EmberWings

2025/1

Firebird

March 2025
www.firebirdsql.org
emberwings@firebirdsql.org

The Firebird project was created at SourceForge on

July 31, 2000

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

Support Firebird

Thank you for your support!

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

In This Issue:

Firebird in the year of the Wood Snake

Dear Readers,

As we step into 2025, the Firebird community finds itself on the path to an extraordinary milestone. This year marks 25 years since Firebird first emerged as an independent open-source database project, and on July 31, 2025, we will officially celebrate a quarter-century of innovation, dedication, and progress.

Born in 2000 from the open-sourcing of Borland's InterBase, Firebird has since carved out its own legacy, evolving through the contributions of a passionate global community. What began as a bold effort to preserve and expand a powerful database system has flourished into a mature, feature-rich solution used by businesses, developers, and institutions worldwide.

The journey has been one of perseverance and ingenuity. Firebird has continuously advanced while staying true to its core values: reliability, efficiency, and independence. Through major releases, performance enhancements, and an unwavering commitment to open-source principles, Firebird has proven its longevity in an ever-changing technological landscape.

The 25th Anniversary is not just a time to celebrate past achievements but also an opportunity to look ahead. The project continues to evolve, with new developments shaping the future of Firebird and ensuring it remains a competitive choice for database solutions. As we move forward, community involvement remains as crucial as ever. Whether through development, documentation, advocacy, or simply sharing experiences, every contribution strengthens the Firebird ecosystem.

This year will be filled with moments of reflection and anticipation. From special events and retrospectives to discussions about the road ahead, 2025 promises to be a defining chapter in Firebird's story. Let's honor the past, embrace the present, and shape the future together.

Stay tuned—there's much more to come!

Warm regards,
The EmberWings Team

The junior developer approached the master with a troubled look. “Master, I seek your wisdom. I am building a scalable application atop Firebird. I have chosen a framework to simplify my task, yet I find my efforts growing slower and more tangled. The framework promises much, yet my application stumbles under its weight. Why is this so?”

The master regarded the junior with a gentle smile. “Tell me, what does this framework do?”

“It converts between objects and tables,” the junior said. “It abstracts the database entirely, so I need not write SQL myself. It promises to work with any database engine, so I am not bound to Firebird alone. And it wraps queries in structures that seem simple but feel… unnatural.”

The master nodded knowingly. “Come,” he said, “let us visit the garden.”

In the garden stood an elegant fountain. Water flowed from a high spout, cascading into a series of bowls, each smaller than the last, before finally spilling into a broad basin below.

The master turned to the junior. “This fountain is like SQL,” he said. “Each bowl is a query, naturally flowing into the next. The database is designed to move data efficiently, one stream at a time.”

The master then placed a series of buckets under the fountain, each with pipes connecting them in loops and spirals. The water sputtered, pooled, and spilled in odd directions.

“This,” the master continued, “is an object-relational mapper. It insists the fountain must behave like a series of buckets. It converts the flow of data into something unnatural and laborious, because it believes the objects and the database must mirror one another.”

The junior frowned. “But Master, does not the ORM make development easier?”

“For small gardens, perhaps,” the master replied. “But Firebird, like any SQL database, thrives when allowed to flow as it was designed. The ORM fights against this flow, endlessly translating, until the fountain becomes clogged with its own effort.”

The master walked to another part of the garden, where a cluster of tools lay. He picked up a watering can and held it aloft.

“This is the Delphi TDataSet,” the master said, tipping the can to dribble water onto a stone path. “It forces the fountain into a single, rigid flow, one drop at a time. It treats each query as though it were a file on a disk. This abstraction is not natural to Firebird, which works in sets and streams.”

The junior hesitated. “Master, what of frameworks that promise to work with any database engine? Surely such versatility is useful?”

The master chuckled and pointed to a row of potted plants. Each pot was different—some tall and narrow, others shallow and wide.

“Now imagine,” the master said, “a single watering can designed to water all these pots. It must pour neither too much nor too little. It cannot adapt to the unique needs of each pot, so it provides only the bare minimum to all. Such frameworks, promising ‘unified access,’ reduce the capabilities of each engine to the smallest common denominator. They waste the strengths of Firebird and constrain its power.”

The junior stared at the tangled fountain, the rigid watering can, and the neglected plants. “What, then, is the right way, Master?”

The master smiled. “The right abstraction is one that respects the nature of Firebird. Write queries in its language, for SQL is its voice. Build your application to work with its transactions, its indexing, its streams. Do not demand that it pretend to be something it is not. When you abstract, do so to enhance, not to obscure.”

The junior bowed deeply. “I understand, Master. I will abandon the frameworks that force Firebird into unnatural shapes and let it flow as it was meant to.”

The master nodded. “Remember, junior: the river flows fastest when the rocks are placed with care. Do not fight the river; guide it.”

And as the junior departed, he felt the weight of his code begin to lift, like a fountain set free to flow once more.

Why are some Firebird apps cursed?

By Pavel Císař (IBPhoenix)

Every application will, at some point, encounter issues when running on Firebird. Some of these are natural consequences of system evolution—changes in dependencies, updates to the database engine, the need for re-optimization as workloads shift, or the occasional performance bottleneck that requires fine-tuning. Such challenges are expected in any long-lived software system and are typically solvable with configuration adjustments or routine maintenance.

However, some applications seem to experience persistent, recurring problems—unexplained slowdowns, erratic behavior, and performance degradation that defy conventional fixes. These applications feel as if they are cursed, doomed to struggle with Firebird no matter what adjustments are made. The real culprit, though, is not Firebird itself but rather fundamental design flaws embedded in the application's architecture.

In this article, we will explore the most common reasons why applications become "cursed" when running on Firebird.

Understanding the Fundamentals of Firebird Application Design

Many of these problems arise because the application is built on assumptions, abstractions, and design choices that are either misaligned with Firebird’s principles or, in some cases, in direct conflict with them. Whether it’s the misuse of ORM frameworks, poor database schema design, excessive reliance on indexing, or the creation of artificial bottlenecks, these issues stem from a failure to understand and respect how Firebird operates. The result is an application that constantly fights against its own database rather than leveraging its strengths.

To properly understand the sources of problems, it is first necessary to know and understand the key elements that define the foundations of a "successful" application. These pillars are:

- Understanding Firebird architecture and implementation.
- Correct identification of data needs within the problem being solved, especially their distribution in time and space, and in individual deployments.

These are then directly reflected in:

- Optimal use and management of key server resources, i.e. connections, transactions and queries.
- Proper database design.
- Efficient data handling.

All these pillars are interconnected, and any deficiency in one will automatically be reflected in all the others.

When designing an application architecture, this is especially evident in the choice of appropriate compromises that define the possibilities and limits of the resulting architecture, which can subsequently be changed only very difficultly (if at all).

Let’s take a closer look at the individual pillars and their role.

Proper Use of Key Resources

Firebird’s performance hinges on the efficient management of three critical resources: connections, transactions, and (prepared) queries. Understanding their specific properties and parameters is vital to leveraging Firebird’s strengths and avoiding common pitfalls.

Connections

A connection defines the parameters and properties of a link to Firebird, both on the application side (e.g. character set, protocol) and on the server side (various caches, session, internal structures, triggers etc.). This makes the connection the most valuable (and most expensive) resource. The chosen way of using and managing them is therefore a fundamental factor that defines the capabilities of entire architecture. For example, using connection pooling affects the ability to use prepared queries.

Transactions

Firebird’s transaction system is one of its standout features, offering fine-grained control over how data is handled, with a particularly notable feature being the ability to run multiple transactions (with different parameters) simultaneously within a single connection.

These options are not available for nothing, but because the Firebird architecture requires careful handling of transactions for optimal function.

Prepared Queries

Prepared queries allow Firebird to compile an SQL statement once and reuse it with different parameters, reducing parsing and optimization overhead. Although newer versions of Firebird provide a cache of executed queries, which makes their repeated calls more efficient, this feature has its limitations and shortcomings. For frequently used queries, it is still preferable to use application-side prepared queries. These are connection-bound, but can be used with any transaction.

Proper Database Design

The importance of proper database design has been pushed into the background, especially in recent years, although a well-designed database is the backbone of any database application. Poor design leads to over-complicated queries, inefficient storage, and maintenance headaches.

For example, the article "Challenges with Primary Keys" published in issue 3/24 describes the direct consequences of choosing a primary key type on query complexity, stored data density and index efficiency. Similarly, choosing a compromise between long varchar and BLOBs, choosing a character set, or using user domains instead of built-in types affects application properties.

The category of database design also includes issues of data distribution, prediction of the evolution of database content, or archiving and removal of data that is no longer needed.

Efficient Data Handling

Firebird excels at set-based operations, but its performance suffers when applications fetch or process excessive data. Efficient data handling ties closely to transaction management and resource use. For example, an often overlooked issue is the impact of data conversion between data types used by the application and the server.

Firebird also allows you to use different strategies for working with data. Some allow you to offload processing to the server, such as calculated columns, stored procedures and functions, triggers, or PSQL blocks, others allow you to address data (such as RDB$DB_KEY or RDB$RECORD_VERSION columns) or process it as cursors (including bidirectional ones), etc.

Now that we have laid the foundations of "successful" applications, let’s take a closer look at the most common application design pitfalls.

Misaligned Abstractions: The Wrong Tools for the Job

In the high-stakes arena of software development, the drive to be "first to market" has shaped the industry over recent decades. This relentless pursuit of speed, amplified by the widespread adoption of agile methodologies, has prioritized rapid delivery and adaptability above all else. Agile's focus on iterative progress and quick feedback loops has spurred the creation and heavy reliance on technologies designed to accelerate development cycles. Tools like Object-Relational Mapping (ORM) frameworks, abstractions such as Delphi's TDataSet, and multi-database connection layers have become staples in the developer's toolkit, promising to simplify database interactions and enhance flexibility.

Yet, this emphasis on speed and versatility has a hidden cost. Over time, the overuse of these technologies has introduced fundamental flaws into database application architectures, flaws that are difficult to unravel once entrenched. While they deliver short-term wins by enabling rapid prototyping and deployment, they often compromise long-term architectural integrity, leading to significant challenges in scalability, performance stability, maintenance, and upgrades. Moreover, they frequently prevent developers from fully leveraging the specific features and strengths of target database systems, such as Firebird, ultimately undermining the potential for optimized, robust applications.

Generic Database Abstraction Layers

Different data stores and databases have a number of common features, which has led to the emergence of APIs that allow them to be accessed in a uniform way (e.g. ODBC, OLEDB, JDBC, Python DBAPI 2.0, .NET Data Provider, etc.). Drivers for individual databases must then fit key functionality into the framework of a given API, which leads to the first level of compromise.

Some APIs provide a certain degree of extensibility into which specific functionality of a given database can be squeezed, but the architecture on which the abstraction is based remains unchanged. If the specific functionality does not correspond to this architecture, it cannot be included in the driver. For example, the ability to run multiple independent transactions with different parameters within a single connection cannot be made available in many generic APIs.

Of course, there are Firebird libraries that don't use any generic APIs, or drivers that go beyond the APIs they implement. But these APIs are the foundation upon which other layers of "rapid application development" are built. These then add additional layers of abstraction, complexity, and tradeoffs.

Essentially all "successful" large applications carefully avoid generic APIs, and in case the application requires support for different database servers, they implement their own specific abstraction (e.g. using a three-tier architecture).

The Pitfalls of Object-Relational Mapping (ORM)

ORMs such as Entity Framework (for .NET), Hibernate (for Java), and SQLAlchemy (for Python) abstract the complexities of database operations, aiming to accelerate development and minimize errors. They map database tables to programming objects, enabling developers to work within familiar object-oriented paradigms. Yet, this abstraction often adopts a generic approach, designed to support multiple database systems, which can clash with Firebird's specific optimizations and features, leading to suboptimal application performance.

The biggest shortcoming of all ORMs is the way they manage connections, transactions, and queries. With some exceptions, they use a generic interface for working with the databases of the given environment, e.g. Python DBAPI 2.0 for SQLAlchemy, DbProvider for EF, or JDBC for Hibernate. It is therefore very difficult and often impossible to use the specific capabilities of Firebird. These systems also introduce their own conventions for a range of issues, from working with transactions to mapping between objects and databases, including migration between versions, to SQL generation, which introduce additional limitations and trade-offs.

In practice, ORM can be successfully used for smaller applications, but as complexity increases, the success rate decreases significantly. Using appropriate strategies can compensate for the problems of ORM to some extent.

- Choose an ORM with strong Firebird support. SQLAlchemy offers a Firebird dialect, and Entity Framework has providers for Firebird, though support for Firebird-specific features varies. ORMs allowing raw SQL execution or custom

- mappings (e.g., for stored procedures) are preferable.
- For performance-critical sections, consider a micro-ORM (e.g., Dapper for .NET) or direct SQL. Micro-ORMs provide lightweight mapping with greater control over SQL, balancing convenience and performance. Combining an ORM for general use with targeted SQL for high-performance areas can leverage both tools effectively.
- ORMs often favor an object-oriented approach, retrieving and manipulating individual objects (rows). This can lead to the "N+1 selects" problem, where multiple queries fetch related data (e.g., fetching an order, then querying each order item separately) instead of a single, optimized join. To mitigate this, developers can use ORM features like eager loading (e.g., Entity Framework's Include or SQLAlchemy's joinedload) to fetch related data in one query.
- ORMs typically manage transactions via sessions or units of work, but their default behavior—such as keeping transactions open across multiple operations—may not align with Firebird's needs. Developers must ensure transactions are committed or rolled back promptly, potentially adjusting the ORM's transaction scope or using explicit transaction control.
- Most ORMs, such as Entity Framework and SQLAlchemy, allow execution of raw SQL. Developers should use this feature for complex or performance-critical queries. For example, employ the ORM for simple CRUD (Create, Read, Update, Delete) operations where it excels, and switch to custom SQL for complex tasks. This balances development speed with performance.
- Use profiling tools to monitor database interactions, identifying slow or inefficient ORM-generated queries. Adjust ORM configurations or replace problematic queries with optimized SQL as needed.

The Limitations of Delphi's TDataSet

Delphi's `TDataSet` is a core abstraction that provides a uniform interface for interacting with various databases, making it a popular choice among Delphi/Lazarus developers. Its integration with Delphi's component-based development model allows for rapid application development, especially when using libraries such as **FireDAC** or **IBX for Lazarus**, which offer Firebird-specific optimizations.

The problem with libraries that work with the `TDataSet` abstraction is multi-faceted. First, the entire abstraction is just another generic API that is implemented in different ways, for example universally using drivers for individual databases, or specific implementations for a particular database. All of them suffer from the limitation of an architecture based on a file-based database abstraction like dBase or Paradox that cannot be satisfactorily mapped to SQL databases, while database-agnostic implementations additionally suffer from the use of a generic API for accessing the database.

One of the most significant problems with TDataSet is that it encourages row-by-row processing instead of set-based operations. This approach, while intuitive for developers accustomed to local databases, is highly inefficient in a client-server environment. Instead of looping through rows in code, developers should structure their queries to process data in bulk whenever possible, reducing network overhead and improving overall efficiency.

Another inefficiency arises from excessive automatic fetching of data. Firebird is most efficient when queries retrieve only the necessary records, keeping network traffic and server load to a minimum. However, TDataSet components often fetch more data than required, sometimes pulling entire tables when only a few records are needed. This problem is exacerbated when using UI elements like data-aware grids, which may automatically request all available rows for display, even if the user only needs a subset. While `TQuery` offers more flexibility than `TTable`, developers must still be mindful of how their queries are executed. The best practice is to design queries to return only the data that is immediately needed, using LIMIT clauses, indexed searches, and careful selection of fields to avoid unnecessary overhead.

Beyond performance concerns, `TDataSet` introduces maintainability challenges that become significant as applications grow. One of the biggest drawbacks of Delphi's component-based database design is that query definitions, parameters, and transaction settings are often stored in component properties rather than centralized in source code. While this makes development faster in the early stages, it can create a fragmented and difficult-to-maintain system as complexity increases. Developers often struggle to track which queries are executed where, as important database logic is scattered across multiple forms and data modules instead of being managed in a structured and version-controlled source file.

The biggest problem, however, is the fact that `TDataSet` is the cornerstone of the implementation of the Model-View-Controller (MVC) concept, and to which the data-aware component system is linked. Developers are therefore faced with a difficult choice: either adapt to the limitations of this abstraction system or create their own MVC system from scratch with better features. For quite understandable reasons, most take the path of least resistance.

Other general libraries and frameworks

The relational model and SQL follow a structured approach with recurring patterns, making them ideal for automation. Many solutions have emerged to streamline these repetitive tasks and save development time. However, they all share a fundamental flaw — the attempt to force different databases into a single abstraction.

The quality and features of each solution depend on the resources invested in its development, which in turn rely on its popularity and user base. As a result, most solutions support multiple database systems — bringing with them all the drawbacks of a one-size-fits-all approach.

These technologies have undeniably simplified and accelerated the development of simple and moderately complex applications. However, their benefits do not scale — beyond a certain level of schema complexity, data volume, or concurrent users, achieving good performance becomes increasingly difficult and costly, or even impossible.

Successful complex applications share a common trait: they either rely on custom technologies for rapid development or adopt a hybrid approach, using generic solutions sparingly within a tailored architecture.

Artificial Bottlenecks

Even applications that successfully avoid the trap of misaligned abstractions can become "cursed" if they contain artificial bottlenecks. These are often introduced unintentionally, either by enforcing unnecessary synchronization, designing workflows that concentrate too much load on specific points, or failing to plan for scalability. These bottlenecks don’t always show up immediately, but as the number of users grows or the dataset increases, they become major performance constraints. The worst part? These problems are entirely avoidable with a better understanding of Firebird’s concurrency model and proper architectural planning.

One of the most common artificial bottlenecks occurs at application startup. Many applications perform expensive database operations when they initialize, such as loading large amounts of data into memory, checking configuration tables, or executing extensive background queries to "warm up" the system. When a handful of users launch the application, this may not be noticeable. But when dozens or hundreds of users start the system at the same time—especially at the beginning of a workday—Firebird can suddenly be overwhelmed with simultaneous, expensive queries, all competing for resources.

A surprisingly common phenomenon is the erroneous work with updating or deleting data. Statistically, in database applications, the most common operation with data is inserting new records (about 75%), followed by updating records (about 20%), and the rarest operation is deleting records (about 5%). While inserting data has practically no synchronization requirements and the throughput of parallel operations is very high, especially in Firebird, the same cannot be said for updating or deleting data. If we ignore the possibility of collision between competing transactions when simultaneously updating the same rows, updating a row (especially repeated ones) creates a problem with the chain of row versions and the removal of unnecessary versions. An often overlooked factor of data changes is that every update or deletion triggers multiple index updates — first during the modification itself and again when outdated row versions are removed — further impacting performance.

When designing an application, it is essential that all updates or deletions of data are well justified and documented, as these are natural bottlenecks in the database. Many applications perform unnecessarily frequent updates, mostly of operational data and metadata (parameters for operations, amounts, etc.), or mass updates at inappropriate times. The biggest sin is combining insert and update operations (e.g. updating the amount in another row).

Conclusion

Applications that struggle with Firebird often do so not because of external factors but because of fundamental design choices made early in development. Many of these issues are avoidable, yet they persist due to reliance on generic abstractions, misguided optimization strategies, and development practices that fail to consider Firebird's unique characteristics.

The difference between a problematic application and a well-functioning one is a matter of approach. Understanding Firebird's strengths and designing with them in mind leads to systems that perform efficiently and scale effectively. Ignoring them results in constant frustration, poor maintainability, and escalating costs.

Firebird is not a database that punishes developers — it rewards those who take the time to understand it. Applications built with this mindset are not only free from unnecessary struggles but can fully leverage Firebird's capabilities to deliver robust, scalable, and high-performing solutions.

Interview with Jim Starkey

Few figures in database history have left as deep an imprint as Jim Starkey. Firebird users know him as the creator of InterBase, the system from which Firebird emerged after its open-sourcing in 2000. But InterBase is just one chapter in a career defined by groundbreaking ideas. Starkey pioneered multi-version concurrency control (MVCC), built database engines that redefined industry norms, and continuously challenged conventional wisdom in database design.

In this rare interview, we step beyond InterBase to explore his entire journey—from his early innovations to later projects like Netfrastructure, Falcon, and NuoDB, all the way to his most recent work. What drove his thinking? What lessons has he learned? And what does he see as the next frontier in database technology? Starkey's insights are always sharp, often provocative, and never dull—so read on.

You’ve been working with databases since the late 1960s, starting with Model 204 and the Datacomputer before moving on to projects like DBMS-11, Datatrieve, and Rdb/ELN. Looking back, what were the biggest challenges and insights from those early projects? Did any particular experiences from that era influence your later work? How do you compare the approaches and ideas from those early systems with what came later?

To be fair, originally, I was more concerned with eking out a living than re-inventing database systems. But since you asked, I liked the Model 204 data model (a record was an abstract collection of attribute/value pairs) but the access language left me cold. On the Datacomputer, I liked the access language but hated the hierarchical data model. But what I really wanted to do was write a relational database system, but CCA would hear none of that (too academic to be commercially viable).

With regard to DBMS-11 (mainframe IDMS ported to the PDP-11), it was hard to say whether I hated the data model or access language more, but I did lift the page structure that I later used for quite a few systems.

Datatrieve-11 reprised the Datalanguage from the Datacomputer. The big challenge was making a large 4th generation language fit on a 16 bit computer with one developer (me) and a year to do it in. VAX Datatrieve had a much larger team but also had to support cross node access, business graphics, VAX DBMS (another hyper-ugly CODASYL database) access, relational access, and a rich API. But to get it done, I had to spin off the Common Data Dictionary and the relational database projects.

The biggest frustration with VAX Datatrieve was the VAX DBMS two phase record locking transaction model that made it next to unusable for interactive users. That led to the epiphany that transactions could be managed by keeping multiple record versions and doing to book-keeping for which transaction should see which version; thus the first JRD (later Rdb/ELN) and MVCC were born. The DEC database guys, of course, hated it, and a database war ensued. The big lesson was that you can’t teach database engineers new tricks, and if you want to explore new technology, starting your own company is usually the only solution (lather, rinse, and repeat). MVCC, of course, did change the world but it did take 30 years.

Unlike your earlier work at DEC, InterBase was fully your own creation—you had complete control over its design and direction. But after a few years, you sold Groton Database Systems and InterBase to Ashton-Tate. What drove that decision? Were you dissatisfied with where InterBase was heading, looking for a break, or eager to explore new database ideas that couldn't be realized within InterBase? Were there plans or features you had envisioned for InterBase that were left unfinished due to the sale?

There comes a point in many new starts where the technology is more valuable to established companies with mature marketing and sales organizations. And, while profitable on paper, we were chronically broke. So, when approached for acquisition by two different companies, the question became which would be the better partner. In fact, however, the deal with Ashton-Tate was a 7-year staged acquisition, so we had plenty of time to work on bells and whistles.

When InterBase was open-sourced, it led to the creation of the Firebird project, which has now been actively developed for 25 years. How do you view Firebird's stewardship of your InterBase legacy? Are you satisfied with how the project has evolved, and is there anything you would have done differently? Do you have any suggestions for Firebird developers as they continue to advance the database?

Mixed feelings. I would have gone for simplicity, fewer esoteric options, more backwards compatibility, an internal SQL engine, and a better SQL API. My rather jaundiced view of current Firebird development is that it is mostly putting warts on warts rather than moving the project architecturally forward.

I have to say that Firebird developers, like MySQL developers, have an intense gut-based aversion to new ideas. New ideas shouldn't be rejected out of hand. New ideas are never perfect but should be seen as a starting point. Going down rat-holes is part of the job.

The Vulcan project was developed with the goal of introducing symmetric multiprocessing (SMP) to Firebird, driven primarily by SAS's interests. Beyond that core objective, were there other requirements from SAS, or improvements and innovations you introduced independently? How do you view the way

Vulcan was later integrated into the Firebird codebase?

The primary goals of Vulcan were to make it 64 bit and port it to Sun OS. A lot of code came over from Netfrastructure, which was SMP-based, so that was mostly a freebie.

I wasn't particularly happy that a number of key Vulcan features such as the provider architecture and the multi-tiered configuration system were originally rejected out of hand. I was amused that much (maybe all) of Vulcan osmosed into Firebird over decade.

With Netfrastructure, you explored integrating databases with application logic in new ways. What was the core idea behind the project, and what were the biggest technical or conceptual hurdles?

The big ideas behind Netfrastructure were a) a SQL engine operating off an in-memory database using the disk for backfill, b) JDBC as the sole API, c) text search, d) an integrated internal Java Virtual Machine, and e) an open ended, multi-tenant architecture for layered apps.

The big problem with Netfrastructure was timing. It came out when everyone thought that all Internet apps should be free and the VCs were obsessed with dot Coms that could be flipped in six months. Just a little bit later, the same VCs were obsessed dealing with a couple of bankruptcies a week.

Netfrastructure remained relatively unknown—was it ever marketed or used beyond a few select cases?

It was superbly successful in a small number of very challenging applications.

Perhaps you may have noticed that marketing has never been one of my long suits.

In 2006, you sold the Netfrastructure web software business to MySQL AB and joined the company to work on Falcon. How much of the Netfrastructure database architecture made its way into Falcon?

All of it. The biggest extension was a bomb-proof serial log for transaction recovery under any circumstance.

Falcon was expected to be a major leap forward for MySQL, and many believed it could redefine MySQL's capabilities. But the project ultimately collapsed after MySQL AB was acquired by Sun, and then Sun was acquired by Oracle. From your perspective, what actually happened? Were the challenges primarily technical, organizational, or something else entirely? Were there fundamental differences in opinion that made development difficult? And were any of Falcon's innovations or design ideas later applied elsewhere?

MySQL was preparing to go public when Oracle bought InnoDB, MySQL's only transactional storage engine out from underneath them. Falcon was Plan B in case Oracle cancelled MySQL's right to ship InnoDB. Oracle didn't, so it came down to a horse race between Falcon and InnoDB performance. Each approximately doubled in speed (much like the competition between JRD and Rdb/VMS at DEC). The big frustration was when Google put the Falcon user mode read/write locks into InnoDB. But then Oracle bought Sun and the horse race was called off.

NuoDB (originally NimbusDB) introduced a fundamentally different approach to database architecture, emphasizing elasticity and cloud scalability. What led you to pursue distributed databases, and what were the most difficult engineering challenges you faced?

I was frustrated by what performance could be achieved on SMP systems and the problems with shared-nothing distributed schemes. I went into a shower on a Sunday morning thinking about how to bring a new node into a running distributed database and came out about an hour later with the more or less complete NuoDB architecture in my head. (Should I say that we designed our house around the need for very long architectural showers?)

Like any new technology, there were at least a dozen intractable problems. What I've learned to do is to repress the panic, take a long look out to sea, find an existence proof that a solution was possible, they wait to get smarter. Hasn't failed yet. A good example of an apparently intractable problem is that when a node re-enters a running cluster, how does it know what part of the database stored locally is current and correct and what parts need to be fetched from other nodes.

While NuoDB has been recognized for its innovations, it has remained a niche product rather than achieving widespread adoption. Why do you think that is? Were there technical limitations, marketing or leadership missteps, or simply less demand than anticipated for databases with these characteristics?

Due to a non-disparagement agreement, I'm afraid that I can't comment.

I have to ask about your last project—AmorphousDB. I first heard about it in your 2016 interview for ODBMS Industry Watch, where you described it as a radical departure from traditional database architecture—potentially 'the last database system ever needed.' It was an intriguing read, but since then, there have been no public updates. Did the project progress beyond the conceptual stage, and what ultimately happened to it?

When I started what became Interbase, it was still possible to boot-strap a company out of revenues. That's no longer the case, and AI has sucked all the oxygen out of the investment and partnership rooms. The technology is fabulous, but nobody gives a hoot about database systems now. In short, no traction. Still working on it, but at the moment, no end game in sight.

Having worked on a wide range of database systems over the years, what are the biggest lessons you've learned—whether they're technical, business-related, or about the database industry as a whole? Looking back, what key principles or insights do you think every database engineer should understand to succeed in this field?

I think there are three essential principles. First, architecture is the art of making all of the pieces fit together. Second, never fall in love with your first idea. Finally, anyone working on a hard problem who isn't a genius or a fool is going to gain significant insight into the large problem. When you get smarter, go back revise what you've done.

Thanks for you time!

Excerpt from ODBMS Industry Watch Interview

What are the most challenging issues in databases right now?

I think it's time to step back and reexamine the assumptions that have accreted around database technology – data model, API, access language, data semantics, and implementation architectures. The "relational model", for example, is based on what Codd called relations and we call tables, but otherwise have nothing to do with his mathematic model. That model, based on set theory, requires automatic duplicate elimination. To the best of my knowledge, nobody ever implemented Codd's model, but we still have tables which bear a scary resemblance to decks of punch cards. Are they necessary? Or do they just get in the way?

Isn't it ironic that in 2016 a non-skilled user can find a web page from Google's untold petabytes of data in millisecond time, but a highly trained SQL expert can't do the same thing in a relational database one billionth the size?. SQL has no provision for flexible text search, no provision for multi-column, multi-table search, and no mechanics in the APIs to handle the results if it could do them. And this is just one a dozen problems that SQL databases can't handle. It was a really good technical fit for computers, memory, and disks of the 1980's, but is it right answer now?

Let me say a few things about my current project, **AmorphousDB**, an implementation of the Amorphous Data Model (meaning, no data model at all). AmorphousDB is my modest effort to question everything database.

The best way to think about Amorphous is to envision a relational database and mentally erase the boxes around the tables so all records free float in the same space – including data and metadata. Then, if you're uncomfortable, add back a "record type" attribute and associated syntactic sugar, so table-type semantics are available, but optional. Then abandon punch card data semantics and view all data as abstract and subject to search. Eliminate the fourteen different types of numbers and strings, leaving simply numbers and strings, but add useful types like URL's, email addresses, and money. Index everything unless told not to. Finally, imagine an API that fits on a single sheet of paper (OK, 9 point font, both sides) and an implementation that can span hundreds of nodes. That's AmorphousDB.

Development update: 2025/Q1

A regular overview of new developments and releases in Firebird Project

Releases:

- Firebird 5.0.2, released 12.2.2025
- firebird-driver for Python 1.10.9, released 3.1.2025
- Jaybird 6.0.0, released 27.12.2024

Official Firebird Docker Images

We are pleased to announce the successful migration of Firebird Docker images to their new repository. The images are now maintained on GitHub and published on Docker Hub, ensuring better accessibility and continued support for the Firebird community. This transition would not have been possible without the pioneering efforts of F.D. Castel, who laid the groundwork, and the invaluable contributions of Adriano dos Santos Fernandes, who played a key role in improving and streamlining the migration process.

News from Firebird Release Planing Meeting

Last year, the Firebird Foundation and Firebird Admins introduced bi-annual Firebird Release Planning Meetings (RPM) with Firebird Partners and Sponsors. These meetings, a revival of the former Technical Task Group (TTG) gatherings, take place every March and September, starting on the second Monday, in a closed group setting.

We aim to keep you updated on the outcomes, including key achievements from the past six months and plans for development and releases in the next six months.

Summary of last six months

Firebird Engine

- Firebird 5.0.2 was released on 12-Feb-2025, containing 11 improvements (with blob data prefetch being a major one) and around 50 bugfixes.
- Adriano dos Santos Fernandes has mostly completed the SQL schema implementation and published it for review in February 2025, with many discussions happened along the road.
- Alex Peshkov continued his work on the shared metadata cache.
- Vlad Khorsun has implemented a number of blob fetch improvements in the remote protocol that were already evaluated and even backported into v5.0.2.
- A number of performance improvements regarding NULL handing in index scans have been implemented. ODS 14 was changed in some aspects (header page and index root page layout), BLOBs over 4GB are now properly supported.
- Other features mentioned in the roadmap:
- Support for tablespaces has been published for review in November 2024, post-fixes are in progress.
- ROW data type feature has been discussed and now in the final adjustments stage.
- JSON support is being discussed (2 parts of the overall proposal were presented, 3rd is coming soon).
- Implementation of the SQL:MED standard (declarative foreign data sources and cross-database queries) has been presented for discussion.

- UNLIST table-valued function implementation has been published for review, now fixing some issues that were reported.

.NET Provider

- Completed support for EF Core 9.
- Releases: EF provider - 10.1.0.0, FirebirdClient 10.3.2.0, EF Core - 12.0.0.0

Jaybird

- Major release: Jaybird 6.0.0 (27-Dec-2024)
- Bugfix release: Jaybird 5.0.6 (16-Oct-2024)

Up to the end of December, most of our work has been on testing Jaybird 6 and writing and updating the release notes and other documentation. Some minor features were added, including support for a configurable socket factory and revising the handling of holdable forward-only result sets.

We started work on Jaybird 7. Copyright information in Jaybird was updated to use SPDX, so that Jaybird users that need it can generate SBOM information.

Python drivers and libraries

- Two `firebird-driver` releases with small bug fixes.
- Work started on revitalization of full Firebird Python stack, starting with `firebird-base` package that's a core dependency (almost finished).

Firebird QA

- Implemented / adjusted / refactored over 100 QA tests.
- QA-scenario: implemented code for show lines from firebird.log that match to selected failed test.
- Changed python script that generates code for QA report so that this code will pass validation of W3C and JS.
- Search the solution of annoying problem with pytest when it sporadically hangs on exit (no return to caller batch).
- Refactored aux report about missed QA runs. Added info about increased values of "F" and "E" counters.
- Problem on Linux host that run QA script using cron: time 'shifts' relatively root settings.

- Completed investigation that relates to performance of INDEX RANGE SCAN in case of NARROW character set, for misc charsets and collations.
- Refactored QA scenario (currently for Windows only) in order to reduce overall time of QA pass for all FB families.
- Valuable performance regression in all tests that use ES after #658abd2. Found thanks to the moving medians report (implemented in sep-2024).
- Refactored QA-report: implemented ability to see full test history for every test, not only for those which have "F" or "E" outcome for last 30 QA runs.
- firebird-driver problem: can not make connection to Services API using NON-ascii user/password/role.
- firebird-driver FR: need ability to get BLOB_ID after blob has been created / changed. QA-problem: pytest: sporadic "Exception ignored in atexit callback: <function _api_shutdown at …hex-addr…>".

Several tests (related tickets in GH tracker) could not be implemented because of lack additional info. They now have status = 'deferred'.

Firebird Documentation

- Various minor improvements to the Language References
- We added links in the old Language Reference Updates to point readers to the Firebird 5.0 Language Reference for more complete/up-to-date information
- Updated and improved various redirects of old links or versions of documentation to the new version
- Work started on reorganizing/improving documentation URLs
- Work on new Firebird Configuration Reference

Firebird Butler & Saturnin

There was no progress on Firebird Butler specifications or Saturnin over last six months.

However, we want to resume work on Saturnin after revitalizing its dependencies and parts of the project itself (see "Firebird drivers" for details).

Plans for March - August 2025

Firebird Engine

- v4.0.6 release was planned to be published shortly after v5.0.2 but still in the pipeline, to be released later in March.
- v3.0.13 release is currently scheduled for Q3 2025.
- Shared metadata cache is expected to complete in May-June.

The Firebird 6.0 roadmap was updated to reflect the current status. We expect that ODS will be finalized during the next 6 months. We'll review and merge the already proposed features and publish the Alpha release in Q3 this year.

.NET Provider

One major plan for 2025 is dropping support for .NET Framework. It's been a decade since .NET (Core) happened and keeping old TFM takes precious resources. The last supported version will be kept in a separate branch. Possibly for security fixes or data corruption issues, through donation to FFzs.

Jaybird

- Likely (depending on content): bugfix releases Jaybird 5.0.7 and 6.0.1
- Implementing support for protocol 19 (Firebird 5.0.3) inline blobs, and backporting it to Jaybird 5 and 6
- Revise FBManager API to allow (amongst others) to set the authentication plugins
- Add support for CALL statement and named procedure/function parameters
- Reimplement and improve CallableStatement (long-term work)

Python drivers and libraries

The Firebird Project currently maintains 9 key Python projects (+ two legacy ones), which support Python 3.8 that is no longer maintained. It was decided to move the full stack forward to Python 3.11 as oldest supported Python version. This will also allow us to starting using new Python features (for example Structural Pattern Matching).

During the upgrade, we also want to clear as much technological debt as possible, introduce improvements, optimizations and improve test coverage. We'll also rewrite tests for these packages from UnitTest to pytest.

- Packages `firebird-base`, `firebird-driver`, `firebird-lib` and `firebird-butler-protobuf` will be moved to new major version: 2.0.0.
- Package `firebird-qa` will be finally promoted to 1.0.0 (currently 0.20.0).
- The `firebird-uuid` package is still in a beta stage (0.3.0), and will likely undergo a significant rewrite, hence it will retain it's 0.x versioning
- Also Saturnin packages `saturnin`, `satirnin-sdk` and `saturnin-core` are still in beta, so they will be bumped to 0.10.0 from current version 0.9.0.

We also consider to make a maintenance release for legacy `fdb` driver with some fixes, mostly related to compatibility with Python 3.13. However, the final decision has not yet been made.

Firebird QA

We are planning to move the entire QA framework to a dedicated machine, separate from IBSurgeon's daily test server, which is shared within our company. This will allow us to consolidate two key scheduled tasks on a single system:

- Running QA scenarios at night.
- Running OLTP-EMUL on Firebird 4.x to 6.x and uploading reports to firebirdtest.com, reviving a process that was discontinued years ago.

Next, we aim to improve reporting by generating a single HTML page that consolidates QA results for both Windows and Linux. Currently, reports are uploaded separately by OS, requiring manual comparison when failures occur—especially in Firebird 6.x, where results must be checked on two different pages.

Additionally, we are considering a full re-implementation of the server-side QA system. The current setup relies on static pre-generated HTML pages with complex JavaScript, which is becoming unsustainable due to the growing size of test history data. Instead, we want to move toward dynamically generated pages, at least for test history, ensuring reports are created only when requested by users.

Firebird Documentation

- Finish reorganizing/improving documentation URLs
- Continue work on Firebird Configuration Reference (long-term work)
- Start Firebird 6.0 Language Reference (long-term work)

BLOB revolution

In the upcoming version of Firebird 5.0.3, there will be significant improvements to BLOB data transfer across high-latency networks like the Internet: for example, transferring 8191 bytes as BLOB will be 3 times faster than using VARCHAR! This optimization dramatically improves BLOB transfer speeds, bringing them on par with VARCHAR data. However, to leverage this enhancement, both server and client components must be updated to the newest version. The performance breakthrough enables efficient cloud hosting of Firebird databases while maintaining full compatibility with traditional client applications. This advancement opens exciting new possibilities for cloud-based Firebird deployments, making remote database access substantially more practical for real-world applications.

Toolbox: Database Workbench Lite

The right database tools can make all the difference. For Firebird developers and database administrators, **Database Workbench** by **Upscene Productions** has been a trusted choice for years. Originally launched in the early 2000s with a focus on InterBase and Firebird, it quickly evolved into a multi-engine database development environment, supporting MySQL, Oracle, and more. Packed with features for database design, SQL scripting, and performance tuning, it has become a comprehensive suite for professionals.

But not everyone needs—or can afford—the full version. That's where **Database Workbench Lite for Firebird** comes in. This free edition, tailored specifically for Firebird, provides essential database management tools without the cost. But does it offer enough functionality, or are its limitations too restrictive? In this review, we'll explore what Database Workbench Lite delivers, where it excels, and where it falls short.

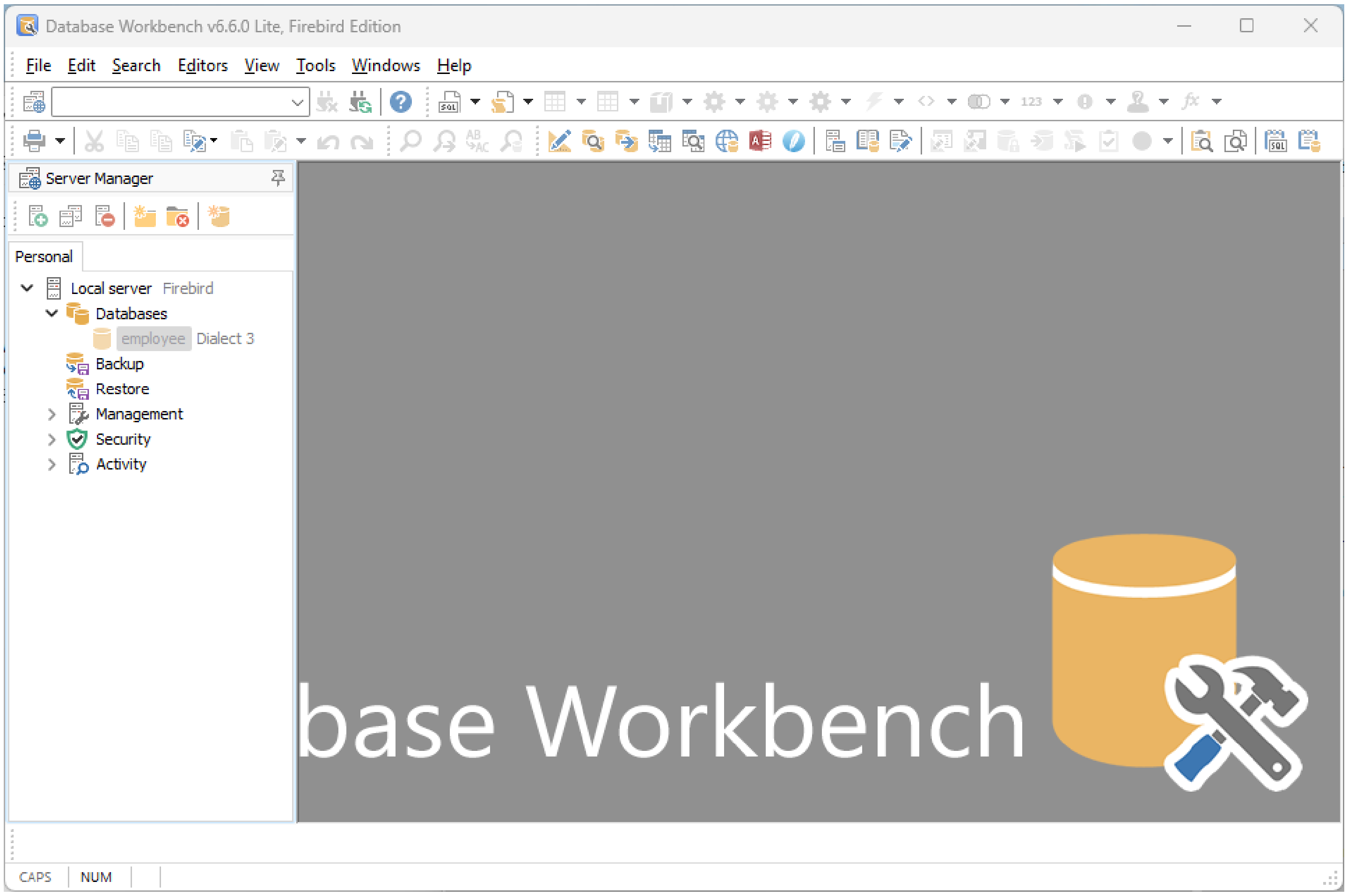
For this review, we used version 6.6.0, released on February 11, 2025.

One key point upfront: **Database Workbench Lite** is strictly for personal, non-commercial use and works only with Firebird. For commercial use, a paid license is required, available in **Basic**, **Professional**, and **Enterprise** tiers, each offering different levels of functionality. The single license includes support for one database type, with additional modules available for purchase.

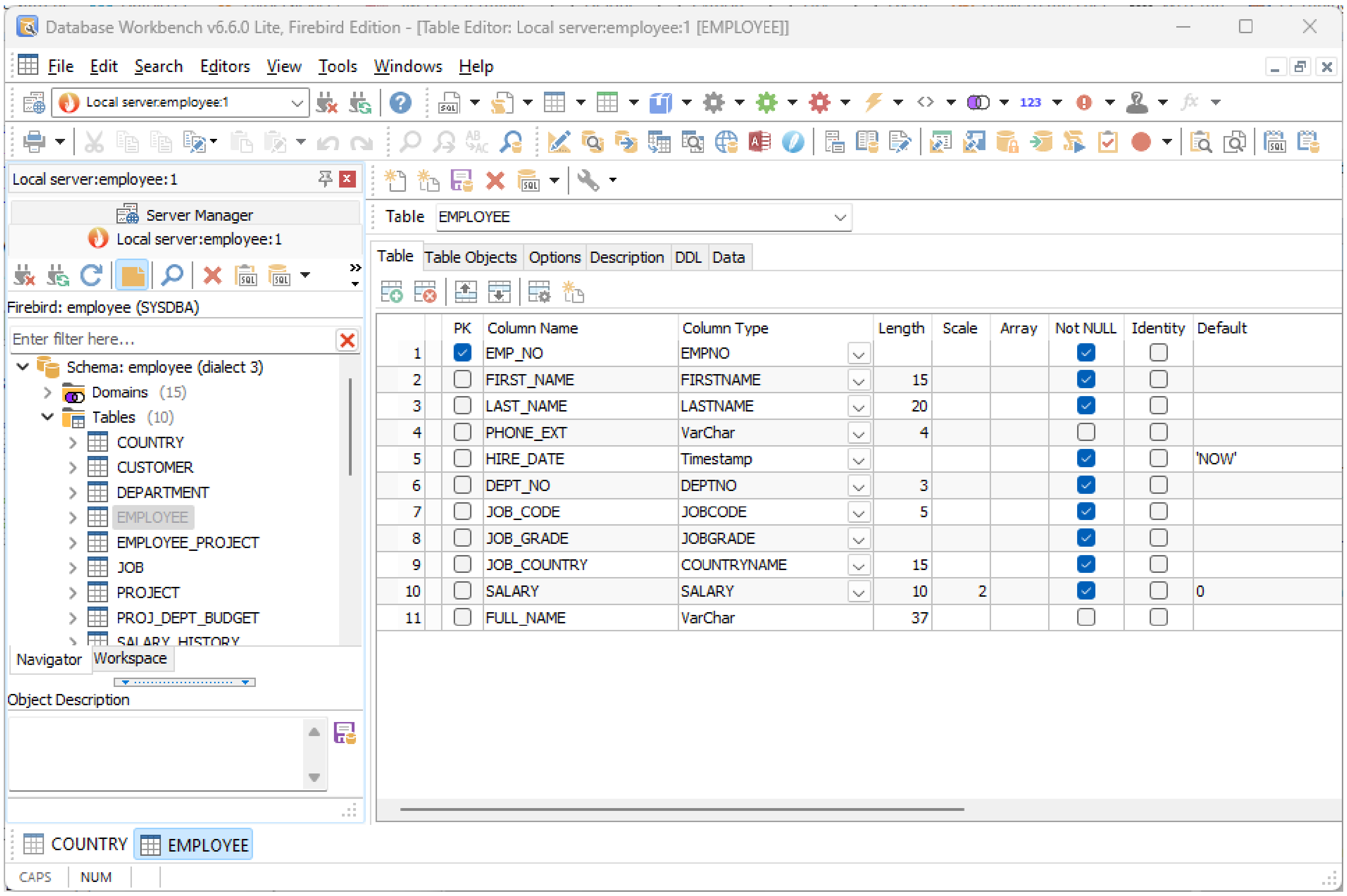
The Lite version also comes with several limitations. Beyond reduced functionality, it allows registration of only two servers and two databases per server.

Database Workbench is a 64-bit Windows application written in Delphi, featuring a standard, single-window interface with toolbars and a side panel. A larger screen is recommended for comfortable use. For this reason, the screenshots in this review are mostly just cutouts of the relevant part of the screen.

While the Lite version lacks most tools from the commercial editions, they remain in the menus and toolbars. Clicking on them triggers a modal window stating the feature isn't available. Hence it is a good idea to disable unnecessary toolbars to reduce clutter and save screen space.



The panel docked at the left edge has a dual function. First, it contains the Server Manager for managing registered servers and databases, and then it contains the Managers of open database connections. Both have the form of a standard treeview, displaying objects according to context. Activating individual objects opens the corresponding editor, and the context menu offers additional available actions.



Database Workbench v6.6.0 Lite, Firebird Edition - [Table Editor: Local server:employee:1 [EMPLOYEE]]

Local server:employee:1

Firebird: employee (SYSDBA)

Enter filter here...

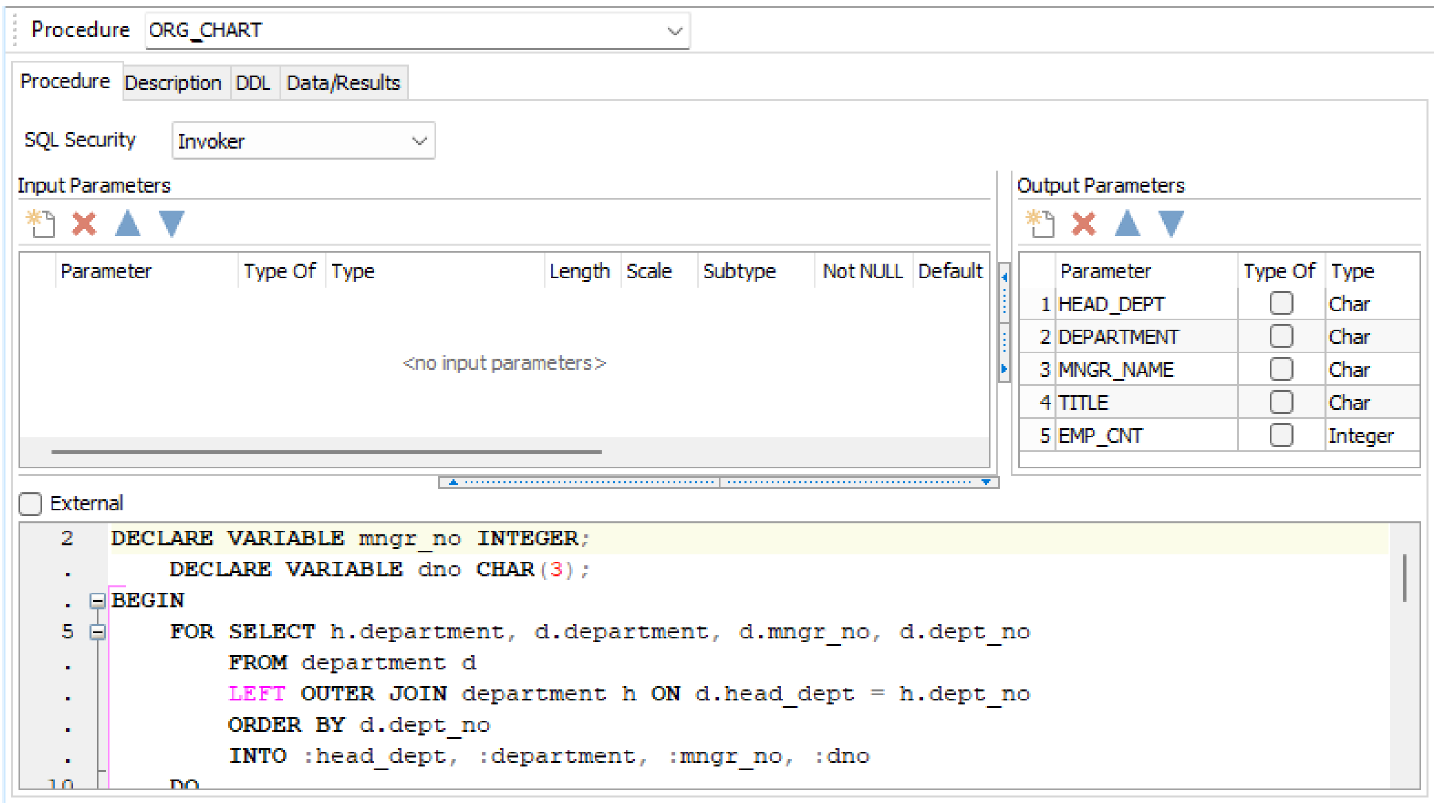
- Schema: employee (dialect 3)
  - Domains (15)
  - Tables (10)
    - COUNTRY
    - CUSTOMER
    - DEPARTMENT
    - EMPLOYEE
    - EMPLOYEE_PROJECT
    - JOB
    - PROJECT
    - PROJ_DEPT_BUDGET
    - SALARY_HISTORY

Table: EMPLOYEE

	PK	Column Name	Column Type	Length	Scale	Array	Not NULL	Identity	Default
1	☑	EMP_NO	EMPNO				☑	☐	
2	☐	FIRST_NAME	FIRSTNAME	15			☑	☐	
3	☐	LAST_NAME	LASTNAME	20			☑	☐	
4	☐	PHONE_EXT	VarChar	4			☐	☐	
5	☐	HIRE_DATE	Timestamp				☑	☐	'NOW'
6	☐	DEPT_NO	DEPTNO	3			☑	☐	
7	☐	JOB_CODE	JOBCODE	5			☑	☐	
8	☐	JOB_GRADE	JOBGRADE				☑	☐	
9	☐	JOB_COUNTRY	COUNTRYNAME	15			☑	☐	
10	☐	SALARY	SALARY	10	2		☑	☐	0
11	☐	FULL_NAME	VarChar	37			☐	☐	

The database object editors offer all the essential functionality expected from a tool of this type, with a few useful enhancements. Notably, the stored procedure editor allows for direct procedure testing.

What is particularly pleasing is the (P)SQL editor, which, in addition to object name completion and syntax highlighting, also offer structural folding (the display of the contents of a collapsed block in a tooltip is particularly elegant) and code templates. The context menu then offers other functions, including specific ones such as inserting stored procedure parameters in the appropriate format.



SQL Security: Invoker

Input Parameters

Parameter	Type Of	Type	Length	Scale	Subtype	Not NULL	Default

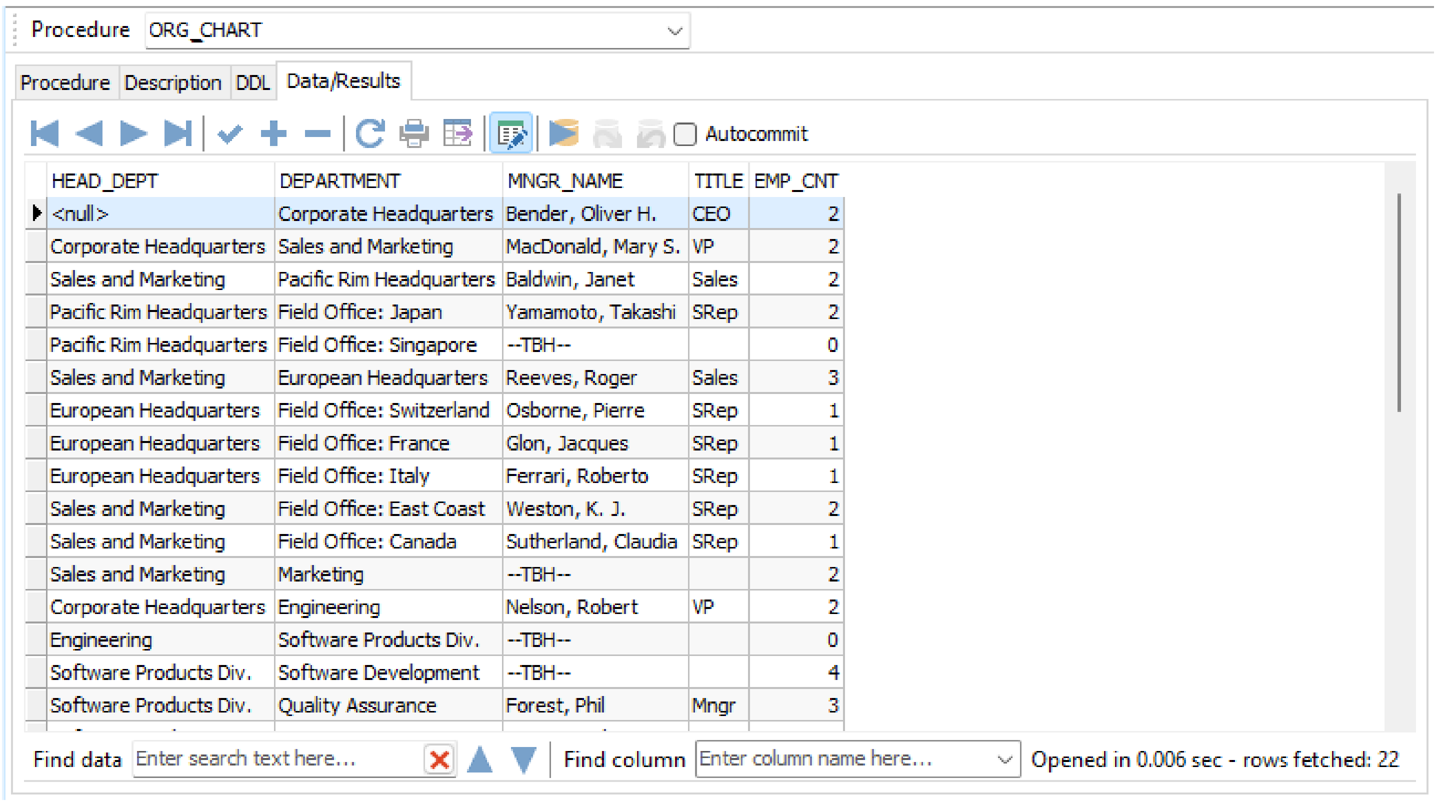
<no input parameters>

Output Parameters

	Parameter	Type Of	Type
1	HEAD_DEPT	☐	Char
2	DEPARTMENT	☐	Char
3	MNGR_NAME	☐	Char
4	TITLE	☐	Char
5	EMP_CNT	☐	Integer

☐ External

```
DECLARE VARIABLE mngr_no INTEGER;
    DECLARE VARIABLE dno CHAR(3);
BEGIN
    FOR SELECT h.department, d.department, d.mngr_no, d.dept_no
        FROM department d
        LEFT OUTER JOIN department h ON d.head_dept = h.dept_no
        ORDER BY d.dept_no
        INTO :head_dept, :department, :mngr_no, :dno
    DO
```



HEAD_DEPT	DEPARTMENT	MNGR_NAME	TITLE	EMP_CNT
<null>	Corporate Headquarters	Bender, Oliver H.	CEO	2
Corporate Headquarters	Sales and Marketing	MacDonald, Mary S.	VP	2
Sales and Marketing	Pacific Rim Headquarters	Baldwin, Janet	Sales	2
Pacific Rim Headquarters	Field Office: Japan	Yamamoto, Takashi	SRep	2
Pacific Rim Headquarters	Field Office: Singapore	--TBH--		0
Sales and Marketing	European Headquarters	Reeves, Roger	Sales	3
European Headquarters	Field Office: Switzerland	Osborne, Pierre	SRep	1
European Headquarters	Field Office: France	Glon, Jacques	SRep	1
European Headquarters	Field Office: Italy	Ferrari, Roberto	SRep	1
Sales and Marketing	Field Office: East Coast	Weston, K. J.	SRep	2
Sales and Marketing	Field Office: Canada	Sutherland, Claudia	SRep	1
Sales and Marketing	Marketing	--TBH--		2
Corporate Headquarters	Engineering	Nelson, Robert	VP	2
Engineering	Software Products Div.	--TBH--		0
Software Products Div.	Software Development	--TBH--		4
Software Products Div.	Quality Assurance	Forest, Phil	Mngr	3

Find data: Enter search text here... Find column: Enter column name here... Opened in 0.006 sec - rows fetched: 22

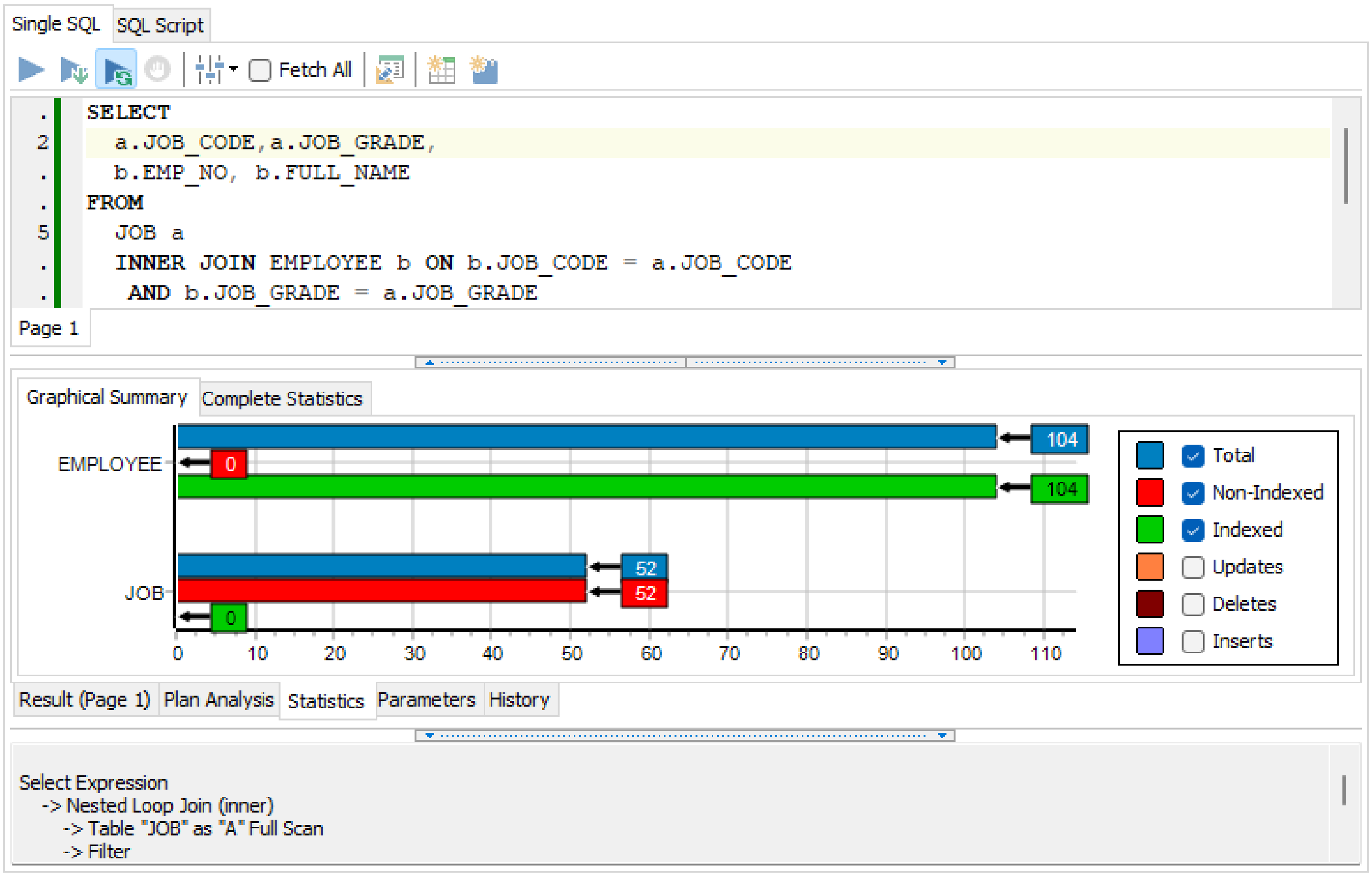
The SQL Editor has two separate panels: one for executing individual commands and another for scripts. However, you can edit multiple statements in the same window and execute a selected one. Depending on the Empty Line is Statement Separator setting, you can also separate statements with empty lines and run a specific one by placing the cursor inside it.

The editor is multi-threaded, allowing you to execute a statement while continuing to work in Database Workbench. Each editor instance opens a new database connection, but this can be disabled.

Tabs below the editor window let you manage multiple SQL statements, which can be saved and reloaded. You can add, delete, or rename tabs via toolbar buttons or the right-click menu. Each tab has its own result set, but only one is visible at a time. Dragging a file from Windows Explorer into the SQL Editor opens it in a new tab.

When an SQL statement contains parameters, you must first click the Prepare button to enter values for them. However, instead of switching to the Parameters tab, it incorrectly defaults to the Plan Analysis tab, which isn't available in the Lite version. On the plus side, the parameter editor maintains a history of previously entered values for convenience.

Although the Lite version lacks Plan Analysis, it still displays the execution plan in an explained format. It also provides execution statistics in both graphical and tabular formats.



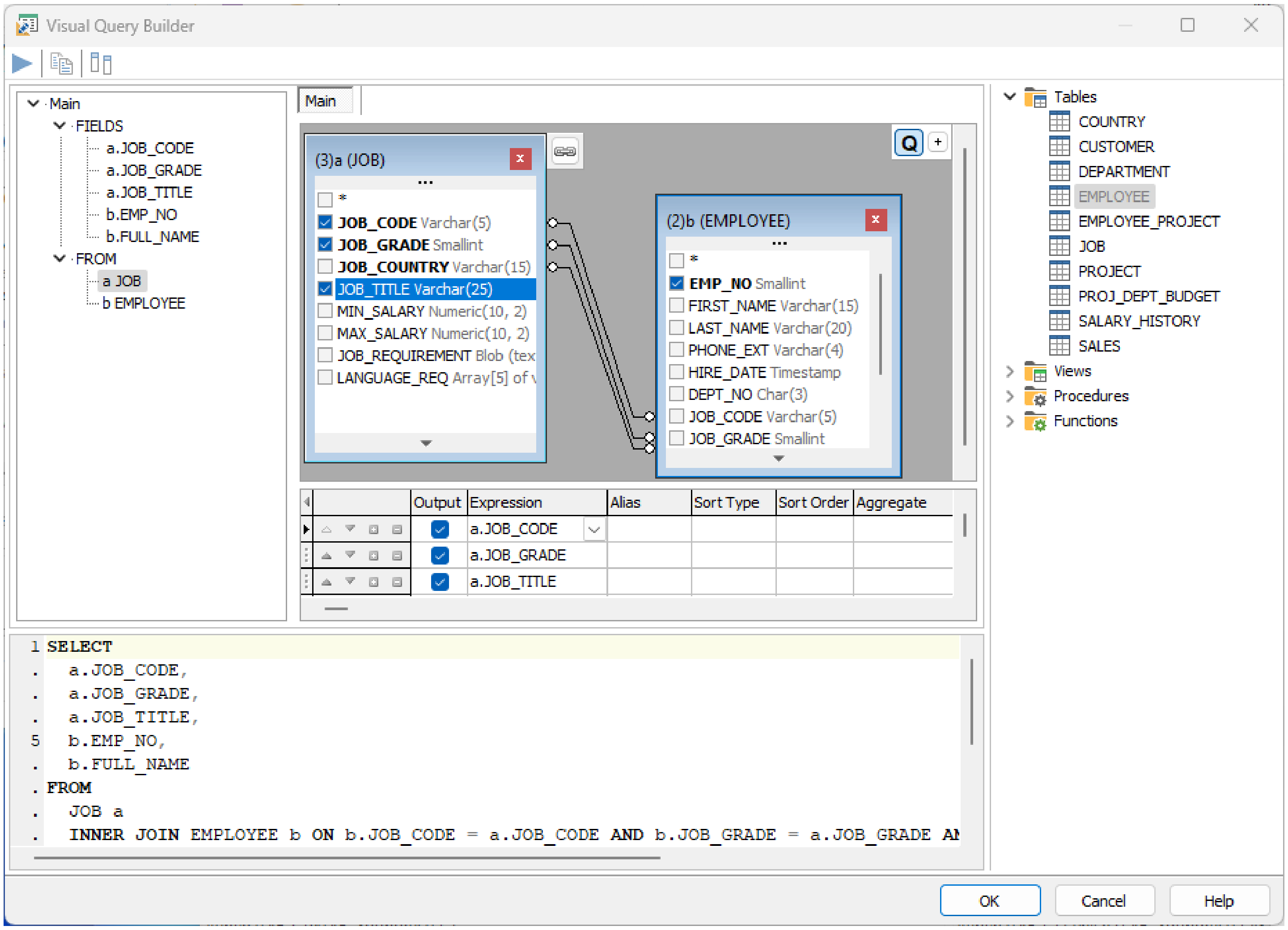
```
SELECT
  a.JOB_CODE,a.JOB_GRADE,
  b.EMP_NO, b.FULL_NAME
FROM
  JOB a
  INNER JOIN EMPLOYEE b ON b.JOB_CODE = a.JOB_CODE
   AND b.JOB_GRADE = a.JOB_GRADE
```

```
Select Expression
   -> Nested Loop Join (inner)
      -> Table "JOB" as "A" Full Scan
      -> Filter
```

The result set is shown in the grid in the bottom half of the SQL Editor. In general, the result set is "live", meaning that you can edit the data in the result set. However, if the result set comes from multiple tables, it could be the case that some columns can be edited while others cannot.

An important feature of the editor is the ability to set limits for the amount of data being loaded (either by the number of rows or the volume of data). The editor will then display a warning when the limit is reached with the option to determine the next course of action.

The Lite version also includes a visual SQL query builder, that also parses the current statement and displays it.



Output	Expression	Alias	Sort Type	Sort Order	Aggregate
✓	a.JOB_CODE				
✓	a.JOB_GRADE				
✓	a.JOB_TITLE				

```
SELECT
  a.JOB_CODE,
  a.JOB_GRADE,
  a.JOB_TITLE,
  b.EMP_NO,
  b.FULL_NAME
FROM
  JOB a
  INNER JOIN EMPLOYEE b ON b.JOB_CODE = a.JOB_CODE AND b.JOB_GRADE = a.JOB_GRADE A
```

The Lite version also offers data export (although the number of formats is limited), metadata extraction, basic database maintenance (backup, restore, sweep, gstat statistics, recompute indices and recompile PSQL) and server administration (user editor, log file inspection, connection monitor)

Summary

Despite all its limitations, Database Workbench Lite offers everything you need to comfortably work with Firebird databases. It is clear that this is a mature product with well-thought-out features, the only weakness of which is a somewhat confusing interface (which is a typical price for complexity).

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.

Details about Oldsest Snapshot Transaction

Greg Kay asked:

We are using Classic server on a Linux with Windows clients. There are about 110 users with 450 connections. The header page printout reports:

```
Database header page information:
      Oldest transaction      22951
      Oldest active           319143
      Oldest snapshot         317998
      Next transaction        572981
```

My question is why is the "Oldest snapshot" considerably less than the "Oldest active"? My understanding is that the "Oldest snapshot" should be close to the "Oldest Active".

Ann W. Harrison answers:

Sigh. The serious problem is the difference between Next transaction and Oldest active. But to answer your question …

Each transaction has a lock on its own identity. Each lock has a name, which is the transaction id of the transaction that owns the lock, and a data space in which it keeps the identity of the oldest transaction that was running when it started.

When a transaction starts, it queries the transaction locks in the lock table, looking for two pieces of information:

1. the identity of the oldest transaction currently running and
2. the identity of the oldest transaction that was running when any active transaction started.

The identity of the oldest running transaction is the minimum value of the lock name (for the transaction lock series, but "lock series" is an advance construct and won’t be discussed until next semester.)

The identity of the oldest transaction running when any active transaction started is the data from the lock with the lowest name - or the minimum value of all data values in the transaction lock series.

The transaction puts the first value (oldest currently running) in its transaction lock data area. It uses the second value to determine what old versions can be garbage collected. Anything older than that value can be, nothing equal to or greater can, unless it’s know to have rolled back.

OK so far? That was the easy part. It was designed for classic, where each connection has its own process but the state of transactions has to be shared among processes.

The oldest active is kept in individual transaction blocks. However, new transactions get their transaction number from the database header page, so the header page must be written whenever a transaction starts. Since the whole page is written, and since people often want to know the state of transaction activity, the oldest active number is also written on the database header page when a transaction starts.

And, to make bookkeeping easier for a shared server, the database block also tracks the oldest active. But for reasons unclear to me, it tracks both the oldest active, and the transaction that was oldest active most recently, which it calls the oldest snapshot. (I think this is a bug). And that value - oldest snapshot - is also written into the database header page.

In a shared server, the oldest active and the oldest snapshot move along pretty smoothly together, barring some unusual combination of activity - lots of commits of old transactions just before the next transaction starts. In Classic, however, the individual processes have different ideas of what the previous oldest active (aka Oldest Snapshot) was because transactions start in different processes, so different processes will have read the lock table at different times. Given the number of connections and clients, that probably explains the difference you see.

Making all indices unique

Firebird User asked:

A colleague read years ago that to improve performance, it is recommended to convert all indexes to unique by adding a primary key at the end. For example, an index on column (A) is defined on columns (A, PK). What is the point of this recommendation?

Pavel Císař answers:

I assume this "recommendation" stems from the inefficiency of removing duplicate index keys during garbage collection. This problem was inherited by Firebird from InterBase, and was completely removed in Firebird 2.0.

The problem was basically that garbage collection on indexes with lots of duplicates could eat up your CPU because duplicates were stored with the most recent at the front of the list, then removed oldest first. Removing a record required reading the whole duplicate chain. Adding the primary key to the end of the index key preserves its basic functionality (Firebird will use only the initial part of the composite key), but eliminates duplicates and thus this problem.

If you're worried that this trick will skew your index statistics, don't worry. Since version 2.0.6, statistics are also stored for individual segments, so if the optimizer is considering using only part of the index key, it has accurate statistics available.

However, indexes now order duplicates by the record number of the record to be deleted, so the garbage collector can look up the entry to be removed by a combination of the value - not very selective - and the record number - very selective. For the purpose of garbage collection, all indexes are now unique.

Currently, the primary key trick does not bring any benefit, nor does it cause any harm, other than unnecessary waste of space.

However, the significant improvement in the efficiency of removing duplicate keys does not mean that removing garbage from indexes is now a cheap matter. It is necessary to realize that each such operation may involve writing several pages, including their reorganization. When cleaning the index for small changes in the database, this may not be noticeable, but it can significantly slow down the server when removing a larger number of records, for example after a mass change or deletion.

How much space does NULL take up?

aiylam_s asked:

I have 2 Firebird databases, each with 10,000 records. The second has 4 VARCHAR columns:

```
- ID VARCHAR(40) indexed not-null
- LOCATION VARCHAR(250) indexed not-null
- LOCATION_PART1 VARCHAR(32000)
- LOCATION_PART2 VARCHAR(32000)
```

The size of the first database on disk is 1941504 bytes (~1.9 MB). The size of the second on disk is 14700544 bytes (~14.7MB). Do 20,000 NULLS really take up 12759040 bytes (~12.7MB)? That corresponds to about 638 bytes per NULL! Can anybody shed some light on this?

Ann W. Harrison answers:

Yup. The null is only one bit, but the record is laid out with all fields fully expanded and null fields zeroed or blanked. Then the record is compressed, using a one byte run-length encoding. Up to 127 identical bytes turn into two bytes, which is OK, but not great when you have 32,000 byte fields - that's a bit more than 500 bytes… and I've probably forgotten something.

> *The record compression method has been significantly improved in Firebird 5, so the amount of unused zeroed space is much smaller than in earlier versions. However, it still takes up more than a single bit.*

Planet Firebird

In this new regular section, we will summarize recent activities and initiatives within the Firebird database community. This will include coverage of events, news, achievements, and notable community projects from the past quarter. Additionally, we'll highlight plans and opportunities for involvement in the upcoming period, such as conferences, meetups, and collaborative efforts within the Firebird ecosystem. This section will keep you informed about ongoing work both within and outside the Firebird project.

We encourage you to support this effort by sharing information about any relevant events, achievements, projects, or any other activities that you believe should be highlighted—whether they have already taken place or are planned for the future. Your input will help us to keep the community connected and informed.

You can reach us at either the "emberwings" or "foundation" @firebirdsql.org email addresses.

Firebird Advent Calendar

This year is special for Firebird as we celebrate the project's 25th anniversary. If you've been using Firebird since its first version, then a quarter-century of stability is certainly worth celebrating—almost like a silver wedding anniversary, right? To mark this milestone, we introduced a special Advent calendar on our website last December—a festive lead-up to the 25th anniversary celebration. It offered a month-long journey through Firebird's history and community spirit, and for those who missed it, it remained available until the end of January 2025.

Each day, a new door on the calendar revealed unique Firebird-themed holiday content. From short stories (a selection of which appeared in the last issue) to festive pictures and songs, there was something for everyone. All the songs from the calendar, along with many more, are available on the Firebird Songs YouTube channel. We also included several fun quizzes to test your knowledge of Firebird's past, present, and future.

In addition to the festivities, we used this event to introduce anonymous questionnaires, allowing users to share insights about how they use Firebird and the challenges they face. If you'd like to help us better understand your needs, we encourage you to take a moment to fill them out. A description of the questionnaires, along with links, can be found on the last page of each issue.

A sad day for the Firebird Project

Helen Borrie, a key figure in the Firebird relational database project and a longtime contributor at IBPhoenix, passed away on January 2, 2025. Her contributions were essential to Firebird's creation and its development over the past 25 years.

Helen's dedication to the project was unwavering. She played a critical role in establishing the Firebird Foundation and managed its operations as the Foundation's Secretary. Her work ensured that the project had the structure and support needed to grow and succeed. She often worked behind the scenes, making sure things ran smoothly and that the community had the resources it needed.

She was also the author of *The Firebird Book*, a comprehensive guide that became an invaluable resource for users and developers. Helen's writing made Firebird accessible to many, helping them understand and use the database effectively. She was always ready to assist others, sharing her knowledge generously within the user community.

Helen's quiet leadership and dedication left a lasting impact on Firebird and its users. Her efforts helped build not just a powerful database but also a strong, collaborative community. She will be deeply missed by all who knew her and benefited from her work.

The Firebird Butler Project

By Pavel Císař (IBPhoenix)

For many years, Firebird users have expressed frustration over the lack of suitable tools for administering and managing their deployments. Firebird inherited only a limited set of command-line tools from InterBase, and due to resource constraints, the project was unable to develop and maintain comprehensive management solutions. Instead, efforts were focused on enabling external development through features like trace services and monitoring tables.

This situation persisted until 2018, when IBPhoenix proposed a new division within the Firebird Project dedicated exclusively to addressing this problem. This division, named Firebird Butler, was established in January 2019 with the long-term goal of creating an ecosystem that provides essential management tools while remaining open for external contributions and enhancements.

The Challenge of IT Infrastructure Management

Managing IT infrastructure boils down to two fundamental aspects: **disaster prevention and recovery**, and **system control to maintain operational stability**. Both elements must work in sync, relying on tools that measure parameters, analyze data, and execute actions—either automatically or with human intervention.

In reality, however, this is far from simple. The diversity of deployment requirements, evolving needs, and an overwhelming number of partial solutions make it an ongoing struggle. No single tool covers all scenarios, and the more capable a system is, the more complex and demanding it becomes. IT teams frequently encounter situations where solutions handle 70%–98% of their needs, but the remaining gap proves frustratingly difficult to bridge without custom workarounds or complex integrations.

Firebird deployment management is no exception. While general-purpose IT tools exist, the number of solutions specifically tailored for Firebird is small—and their usefulness stops at Firebird-specific issues. Yet, database management is only one part of a broader IT landscape. Consider a scenario where an IT team must manage a Firebird-based application that empower multiple semi-automated production facilities located in different countries—Firebird is just one piece of that puzzle.

The unavoidable reality is that every organization ends up using some custom parts in their final solution. It’s also not uncommon that the more effort you put into tailoring that solution, the more effective, scalable, and reliable it becomes. So, instead of searching for a so-called "Golden Product" that attempts to cover all possible needs, a better approach is to seek a **Solution Construction Kit**—a framework that allows users to assemble only the tools they truly need in the most efficient way. This philosophy enables a tailored, adaptable, and scalable solution rather than a rigid, one-size-fits-all approach. This is where Firebird Butler enters the picture, offering an open, flexible, and modular approach to Firebird management that empowers users to build exactly what they require with minimal overhead.

The Vision Behind Firebird Butler

The Firebird Butler Project aims to address the long-standing gap in Firebird database management by leveraging open standards for interoperability. Rather than relying on a single monolithic tool, Firebird Butler defines a modular framework where interoperable services can be assembled into customized solutions tailored to specific needs.

A useful analogy is the **LEGO system**: Firebird Butler establishes guidelines for designing individual components that seamlessly fit together, regardless of their origin or implementation. This ensures that services developed by the Firebird Project or third parties can integrate smoothly, allowing for flexible, purpose-built solutions. By adhering to these open standards, Firebird Butler fosters seamless interoperability across platforms and implementations.

"That sounds great," you may say, "but how exactly would such a software LEGO system work?" To explain, we must first look at another analogy.

One of the best historical examples of a component-driven architecture is the **Delphi VCL** (Visual Component Library). The VCL revolutionized software development by offering structured, reusable components that developers could easily integrate into their applications. Firebird Butler adopts the same component-based philosophy but adapts it to meet modern requirements.

Since Delphi's launch in 1995, the IT landscape has evolved dramatically. Today, key concerns include availability across multiple platforms, distributed architectures, and scalability. Modern systems must process vast amounts of data, increasingly incorporating AI and adaptive learning mechanisms to handle complexity efficiently. As interconnectivity expands, software components have transitioned into service-oriented, messaging-based architectures to meet these increasing demands.

However, rapid technological advancements have created two major issues:

1. Service integration has prioritized convenience over efficiency, leading to suboptimal solutions that rely on web technologies like HTTP and JSON, as well as complex messaging protocols like AMQP, which often require entire stacks of additional software, instead of well-optimized communication methods.

2. The software world has become fragmented, with small component systems and large distributed systems existing independently, lacking a seamless way to bridge them.

Had Firebird Butler been built on existing systems and widely used technologies for component interaction, it would have faced a difficult tradeoff: prioritizing either efficiency but limited scalability or scalability at the cost of increased complexity and lower efficiency. Ultimately, this would restrict the range of viable use cases.

Since one of Firebird's most valued features is its scalability philosophy—"from embedded to enterprise", the Firebird Butler Project sought an **innovative solution** that could bring this principle into the world of software components.

Thus, the core idea behind Firebird Butler was to design a system of software components that, when used within a monolithic application or process, would function similarly to Delphi VCL, while also supporting distributed solutions spanning multiple processes or network nodes **using the same components**.

Technological Challenges and Solutions

One of the key challenges in designing Firebird Butler is balancing the efficiency of in-process communication with the flexibility of distributed services. Traditional component-based architectures, such as Delphi's VCL, rely on direct method calls that enable fast and predictable interactions between components. These calls, however, do not scale well across multiple threads, processes, or networked environments. In contrast, distributed systems favor TCP/IP-based communication, which allows components to operate independently but often introduces latency and complexity due to its inherently stateless and asynchronous nature.

To bridge this gap, Firebird Butler takes a **hybrid approach** by adopting messaging as the core communication method between components. Instead of relying on direct calls or remote procedure calls (RPCs), which can hinder scalability, Butler uses **ZeroMQ** as its messaging backbone. ZeroMQ is designed for high-performance, lightweight communication, making it **equally effective** for intra-process, inter-process, and networked communication. This ensures that services can scale seamlessly while maintaining efficient message delivery.

Another major challenge is balancing tight and loose coupling between services. In a tightly coupled system, components expect stable, persistent connections, which works well within a single application but becomes unreliable when extended over a network. On the other hand, loosely coupled services allow for more resilient and scalable deployments, but they introduce complexity in managing interactions.

Firebird Butler addresses this by supporting both tight and loose coupling based on deployment needs. Within a local environment, components can communicate with minimal overhead. For networked services, Butler ensures robustness by enabling stateless, asynchronous messaging while still allowing for stateful interactions where necessary.

Finally, the choice between synchronous and asynchronous processing plays a critical role in system performance. Synchronous execution is straightforward but limits scalability, as each operation must complete before the next begins. Asynchronous processing, in contrast, enables better resource utilization by allowing multiple tasks to execute in parallel.

Firebird Butler prioritizes asynchronous service communication without enforcing a rigid implementation model. Developers can choose the best approach for their needs while ensuring compatibility with the overall architecture.

By integrating messaging-based communication, flexible coupling models, and scalable processing strategies, Firebird Butler creates a foundation that combines the efficiency of component-based development with the adaptability of modern distributed architectures. This approach ensures that solutions built with Butler remain both highly performant and scalable, capable of handling anything from local applications to large-scale distributed systems.

ZeroMQ: The Messaging Backbone

Building an effective messaging system for Firebird Butler required a solution that was not only fast and lightweight, but also flexible and scalable enough to accommodate diverse deployment scenarios. Many traditional messaging systems rely on centralized message brokers such as RabbitMQ or Kafka, which come with built-in reliability features but also introduce complexity and overhead. Firebird Butler needed a different approach—one that allowed for direct communication between services without forcing a rigid architecture. This is where ZeroMQ stood out as the ideal choice.

Unlike full-scale message queue systems, ZeroMQ does not require a central broker, but importantly, it does not prohibit developers from implementing one either. Instead, it provides the building blocks for messaging, allowing developers to construct as much infrastructure as they actually need, rather than being forced into adopting heavyweight, pre-defined solutions. This aligns perfectly with Firebird Butler's philosophy of modularity and flexibility—we needed a system that could operate with minimal infrastructure yet scale up when necessary.

ZeroMQ's appeal also lies in its diverse messaging patterns, which include request/reply, publish/subscribe, push/pull, and dealer/router models. This means that Firebird Butler services can communicate synchronously or asynchronously, with tight or loose coupling, depending on the specific needs of the deployment. Additionally, ZeroMQ supports multiple transport protocols, including TCP for distributed communication, IPC for inter-process messaging, and INPROC for ultra-fast intra-process communication. This versatility ensures that Firebird Butler can be deployed in small, single-machine setups as well as complex distributed networks without re-engineering its messaging layer.

Another advantage is performance. ZeroMQ is designed to be lightweight and high-speed, outperforming many traditional TCP-based applications through efficient message batching and an internal threading model. This makes it an excellent fit for Firebird Butler, where low-latency communication between services is essential.

Of course, ZeroMQ is not without its drawbacks. Unlike broker-based messaging systems, it does not offer built-in message persistence or guaranteed delivery, which means applications using it must handle these concerns at a higher level. However, for Firebird Butler's purposes, ZeroMQ provides the right balance of performance, flexibility, and scalability. It gives us the freedom to implement only what is required while avoiding the overhead of a fully-fledged enterprise messaging system.

ZeroMQ has gained widespread adoption across industries, proving its reliability and flexibility. It provides a low-level C API with high-level bindings in over 40 languages, including Python, Delphi, FreePascal, Java, PHP, Ruby, C, C++, C#, Erlang, and Perl. Additionally, it is available as pure Java (JeroMQ) and pure C# (NetMQ) implementations, both officially maintained by the ZeroMQ community.

Its adaptability has made it the choice of numerous organizations, including AT&T, Cisco, EA, Los Alamos Labs, NASA, Weta Digital, Zynga, Spotify, Samsung Electronics, IBM, Microsoft, and CERN. Developers can explore the **ZGuide** for in-depth explanations and practical examples in over 30 programming languages, reinforcing its role as a versatile messaging solution.

Ultimately, although ZeroMQ is not perfect, it is well-suited for the goals of Firebird Butler. It allows us to create fast, adaptable, and efficient communication channels between services, reinforcing the project's fundamental vision: a modular, scalable, and lightweight system that grows with the needs of its users.

From specification to implementation

Firebird Butler Platform specifications (a set of RFC documents) serve as blueprints for implementing the platform in various programming languages. The reference implementation is in Python, but the Butler division envisions supporting C#, Java, Delphi, and Free Pascal. However, at present, Firebird Project resources allow for implementation only in Python.

Since the platform specification does not dictate an exact API or implementation details, individual implementations may differ in design, architecture, features, and API. Multiple implementations in the same language could even coexist, offering different architectural approaches—such as a traditional thread/process model versus an asynchronous processing model.

The Firebird Butler Platform is essential for creating **Butler Services**, which are software components that communicate using **ZeroMQ sockets** and the **Firebird Butler Service Protocol**. A service can use multiple ZeroMQ sockets for different tasks, but only one primary socket is required to comply with the Butler Service Protocol.

Butler Services could do anything, but a well designed service does only one task, or a small set of closely related tasks within single category. While respecting the rule of simplicity, services can be divided into several basic types:

- **Measuring services**, which collect and transmit data, such as monitoring table statistics.
- **Processing services**, which receive data, process it, and output results—examples include analytics engines, data transformers, brokers, routers, and load balancers.
- **Provider services**, which execute tasks on request, such as performing database backups.
- **Control services**, which manage other services within the ecosystem.

Saturnin: A Reference Implementation

The Saturnin project has several goals:

- Provide reference implementations for Firebird Butler specifications, serving as examples for implementations in other languages.
- Act as a proof-of-concept for Butler specifications and a real-world test of its capabilities in managing Firebird databases through a service-oriented, message-driven approach.
- Deliver a platform for developing and deploying Butler services and complete solutions.
- Offer a library of selected services for direct use.

Development began in February 2019, with the first prototype demonstrated at the Firebird Conference in Berlin in October 2019. This prototype implemented the Firebird Butler Service Protocol (FBSP), Firebird Butler Datapipe Protocol (FBDP), test services, and a special "Node" service for running and managing other services, including a CLI console for interaction. This milestone primarily aimed to validate the Butler specifications and protocols.

Initially, the `saturnin-sdk` repository contained development tools and protocols, while the `saturnin` repository handled service development and deployment. Development continued through 2020, despite the COVID outbreak, focusing on core fundamentals and a new Firebird driver for Python. The `firebird-base` package was introduced to handle configurations, registries, logging, tracing, buffer management etc., which were later used in the new `firebird-driver` package and Saturnin itself.

During development, the project underwent its first major restructuring. Code for running services was moved to a new `saturnin-core` repository, leaving `saturnin-sdk` with only development tools and sample services. The `saturnin` repository became a service library for Saturnin-based solutions, with both `saturnin` and `saturnin-sdk` depending on `saturnin-core`. The Node service remained the primary service deployment platform, but by the end of the year, it became clear that a different approach was needed for deployment.

In 2021, Saturnin underwent a significant architectural redesign. The `saturnin` repository evolved into a comprehensive development, deployment, and management platform, absorbing `saturnin-core`. The `saturnin-core` was repurposed as a library of deployable services, while `saturnin-sdk` became an add-on for development-related extensions. With this shift, `saturnin` became the primary dependency.

The transition was set to complete in 2022 but was delayed for six months due to work on a new QA system for Firebird, based on pytest and the new Python driver. It was completed in March 2023, resulting in 0.8.0 release with a core library of 13 micro-services, and support for service bundles (solutions) built as standalone executables produced by **Nuitka** Python compiler.

At this stage, Saturnin was ready for real-world testing. Discussions began with several Firebird users about piloting a simple Firebird-management solution, but external factors halted the initiative. However, development continued, culminating in the release of version 0.9.0 in October 2023. Attention then shifted to transitioning the Firebird Foundation from Australia to the Czech Republic.

Development resumed in 2025, beginning with a revitalization of the entire stack of eight Python packages currently maintained by the Firebird Project.

The project is named Saturnin after the clever and composed butler from Zdeněk Jirotka's novel Saturnin. Much like Jeeves from P.G. Wodehouse's stories and Jarvis from Iron Man, Saturnin blends subtlety with a touch of humor and authority. His character, known for his refined intelligence and quiet efficiency, orchestrates events behind the scenes with grace. The name reflects this balance of playful elegance and unwavering loyalty, capturing the essence of a system that operates with quiet precision and composure.

Conclusion

For the average Firebird user, Butler represents a forward-thinking step in database automation. Its open architecture means that future service implementations can extend its capabilities far beyond what’s possible today. Instead of relying on closed, monolithic tools, Butler enables an ecosystem of flexible, interoperable services that can be mixed and matched as needed.

Firebird Butler and Saturnin are still in their infancy, but their potential is immense. The project needs developers to contribute, early adopters to test its capabilities, and sponsors to support its continued growth. If you believe in the power of open standards and service-oriented database management, now is the time to get involved.

In the next issue, we will introduce you to the architecture and capabilities of Saturnin.

Resources

1. The Firebird Butler, An Introduction presentation from Berlin 2019 Conference
2. Firebird Butler website with specifications
3. Butler Hub on Firebird website
4. ZeroMQ and ZGuide
5. Nuitka Python Compiler

…And now for something completely different

The Secret of the Firebird

Firebird was a legend in the world of databases. Over decades, the open-source powerhouse had earned its place as a reliable and efficient system, quietly powering industries across the globe. Its user community stretched far and wide, and its long history was intertwined with contributions from countless developers—some independent enthusiasts, others employees of companies heavily invested in its success.

With such a sprawling history, myths inevitably grew around Firebird. Tales of hidden tricks, mysterious parameters, and forgotten experimental branches swirled through forums and mailing lists. Some stories were true, some exaggerated, and others outright fabrications. And after decades, it is hard to tell them apart.

Renata Alvarez was an experienced Firebird user, known among her colleagues for her sharp intuition and tenacity. Yet, even her skills couldn't solve the performance issues plaguing her team's latest project—a resource-intensive financial application.

Her team had tried everything: optimizing queries, tweaking indices, and experimenting with every documented configuration option. Still, the database groaned under the weight of their workload, and their deadlines loomed ever closer. Frustrated but unwilling to give up, Renata turned to the community for help.

In her search, she stumbled across an old forum post that caught her attention:

> “Performance bottlenecks? Maybe you haven’t unlocked all the Firebird has to offer. Ever heard of TRB? 😉”

The cryptic comment led her down a rabbit hole of old discussion threads, uncovering fragments of lore about Firebird’s development history. One name kept surfacing: Ignis, a long-retired core Firebird developer. Few in the community seemed to remember him, but those who did described him as a "loner genius" who avoided public forums and conferences, communicating only within a circle of Firebird’s core developers.

“Ignis wrote some of the most brilliant code in the project,” one user claimed.

Through fragments of old public forum discussions and development mailing list archives, Renata pieced together the story of Ignis. Among his contributions to Firebird was an ambitious project he called the Throughput Rate Booster (or TRB for short). According to old conversations, TRB was designed to push Firebird’s performance to its absolute limits, extracting every ounce of potential from the hardware while consuming minimal resources.

But TRB had never been publicly announced. One archived message explained why:

> “Ignis didn’t want to raise false hopes. He insisted we keep it under wraps until it's perfect. No public announcements, no documentation—just testing, iteration, and silence.”

Discussions among Ignis's collaborators, with cryptic mentions of "overheating flames" and warnings to "tread carefully," hinted at potential risks inherent in TRB's architecture. While Renata couldn't fully decipher their meaning, the secrecy surrounding the project suddenly made sense.

Renata searched for any mention of the project's conclusion, but the trail vanished into obscurity. Did Ignis succeed? Or did the risks prove insurmountable?

There was only one way to find out.

Renata opened Firebird's source code repository for the first time. It was overwhelming—thousands of files, decades of commits, branching histories, and countless pull requests created a tangled web. But she was determined to uncover the truth.

She started to follow the Ignis's trail, and at first, the commits seemed unrelated: simple bug fixes or minor optimizations. But as she searched for clues tied to Ignis, patterns began to emerge. Ignis's fingerprints were scattered throughout the codebase—alterations to the query planner, threading model, and transaction handling. In one commit related to query optimization, she found:

```
// Optimization hooks added. For testing only. Handle with care.
```

Another commit, touching on multi-threading, carried an equally mysterious note:

```
// Experimental flightpaths – requires manual tuning.
```

The more Renata followed Ignis's trail, the more pieces of the puzzle she uncovered. Each comment hinted at a larger picture, a hidden layer of functionality buried within Firebird's architecture. One commit led her to a later removed file labeled flightpaths.h, which contained only bunch of enigmatic constant definitions with strange comment:

```
// The Firebird soars when its flames are balanced.
```

In the query planner module, she uncovered a tantalizing line:

```
// TRB logic added. Testing incomplete.
```

Each discovery led her deeper into the labyrinth. Ignis and his collaborators had buried the TRB code behind layers of indirection, conditionals, and seemingly unrelated changes. It was so well hidden that even current Firebird maintainers seemed unaware of its presence. Finally, after hours of meticulous searching and comparisons, Renata found the core of TRB's logic present in the master branch.

Her excitement grew. TRB was still in the Firebird code! But a major question remained unanswered: how was it activated?

Renata pored over the comments and surrounding code but found no clear instructions. The developers had deliberately kept the activation method obscure. Still, she reasoned that they must have needed a way to test it during development.

"It has to be external," she thought. "Something they could control without touching the code every time."

Two possibilities stood out: an environment variable or a configuration file entry. Both were standard practices for enabling hidden features during testing. Renata decided to start with the simpler option.

Opening her firebird.conf file, she added a single line based on everything she had learned:

```
turbo = True
```

Renata restarted the server and immediately opened the firebird.log file. Among the usual startup messages, one stood out:

```
[INFO] TRB Mode Engaged: The Firebird takes flight.
```

Her heart raced. She had done it.

Renata ran her benchmarks, and the results were astonishing. Queries that had dragged for minutes completed in seconds. Complex joins and massive data migrations flew by effortlessly. It was as if Firebird had been holding back its true potential all along.

But the exhilaration was short-lived.

Soon, she noticed anomalies—missing rows, corrupted values, and inconsistencies in certain operations. The database logs filled with warnings:

```
[WARNING] Flightpath integrity degraded. Risk of instability.
[WARNING] Hardware limits exceeded – system overload imminent.
```

Her server's CPU temperatures spiked dangerously. One machine shut itself down to avoid frying, while another suffered permanent hardware damage.

Frustrated and alarmed, Renata turned to the online developer community. She posted a carefully worded inquiry, sharing her findings without revealing too much detail. Most responses were skeptical, brushing off her claims as impossible or fabricated. But one private message stood out, from a user named Phoenix42:

> "You've unlocked the Firebird's secret, haven't you? TRB was Ignis's greatest work—and his greatest failure. It wasn't safe then, and it's not safe now. We hid it for a reason. Use it wisely, or let it rest."

Renata realized that Phoenix42 must have been one of Ignis's collaborators. At last, she understood why Ignis and his team had concealed TRB so carefully. The weight of that truth pressed down on her—a mix of awe at their ingenuity and the sobering reality of the dangers they had tried to contain.

Renata stared at her notes for hours, weighing her options. She could share her discovery with the Firebird community, mobilizing developers worldwide to stabilize TRB. But the risks were enormous. If mishandled, TRB could cause catastrophic failures and harm Firebird's reputation as a solid, dependable database system.

In the end, she made the hard decision. Renata deleted her detailed findings, leaving only cryptic references in her personal notes. She told no one about Turbo Mode, trusting that someday, someone better equipped might revisit it.

But she couldn’t forget what she’d found. Her research hinted at other forgotten features and experiments within Firebird’s code—untapped potential that might one day change the database world. And unlike TRB, some, she believed, would be safe to explore.

Meanwhile, the legacy of Ignis burned in Firebird’s source code, waiting for the next brave soul to unlock its hidden potential.

Although this story is fictional, it contains a grain of truth. Because Firebird, like any complex software, has its secrets, known only to insiders. Don't believe it? Then try running the following query on the sample EMPLOYEE database:

```
with s1 as (select ORDER_STATUS, count(*) as sc from SALES group by 1),
s2 as (select ITEM_TYPE, count(*) as sc, avg(QTY_ORDERED) as sa,
sum(QTY_ORDERED) as ss, min(QTY_ORDERED) as si, max(QTY_ORDERED) as sx
from SALES group by 1)
select first 1 skip 2 sc from s1
union select first 1 sc from s1
union select first 1 skip 1 sx + sx + sx from s2
union select first 1 skip 1 sc + sa + ss + sx from s2
union select first 1 sa - sc - sc - si from s2
union select first 1 skip 1 sc + sc from s1
```

If you can't figure out what the result means, wait for the next issue.

Most secrets (or Easter eggs, if you will) fall into the category of forgotten knowledge, not-so-obvious uses of certain features and their combinations, internal implementation details, etc. They usually have no real meaning and use for ordinary users, but they can be a secret ace up your sleeve in solving some unusual situations. The truth is that most professionals don't like to share such secrets, just like magicians don't reveal their tricks, or chefs their special ingredients. So if you discover some, good for you.

However, there are also secrets that are hidden on purpose (and we really don't mean the infamous LOCKSMITH account, which has long since been deleted). One such secret of the Firebird, although hidden in plain sight, is particularly hilarious. And if you can uncover it, it will bring you fame.

This issue contains several clues on how to find it, and we will gradually present you with more.

Write to us if you think you have discovered it

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!