server, its case coated in a fine gray film. A worktable sagged under piles of papers and notebooks, boards pinned with curling sheets scrawled in equations and diagrams no one immediately recognized—half physics, half mathematics, and more than a little madness.

But what stopped them all was the device.

It loomed in the center of the lab: a cage of copper coils, crystalline plates, and blackened capacitors wrapped around the charred carcass of a computer. Parts looked handmade, solder joints uneven, wires braided in strange patterns. Other components seemed almost alien, the kind of thing Tesla might have built in a fever dream.

“Holy hell,” Julian breathed. “What is this?”

Kira circled it cautiously, her flashlight tracing the burn marks across its surface. “Looks like it… fried itself. Whatever it did, it wasn’t safe.”

Phoenix42’s voice crackled softly in their ears. “Ignis. This is his work. No one else would have built something like that.”

They moved to the server. After brushing off layers of dust, Julian connected his laptop. The fans whirred weakly, and an ancient splash screen flickered to life.

“OpenSuSE,” Julian said, eyebrows raised. “Version’s ancient. And… Firebird’s here too. Prehistoric build.”

Directories bloomed on his screen—hundreds of text files, code fragments, and rambling documents. “We’ve got his files,” he whispered. “Notes, logs… this is everything.”

They copied the trove, progress bar crawling as Renata and Phoenix42 guided them remotely.

While the transfer ran, Marta leaned over one of the chalkboards, squinting at the swirling equations. “This isn’t just programming. It’s… physics. Look—he’s modeling particle fields.”

Renata’s voice was hushed. “That device… it was his stabilizer. A way to hold the hardware steady under Turbo Mode. According to these notes… it worked. For about a minute. Then it overloaded. Needed hours to recharge before another test.”

“That’s how he managed to develop the Solstice Code,” Kira realized. “He couldn’t wait for solstices—he built his own artificial one.”

“But it failed,” Phoenix42 added quietly. “Burned itself out before he perfected it.”

The deeper they read, the more unsettling the story became. Ignis had been close—too close. He wrote of new approaches, of leads he hadn’t yet pursued. Yet threaded through his notes were anxious lines, doubts about whether he would have time to finish. Some passages read like a man racing against the clock.

“Was he dying?” Marta wondered aloud. “Or… afraid of something?”

Several entries repeated the same phrase: Phoenix Protocol. No explanation, only a scattering of references—half warnings, half promises.

“Whatever it means,” Renata said, “it mattered to him. Maybe it was his fallback plan. Or maybe… his warning to us.”

On another board, a pattern of solar diagrams caught Julian’s attention. He traced a circle with his finger. “This is it. Look—he mapped solar flares, geomagnetic alignments, the exact tilt of Earth’s axis. He proved that during the summer solstice, nature itself recreates the stabilizing effect his device provided. But bigger. Stronger. Ten minutes instead of one.”

Renata nodded slowly. “That’s why it works. The solstice isn’t a trick—it’s the Sun and Earth aligning to do what his machine could only imitate.”

For a moment, none of them spoke. The enormity of it settled over them: the Solstice Code wasn’t just a quirk of software. It was Ignis’s legacy, a piece of him encoded into both machine and cosmos.

They finished the transfer, shut down the ancient server, and left the lab as they found it. The door closed with a hollow thud behind them, the flame icon fading once more into shadow.

Outside, the night air felt sharp and cold. The three walked back through the industrial ruins, laptops heavy with stolen secrets.

Renata broke the silence over the comm. “We know the *why* now. But the reward… stabilizing it permanently? We’re nowhere close. Ignis didn’t solve it. And if he couldn't…"

Phoenix42 cut in, his voice steady but grim. "He left us pieces. Maybe enough. But understand—he also left warnings. Whatever the Phoenix Protocol is, it wasn't for nothing."

Marta exhaled, her breath misting in the dark. "Then our real work starts now."

The six of them fell into silence, each carrying the weight of the night's discoveries. They had uncovered Ignis's hidden lab, solved the mystery of the Solstice link, and yet the true puzzle loomed larger than ever.

The Solstice Code's flame still burned—but so did its shadows.

Epilogue

When the team left, the lab fell back into silence. The broken machine in its center would never stir again.

But elsewhere, beyond their sight, something had changed. The moment they accessed Ignis's old server, a dormant process—buried so deep it escaped their notice—had stirred. For decades it had waited, a lock without a key. Their intrusion had been the key.

Across forgotten lines of network, in places none of them suspected still connected, a signal moved. Outdated routers blinked awake. Consoles long darkened flickered with life.

And then, on a hidden terminal somewhere in the world, words appeared:

```
PHOENIX PROTOCOL – INITIATED.
```

Open-source work often doesn't come with a SALARY, but it does bring a kind of recognition. Each contributor leaves a trace of their work, whether in code, drivers, or documentation, and the community remembers them in subtle ways.

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!