EmberWings

2024/3

IBPhoenix

THE POWER WITHIN

September 2024

www.ibphoenix.com

emberwings@ibphoenix.com

IBPhoenix

THE POWER WITHIN

Support & tools for Firebird

IBPhoenix is the leading provider of information, tools and services for Firebird users and developers

IBPhoenix website & e-shop

IBPhoenix is directly involved in the Firebird Project and is also a member of the Firebird Foundation z.s.

In This Issue:

- Editorial
- Wisdom of the elders
- Challenges with Primary Keys
- Interview with Pavel Zotov
- Development update: 2024/Q3
- Toolbox: Red Expert
- Answers to your questions
- Automatic Test Creation
- ..And now for something completely different

EmberWings Under Firebird's Wing

As we release the third issue of *EmberWings* magazine, we are excited to share an important update regarding the future of our publication. This issue marks the final edition published under IBPhoenix, as we proudly announce the transition of *EmberWings* to the **Firebird Foundation**, where it will become the official publication of the **Firebird Project**.

This shift brings new opportunities for growth, allowing *EmberWings* to further align with the Firebird Project's mission of advancing the development and use of Firebird database technologies. The transition will not only maintain the unique style and tone our readers have come to appreciate but also bring a broader range of content. Readers can expect more diverse technical articles, deeper insights into Firebird database server innovations, and expanded coverage of related software and tools.

Starting with the December issue, the magazine will be offered under a new publication model. *EmberWings* will continue to be available for free, but with a 12-month delay for the general public. However, Firebird Foundation donors will enjoy early access to each new edition. We are pleased to introduce two new subscription programs to support the Firebird Project: **Firebird Associate** and **Firebird Partner**. These programs are designed for dedicated supporters of the Firebird, offering access to the latest issues of *EmberWings*, along with other benefits.

We are deeply grateful for the community's continued support of *EmberWings* as it enters this new phase under the Firebird Foundation. We believe this transition will provide the magazine with a solid foundation to grow, offering even more value to our readers while reinforcing our connection with the Firebird Project.

Thank you for being part of this journey with us. We look forward to continuing to share insights, innovations, and technical expertise as we take flight into this new chapter.

Your EmberWings editors

In a serene village surrounded by rolling hills, a young apprentice was given the task of managing the tribe's extensive records using the Firebird system. With great enthusiasm, the young man began organizing the data but soon encountered a perplexing problem.

"Wise Shaman," the young apprentice said, "our records use a primary key based on a combination of multiple attributes, but now I'm finding that the system is becoming slow and inefficient. The complexity of the composite key is affecting performance. What should I do?"

The shaman, known for his deep insights, nodded thoughtfully and said, "Let me share a story that may offer you guidance."

In a vast and ancient garden, each plant was identified by a complex label that included multiple attributes such as species, age, and color. The gardener used these composite labels to keep track of the plants. However, as the garden grew, the complexity of managing these detailed labels led to inefficiencies and slowdowns in the garden's upkeep.

The gardener sought the wisdom of the old tree, who was known for its profound knowledge of nature. 'Wise Tree,' the gardener said, 'our complex labels are causing inefficiencies and slowing down our work. How can we improve our system?'

The old tree replied, 'Consider simplifying your approach. Instead of relying solely on the composite labels, create an additional, simpler identifier that can serve as a quick reference. Use this identifier in conjunction with your detailed labels to streamline your tasks. For example, you could use a unique plant number that links to a more detailed description. This way, you maintain clarity without sacrificing efficiency.'

The gardener followed the old tree's advice and introduced a simpler, unique identifier for each plant. The detailed labels remained, but the new identifier made it much easier to manage the garden. Efficiency improved, and the gardener could tend to the plants with greater ease and speed.

The shaman turned to the young native and said, "Just as the gardener learned to simplify the identification process by adding a unique, simpler identifier, so must you consider refining your primary key strategy. Introduce a more straightforward key to enhance performance and use it in conjunction with your composite key. This will help you maintain efficiency and clarity in managing your records."

The young apprentice took the shaman's advice to heart and began to implement a simplified primary key system alongside the existing composite key. The performance issues were resolved, and managing the records became much more efficient.

And so, the young man learned that balancing complexity with simplicity can lead to greater efficiency and effectiveness in managing data.

Challenges with Primary Keys

Primary keys undoubtedly play a vital role in relational databases. That is why it is necessary to properly understand their nature, implementation details and ways of using them within the database and applications that work with the database. Without this understanding, it is very easy to make fundamental mistakes in database design that are very difficult (if at all) to correct later.

This article focuses on the issues and challenges you will face when working with primary keys in Firebird, and relational databases in general.

The root of all problems

At the heart of most of the problems we encounter is a lack of understanding of the concepts and their role in the implemented system.

First, when talking about primary keys, it is always necessary to distinguish between the terms `primary key` and `PRIMARY KEY constraint`.

- A PRIMARY KEY is defined by Relational Model as a set of one or more attributes that uniquely identify tuples (rows) in a relation (table).
- A PRIMARY KEY CONSTRAINT is a rule applied to a database table that ensures **uniqueness** and **non-nulability** of column(s) that make up the PRIMARY KEY. It's defined in SQL standard and implemented by relational database systems (such as Firebird) with use of NOT NULL constraints defined on all PRIMARY KEY columns, and an unique index.

The devil is in how these two concepts are linked, with all problems starting with the choice of primary key.

To have a key or not

When designing tables, it is usually not a problem to determine whether the table has a (natural) primary key. It is much more difficult to decide whether a table that does not have a natural primary key should have a (artificial) primary key. These are primarily auxiliary tables, e.g. those used to implement N:M links, request queues, etc.

Some developers tend to define primary keys for all tables indiscriminately because they want to retain the ability to delete or update specific rows (from a given set). However, this is completely unnecessary because the ROWS clause of the DELETE and UPDATE statements can be used for the same purpose.

A specific case are frameworks for working with the database, which for their functionality require the presence of a primary key even for tables that may not otherwise have one. From practice, it can be concluded that the presence of an unnecessary primary key causes more trouble than benefit in the long run. If you

have a choice of used technologies, we recommend that you consider other options.

In practice, it is advisable **not** to define artificial primary keys for any table for which a natural primary key cannot be defined.

Natural vs. Artificial keys

Nowadays, the prevailing belief is to use only artificial keys instead natural ones. It cannot be denied that the use of artificial keys has many significant advantages (and support with IDENTITY columns). Therefore, in general, we can recommend using an artificial key of a uniform type. However, using artificial keys also have a major drawback.

> *Artificial keys are ideal for processing, but absolutely useless from a user's point of view.*

This results in the need to retrieve data from linked tables, as the foreign key value cannot be displayed to the user (or used for selection). The consequence is increased query complexity (number of additional table joins) with all the consequences for performance.

Unfortunately, the negative effects of complex queries on performance will become apparent only after a certain period of time, when the amount of processed data increases. Since in practice it is common for the tables with the most foreign keys to be the most used (and largest) as well, these problems will show up sooner rather than later (but rarely in testing).

However, the existence of this fundamental disadvantage is not a sufficient reason for not using artificial primary keys. But it requires appropriate measures in the way of working with data and at the application level, which will make it possible to reduce the complexity of queries.

Possible measures include:

- For small and/or mostly static tables, in-app lookup tables can be used to ensure that user-friendly data is displayed instead of a loaded foreign key.
- The display of user-important data from linked tables can be separated in the application into a separate form displayed only on request.
- For frequently referenced tables with a simple (one column) and stable natural key, consider using that key instead of an artificial key **for references**. It also means that referenced table can still have an artificial primary key, but the natural key is defined as an UNIQUE constraint, and used as a foreign key in other tables.

Unfortunately, there are no good simple or one-size-fits-all solutions for reducing query complexity when filtering by values from linked tables. The only way is to avoid such filters, or at least reduce the number of such filters used at the same time.

Choosing the type of artificial key

The rule of thumb is to use only a single data type (domain) for all artificial primary keys in the database. A uniform type greatly facilitates working with data in the application, especially when using frameworks that isolate work with the database from the rest of the application logic (either in the form of an ORM or a custom solution).

Of course, this rule is not a dogma, and it can be broken in justified cases. Sometimes this is the only way to avoid major problems (see the example using a natural key for references). So if you are isolating database work into a separate subsystem or layer within your application (recommended), make sure it supports more than one data type for primary (and thus foreign) keys.

Since primary key values must be unique, the choice of type is directly related to the way new unique values are created, and the scope of uniqueness.

The requirements for the scale of uniqueness have the highest priority when choosing a type.

- If uniqueness within a table is sufficient for you, integer types with a range corresponding to the expected maximum number of rows in the table will suffice.
- If you require database-wide uniqueness, it may still be possible to use an integer type as long as its range is still sufficient. However, it is usually necessary to use a different data type or keys composed of several columns.
- If you require uniqueness within multiple databases, it is essential to use a data type that best matches the chosen method of generating such a unique key.

If tables in a database have different uniqueness requirements, it is advisable to use a single mechanism for all tables, that is, the mechanism with the highest identified uniqueness scope.

In practice, determining the scope of uniqueness comes down to whether there may be a need, now or in the future, to consolidate data from multiple databases into a central system or to integrate data across various databases.

If the answer is negative, uniqueness within the table is enough for you, and thus keys of type BIGINT.

Advantages of choosing this type:

- Easily create new unique values using generator or SQL IDENTITY clause.
- Inserting ascending values into a primary key index requires a minimal amount of page splitting, and also the occupancy of the leaf pages is maximal.
- Option to later extend the scope of uniqueness beyond a single table (more on that later).

If the answer is yes, you can choose from a wide range of technical solutions, each of which has its own specific advantages and disadvantages.

All these solutions have two components:

- The method of generating unique values, and the resulting basic data type of the value.
- Way of representing values in the database.

UUIDs

The easiest way to generate globally unique values is to use UUIDs (Universally Unique Identifiers) from widely recognized standard RFC 4122. This standard offers several versions of global identifiers of the same length, which differ in the way they are generated and the properties of the generated values. Version 1 or 4 identifiers are most commonly used for primary keys, as versions 2, 3, and 5 are not suitable due to the way the values are generated.

However, the structure of these values makes indexes that use them as keys quite inefficient (especially with Firebird, which uses prefix key compression). For that reason, RFC 4122 was superseded by RFC 9562 in May 2024, which defines additional versions 6 and 7 that are much more suitable for primary keys.

> *Due to the novelty of the standard, UUID version 7 support is currently only implemented in the development version of Firebird 6. However, it is likely that this support will also be added to one of the upcoming Firebird 4 and 5 updates.*

While the UUID standard elegantly solves the way of creating globally unique values, it faces the problem of how to store these values in the database. The UUID is a binary sequence of uniform size of 16 bytes. These can be stored directly as CHAR(16) CHARACTER SET OCTETS, but using the OCTETS character set causes problems when working with these values in user applications and database tools. Therefore, these values are typically converted to a character string, and stored as a CHAR type in the ASCII character set.

The most commonly used conversion to hexadecimal representation is 32 characters long (or 36 if you keep the standard formatting with block separators), which is quite a lot, especially compared to the 8 bytes of BIGINT keys. With a primary key that is only one per table, it would still be acceptable, but with foreign keys that can have a larger number in one table, it has a major impact on performance and resource requirements.

> *Currently, the best solution is probably to use Base64, Base85 or Base128 encoding, which reduces 16 bytes instead of 32 characters to 24 (Base64), 20 (Base85) or 19 characters (Base128). Additionally, this conversion changes the characteristics of the values somewhat, which may (or may not) improve indexing.*
>
> *Although Base64 offers the least compression, you can use Firebird's internal functions BASE64_ENCODE and BASE64_DECODE. We hope to see support for other variants in the future.*

But using UUID is not the only way to create globally unique identifiers. Popular alternatives include the Hi-Lo method and Snowflake ID, which we will now introduce.

The Hi-LO method

The Hi-Lo method is a technique used to generate unique identifiers in distributed systems and databases that minimizes contention and improves performance in high-concurrency environments.

In short, it generates sequence identifiers that consist of two whole parts, where the first (HIgh part) is generated by the central authority and the second (LOw part) is generated locally.

Generated values can be stored in different ways. You can use a primary key composed of two columns, one for each part. However, it is much more advantageous to merge both segments into a single BIGINT value. For example, if you use the upper two bytes of the 8-byte BIGINT for the upper segment for database identifiers, and the remaining lower 6 bytes for the local ids, then you can create up to 281,474,976,710,655 unique keys for each database in a total of 65,535 databases.

Another advantage of this solution is the possibility to use standard Firebird generators for generating the LOw segment, and global setting of the HIgh segment using the configuration (e.g. a dedicated table). Then, for example, in the ON CONNECT trigger, you can store the segment's HIGH value in a context variable, and create a stored function that uses this value together with a

generator to create a primary key.

The disadvantage of this method is the inability to use the IDENTITY clause when defining a primary key column. Instead, you must create the appropriate triggers manually.

This method can also be used if you use plain BIGINT primary keys with local generation, and later decide to convert them to global identifiers.

If the following prerequisites are met, you can also make such a change directly in the database:

- No key value will exceed the maximum value that can be stored in the number of bytes reserved for the lower segment.
- You use custom triggers and generators instead of the IDENTITY clause.
- You have defined referential integrity with ON UPDATE CASCADE rules.

However, although direct key conversion is possible in this case, it will be quite time-consuming. The alternative is of course to create a new database and copy the data with continuous conversion of key values.

Snowflake ID

`Snowflake IDs`, or `snowflakes`, are a form of unique identifier created by Twitter (now X) for the IDs of tweets.

They are 8 bytes long binaries, so they fit into BIGINT type. The first 41 bits are a timestamp, representing milliseconds since the chosen epoch. The next 10 bits represent a machine ID, preventing clashes. Twelve more bits represent a per-machine sequence number, to allow creation of multiple snowflakes in the same millisecond. The final number is generally serialized in decimal.

Snowflakes are sortable by time, because they are based on the time they were created. Additionally, the time a snowflake was created can be calculated from the snowflake. This can be used to get snowflakes (and their associated objects) that were created before or after a particular date.

There are several slightly different variants used by different projects or companies (such as Discord, Mastodon or Instagram), so there are plenty to choose from. You can then find a variety of implementations for different

languages on GitHub.

Snowflakes are very suitable for use as primary keys in a database, and seem to be an ideal replacement for UUIDs in the future. Unfortunately, their use is currently not the most convenient, because no standard has been established, nor is support available in Firebird (which we will hopefully see in the future).

To have a constraint or not

Just as not every table needs to have a primary key defined, neither does an appropriate constraint need to be defined for every primary key. You can achieve the same effect by defining a NOT NULL constraint on all columns of the key, and by creating a unique index with the given key.

> *The only real reason to define a constraint on primary key is that you cannot create a foreign key constraint without it. It follows that if the given table is not referenced by a foreign key, it is not necessary to create a constraint on the primary key.*

But why not simplify your life and create constraints for all primary keys without distinction?

One of the reasons could be, for example, the fact that an index bound to a constraint cannot be deactivated, and therefore not rebuilt (reoptimized) without having to delete (and recreate) the constraint. Reoptimization of such indexes is therefore a rather complex process, and running it occasionally is important when using a certain type of keys, for example UUIDs. Unfortunately, Firebird currently does not allow disabling and re-enabling constraints (cascading propagation to foreign keys in the case of a primary or unique key would be handy), which would make this operation much easier.

Another (supplemental) reason may be that although the table is referenced from other tables, these references do not require foreign key constraints (since their functionality is implemented in a different way), and only an index on the foreign key is sufficient to speed up queries. And where there is no foreign key constraint,

there may not be a primary key constraint.

However, none of the above means that you shouldn't define a primary key constraint on all tables with a primary key. On the contrary, it is a recommended strategy. We present this alternative only to clarify the technical reasons and consequences behind this choice. Many young developers are not aware of them and mindlessly perform the learned rituals without understanding their meaning.

Which constraint

If you define a constraint on primary key, you have two options: define a PRIMARY KEY constraint or a UNIQUE KEY constraint. Both constraints ensure that a key is unique within a table with the important difference that a unique constraint allows a key to contain one or more NULL values and can therefore be created on columns that do not have the NOT NULL flag, while a primary key constraint does not allow NULL values and all key columns must have the NOT NULL flag. Another difference is that a table can have only one primary key constraint, but it can have multiple unique key constraints.

When defining a FOREIGN KEY constraint, Firebird does not discriminate between a primary or unique key constraint, and foreign key constraints can therefore be linked to both types. This feature is important because it allows you to create solutions that benefit from referential integrity but:

- allow an alternative (typically natural) key to be used for referencing.
- allows you to assign a meaning to a NULL value in a reference because it can refer to an existing record in the primary table. This has a significant impact, for example, on the results of inner joins, where a null value in a foreign key causes a row with NULL to be omitted from the result, and an outer join must be used to prevent that.

> *The choice of constraint used for the primary key is therefore a crucial choice, determining your options and the way you work with the stored data.*

Final words

Here is a brief summary of key points and recommendations:

- Do not forget the difference between PRIMARY KEY and primary key CONSTRAINT.
- It is advisable not to define artificial primary keys for any table for which a natural primary key cannot be defined.
- Artificial keys are ideal for processing, but absolutely useless from a user's point of view.
- Artificial keys are better than natural keys, but they come at a price. Without compensation strategies, you are setting yourself up for big problems in the future.
- Choosing the type and method of generating the primary key is an absolutely fundamental decision that cannot be easily changed, but always has consequences that may not always be clear.
- The best key is: simple (one column), easy to generate and process, doesn't take up much space, and has good indexing properties. If it can carry information other than uniqueness, that's a huge bonus.
- Before you define a constraint, it is better to think twice.
- The PRIMARY KEY constraint is not the only option. The UNIQUE constraint has its important uses.

Interview with Pavel Zotov

A database server is complex software, and the requirements for its reliability are extreme. In the last issue, you could get acquainted with the tool used in the development of Firebird to test its functionality. In the following interview, you can look under the hood of the entire quality assurance process and meet the man for whom Firebird quality is above all else.

Hi, can you introduce yourself to our readers? Where do you live and how was your journey to Firebird?

My name is Pavel Zotov. I live in Russia, in the Moscow region. I have been working with Firebird since 2009, although my first experience with this DBMS appeared in 2002 (that was the time of InterBase 6).

If I remember correctly, you joined the Firebird project in August 2013 when you got a grant from Firebird Foundation to work on Firebird QA. How do you see your beginnings in the project? Did your expectations come true, or did you imagine something else?

Yes, I started working as a QA engineer in August 2013. At that time, Firebird was in intensive development of version 3.0 (Alpha1). I remember that my first work was about performance problems that I had previously found and which were published on the Russian DBMS forum `sql.ru`.

Lot of examples were taken from my own posts on that forum. There is beautiful education site, `sql-ex.ru`, where I have studied SQL. Many complex SQL queries (from its world-open part that is called 'Training Stage', with permission of this site owner) have been adapted to QA framework. It was done mostly for verifying 'robustness' of Firebird parser/compiler and correctness of SQL results. I have noticed that some of these queries produced wrong output.

Some tests have been ported from InterBase and Postgres support forums. Also, there was a lot of work to verify correctness of new features introduced in Firebird 3 (Srp authentication, encryption etc.). Some complex SQL tests that deal with new Firebird features (first of all window functions) have been added at that time. Also, hundreds of relatively simple tests were implemented after investigation in old Firebird tracker, and obtaining list of fixed tickets which have not appropriate test.

It was very interesting for me to work on these tests. Although it was time consuming. But these tests were created quite quickly because I could discuss any issue with the Firebird developers using our native language.

Your first contributions to Firebird QA were for the OLTP-EMUL benchmark, and you've been maintaining it ever since. What role does this project play within the development of Firebird?

The first idea for such a test was born long before I joined the Firebird project, maybe in 2010. At that time I was working in big trade-servicing enterprise, developing an OLTP application for them (however, it used a very old programming language). This application allowed to make a lot of operations related to stock, logistics, servicing, paydesk, accounting department etc. A lot of effort has been put into ensuring acceptable scalability and performance, and of course getting the results right, including arithmetic.

When I started working on the Firebird team at the same company, the performance issue was the first and most serious thing we encountered. Of course, we made some "stupid mistakes" at the initial stage, and many people from the Russian-speaking Firebird community helped us. I would especially like to thank Vlad and Dmitry for a lot of useful suggestions.

After I joined the Firebird project, it became clear to me that some benchmark ideas could be taken from the old system I used in my work. I discussed this with Dmitry Yemanov, and he pointed out the most important things that are desirable in such a test, as well as what to avoid (security, user rights, i18n, etc.). After about 6 months, this test was implemented and most of its algorithms were similar to those that worked in the real application.

But also (and this was quite surprising to me) almost 50 bugs were found in Firebird 3, which was being developed at the same time (not yet officially released). So I hope this test has become useful to achieve the stability of the first release of version 3.

In addition, this test turned out to be suitable for investigating problems related to performance regressions after the release of Firebird 3. I remember that such question came up at least 3-4 times, specifically: we received reports from various customers about a performance problem, but it was not clear when such regression appeared. So I ran the test on different builds and compared the performance scores. Using this, several regressions were found accurate to a specific build date/number.

In 2019, we (the iBase company) had the opportunity to use a fairly powerful server of one of our customers, which allowed us to run an OLTP-EMUL test regularly using a scheduler. I have developed some new reports to compare results between Firebird 3 and 4, you can see them at https://firebirdtest.com/oltp-emul/

This report was created primarily to detect "slow regression", i.e. when the performance drop has a trend, but is difficult to detect by comparing the results of several adjacent runs. Unfortunately, we are currently unable to use this server, so this report is out of date. There is also no data for Firebird 5 as it was performed in 2021. But I hope we can resume this activity on our own hardware soon.

At the time you joined Firebird QA, the project was using a homegrown system (fbtest) written in Python. Was it difficult for you to learn to work with it? What surprised you the most, pleasantly and/or unpleasantly?

I used to and still believe that FBTEST is **very** simple yet powerful framework!

But it took me a while to get used to some specifics, namely: the necessity to strictly switch to the appropriate folder before running the tests and to use test names instead of test filenames/paths.

However, over time, several problems arose. First, the FDB driver (which fbtest uses to communicate with Firebird) did not support the new data types that were implemented in Firebird. Also of concern was the problem with the teardown phase of some tests (mainly due to the LINGER feature in SuperServer). I also missed some useful features that are now in the new QA framework, namely: comparing firebird.log content collected before and after some actions; no built-in ability to give each line that is executed an exact timestamp (on Windows you had to use something like mtee.exe); no option to specify client library path (fixed later); the result of running fbtest did not affect the value of ERRORLEVEL (this was fixed later); no option to specify something like "skip if ServerMode = 'Classic'"; no option to quickly extract metadata etc. The funny problem that show up for time to time was the need to look for a non-ascii character in some part of the test that could be entered occasionally: it might be a test that I didn't check at the moment, but was edited few minutes ago (say I changed a comment etc. in it).

A few years ago, Firebird QA was converted to a new testing system based on pytest. How do you rate the new system compared to the original one? What has improved or worsened for you?

I think it was definitely a big step forward for us. And the first reason for this is that our new quality control is now based on a widely used (industry) framework - pytest. We can take advantage of a lot of its features and plugins, including py.mark, py.skip/skipif, console output style, settings that are common to a number of tests (eg encryption plugin name, values for setting replication parameters, etc.).

The new Firebird-driver that replaced FDB now also supports all Firebird data types and most functions (currently only the batch API is missing).

Also, the new command switch "--extend-xml" is very useful for generating detailed reports about the QA run.

There is a new useful feature in the QA plugin: we can skip recreating the test database before each next test and run a file-level copy of an already existing one. All such empty databases are collected in a special folder ('$QA_HOME/dbcache'). This capability greatly reduces the overall QA runtime, even though we have to clear the cache directory before each new run. It is on by default, so if you don't want it, you have to use the '--disable-cache' switch.

After much discussion, a solution was found for the problem related to using databases with non-default settings, e.g. that are involved in replication or must be self-security (this is now done using the pre-built "qa-databases.conf" file in the QA_HOME/files directory).

However, some problems still exist. We need to escape backslashes even if they are in comments. Also, pytest will fail and the entire test suite will not be executed if any file contains a non-ascii character in the wrong place. More seriously, firebird-driver still has some problems when we try to run complex python code that calls a function and tries to use instances of the `Connection` and `Cursor` classes there (this can cause python to crash!). This issue occurred during a migration from the old to the new QA framework. However, a workaround was found (it's necessary to pass the above instances to the function as parameters).

Readers are surely interested in how your work fits into the overall framework of Firebird development. Can you explain how Firebird QA works, and what is your usual routine?

First of all, I would like to sincerely thank my supervisors (Dmitriy Kuzmenko and Alexei Kovyazin). They understand that I participate in the Firebird project as a QA engineer and have to spend valuable time running tests, verifying some issues, doing something based on requests from Firebird developers, etc. They allow me to handle this during my working hours - but of course only if there is nothing urgent related to our customers. They also know that any bugs that are detected in the standard version of Firebird will eventually be carried over to our commercial product - the HQBird family. For that reason, my quality control effort can be considered something of a "filter" that prevents many bugs from appearing in HQBird.

The QA framework and some useful tools (batch scripts) are installed on several computers. Two of them (with Windows and Linux) belong to iBase and two are my own. Also at iBase we have dedicated storage for all the builds that are created after each new commit in the master/v5/v4/v3 branches. This makes it possible to very quickly "jump back in time" to confirm whether a problem can be reproduced or not.

I have an account on the Firebird github repository and am subscribed to all commit information related to core development (sent to my email). When I see a commit with a message like "Fixed..." and it's related to the Firebird core, I open the relevant tracker ticket and see if it's related to QA. If a test seems too difficult to implement, or if I can't reproduce the problem, that test is marked as "deferred". In other cases, this test may be labeled as "not enough information" or "cannot reproduce". After some time (months or even years) I review all such tests and many of them no longer seem so difficult to implement.

The simplest case of course is if the problem contains some SQL/PSQL code that illustrates the problem and gives the build number. If I can reproduce the problem using given build (or build recent for the date that was given in the ticket), I will check whether this problem is actually fixed or not. If so, I'll create a test that almost always looks like the ticket, verify it, and write a comment about which build had that problem and where it actually got fixed. Finally, I push the test file to

to our QA repository.

But of course it's often not that simple. Currently there are a dozen tickets with missing information about the build or some parameters, etc. If I don't know how to reproduce it, I will write a question to the Firebird developers. In almost all cases, they can provide an answer that helps me reproduce the problem. But only if I don't delay too much with my question (I mean we have to "follow fresh footprints").

Also, from time to time some Firebird developer asks me to somehow verify the code that they don't want to commit to the Firebird repository yet (because they think it's premature). This could be about various things: encryption, authentication, query optimizer, cache usage, etc. Usually no tests are created after this task. Only results and conclusions are sent to the Firebird team.

Some tests were implemented after reading forums / blogs of other DBMS (Interbase, PostgreSQL, Oracle). There are many tests that have been adapted to our framework from one we inherited from InterBase, named 'GTCS' (AFAIK, still supported by Adrian and Alex).

In a separate group are tests related to performance problems. We CANNOT rely on the ratio of time values that were measured by the 'clock' for some code before and after the patch. I mean we can't use either the 'elapsed time' in the ISQL output or the datediff() result due to the presence of concurrent load on the test machine during this test. The only way that seems reliable is to measure CPU 'user time' values, which can be obtained using the `psutil` package. However, such tests can fail from time to time - mostly due to wrongly chosen ratio threshold values.

About my usual routine. As for QA itself - it's… pytest (gee). If I want to find a test or ticket that has already been fixed and looks similar to the one we're currently investigating, then of course I'll use github and its search mechanism (which allows for fuzzy searches). I use FAR Manager which is extremely convenient for working with files/commands etc. I also find its text editor quite comfortable with the text coloring scheme turned on. I use a simple batch file to submit tests, which (beside of git actions) sends me a letter with details - and I use such letters when creating QA-reports.

What is your strategy for creating tests for Firebird? How do you go about deciding what to test and how?

First, if the issue recorded in the Firebird tracker is fixed, I will receive a notification in the email. Of course, I pay attention to issues that provide a SQL test case, or at least have a description of the problem and the exact build number of the affected Firebird build (ie in full "5.0.1.1477" format). If the issue has too little detail or cannot be tested at all, I mark it in Firebird tracker with "lack of information" or "cannot be tested". If the problem can't be reproduced (or I don't know how to do it), I'll try asking the Firebird team about a way how to do it. A test will never be submitted to our QA repository until it is reproducible on at least one of the major Firebird versions.

In some rare cases, the test needs to be implemented using a completely different scenario than the one listed in the ticket. In particular, it can be a scenario that looks like a brute force attack. For example, tests that check for issues with Unicode characters; or a test that does a lot of combinations with scrollable cursors; or a test that verifies the impossibility of accessing RDB$ tables, etc.

When test for a ticket is done and committed, I will mark it in ticket in order to see only 'non-resolved' ones easily.

I also check for new posts related to the `$FB_HOME/doc` directory as they contain information about new features that need to be tested as well. If any issue (in the tracker) or example (from `$FB_HOME/doc`) contains SQL/PSQL code, I'll try to create a test as soon as possible. Otherwise, if I think something like this has already happened, I'll look for it in the existing tests. If the problem seems to be "completely new and complex", I will try to first implement a test "prototype" using only Python code (ie without the QA-plugin). A lot of information/examples related to Python can be found on StackOverflow and other sources, so the "main problem" is getting the question right for Google (wink).

Do you see any flaws in how Firebird quality control is currently done? What would you like to improve?

I cannot judge the quality of my work. Rather, it should be done by those who benefit from its results (smile). At the end of 2019, I decided to re-implement the scheme of how QA results are stored and - most importantly - how they are displayed. After many discussions with Firebird developers, the current 'cross-reporting' view was born (see `firebidtest.com` and jump to any page with specific results for the OS / FB family). It allows (I hope) a quick assessment of the last 30 Firebird builds on one page. Number of columns with pushed SHA (which uniquely identifies concrete snapshot) can be easily increased up to ~85. This cross-report can also be used to quickly find the build that caused a particular regression. For each failed test, you can see a complete history of how that test was completed.

There are also tables with the time spent to run each test. However, we still don't see (in a suitable form!) performance regressions in our tests. Last year I made several attempts on this and applied complex algorithms from statistical analysis. It showed me that performance regressions **mostly** look like sudden jumps (spikes) rather than smooth trends. But I haven't implemented a proper procedure to detect them yet. That is my task this year.

Another thing I want to implement is the ability to see the results of a (selected) test in **one** table for all processes from the Firebird family, ServerMode values and operating systems (Windows / Linux).

Many consider testing and QA in general to be the most boring and unrewarding IT job. Is it true or do you enjoy it?

I completely disagree with such an opinion. When I was developing my own OLTP application, finding errors took about 70% of the time. And it was a real pleasure when I finally found that some code no longer had any problems/bugs after the effort. I saw it as a reward for hard work. So my current job definitely satisfies me.

Can you enlighten us about the popularity of Firebird in your country, and how it is with the Firebird user community there in general?

Firebird was and is popular in Russia. Maybe for historical reasons (because since the 90s there have been a lot of small businesses that didn't want to pay for a commercial DBMS), but I think also because the product is really easy to work with. Recent years show that the number of companies using Firebird is constantly growing. Two conferences organized by RedBase proved it.

It is also very difficult to find other DBMS that have such support from the main developers, especially for people who speak Russian. I will give just one example from my own experience. In 2011-2012 (when I worked at the previous company) we were tasked with migrating from Firebird to Oracle. We soon ran into some code example that caused Oracle to crash (smile). We looked for a solution but couldn't find one. When we asked Oracle Support (it's called "metalink") about the problem, it turns out that they know about this bug, but it's been around for… about 1.5 or 2 years! And it hasn't been fixed yet. Next, we figured out that if we want this bug fixed sooner, we have to pay Oracle **a lot** for it. Of course, we did not continue the search for a solution to this problem and moved on to other tasks (smile). I can't imagine the Firebird developers letting this situation go unnoticed and not trying to address it.

Since I started working with Firebird in 2009, I have noticed many times that almost all major bugs (including crashes of course) have been fixed within a few days. The longest the bug was not fixed (IIRC) was about 2-3 months (it was an index corruption issue, the infamous 'missing records'). Such a high level of support from its developers (over many years) is a good reason to choose this DBMS for doubters.

Thanks for you time!

Development update: 2024/Q3

A regular overview of new developments and releases in Firebird Project

Releases:

- Firebird 5.0.1, released 2.8.2024
- Firebird 4.0.5, released 8.8.2024
- Firebird 3.0.12, released 8.8.2024
- Firebird ODBC driver 3.0, released 10.9.2024
- firebird-driver for Python 1.10.6, released 15.8.2024

Import library for Borland compilers

Historically, Firebird provided `fbclient_bor.lib` — import library that can be used with Borland compilers — in x86 Windows packages. However, it's missing in the v5.0.0 release. The reason is simple — the build script didn't attempt to make this library if `implib.exe` is missing and Firebird GitHub environment obviously misses it.

It has been decided that this library will no longer be part of the v5 packages and this change is now documented in the Release Notes.

JSON support for Firebird 6

In July, Red Soft submitted the first part of its proposal for implementing JSON support in Firebird 6 for discussion. The main points of the proposal are:

- Based on the JSON SQL standard 2017 (ISO/IEC TR 19075-6:2017).
- There is no explicit JSON data type (which appears only in SQL 2023) or a binary JSON type. JSON is stored as text via text types.
- The implementation adds JSON functions, JSON_TABLE record source, and all its associated components.
- All functions are named according to the SQL standard.
- Nearly all functionality has already been implemented in RedDatabase and is awaiting submission for merging. To avoid merging over 15,000 lines of code at once, the work has been divided into eight parts. Parts 1-3 focus on JSON generation functions, parts 4-6 handle JSON extract (query) functions, and parts 7-8 cover JSON_TABLE.
- The JSON code is separated from the engine as much as possible to facilitate its unit testing.
- Internally, all the basic JSON functions (JSON_VALUE, JSON_QUERY, JSON_ARRAY, and JSON_OBJECT) are implemented as expression node, JSON_TABLE is a record source node and JSON_ARRAYAGG and JSON_OBJECTAGG are aggregation nodes.
- All the functions produce compact JSON. Therefore, all excess spaces, tabs, and new lines will be omitted.

- Certain aspects of the standard are not very user-friendly, so a few additional features will be introduced that extend beyond the standard.

The subsequent discussion among developer covered various aspects of the design and the optimal implementation approach. Key topics included the internal representation of JSON values, the efficiency of storing and processing them, and potential future extensions, such as binary JSON, the JSON type in the 2023 standard, and indexing.

You can read the whole discussion in `firebird-devel` googlegroup under *"Proposal for SQL support for JavaScript Object Notation (JSON). Part 1"* thread.

Tablespaces for Firebird 6

In September, Red Soft submitted the updated proposal for implementation of tablespaces in Firebird 6 for discussion.

Syntax

TABLESPACE

```
CREATE TABLESPACE [IF NOT EXISTS] <TS NAME> FILE '/path/to/file'
ALTER TABLESPACE <TS NAME> SET FILE [TO] '/path/to/file'
```

Either absolute or relative path is allowed.

```
DROP TABLESPACE [IF EXISTS] <TS NAME>
```

The development of the INCLUDING CONTENTS option has been postponed.

For an existing tablespace, it is possible to add a comment using the COMMENT ON statement.

```
COMMENT ON TABLESPACE <TS NAME> IS {'text' | NULL}
```

TABLE

```
CREATE TABLE ... [[IN] TABLESPACE {<TS NAME> | PRIMARY}]
```

It is also possible to specify a tablespace when creating a column or table constraint:

```
<column/table constraint> ::=
 ... UNIQUE ... [[IN] TABLESPACE {<TS NAME> | PRIMARY}] |
     PRIMARY ... [[IN] TABLESPACE {<TS NAME> | PRIMARY}] |
       REFERENCES ... [[IN] TABLESPACE {<TS NAME> | PRIMARY}] ...
ALTER TABLE <TABLE NAME> SET TABLESPACE [TO] {<TS NAME> | PRIMARY}
```

The table data will be moved to the specified tablespace or to the main database. It is also possible to specify a tablespace when adding column or table constraints.

INDEX

```
CREATE INDEX ... [[IN] TABLESPACE {<TS NAME> | PRIMARY}]
```

By default, table indexes are created in the same tablespace as the table itself.

```
ALTER INDEX ... [SET TABLESPACE [TO] {<TS NAME> | PRIMARY}]
```

The index data will be moved to the specified tablespace or to the main database.

ODS changes

- A new table RDB$TABLESPACES
- New field in RDB$INDICES
- New field in RDB$RELATION_FIELDS
- New fields in RDB$RELATIONS

gbak

Backup works as usual for now. It gets data from a database transparently working with tablespaces.

Restore has new switches.

```
-ts_map[ping] <path to file>
```

Option is required for correct database recovery if its backup contains tables or indexes saved in tablespaces. To do this, specify the path to file, which consists of lines with two values: the first column is the name of the tablespace, the second column is the new location of the tablespace. You can specify either an absolute path or a relative path.

Example:

```
  TS1 /path/to/tablespace1.dat
  TS2 /path/to/tablespace2.dat
```

`-ts <tablespace> <path>`

Option allows to specify the path for the tablespace. You can specify either an absolute path or a relative path. The option can be used as many times as required. It can also be used together with -ts_map.

`-ts_orig[inal_paths]`

To restore tablespaces to the original paths they were on when the backup was created. It is still possible to override paths for some tablespaces using the -ts and -ts_map options. This is an explicit option, not a default action.

If you do not specify the above options when restoring a database that has tablespaces, an error about the inability to determine the path to restore tablespaces will occur.

Replication

There is an `apply_tablespaces_ddl` parameter for replication. If this parameter is disabled, tablespace-related DDL statements and CREATE/ALTER TABLE/INDEX clauses will not be applied to the replica. This is used if the replica has its own set of tablespaces or none at all.

Other

`ALTER DATABASE {BEGIN | END} BACKUP` will put not only the main database file, but also all tablespaces into safe copy mode. A delta file will be

created for each tablespace.

Command `SHOW {TABLESPACES | TABLESPACE <TS NAME>}` displays a list of all tablespace names in alphabetical order or information about the specified tablespace.

Limitations and future plans

- It's possible to create up to 253 tablespaces.
- Operators to move an index or table to a tablespace require an exclusive database lock.
- Red Soft plans to add support to move blobs into separate tablespace.

Toolbox: Red Expert

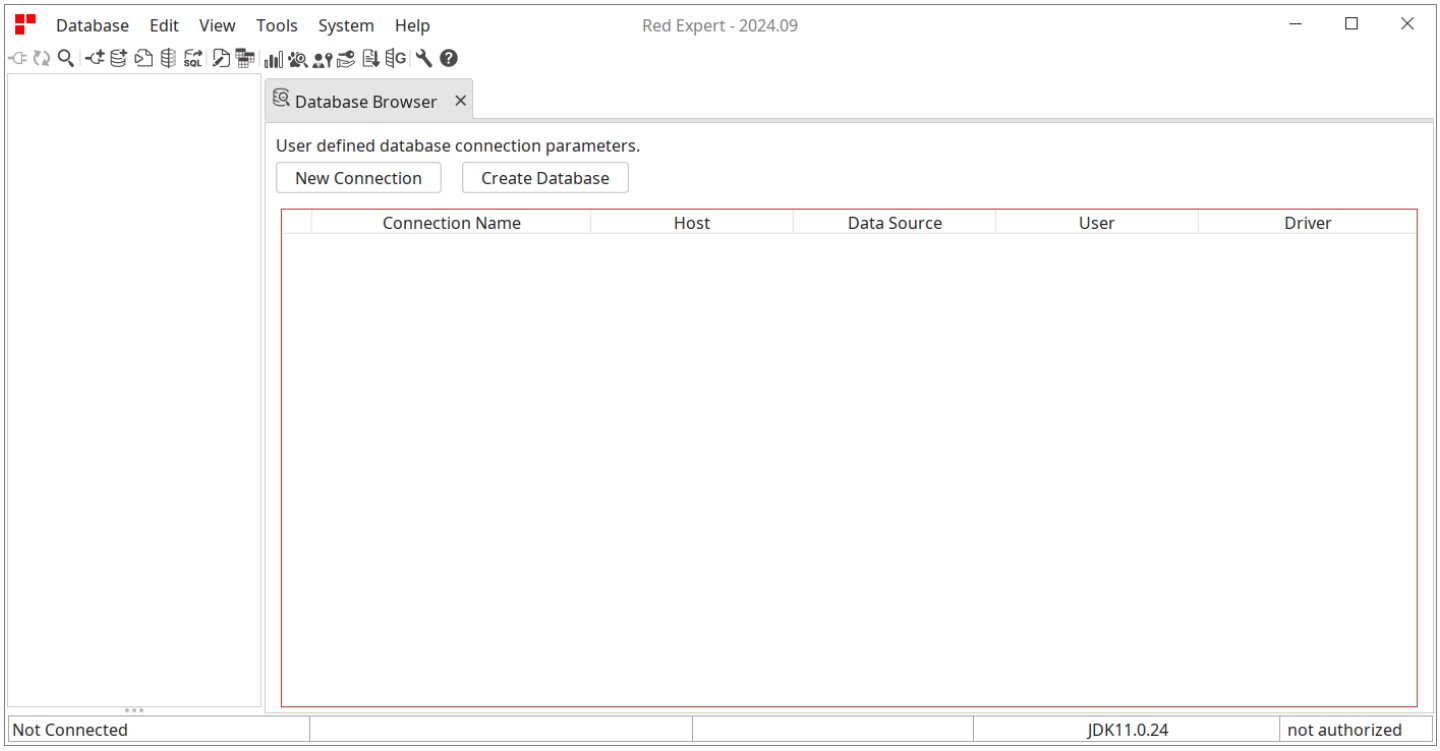
The Red Expert [1] is an open-source cross-platform database manager for RedDatabase and Firebird developed by Red Soft. It's written in Java and uses the JayBird drivers, with version 3, 4 and 5 of the driver directly included. It's licensed under GPL 3.0, and you can find it on GitHub [2].

We used the version 2024.09 for this review.

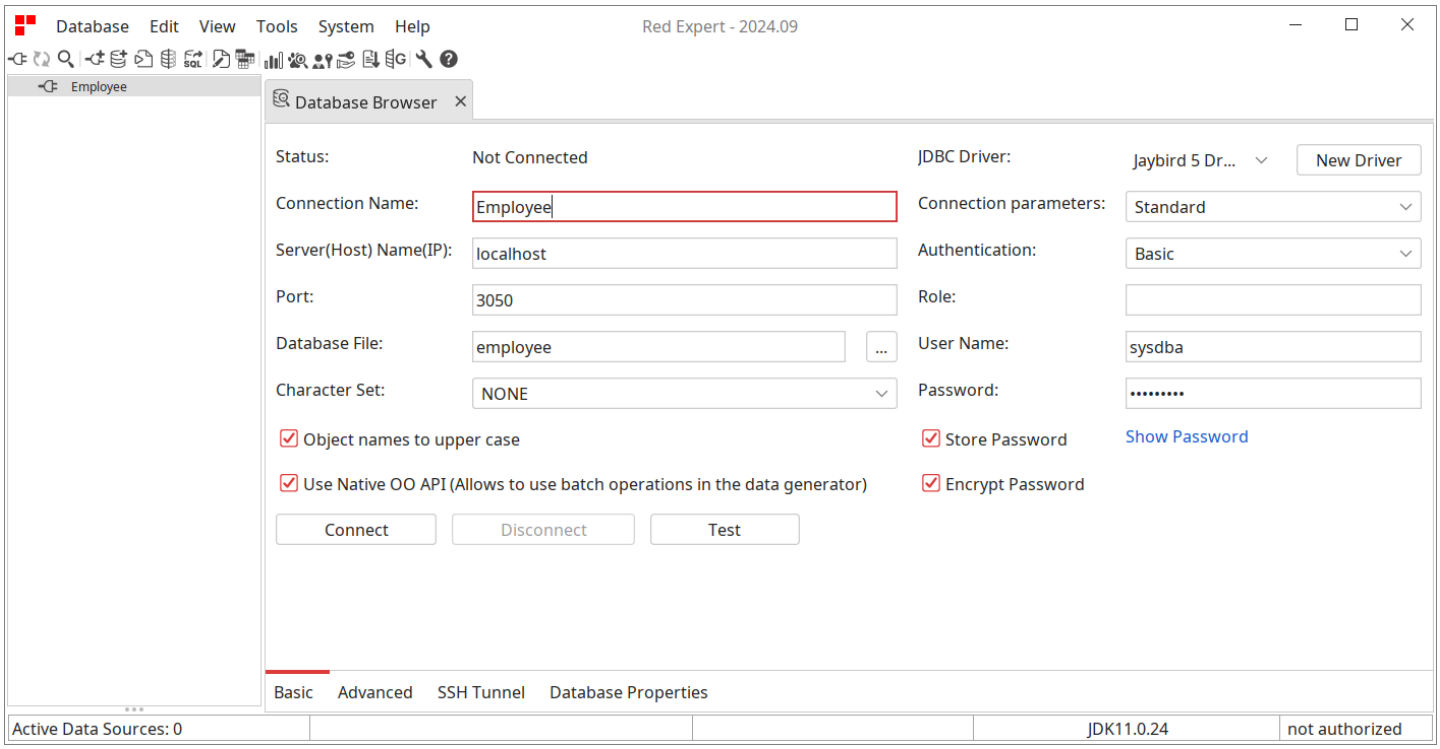
1. Red Expert download
2. Red Expert repository

First steps

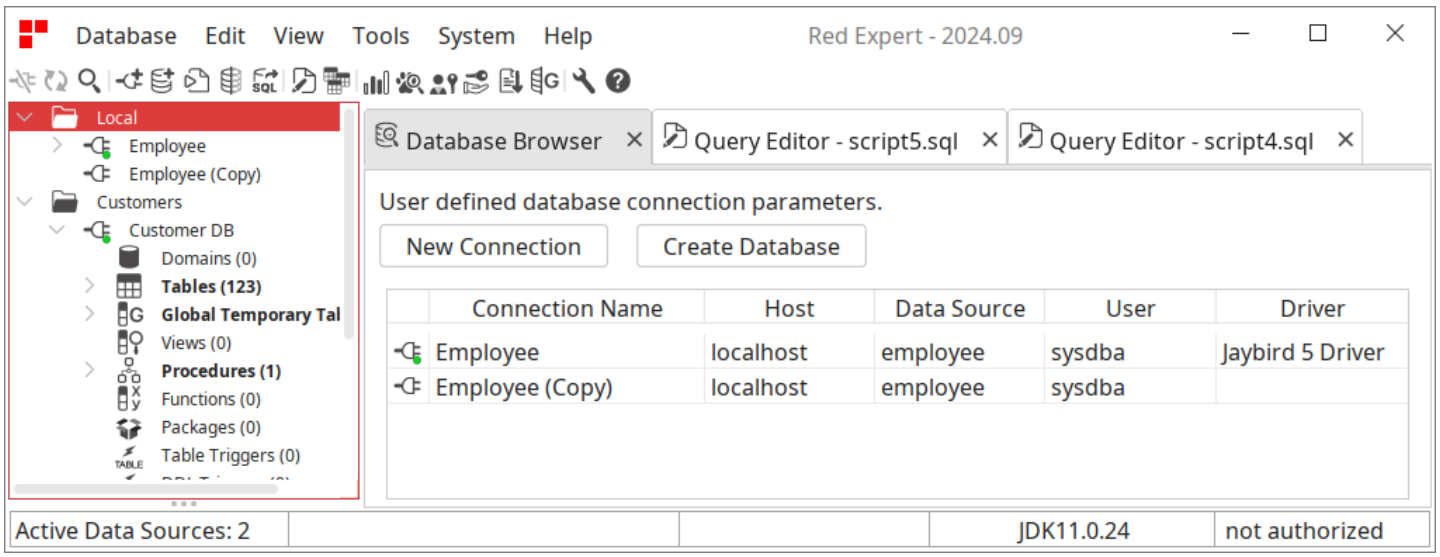
When you launch Red Expert for the first time, you will be presented with the Database Browser panel that is initially empty.



So your first step is to define a new database connection.



The left panel is used not only to organize connections, but also to display and navigate through database objects of connected databases.



Interface

The user interface consists of a single window with standard elements: a menu bar, a toolbar, a workspace with object views organized in tabs, and a status bar. Unlike *DBeaver*, which we reviewed in the previous issue, the individual panels cannot be detached, minimized, or moved—a feature that some may consider an advantage. However, the inability to reorder tabs within tab views is a drawback, as this feature could be quite useful. As a result, layout customization is limited to resizing certain panels using sliders."

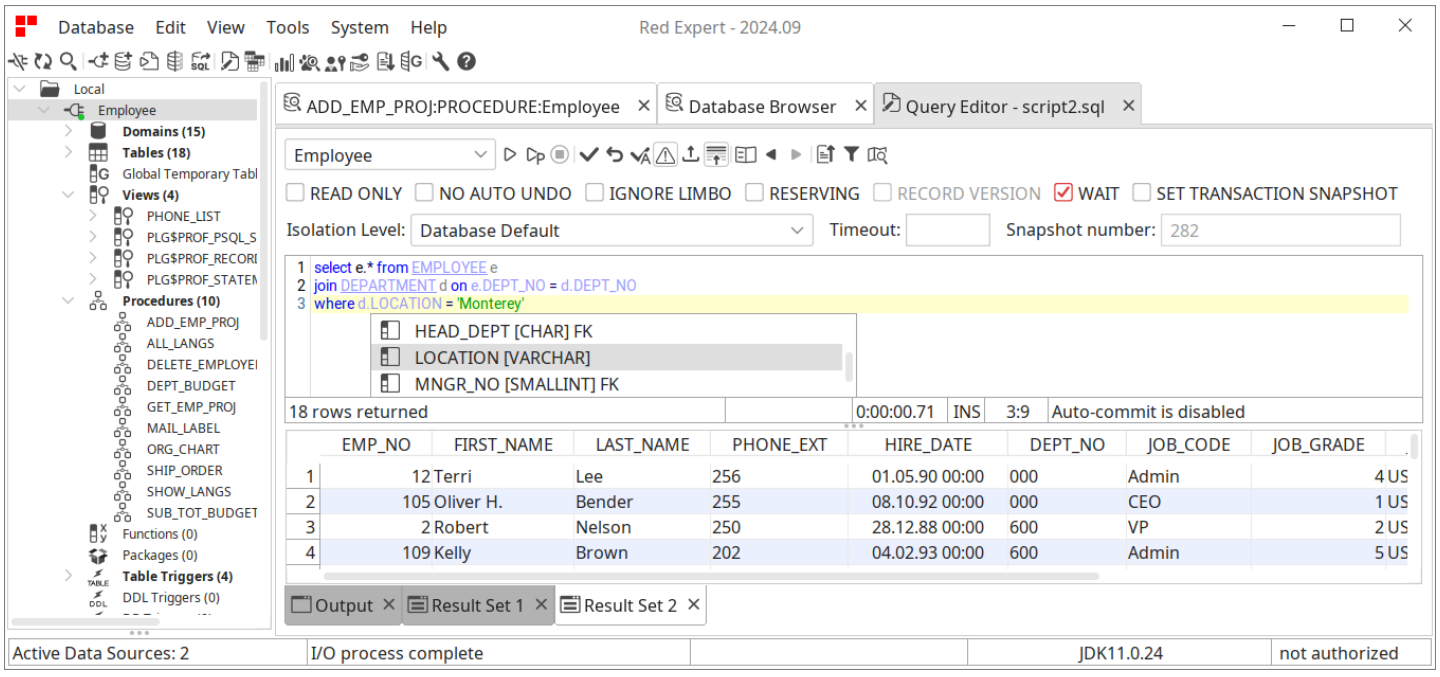
The interface appearance can be customized with one of six themes, including dark modes. Users can also configure keyboard shortcuts, customize colors in the editor and Result Set, and adjust toolbars, among other options. Overall, the interface is pleasant, clean, and easy to navigate. This is especially valuable considering that the documentation, including the built-in help, is available only in Russian.

Functionality

Red Expert is an ambitious, feature-rich tool designed for daily use, especially by database application developers, though database administrators will also find it highly valuable. It offers everything from a clear overview of database objects to tools for their creation and modification, data display, a robust query editor, script generation from metadata, an ER diagram editor, user and permission management, database validation, and much more. However, it lacks a dedicated feature for handling database backups.

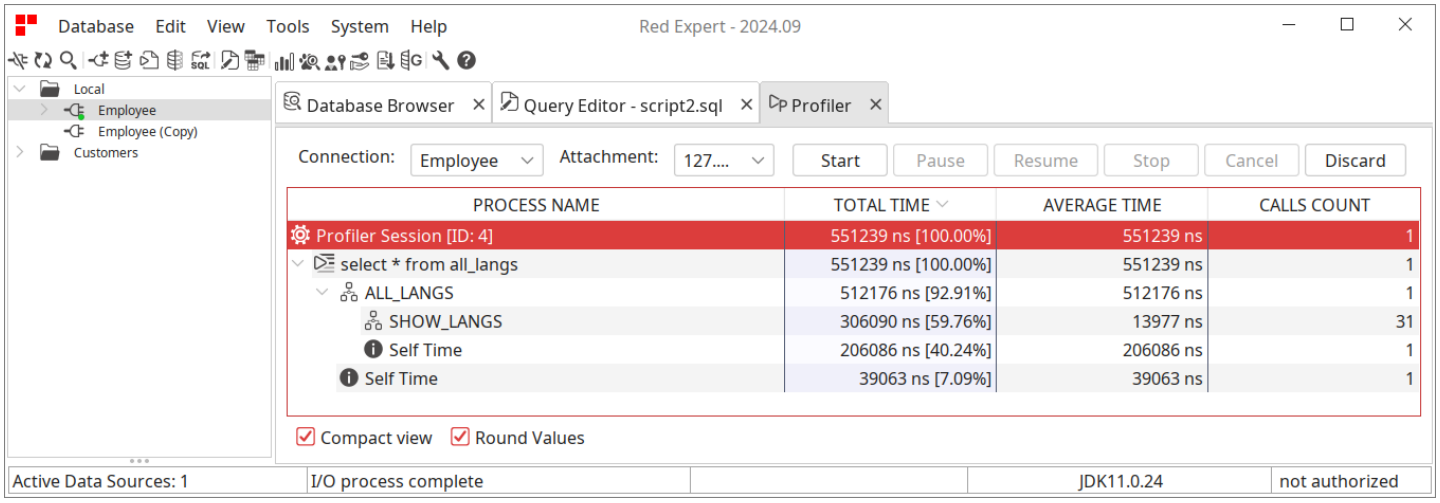
The SQL editor, a crucial tool for any Firebird user, is well-implemented in Red Expert, featuring auto-completion, shortcut-activated templates, and live object references. Execution statistics and execution plans are available in both standard and extended formats. Query results can also be exported as CSV, XLSX, XML, or SQL files.

Two standout features of the editor are worth highlighting. First, each execution of an SQL command opens a separate tab for the Result Set, allowing users to manage multiple outputs independently. You can easily switch between these tabs, and hovering over a tab reveals a window with the corresponding SQL statement, enabling quick navigation between outputs or copying commands to the clipboard or editor.

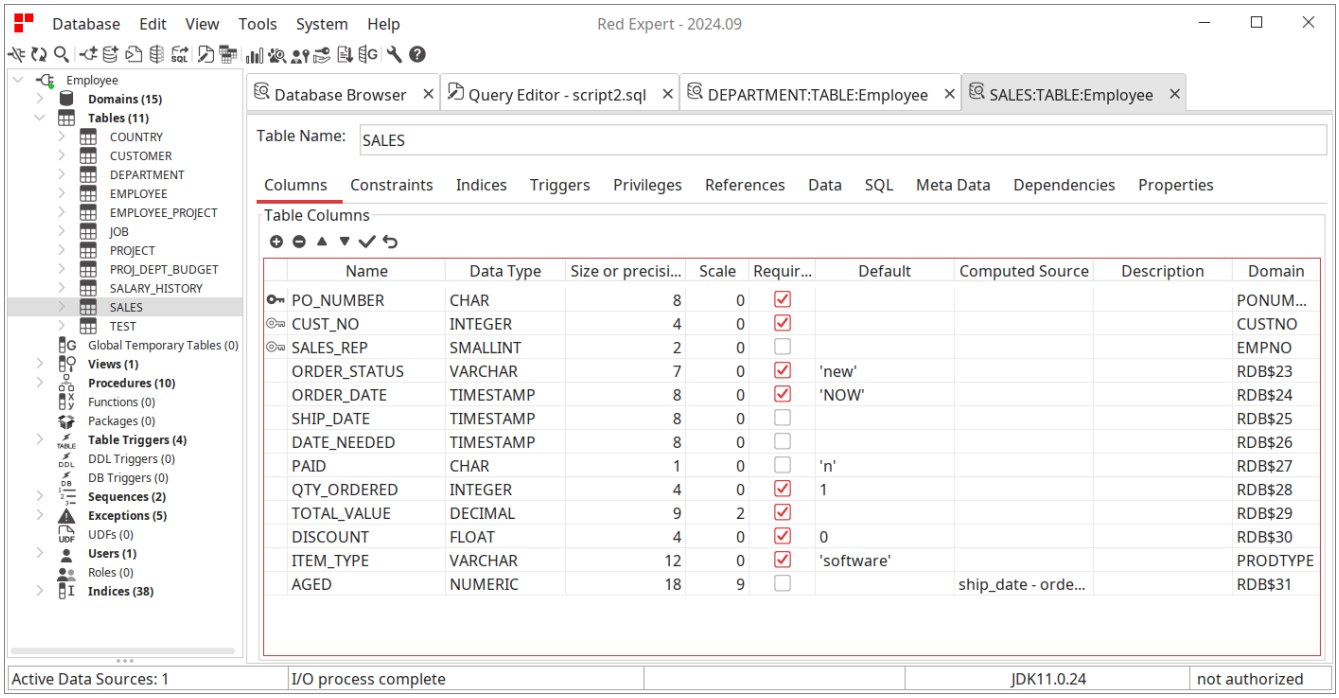


However, the query output display (which applies to tables as well) is fairly basic, offering only column sorting and reordering. It would be beneficial if future updates included more advanced features, such as column aggregation.

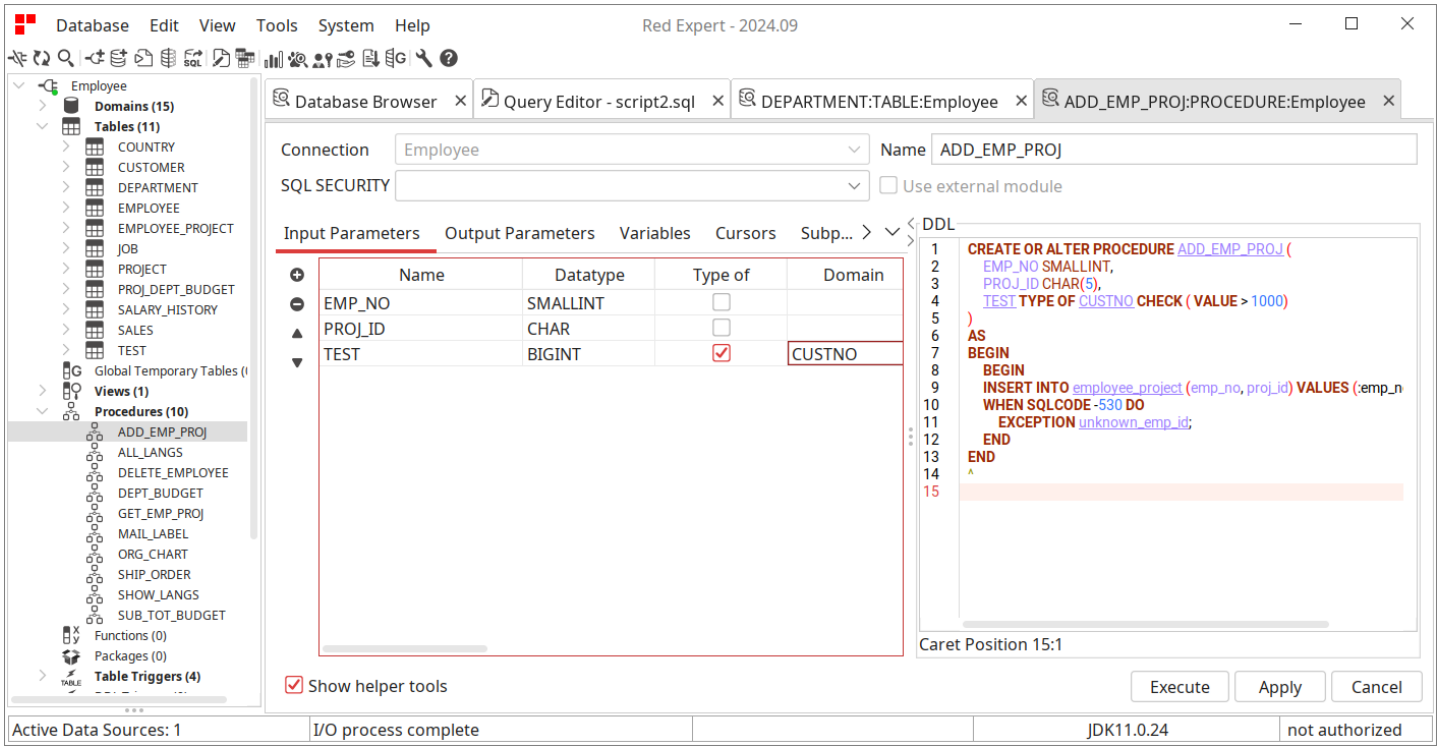
The second outstanding feature is the ability to execute SQL commands directly within the profiler, a functionality developers are sure to appreciate.



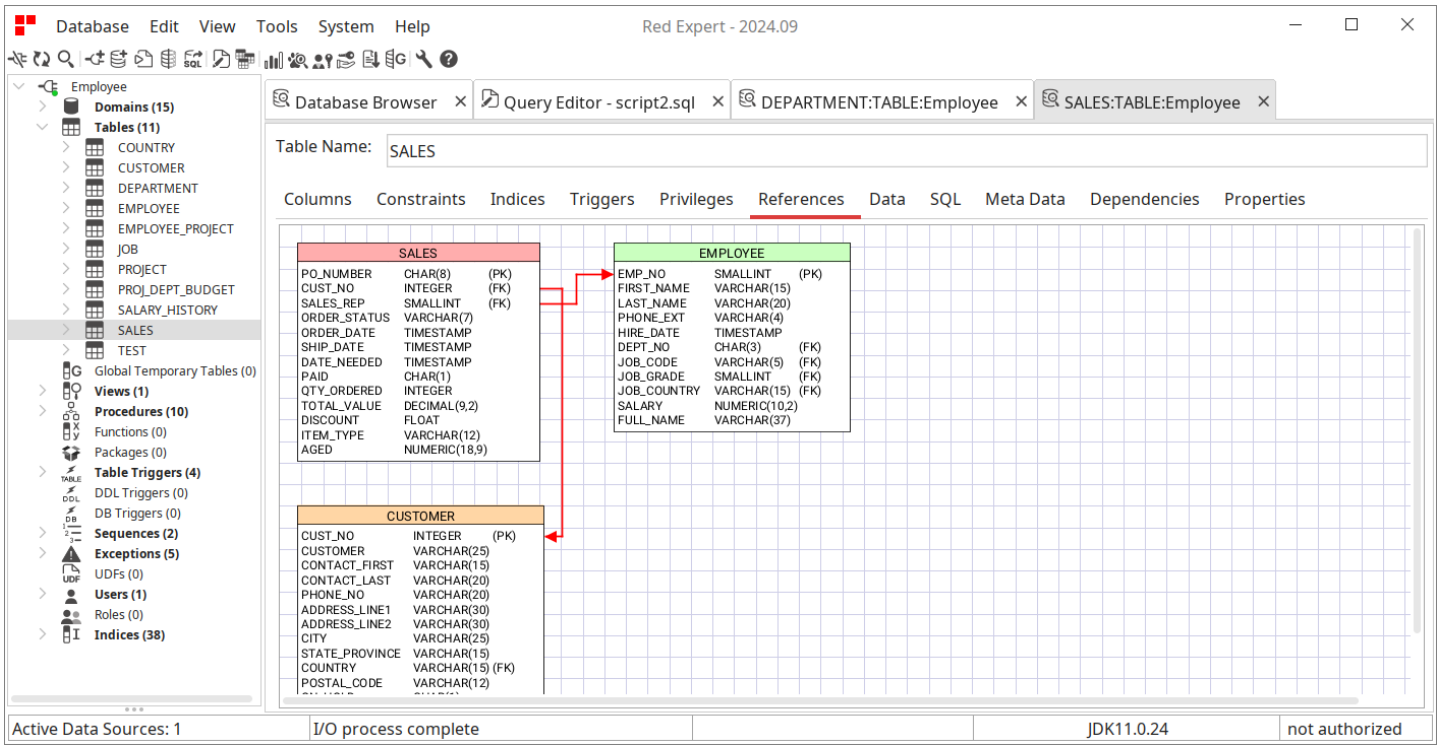
The display of individual database objects follows the standard format seen in similar tools, making navigation straightforward. However, some displayed information and properties apply only to RedDatabase and are inactive when working with Firebird. That said, since many RedDatabase features are gradually being implemented in Firebird (for example, tablespaces are planned for Firebird 6.0), the inclusion of these features can be seen as a forward-looking advantage.

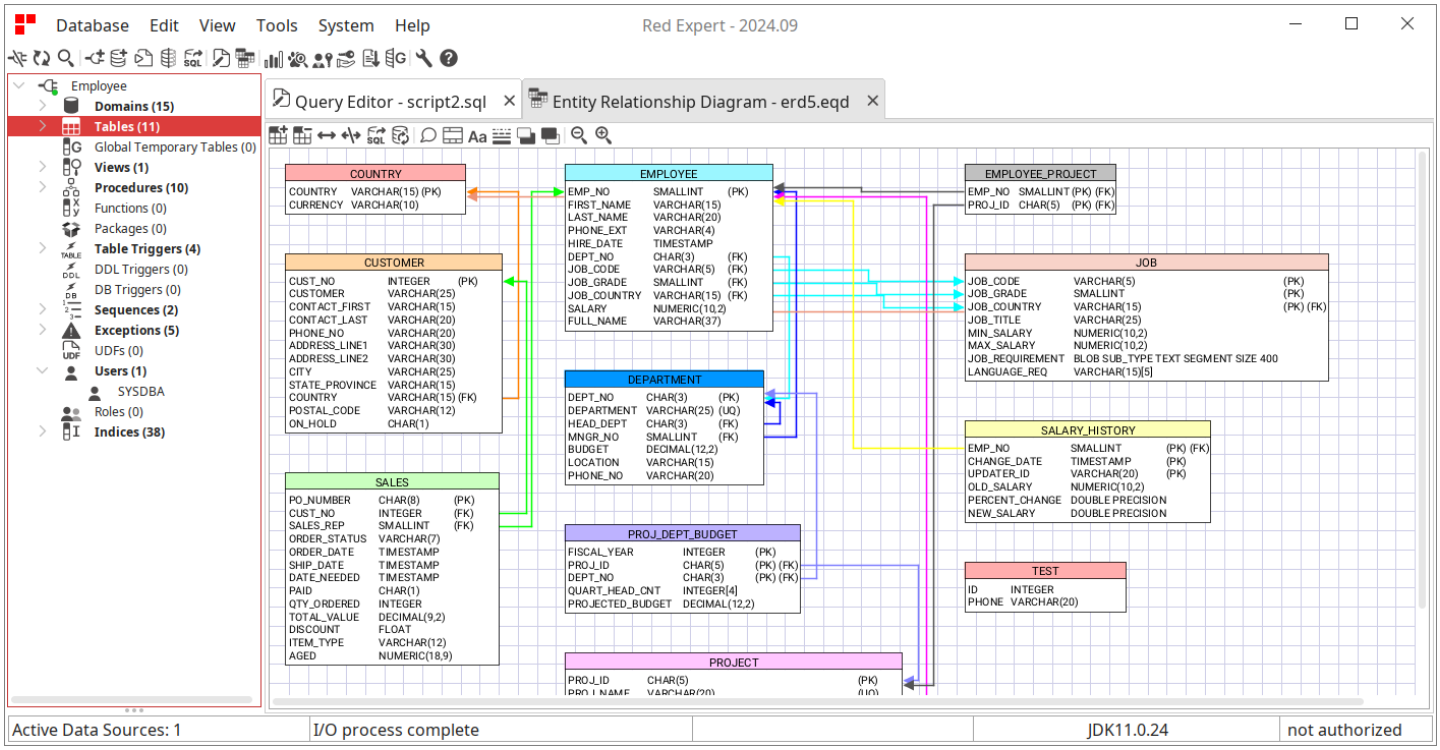


A particularly pleasant surprise is the stored procedure editor, enhanced with useful tools for editing parameters, variables, cursors and more.

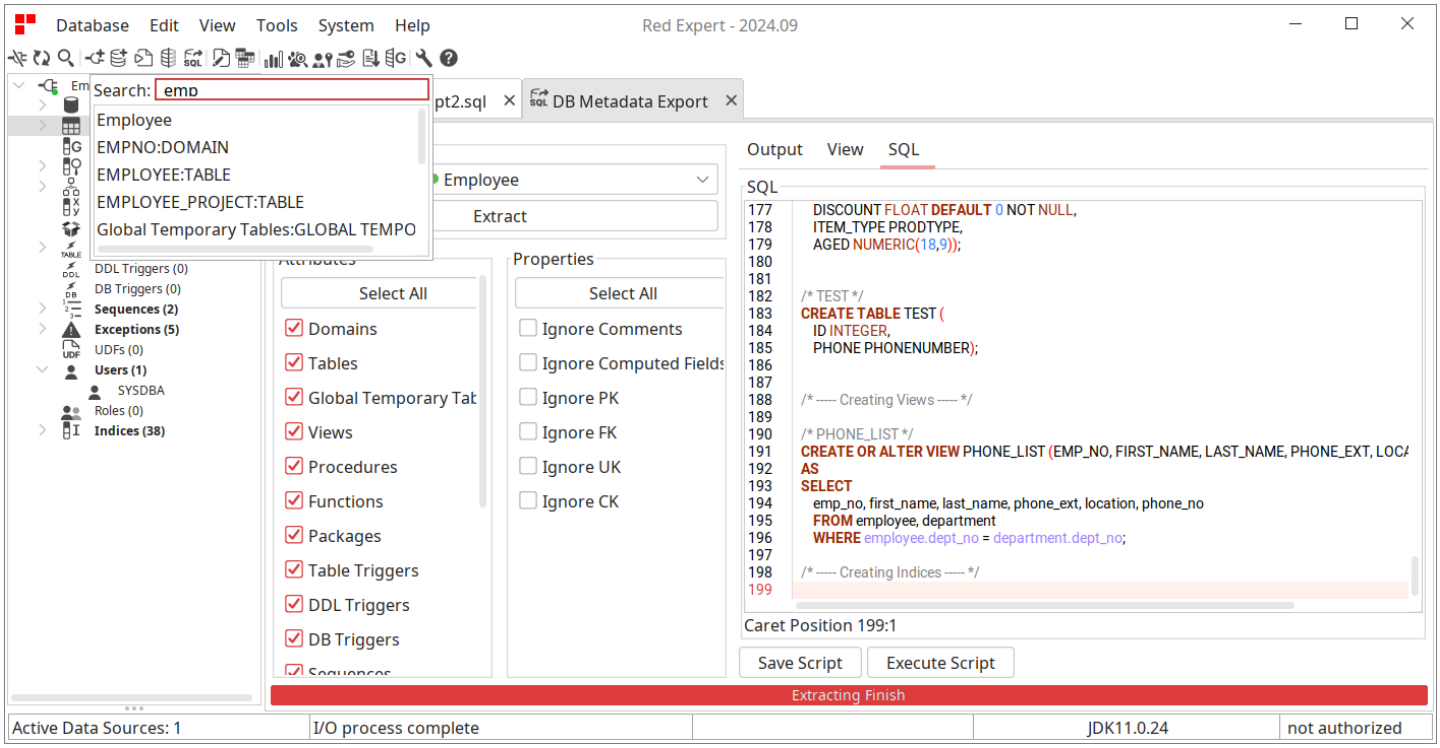


ER diagrams are available for visualizing relationships, both for the entire database and for individual tables.

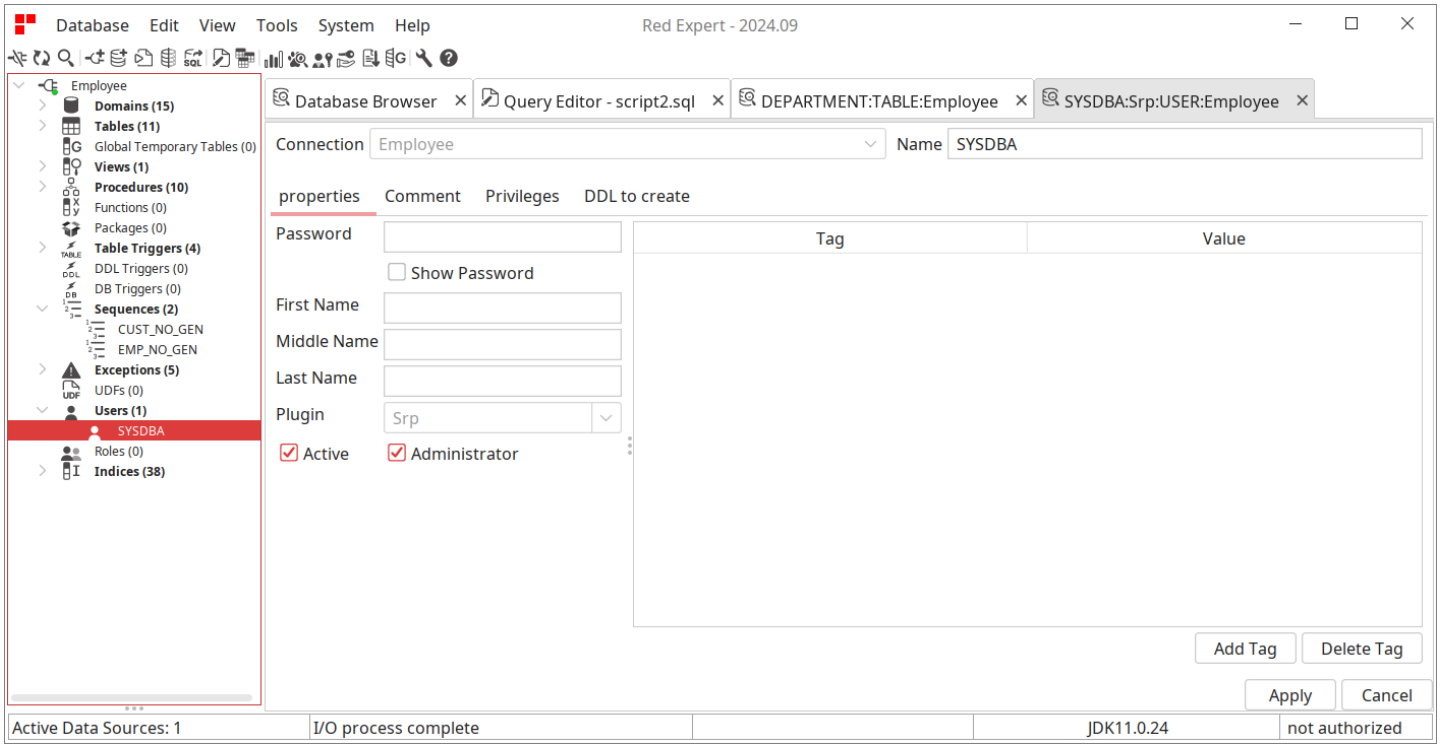




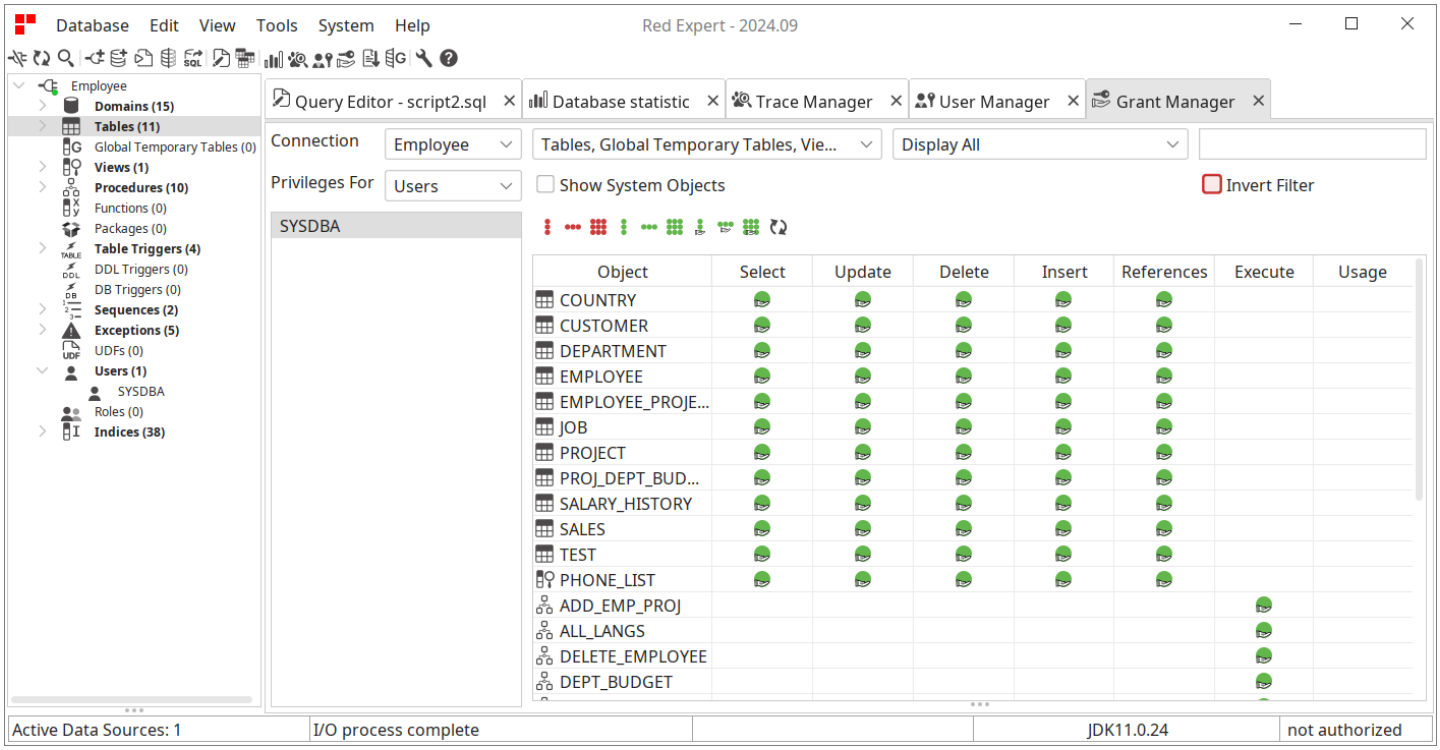
You will undoubtedly appreciate the additional functions and tools, which include the ability to recompile procedures and triggers from a tree view of objects, DDL script generator from metadata, database structure comparison (with the option to create a difference script), Data Importer and Data Generator, User and Grant managers, database statistics, table validator and (P)SQL profiler.



User manager



Grant manager



Data generator

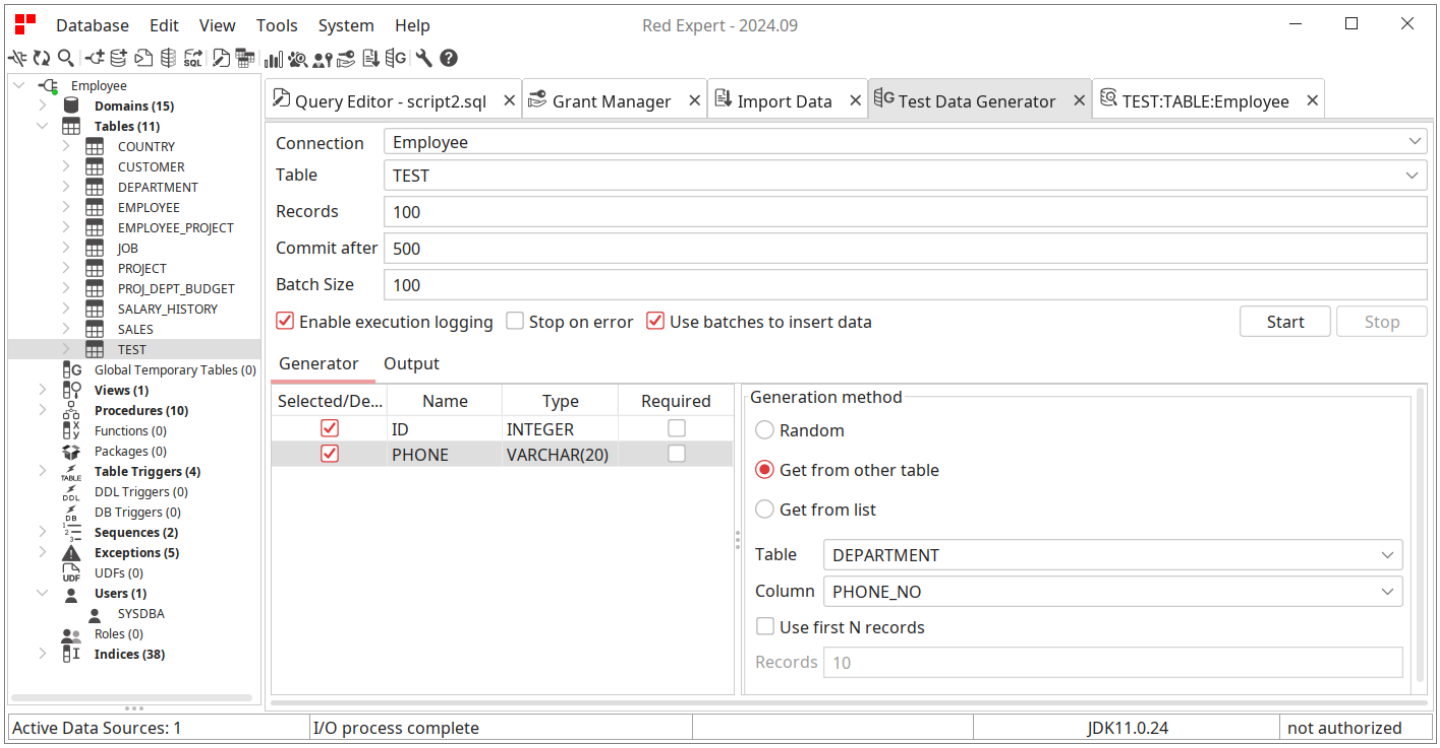
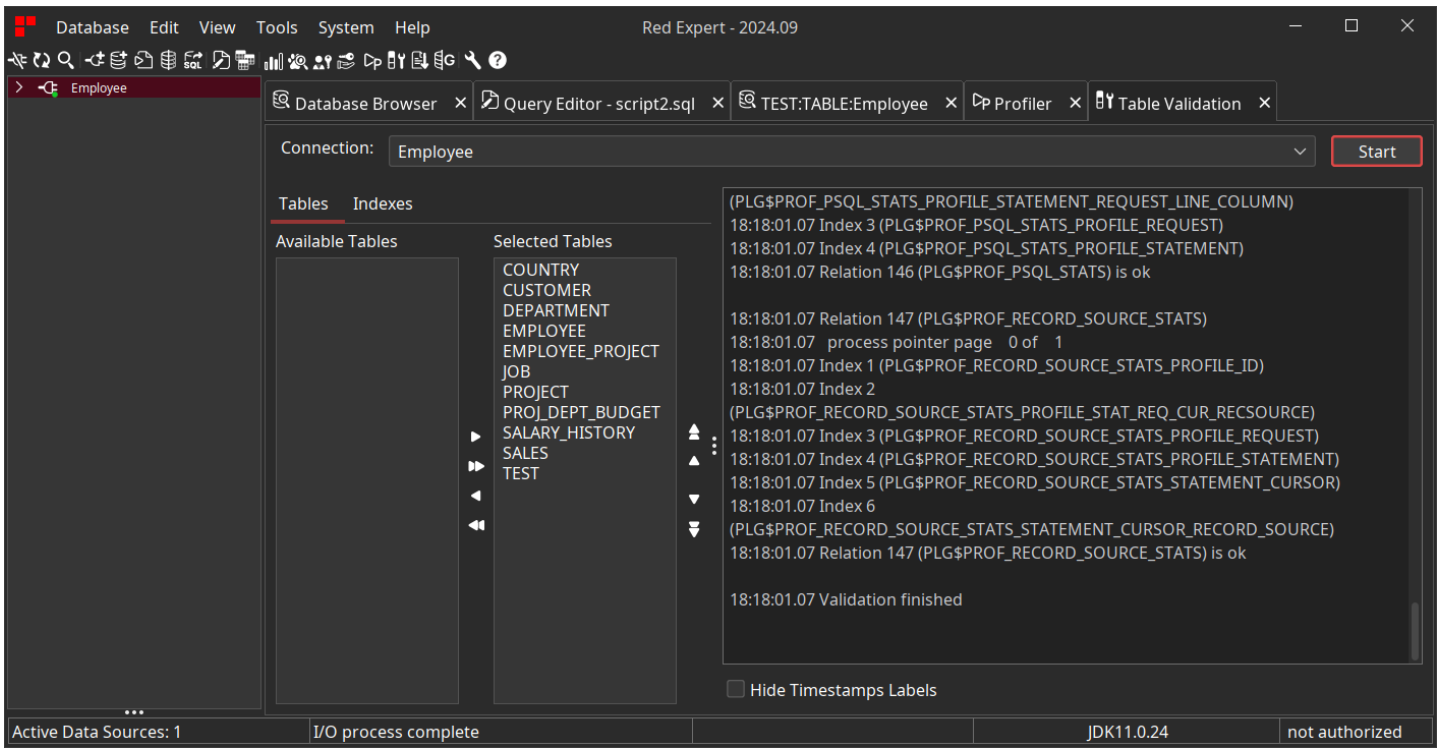


Table validator



The only disappointment is the Trace Manager, which we were unable to activate with Firebird. The built-in trace configuration support is exclusive to RedDatabase, resulting in parameters incompatible with Firebird. Even using a manually created configuration file proved ineffective. Unfortunately, Red Expert provides no feedback on whether the trace session was successfully initiated or if an error occurred.

Summary

Red Expert is definitely a solid contender among tools for Firebird, especially those aimed at database application developers. Its only weakness is the fact that it was (and to some extent still is) primarily designed to work with RedDatabase (and for Russian users). If future versions improve Firebird support and offer English documentation, Red Expert undoubtedly has the potential to gradually become "The standard GUI tool for Firebird", replacing the lagging FlameRobin.

But you definitely won't make a mistake if you try Red Expert today.

Advantages:

- Multiplatform
- Open Source
- Profiler integration
- ER diagrams
- User and Grant managers
- Data generator and importer

Disadvantages:

- Documentation only in Russian language
- Trace support is focused on RedDatabase and not Firebird
- You may encounter some quirks when working with Firebird

Our Rating: 8/10

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.

About transaction numbers in index nodes

Svend Meyland Nicolaisen asked:

Wouldn't it be possible to add transaction information to the index entries so that the engine doesn't have to lookup the record for validation?

Ann W. Harrison answers:

Yes, it would be possible, but would greatly increase the size of indexes and the amount of maintenance they require. Here's why.

Size matters a lot in index performance - there's less information per page with larger entries and the index depth increases.

An index entry currently consists of eight bits of prefix, eight bits of length, the compressed key value, and a 32 bit value which is a page number (for upper levels of the index) or a record number (for the bottom level). Record numbers are 40 bits in V2 but the compression is better, so the net is smaller. Transaction ids are 32 bit values and you need two of them - one to say what transaction created the value and one to say which transaction superseded the value. Adding two transaction ids to each index key nearly triples the "overhead" size of an index entry.

But that's no all. If you have an indexed field that switches between two values, it is currently represented in the index by one entry for each value - not one entry for each version. For example, transaction 1 creates the record and stored 'ABC' in the indexed field. Transaction 2 updates the record, changing 'ABC' to 'DEF'. The index now has two entries for the same record, 'ABC' and 'DEF'. Transaction 3 updates the record again, setting the field's value back to 'ABC'. Since there's already an index entry for 'ABC' for this record, nothing gets added to the index.

If we kept transaction information in the index, we would need two 'ABC' entries for that record, one for the 'ABC' created by Transaction 1 and superseded by transaction 2, and a second for the one created by Transaction 3 and not yet replaced. So instead of two entries with 6 bytes of overhead, we now have three entries with 14 bytes of overhead each.

Additional index entries mean more maintenance.

When transaction 2 updated the record, it would have to modify the old index entry in addition to inserting its new index entry.

When the record version created by transaction 1 is garbage collected, there's no need to change the index, because there is still a valid 'ABC' in the record version chain. If the index entry were tagged with the ids of the transactions that created and obsoleted it, the garbage collection would have to include removing the older of the two 'ABC' entries.

Why you should occasionally rebuild your indices

Aage Johansen asked:

Is it necessary to rebuild indices (by ALTER INDEX INACTIVE/ACTIVE or by backup/restore)? Doesn't Firebird balance the indexes dynamically?

Ann W. Harrison answers:

Yes, but the bucket split algorithm tends to leave partially filled buckets when indexes are built incrementally, while the fast-load algorithm used when an index is added after data exists produces dense indexes.

Table fill ratio after restore

Christian Kaufmann asked:

I always thought, that the fill ratio is corrected to about 80% when doing a restore. But after my restore, I get the following values for the biggest table:

```
Data pages: 25095, data page slots: 25095, average fill: 59%
Fill distribution:
       0 - 19% = 0
      20 - 39% = 1
      40 - 59% = 25094
      60 - 79% = 0
      80 - 99% = 0
```

The table contains about 7'000'000 records. Most of these will remain unchanged, but new records will be added in the future. The data fields of my record are two *smallint* and two *integer* fields.

- should I do a restore with -use_all_space ?
- should I just leave it like this ?
- other suggestion for changing it ?

Ann W. Harrison answers:

It is well - and incorrectly - known that the fill ratio is 80% if the database is set up to retain space for new versions. I've read it often from sources who should know. But I've also read the code and was around when it was designed. When space is reserved, the system leaves space for one fragmented record header for every primary record version stored on a page.

A primary record version is the newest version of a record - back versions, blobs, and fragments don't count.

A fragmented record is one that is split across pages, either because it was longer than a page or because it was forced to be on a particular page and didn't fit completely there.

A fragmented record header is a normal record header plus a pointer to the page and line of the fragmented part of the data. Counting on my fingers, I think it's 22 bytes, of which six plus three fill bytes occur only in fragmented records.

So, if your records compress to less than 9 bytes, you'll have more than 50% reserved space.

In this case, I would use that switch. The reserved space is used only for modified and deleted records. If there is no space on page when a modification or delete is done, Firebird will move the back version to a different page. That's expensive, but not nearly so expensive as having all pages half filled.

Christian Kaufmann continues:

OK. But if I understand correctly, even an update of a single record, where I change only one field forces Firebird to add a new copy of it and then, a new page will be necessary?

Ann W. Harrison answers:

Data is stored on data pages and index entries are stored on b-tree pages, so the number and types of indexes don't affect data storage.

If you use the -use_all_space switch, the database will put as many records as can possibly fit on each page. When you modify a record, Firebird first checks to see if the old version will fit on the same page with the new version. If not, then the old version will be put on a previously allocated data page for that table that does have space. First choice goes to dirty pages in the cache, second to other data pages in the cache, third to pages on disk.

If there is no previously allocated data page for that table with space, Firebird will take a free page in the file and allocate it to the table. If there are no free pages in the file, Firebird will extend the file and allocate the new page to the table.

When the back version is stored on a page, that page is available for new records or back versions. Eventually, the back version will become unnecessary and it will be removed, leaving space on that page for a back version of one of the newer records.

Until *InterBase 3*, we didn't reserve space for modifications. The first set of updates and deletes after the data was loaded tended to create a lot of I/O as back versions required new pages but eventually removing old versions and deleted records would create space and things got faster. The reserved space was added to reduce that "settling in" cost, but given the size of your records, the algorithm produces too much free space and you do excessive I/O because pages are only 60% full.

> *You can only compress entire databases with the -use_all_space restore parameter, which can cause problems with frequently updated tables. With FBOpt, you can compress individual tables where it really matters.*
>
> IBPhoenix

Automatic Test Creation

In the last issue, we introduced the testing system utilized in the development of Firebird and explored its capabilities for creating automated tests for database interactions in any Firebird-dependent applications. These tests can greatly simplify the transition between different versions of Firebird.

Creating tests for existing applications can be challenging. These applications usually involve numerous SQL queries and data operations, necessitating the creation of a substantial number of tests. Consequently, we will now explore various options for programmatically generating these tests.

The main challenges in automated test creation

Tests that share similar characteristics can be created programmatically with relative ease. This can be achieved by augmenting a test template with variable components, including:

- The SQL command (along with any parameters)
- The execution plan
- The data used by the command
- The expected output of the SQL command

Thus, the primary challenge lies in gathering this information and storing it in a format that the test generator can effectively utilize.

An essential step in the process is identifying testable commands and grouping them according to test templates. SQL queries are ideal candidates for automated test generation because they can be tested independently, follow a consistent structure, and can share a common test database. Queries can be further categorized into two main groups: static and parameterized.

Generating tests for data modification operations is more challenging, primarily because it's difficult to create a unified template with a reasonable number of variable elements. Additionally, many of these operations must be executed together due to business logic, making it impractical to test them in isolation. The significant differences between these operations often make it challenging—if not impossible—to develop a single template for testing.

The exception is modifications encapsulated within stored procedures, which significantly simplify template creation.

Gathering information about SQL statements can be challenging for several reasons:

- They are scattered across various parts of the application's source code.
- They are defined in hard-to-reach structures, such as within Delphi components.
- They are generated programmatically in ways that hinder easy access, such as with various ORMs

When SQL statements are defined in hard-to-reach structures, a similar approach can often be applied. For instance, if they are stored within Delphi components, the DFM files can be processed in text format.

The primary challenge in extracting queries from source code is dealing with parameterized queries, as it is impossible to determine the exact values of the parameters used during execution.

When it comes to programmatically generated SQL statements, the ability to collect executed SQL largely depends on how you create them. If you're using an ORM, it is highly likely that it includes built-in support for logging SQL statements to a file. For example, frameworks like Entity Framework, Hibernate, Django ORM, SQLAlchemy, ActiveRecord, Eloquent, and GORM offer this functionality. If you have your own SQL generation system, it's straightforward to integrate logging capabilities as well.

If none of the aforementioned options are viable, you can obtain the necessary information using the Firebird trace feature.

Extracting SQL statements through logging is much easier if you're using a dedicated system to test your application, as the logs are typically smaller and provide better coverage of the full range of operations. If such a system is unavailable, the only option is to generate logs from the application's live operation.

Using Firebird trace

Using Firebird trace is simple and straightforward. If you're not using a Firebird trace tool such as Upscene's TraceManager, you can always use the `fbtracemgr` utility that comes with Firebird along with a manually created configuration file (use redirect to file to save the trace).

You can use the `fbtrace.conf` file from your Firebird installation as a template for your configuration file, or use the one below:

```
database = %[\\/]my_database.fdb
{ # Do we trace database events or not
  enabled = true
  # Put sql statement execution finish\fetch to eof records
  log_statement_finish = true
  # Trace only SELECT statements
  include_filter = %(SELECT)%
  # Print access path (plan) with sql statement
  print_plan = true
  # Use legacy (false) or explained (true) plan format
  #explain_plan = true
  # Print detailed performance info when applicable
  #print_perf = true
  # Put finish record only if its timing exceeds X milliseconds
  time_threshold = 0
  # Maximum length of SQL string logged
  max_sql_length = 30000
  # Maximum length of individual string argument we log
  max_arg_length = 800
  # Do not log session init/finish events
  log_initfini = false
}
```

If you wish to collect execution plans in extended form instead of the classic version, simply uncomment the `explain_plan` option. If you wish to collect also the performance counters (table access statistics), uncomment the `print_perf` option.

The resulting trace log needs to be processed further because it contains many entries similar to the following:

```
2024-08-28T09:58:36.1840 (1974:0x7f004dfb0640) EXECUTE_STATEMENT_FINISH
       employee.fdb (ATT_96, SYSDBA:NONE, NONE, TCPv4:127.0.0.1/38798)
       /opt/firebird/bin/isql:11033
               (TRA_2935, CONCURRENCY | WAIT | READ_WRITE)

Statement 102:
-------------------------------------------------------------------------------
select * from country
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
PLAN (COUNTRY NATURAL)
16 records fetched
     0 ms, 20 fetch(es)
```

```
2024-08-28T10:28:31.4820 (1974:0x7f004df91dc0) EXECUTE_STATEMENT_FINISH
       employee.fdb (ATT_105, SYSDBA:NONE, NONE, TCPv4:127.0.0.1/36458)
       /usr/bin/python3.11:13225
               (TRA_2942, CONCURRENCY | WAIT | READ_WRITE)

Statement 102:
-------------------------------------------------------------------------------
select * from country where currency = ?
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
PLAN (COUNTRY NATURAL)

param0 = varchar(10), "Euro"

6 records fetched
     0 ms, 21 fetch(es)

2024-08-28T10:36:38.0520 (1974:0x7f004df923c0) EXECUTE_STATEMENT_FINISH
       employee.fdb (ATT_111, SYSDBA:NONE, NONE, TCPv4:127.0.0.1/33016)
       /usr/bin/python3.11:13665
               (TRA_2945, CONCURRENCY | WAIT | READ_WRITE)

Statement 203:
-------------------------------------------------------------------------------
select * from country where currency = ?
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
PLAN (COUNTRY NATURAL)

param0 = varchar(10), "Dollar"

6 records fetched
     0 ms, 21 fetch(es)

2024-08-28T10:37:14.7370 (1974:0x7f004df923c0) EXECUTE_STATEMENT_FINISH
       employee.fdb (ATT_112, SYSDBA:NONE, NONE, TCPv4:127.0.0.1/52064)
       /usr/bin/python3.11:13716
               (TRA_2946, CONCURRENCY | WAIT | READ_WRITE)

Statement 304:
-------------------------------------------------------------------------------
select * from country where currency = ?
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
PLAN (COUNTRY NATURAL)
param0 = varchar(10), "Euro"

1 records fetched
     0 ms, 21 fetch(es)
```

The simplest way to achieve this is by using a Python script with the `firebird-lib` library, which includes a trace log parser that outputs information as structured objects. A key advantage of this parser is its compact output, as it identifies repeated information and emits relevant objects only on their first occurrence. On subsequent occurrences, it references these objects, reducing redundancy. Specifically, this applies to `SQLInfo` and `ParamSet` objects, which contain details about the SQL command and its unique combination of parameters.

The following script processes the trace log and lists unique SQL queries along with their corresponding (unique) parameter sets:

```
# /// script
# requires-python = ">=3.11"
# dependencies = [
#   "firebird-lib~=1.5",
# ]
# ///
import sys
from collections import defaultdict
from firebird.lib.trace import TraceParser, ParamSet, SQLInfo, EventStatementFinish
params = {}
queries = {}
query_params = defaultdict(set)
parser = TraceParser()

with open(sys.argv[1]) as f:
    for obj in parser.parse(f):
        if isinstance(obj, ParamSet):
            params[obj.par_id]= obj
        elif isinstance(obj, SQLInfo):
            queries[obj.sql_id] = obj
        elif isinstance(obj, EventStatementFinish):
            if obj.param_id is not None:
                query_params[obj.sql_id].add(obj.param_id)

for q in queries.values():
    print(q)
    for p in query_params.get(q.sql_id, []):
        print(' ', params[p])
```

If you save the sample trace log to a file named `sample-trace.log` and the script to `process_log.py`, you can run the following command:

```
pipx run process_log.py sample-trace.log
```

To get next output:

```
SQLInfo(sql_id=1, sql='select * from country',
        plan='PLAN (COUNTRY NATURAL)')
SQLInfo(sql_id=2, sql='select * from country where currency = ?',
        plan='PLAN (COUNTRY NATURAL)')
  ParamSet(par_id=1, params=[('varchar(10)', 'Euro')])
  ParamSet(par_id=2, params=[('varchar(10)', 'Dollar')])
```

> *An alternative to using a parser is the IBPhoenix Trace Plugin, which directly generates data in JSON or protobuf formats, making it suitable for machine processing. However, in this case, you'll need to handle the reduction of redundant output within your own code.*

Data storage

While it may be possible to merge the process of extracting SQL command data with test generation, it is advisable to separate these steps. First, store the extracted data in a suitable format, then use it with the test generator. This approach simplifies test updates when SQL statements are modified.

Data should be stored in a structured format that allows for easy manual or automated processing later. While it may be tempting to store the data in text files under version control, using a database for storage is far more efficient and convenient.

The design of the database structure is beyond the scope of this article, as it depends on various factors, including the nature of the data, the method of acquisition, and the requirements for updates and future use. However, designing such a database should not be difficult for an experienced developer.

Test templates can also be stored in this database, along with information regarding the association of individual SQL commands with their corresponding templates.

Test generator

Test templates form the foundation for generating automated tests. To begin, it's advisable to manually create a test for one representative SQL statement from each category you've collected. From this original test, you can then modify it to create new tests for different statement. This approach makes it easier to identify both commonalities and differences across tests.

Next, you'll need to decide how to represent your templates and how to implement them in the test generation process. There are several options to consider. If your requirements are simple—such as creating tests by replacing text in the template or making minor adjustments based on straightforward branching rules—it may be most efficient to write a custom generator in the programming language you're most familiar with.

However, if your generation process demands more complex rules for creating variations, it is generally better to leverage a dedicated templating system. Numerous options are available for this purpose, including Jinja2 [1], Mako [2], Cog [3], CTemplate [4], Mustache [5], Handlebars [6], Apache FreeMaker [7]. Your choice will determine whether you can utilize an existing tool for test generation or if you will need to develop your own solution.

1. Jinja2
2. Mako
3. Cog
4. CTemplate
5. Mustache
6. Handlebars
7. Apache FreeMaker

...And now for something completely different

The Tale of the Three Servers

Once upon a time, in a bustling data center, there lived three little servers. They were brothers, and each one wanted to build the best system to store and protect the world's precious data.

The first little server was eager and didn't want to waste time. "I'll keep things simple," he said. "I'll build my system using basic flat files. It's easy and fast!" So, the first little server built his system by saving data in plain files, scattered across directories.

Before long, the data began to pour in, and the first little server was feeling quite proud of his speedy setup. But one day, the Big Bad Query came along. This was no ordinary query—it was complex, heavy, and hungry for data.

"Let me in, let me in, little server," the Big Bad Query demanded, "or I'll crash your system down!"

"Not by the code on my hard drive!" replied the first little server. But the Big Bad Query was powerful. It huffed and puffed and sent complicated search requests that overwhelmed the little server's flat-file system. In no time, the system crashed, and data was lost in the chaos. The first little server had no choice but to flee to his brother's system for help.

The second little server had taken more time to think things through. "I'll be smarter than that," he said. "I'll use a <CENSORED> relational database system. Structured tables, relationships, and indexes will protect me from any trouble!" So, the second little server built his system with SQL, neatly organized tables, and strong constraints.

The data flowed in smoothly, and for a while, everything seemed perfect. But soon enough, the Big Bad Query came back, more complex than ever. "Let me in, let me in, little server," it roared, "or I'll crash your system down!"

"Not by the code in my relational tables!" cried the second little server. But the Big Bad Query was even more cunning this time. It unleashed massive joins, nested subqueries, and overloaded the system with concurrent requests. The relational database struggled under the pressure, slowing down until it became unresponsive. The second little server had to abandon his system and run with his brother to the third server's setup.

Now, the third little server had been quietly working on his system for a long time. He was careful and wise, and he had chosen to use Firebird—a strong, reliable database known for its resilience and power.

"I'll build my system using Firebird," he said confidently. "It's lightweight, yet robust. It can handle heavy loads, and it won't let me down."

The third server set up his Firebird database with careful planning. He used its powerful features like multi-generational architecture, high concurrency, and strong support for stored procedures and triggers. Firebird's small footprint didn't fool him; he knew it packed a punch.

When the Big Bad Query returned, it was more fearsome than ever. "Let me in, let me in, little server," it bellowed, "or I'll crash your system down!"

"Not by the code in my Firebird database!" answered the third server bravely.

The Big Bad Query tried everything—complex transactions, enormous datasets, simultaneous users—but Firebird was ready. It managed resources efficiently, handled concurrency with ease, and didn't let the heavy load bring it down. The Big Bad Query couldn't crash the system, no matter how hard it tried.

Exhausted and defeated, the Big Bad Query finally gave up and slunk away, leaving the third little server's system running smoothly. The three brothers celebrated, and the first two servers learned an important lesson: sometimes, the best solution isn't just about being simple or building quickly. A smart, well-designed system using Firebird can stand strong against even the toughest of queries.

And so, the three servers lived happily ever after, with Firebird as their protector, ensuring data flowed smoothly and safely for all time.

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.

Firebird website

Firebird on GitHub

Thank you for your support!

EmberWings is a quarterly magazine published by the **Firebird Foundation**, free to the public after a 12-month delay. Regular donors get exclusive early access to every new edition upon release.