

# *EmberWings*

2024/2



**IBPhoenix**  
THE POWER WITHIN

June 2024

[www.ibphoenix.com](http://www.ibphoenix.com)

[emberwings@ibphoenix.com](mailto:emberwings@ibphoenix.com)



## Support & tools for Firebird

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

IBPhoenix website & e-shop

If you are interested in publishing or advertising in EmberWings magazine, please send us an [email](#) or [message](#).





## In This Issue:

- Your Voice, Our Vision: The Future of EmberWings Magazine
- Wisdom of the elders
- Using Firebird QA tools
- Interview with Hajime Nakagami
- Development update: 2024/Q2
- Toolbox: DBeaver CE
- Answers to your questions
- Optimization of the database structure
- ..And now for something completely different

# Your Voice, Our Vision: The Future of EmberWings Magazine

As we launch the second issue of EmberWings Magazine, we are thrilled by the positive feedback received from our debut edition. Your kind words have been truly encouraging. We were delighted to hear that you enjoyed our articles, and that our content struck the right balance between technical detail and readability. Your feedback has reinforced our commitment to delivering high-quality, relevant content.

Because you have expressed satisfaction with the basic layout of the magazine, we will therefore continue to maintain it. So each issue will have two main articles, a review of a product that caught our attention, an interview with interesting people from the Firebird community, answers to tricky questions, an overview of developments within the Firebird project, and, of course, some thematic humor.

We are also pleased that you like the graphic style of the magazine. We will continue to try to bring you attractive Firebird themed images.

Thank you for your interest and support. They are a great encouragement to us in our quest to make EmberWings a stable resource for all Firebird RDBMS fans and users. We hope you will keep us in your favor.

*Your EmberWings editors*



*Under the ancient oak, by the flickering campfire, a young native approached the tribal shaman. "Wise Shaman," the young one began, "how can we keep our paths clear when the spirit of the land changes?"*

*The shaman picked up a smooth stone and cast it into the river. "Tell me, young one, how does the river respond to the stone?"*

*The young native watched the ripples and replied, "It flows and adapts around the stone, continuing its journey."*

*The shaman nodded, then pointed to the sky. "How do the stars maintain their place in the heavens?"*

*The young native looked up and said, "They remain constant, guiding us through the night."*

*The shaman smiled, "And how does our spirit ensure harmony with the new rhythms of the earth, like the changing of the database or upgrading the server?"*



*The young native pondered and responded, "With sacred tests that trace every query and procedure, revealing hidden paths and shifts."*

*The shaman nodded again, "Yes. These automatic tests are like the river and the stars. They adapt to changes, yet remain steadfast in their guidance. When the database transforms, or the server changes its behavior, these tests uncover new paths, ensuring our journey remains smooth and true."*

*The young native, understanding now, asked, "So, the tests are the silent guardians, guiding us through unseen transformations?"*

*The shaman closed his eyes and replied, "Indeed. The true power lies in their silent vigilance. Trust in these tests, for they reveal hidden mysteries and ease our passage through the ever-changing landscapes of our spirit and the earth."*





## Using Firebird QA tools

Automated testing is an integral part of Firebird development. Because standard tools were not sufficient for the unique requirements of database server testing, the Firebird project developed specialized test tools that have undergone a number of changes over the years. Currently, Firebird uses a testing system based on the pytest tool, supplemented by a specialized plugin designed for Firebird's specific testing requirements.

Although primarily designed for internal use, its features make it easy to create automated tests for database interactions of any Firebird-dependent application. As already mentioned in the last issue, such tests can significantly facilitate the transition between individual versions of Firebird. In this article you will learn how to create and run such tests.

# Introduction

---

The Firebird QA tools are built on pytest, which is a powerful and widely-used testing framework for Python.

If you're new to Python, don't worry. Using pytest is simple, and writing tests requires only basic knowledge of Python. In fact, if you know how to use Firebird ISQL tool, you can get along with simple tests just by replicating the common pattern for test files without any knowledge of Python. Of course, you will need to know Python to write more sophisticated tests, but writing tests is actually the easiest way to gradually improve your Python skills without a steep learning curve.

To install the QA tools, you only need Python version 3.8 or later (we recommend Python 3.11 or 3.12), and the `pipx` tool installed. Details on their installation can be found in the article "Using tools written in Python" published in the first issue.

Installation command from console:

```
pipx install --include-deps firebird-qa
```

You can verify that pytest is properly installed with:

```
pytest --version
```

If you later decide to use any of the plugins from the vast library of pytest extensions, you can install them using the `pipx inject` command. For example, the `pytest-repeat` plugin, which enables repeated execution of tests, can be installed using:

```
pipx inject firebird-qa pytest-repeat
```



# Testing database interactions

---

Testing the interactions of a user application with a database is not, in principle, a complex problem. The goal of these tests is not to verify the correct functionality of the application, but whether the server responds to the application's commands in the desired and optimal way. Since the application communicates with Firebird through SQL commands, it is basically just executing them with controlled conditions, verifying that they will run without problems and with the required parameters, and returning the required results.

Tests can be very simple because we focus on individual interactions and not on entire functional units (which consist of a sequence of simpler interactions). In practice, this means that we will test individual SQL commands entered by the application, whether they are queries, data modifications or procedure calls.

We will need at least two databases for testing. One database will be used by tests that do not modify data, and should approximate a typical production database in terms of data structure and scope. Other databases can be either without data, or with only the necessary amount of data to test commands that change the contents of the database.

The reason for such a division of databases is simple. Each test must have a well-defined environment, including the database. For tests that modify the content of the database, it would be necessary to return it to its original state at the end of the test, which unnecessarily complicates either the implementation of the tests or their runtime environment. It is much easier to create a new database (either using an SQL script, from a backup or by simply copying) before starting the test and delete the database after it has run. Thus, the size of the database greatly affects the speed of running the tests. Tests that do not change the database can directly use the same database, so its size does not affect the speed of running the tests in any way.

Each test has basically the same structure:

- Uses a specific database

- Executes one particular SQL statement
- Verifies that command execution conforms to expected values or parameters

SQL statements should closely copy the statements used by the application. However, applications often use parameterized queries. In such a case, it is possible to either create a simple test with a single variant of the command with fixed parameters, or to create a parameterized test with parameters given by a set of values or using a function.

Validation of command execution against expected parameters may vary and depends on the nature of the command. Correct execution is automatically verified, however, it is advisable to verify return values (in case of queries) or changes made. In the case of queries, it is also advisable to check whether the execution plan corresponds to what is expected. Typically, it is not necessary to test the execution speed of the command, because for simply built tests it is sufficient to work with the execution time of the test as a whole.

## Basic concepts

---

Pytest is built around a few core concepts:

**Test Functions:** Tests in `pytest` are simply Python functions that begin with the prefix `test_`. These functions contain the code to be executed and typically include assertions to verify expected behavior.

**Fixtures:** Fixtures are functions that provide a baseline set of data or functionality for tests to use. They are used to set up preconditions, such as initializing a database or establishing connections, before test functions are executed. They are also responsible for proper cleanup of transient resources.

**Assertions:** Assertions are statements used in test function to verify the expected behavior. They typically have form of standard Python `assert` statement:

```
assert <boolean expression> [, <message>]
```

Optional `<message>` is printed along assert expression introspection when assertion is evaluated as False.

**Test Discovery:** Pytest automatically discovers and runs tests within the project directory structure, eliminating the need for manual test suite configuration. Tests can be organized into modules and subdirectories, and pytest will recursively search for and execute all test functions.

**Plugins:** Pytest offers a plugin architecture that allows users to extend and customize its functionality. There are hundreds of community-maintained plugins available for pytest, covering a wide range of use cases, including test coverage, parallel execution, and integration with other tools and frameworks.

The firebird-qa plugin then creates a superstructure on top of this basic architecture, the elements of which can be divided into several groups:

- Uniform format of tests. Each test is a separate file that can contain one or more test functions (which, however, represent different variants of the same test, e.g. for different platforms or versions of Firebird). The plugin also defines descriptive test metadata (various identifications, name, notes, etc.) and support for test function metadata (eg assignment only to a certain test platform or Firebird version, etc.)
- Uniform test repository layout.
- Configuration of the environment for running tests according to defined parameters. This is primarily a selection of only those test functions (test variants) that correspond to the current platform and version of Firebird, the configuration of the connection to the tested server and its tools, data files, etc.
- Fixtures for securing and managing resources used by Firebird tests. These are primarily databases, users, roles, etc. A special role is played by the fixture, which provides a multifunctional Action object for working with Firebird.



# Test repository

---

Tests and other necessary files are organized in the directory structure of the so-called test repository. Here is layout of sample repository:

```
.
├── backups
├── databases
│   ├── employee3.fdb
│   └── employee4.fdb
├── files
├── tests
│   ├── parametrized
│   │   ├── __init__.py
│   │   └── test_01.py
│   ├── __init__.py
│   ├── test_01.py
│   ├── test_02.py
│   ├── test_03.py
│   └── test_04.py
├── firebird-driver.conf
└── pytest.ini
```

- Backup files that would be used to create test databases should be stored in `backups` directory.
- Databases that will be either copied to test databases or used directly should be stored in `databases` directory.
- Any data files used by tests should be stored in `files` directory
- All tests should be stored in `tests` directory structure (you can organize them freely in sub-directories there). Because tests are written in Python, the test suite directory is a Python package, so each directory **MUST** contain an empty `__init__.py` file.

- The `firebird-driver.conf` file is mandatory, and must contain configuration for at least one Firebird server. Package with examples for this article contains configuration file that defines server `local` with standard `SYSDBA/masterkey` credentials.
- File `pytest.ini` is an optional `pytest` configuration.

To run your tests, you need to open the terminal window in this directory and run `pytest` from there.

## Your first test

---

Here is an example of simple test that uses standard ISQL tool to execute a query against `employee` database, and verifies its output and execution plan:

---

```
"""
ID:          query.01
TITLE:       Query from JOB with index filter
DESCRIPTION: This test uses database specified by alias
              (in databases.conf), and has description that
              spans multiple lines.
"""

import pytest
from firebird.qa import *

# fixture providing test database
db = existing_db_factory(filename='#employee')

# isql script executed to test Firebird
test_script = """
set plan on;
select JOB_CODE, JOB_TITLE, JOB_COUNTRY from JOB
where JOB_COUNTRY = 'France';
"""
```

```
# this is expected output from test script execution
expected_stdout = """
PLAN (JOB INDEX (RDB$FOREIGN3))

JOB_CODE JOB_TITLE                JOB_COUNTRY
SRep      Sales Representative     France
"""

# fixture for test execution and evaluation
act = isql_act('db', test_script,
               substitutions=[('=', ' '), ('[ \t]+', ' ')])

def test_1(act: Action):
    act.expected_stdout = expected_stdout
    act.execute()
    # This evaluates test outcome
    assert act.clean_stdout == act.clean_expected_stdout
```

---

## Test metadata

The first block of text enclosed by triple double quotes is so-called `docstring` of the Python module with your test. The `firebird-qa` plugin uses it for definition of test metadata. Each metadata item must start on separate line starting with item tag followed by colon. The only mandatory items are:

**ID:** Unique test identification. Can contain alphanumeric characters, dot, underscore and hyphen. Must start with alphanum character.

**TITLE:** Test title. Multiline titles are concatenated into single line (line breaks removed and line contents separated with single space).

**DESCRIPTION:** Test description. Could span multiple lines.



# Module imports

Imports listed in example are mandatory for all tests. You may import additional Python modules that your test needs.

## Test database

Each test typically requires a database. These are provided as fixtures for your tests that are created either by `existing_db_factory` or `db_factory` function call.

The `existing_db_factory()` function creates fixture for directly used databases (i.e. for tests that do not modify its content). You can identify the database via its alias name in `databases.conf` file using `#alias` format like in this example, or with filename of the database located in `databases` sub-directory. The returned fixture must be assigned to module-level variable. Name of this variable is important, as it's used to reference the fixture in other fixture-factory functions that use the database, and the test function itself.

The `db_factory()` function creates fixture for temporary test databases, where function arguments specify how database is created, initialized and removed:

- If not specified otherwise, the fixture creates new empty database.
- To create database from backup file, use `from_backup` argument. File must be located in `backups` directory.
- To use copy of prepared database, use `copy_of` argument. File must be located in `databases` directory.
- The name of created temporary database could be specified with `filename` argument. Default database name is `test.fdb`.
- It's possible to specify `page_size` and `sql_dialect` of created database. These options are ignored if database is created as a copy, or from backup.
- It's possible to specify database `charset` (not applied for backups and copies) that is also default connection charset.

- After temporary database is created (by either method), it could be initialized with SQL commands (executed via `isql`) specified using `init` argument.
- Database is created using default server user and password. It's possible to specify alternate credentials with `user` and `password` arguments.
- The fixture ensures that database is created and initialized during test setup, and removed during test teardown. To disable either phase (because create/drop is performed by test itself), use `do_not_create` or `do_not_drop` arguments.
- By default, database is set to async write after creation to speed up database operations. It's possible to change that with `async_write` argument.

Because almost all Firebird tests need a database, the QA plugin works with concept of primary test database. This fixture is typically named `db`, and is used by other fixtures that work with database.

## Test script and expected output

Our first test uses ISQL to execute a SQL script, captures its output and compares it with expected one. Hence we need to define these. The recommended way is to use variables defined at module level for such blocks of text.

## Action fixture

The Action object provided by action fixture is a Swiss Army Knife for your tests. This fixture is created by `isql_act()` or `python_act()` factory functions. These functions are identical (it's in fact only one function available under two names). It's a legacy from old `fbtest` QA system that had two types of tests, and such distinction was retained during conversion of tests from the old system to the new one. Although there is no difference, it's recommended to retain the distinction in new test by using:

- `isql_act` for simple tests that use single ISQL test script.
- `python_act` for more complex test implementations.

This function has next arguments:

**db\_fixture\_name:** REQUIRED name of database fixture (primary database). Note that database fixture is referenced by name, not by value, so you have to pass the fixture variable name as string!

**script:** Optional SQL script that tests the server.

**substitutions:** Optional list of additional Regex substitutions for stdout/stderr cleanup. In the example, there are substitution that remove = characters and replace tabulators with single space.

The returned fixture must be assigned to module-level variable. It's typically named `act`.

## The test function

All test functions have at least one parameter: the Action object provided by action fixture created earlier.

Our simple test has only three lines of code:

First line assigns the expected output defined at module level to the `expected_stdout` member variable in the Action object.

Second line executes the test script via ISQL.

The third line asserts that output from script execution is equal to expected output. It uses "cleaned" version of both strings (that's the reason why we assigned the expected output to the Action object on first line) that have noise content removed or normalized in other ways for simpler comparison. Default cleanup removes `SQL>`, `CON>` and other ISQL "noise" strings and all leading and trailing whitespace characters from each line of text. In our example with additional substitutions, it also removes the header/data separator from query output.

# Test execution

To execute tests from your repository, just open the terminal window in it and execute the `pytest` there. However, you must always provide `--server NAME` parameter where `NAME` is the server configuration name in `firebird-driver.conf` file. As you'll typically use only one test server, it's practical to create `pytest.ini` file in your test repository with command line and other `pytest` configuration options you want to always use. The recommended content is:

```
console_output_style = count
testpaths = tests
addopts = --server local --install-terminal
```

Now you can just execute:

```
pytest
```

to run all tests in your test repository. With the above sample test there, you should see output like this:

```
===== test session starts =====
platform linux -- Python 3.11.9, pytest-8.2.0, pluggy-1.5.0
System:
  encodings: sys:utf-8 locale:UTF-8 filesystem:utf-8
Firebird:
  configuration: firebird-driver.conf
  ODS: 13.1
  server: local [v5.0.0.1306, SuperServer, Firebird/Linux/AMD/Intel/x64]
  home: /opt/firebird
  bin: /opt/firebird/bin
  client library: libfbclient.so.2
rootdir: /home/job/qa-user
configfile: pytest.ini
plugins: firebird-qa-0.20.0
collected 1 item

query.01 .                                                                    [1/1]
===== 1 passed in 0.12s =====
```

## More tests

---

The vast majority of database interaction tests have a uniform structure, which greatly simplifies their creation. You can use the above example as a template for any number of other tests that work with a single static SQL query. Just copy the original file under a new name, change the description in the metadata, the text of the SQL query and the expected output from ISQL.

*In the next issue, we'll show you how all the steps can be automated using simple Python script.*

Although most tests have a uniform structure, some tests may require slightly different implementation. Therefore, we will now demonstrate the most typical ones.

## Test variants

The reasons for creating alternative test implementations can be varied. The most typical is the need to modify the test for different versions of Firebird. Another common reason is the difference in implementation for different platforms.

When applying tests to different versions of Firebird, it is necessary to use a database with the corresponding ODS structure. If the database used by the test is created using a script or from a backup, there is no problem. However, if the test uses the database directly or a copy of it, it is necessary to copy the database with the corresponding structure to the `databases` directory before starting the test. An alternative solution is to create several test functions and assign each different database to the corresponding version of Firebird. The pytest plugin will then automatically select the test function corresponding to the currently used version of Firebird.

Modification of first test that uses different databases could look like:



---

```
# -- Test version for Firebird 3

# fixture providing test database for Firebird 3
db3 = existing_db_factory(filename='employee3.fdb')

# fixture for test execution and evaluation
act3 = isql_act('db3', test_script)

@pytest.mark.version('>=3,<4')
def test_for_fb3(act3: Action):
    act3.expected_stdout = expected_stdout
    act3.execute()
    assert act3.clean_stdout == act3.clean_expected_stdout

# -- Test version for Firebird 4 and 5

# fixture providing test database for Firebird 4 and 5
db4 = existing_db_factory(filename='employee4.fdb')

# fixture for test execution and evaluation
act4 = isql_act('db4', test_script)

@pytest.mark.version('>=4,<6')
def test_for_fb_4_and_5(act4: Action):
    act4.expected_stdout = expected_stdout
    act4.execute()
    assert act4.clean_stdout == act4.clean_expected_stdout
```

---

The test functions remain essentially the same, only using different fixtures for the database and the Action object. Assigning a function to a Firebird version is done using the `pytest.mark.version` decorator, which is passed the specification of the required version. Similarly, a function can be assigned to a specific platform with the `pytest.mark.platform` decorator.

Because Firebird 5 can access Firebird 4 databases, we can use single database for both Firebird versions.

## More complex tests

Some tests cannot be implemented using ISQL scripts, or the use of ISQL is too impractical. For example, if a test query returns a large number of rows, it may not be practical to test its output because of the amount of text that would need to be included in the test. Testing only the execution plan without actually executing the query is not a suitable solution, as is limiting the query output with, for example, the `FIRST` clause (it can change the query execution time, which we are also interested in). The solution is to execute the query directly from Python, instead of using ISQL, which allows us to retrieve the result of the query but ignore it entirely.

A corresponding modification of the initial test may look, for example, as follows:

---

```
"""
ID:          query.04
TITLE:       Query from JOB with index filter
DESCRIPTION: This test uses Python code to fetch all returned data
              without output.
"""

import pytest
from firebird.qa import *

db = existing_db_factory(filename='#employee')

test_query = """
select JOB_CODE, JOB_TITLE, JOB_COUNTRY from JOB
where JOB_COUNTRY = 'France';
"""

expected_plan = """Select Expression
-> Filter
  -> Table "JOB" Access By ID
    -> Bitmap
      -> Index "RDB$FOREIGN3" Range Scan (full match)
"""
```

```

# fixture for test execution and evaluation
act = python_act('db')

def test_1(act: Action):
    # Open the connection ("with" ensures cleanup)
    with act.db.connect() as con:
        # Create cursor
        with con.cursor() as cur:
            cur.execute(test_query)
            # Fetch all data so test execution time contains the fetch
            # time, but do not store anything anywhere
            for _ in cur: pass
            # This evaluates test outcome
            act.stdout = cur.statement.detailed_plan
            act.expected_stdout = expected_plan
            assert act.clean_stdout == act.clean_expected_stdout

```

---

This version of the test uses the EXPLAINED version of the execution plan. To avoid the need to preserve the indentation of the expected output, we will use a cleaned version of the output for comparison.

Also this test can be used as a template in which you just need to change the query being executed and the expected execution plan.

## Parameterized tests

Another common case is the use of parameterized queries. For example, for our first test, the query used by the application might look like this:

```

select JOB_CODE, JOB_TITLE, JOB_COUNTRY from JOB
where JOB_COUNTRY = ?;

```

Also in this case, ISQL will not suffice, and we will have to use code in Python. The Firebird driver for Python makes working with parameterized queries very easy, all we need to figure out is how large sets of parameters we want to use and how to supply them to the test function.

First of all, we recommend using only a limited set of parameters. Collections in the range of two to ten different query calls are usually fully sufficient to verify common and borderline query call cases. Additionally, smaller sets can be directly part of the test code. Larger sets should be loaded from a file.

To supply parameters to the query we can, for example, use a cycle. But it is much more convenient to use the support for test parameterization provided directly by pytest. Tests parameterized using pytest are executed separately for each set of parameters, as separate variants. So if an execution fails with a certain set of parameters, it won't affect the results of other calls, making it much easier to identify the real source of the problem.

The parametrized test may look as follows:

---

```
"""
ID:          query.parametrized.01
TITLE:       Query from JOB with index filter
DESCRIPTION: This one demonstrates the use of parametrized query and
              parametrized test
"""

import pytest
from firebird.qa import *

db = existing_db_factory(filename='#employee')

test_query = """
select JOB_CODE, JOB_TITLE, JOB_COUNTRY from JOB
where JOB_COUNTRY = ?;
"""

# List of expected outputs from parametrized query
expected_outputs = [
    """JOB_CODE JOB_TITLE                JOB_COUNTRY
-----
SRep      Sales Representative          France
""",
    """JOB_CODE JOB_TITLE                JOB_COUNTRY
-----
Eng       Engineer                     Japan
SRep      Sales Representative          Japan
"""]
]
```



```

expected_plan = """Select Expression
-> Filter
-> Table "JOB" Access By ID
-> Bitmap
-> Index "RDB$FOREIGN3" Range Scan (full match)
"""

# fixture for test execution and evaluation
act = python_act('db')

# Test parameterization
@pytest.mark.parametrize('query_parameter, output_index',
                        [("France", 0), ("Japan", 1)])
def test_1(act: Action, capsys, query_parameter, output_index):
    # Open the connection ("with" ensures cleanup)
    with act.db.connect() as con:
        # Create cursor
        with con.cursor() as cur:
            cur.execute(test_query, [query_parameter])
            # Print data from cursor to stdout
            act.print_data(cur)
            # capsys is pytest fixture to capture stdout/stderr content
            act.stdout = capsys.readouterr().out
            act.expected_stdout = expected_outputs[output_index]
            assert act.clean_stdout == act.clean_expected_stdout
            # It's necessary to reinitialize the output cleaning because
            # it's cached
            act.reset()
            # Check the execution plan
            act.stdout = cur.statement.detailed_plan
            act.expected_stdout = expected_plan
            assert act.clean_stdout == act.clean_expected_stdout

```

---

Test parameterization is defined by the `@pytest.mark.parametrize` decorator. The first parameter of the decorator is a list of test function parameters to be filled from the dataset. The second parameter is a data set consisting of a list of tuples of parameters. In this case, each tuple consists of a query parameter and an index into a list of expected query outputs (since the query output is not large, we include it in the test).

Data from the cursor is written to standard output by the helper function `print_data` provided by the `Action` object. The `capsys` fixture provided by `pytest` is used to capture the printout.

Since we do not expect the execution plan to vary according to the parameters passed, it is not used to parameterize the test function. However, if histogram support will be added to Firebird, the plans may start to differ and will need to be included in the parameterization.

If we run the test, we get output like this:

```
===== test session starts =====
platform linux -- Python 3.11.9, pytest-8.2.0, pluggy-1.5.0
System:
  encodings: sys:utf-8 locale:UTF-8 filesystem:utf-8
Firebird:
  configuration: firebird-driver.conf
  ODS: 13.1
  server: local [v5.0.0.1306, SuperServer, Firebird/Linux/AMD/Intel/x64]
  home: /opt/firebird
  bin: /opt/firebird/bin
  client library: libfbclient.so.2
rootdir: /home/job/qa-user
configfile: pytest.ini
plugins: firebird-qa-0.20.0
collected 2 items

query.parametrized.01 ..                                     [2/2]

===== 2 passed in 0.14s =====
```

Note that the test was performed twice. Each time with a different set of parameters.

## When a test fails

---

In addition to successful execution, the test can end with either an error or a failure. An error is reported if an unexpected exception occurs during test execution. Test failure is reported if any test condition is evaluated as false. In such a case, pytest will print the test and failure location, and an analysis of the tested condition.

## An example of a test failing due to a difference with the expected query output:

```
===== test session starts =====
platform linux -- Python 3.11.9, pytest-8.2.0, pluggy-1.5.0
System:
  encodings: sys:utf-8 locale:UTF-8 filesystem:utf-8
Firebird:
  configuration: firebird-driver.conf
  ODS: 13.1
  server: local [v5.0.0.1306, SuperServer, Firebird/Linux/AMD/Intel/x64]
  home: /opt/firebird
  bin: /opt/firebird/bin
  client library: libfbclient.so.2
rootdir: /home/job/qa-user
configfile: pytest.ini
plugins: firebird-qa-0.20.0
collected 1 item

query.03 F [1/1]

===== FAILURES =====
_____ tests/test_03.py::test_1 _____
act = <firebird.qa.plugin.Action object at 0x7fa529a11b50>
  def test_1(act: Action):
    act.expected_stdout = expected_stdout
    act.execute()
    # This evaluates test outcome
>    assert act.clean_stdout == act.clean_expected_stdout
E      assert
E          Select Expression
E          -> Filter
E          -> Table "JOB" Access By ID
E          -> Bitmap
E          - -> Index "JOB_FK1" Range Scan (full match)
E          ?           ^  ^^^^^\n
E          + -> Index "RDB$FOREIGN3" Range Scan (full match)...
E
E      ...Full output truncated (4 lines hidden), use '-vv' to show
tests/test_03.py:41: AssertionError
===== short test summary info =====
FAILED tests/test_03.py::test_1 - assert
===== 1 failed in 0.15s =====
```

# Processing test results

---

The results of the test run are printed on the standard output, which is enough for quick orientation. For a more detailed analysis of the results, the XML format is more suitable. Pytest allows you to save results in the standard JUnitXML format using the `--junit-xml=filepath` switch.

JUnit XML specification seems to indicate that "time" attribute should report total test execution times, including setup and teardown. It is the default pytest behavior. To report just call durations instead, add next option to the `pytest.ini` file:

```
junit_duration_report = call
```

The Firebird plugin defines the `--extend-xml` switch, which when used together with `--junit-xml` produces JUnitXML file with additional metadata for `testsuite` and `testcase` elements recorded as property sub-elements. Please note that using this feature will break schema verifications for the latest JUnitXML schema. This might be a problem when used with some CI servers.

With a sufficient set of tests, you can very quickly detect any problems with the transition of your applications to the new version of Firebird. Any test failures will show you problem areas, as well as the source of problems. By processing the JUnitXML output, it is possible to compare the execution times of individual tests for individual versions of Firebird. However, for a more accurate evaluation of the running time, we recommend repeating the tests, e.g. using the `pytest-repeat` plugin.





# Conclusion

---

In this short introduction to the use of Firebird QA tools, we have only touched very superficially on the possibilities of their use, and we have completely omitted some types of tests (e.g. writing to the database, calling procedures, etc.). However, our goal was not an exhaustive instruction manual, as detailed description of the features and various guides can be found in the documentation for pytest, Firebird QA plugin and Firebird driver for Python. Our goal was primarily to demonstrate two aspects of the use of these tools. First, that even with a minimum of work (and even without knowledge of Python if you have a usable template) you can create full-fledged tests of the application's interaction with the database, and secondly, that the presented tools and methodology have a much greater potential for use. We hope that we have succeeded in achieving it.

In the archive attached to this issue you will find a repository with used tests and databases. Another source of inspiration for possible test implementations can be the Firebird test repository, especially the tests in the `tests/functional` directory.

## Resources:

1. pytest <https://docs.pytest.org>
2. Firebird QA tools <https://firebird-qa.rtfld.io>
3. Firebird tests <https://github.com/FirebirdSQL/firebird-qa>
4. Firebird driver for Python <https://firebird-driver.rtfld.io>



## Interview with Hajime Nakagami

If you program for Firebird in Python, Go, Rust, Julia, Erlang or Elixir, then you probably use a Firebird driver created by Hajime Nakagami. That's why we asked him a number of questions for you.

**First, can you introduce yourself to our readers who don't already know you? Where do you live and what do you do for a living? Is your job in any relation to Firebird or your Open Source projects?**

I was born in 1967 and have lived in Japan my entire life. Currently, I work at BeProud (<https://www.beproud.jp/>), focusing on developing web systems using Python, Django, and MySQL. Firebird isn't part of my job toolkit, and I don't engage in Open Source Software development as part of my professional duties. This lack of dependency on Firebird and OSS in my role has had a positive influence. For instance, it grants me the freedom to participate in interviews like this one.

**Your portfolio of projects on GitHub is indeed impressive, and almost all of them are active. How do you manage it over all these years, and how much time does it cost you?**

I'm mostly active on weekends. When starting a new project, I spend a few months going through the documentation and protocols to understand if it's feasible. If it seems doable, I use my weekends to finish the first version in about 2-3 months. After that, I work on it when I want to or as issues come up. I handle several projects simultaneously, but I might not invest as much time in each as you might expect. Sometimes I lose interest and switch between them, so my time isn't evenly distributed across all projects.

**If I remember correctly, and feel free to correct me if I'm wrong, the oldest project related to Firebird is a Python driver called firebirdsql. What was your reason for creating this driver?**

Yes, pyfirebirdsql is probably the oldest project in my repository on Github.

Back in 2002, I was working on a simple web application using Zope and Firebird on a Windows platform. I opted for Firebird due to its straightforward installation process and usability on Windows.

However, when the maintainer of KInterbaseDB passed away, I realized there was no way to access Firebird from Python3. Upon examining the KInterbasDB source code, I concluded that it would be impossible for me to maintaining it.

Firebird is a pretty RDBMS, compact and compliant with SQL standards, and I thought it will be a great loss for Firebird if it became inaccessible from Python. This prompted me to begin developing pyfirebirdsql. Initially it worked only with Python3. Later, due to many requests, I made it work with Python 2.7 as well.

I would stay with this driver for a while, because as the author of the "official" Firebird drivers for Python, I have been following your work on firebirdsql almost from the beginning. I have to admit that when I was creating the FDB driver in 2011 as a replacement for KInterbasDB, which lost its main developer (and on which Firebird testing was built), I had no idea that someone else was working on something similar. When I proudly published the first version of FDB in December 2011, thinking that Firebird finally has a supported driver for Python again, it was a big surprise for me when I discovered your driver on PyPI the very next day, which at that time (in version 0.6.5) was already a year in development and with 20 released versions. So I hereby publicly declare that although FDB is (was, now it's firebird-driver) the "official" Firebird driver for Python, your firebirdsql is older and thus more original. Anyway, for 14 years now users can choose which driver they want to use (and I know that both are used). So the question is, how do they differ from each other, when and how is it more advantageous to use particular driver? The main difference between the drivers is that FDB (and its successor firebird-driver) uses the Firebird API through the client library, while firebirdsql directly implements the Firebird communication protocol and therefore does not need a client library at all. Can you think of any other differences?

Another difference is that pyfirebirdsql can connect to Firebird from version 2.5 to version 5.0 (and possibly 6.0), and supports both Python 2.7 and 3+.

And I think that just as there are multiple driver implementations of MySQL and PostgreSQL, it is worthwhile to have multiple implementations for Firebird.

Yes, I fully agree. While FDB supports Firebird 2.5/3.0 and both Python lines, it does not support Firebird 4+, and firebird-driver supports only Firebird 3+ and Python 3. With pyfirebirdsql, users are not forced to choose the path they want to constraint themselves. Especially for users that are still on Python 2.7, your driver is a better choice.



However, from personal experience I know that it is quite challenging to maintain a project for two so much different versions of Python. So, are you considering deprecating support for Python 2.7?

If possible, I would like to continue to support Python 2.7 as well. However, there's a possibility that future developments may make it challenging to sustain compatibility with both Python lines using a single source. If that happens, I will have to drop 2.7 support.

But maybe there is already no demand for Python 2.7? I don't know.

Most of your projects are written in Python. Considering how Python is rapidly developing, are you considering changing the minimum supported version to a higher one? Personally, I have been considering moving to at least version 3.10 for some time, mainly for the possibility of using Structural Pattern Matching to simplify and clarify the processing of variable data structures (there are many such places in drivers).

When new versions of Python adds new features (such as time zone support), I may discontinue support for older versions to use those features. I also discontinue support for Python versions that are no longer supported by Github Actions. However, I do not actively use the latest version features to clean up internal processes.

I think I would use Pattern Matching when starting some new project.

It could be said that your projects mainly fall into two large groups. The first consists of Python drivers for various database servers (MySQL, PostgreSQL, SQLServer, MongoDB, Redis, Cassandra, and others), and the second consists of Firebird drivers for various languages (besides Python, it also includes Rust, Golang, Julia, Erlang, and Elixir). What was your reason for creating these projects? Was it needed for other projects/work, or just purely out of interest in the issue or for fun?

Just for fun.

I write drivers for various databases in Python because Python is the programming language I am most familiar with, and I choose Firebird drivers as a subject to learn a new programming language because I am familiar with Firebird's wire protocol.

**If I'm not mistaken, almost all your drivers are based on the network protocol of individual databases. Can you compare the complexity of these protocols? How Firebird's protocol stands in comparison to other databases? Which implementation was the most interesting and which the most challenging for you, and why?**

I can only judge by experience with protocols that I worked with, but among RDBMS, Firebird protocol is the most complex and sophisticated. I think that Interbase developers were clever.

PostgreSQL's protocol is the simplest, but this does not mean it has poor performance. Since there are many databases that follow PostgreSQL's wire protocol, PostgreSQL's simple wire protocol is probably the best.

Drivers for non-OSS RDBMS are more difficult to implement, but for reasons other than being complex or sophisticated. That was not fun for me.

**When you evaluate your experience with creating Firebird drivers in different languages, which language was the best / most interesting experience? Did you come across any surprises?**

I was happy when the first Python driver I implemented worked well. However, I can't say that working with these programming languages would bring too different experiences.

Rather, I think good experiences come from collaboration. Drivers that get a lot of feedback and pull requests are good experience. I had the best experience with the Go driver, because it has many users and I have got a lot of feedback.

Currently, most boring for me is the Rust driver, because there is very little feedback on the code!

<https://crates.io/crates/firebirust>

For me, a programming language that is used by many people provides a good experience.

**What are your plans for the future? What are you up to this year?**

I don't have a plan. I don't want to learn a new programming language right now, but I want to get more experience with Elixir. So I would like write something in it.

Since there hasn't been an international Firebird conference for many years, we've lost even the little contact with the community in Japan. Can you enlighten us about the popularity of Firebird in your country, and how it is with the Firebird user community there in general?

Unfortunately, Firebird is not used much in Japan. Therefore, the user group is also mostly dormant. I think it's because many systems use managed cloud databases.

However, Firebird is getting better with each new version, and as long as the development of Firebird continues, I believe that the time will come when many people will use it and gather again.

Thanks for you time!







## Development update: 2024/Q2

A regular overview of new developments and releases in Firebird Project

### Releases:

- [Jaybird 5.0.5](#), released 14.6.2024
- [DBD:Firebird PERL extension 1.38](#), released 21.5.2024
- [Firebird SQL Client Library for Kotlin Multiplatform 1.0](#), released 2.6.2024
- [Python firebird-base 1.8.0](#), released 3.5.2024
- [firebird-driver for Python 1.10.4](#), release 7.5.2024
- [firebird-qa 0.20.0](#), released 9.5.2024

# Firebird release schedule

---

The Firebird 5.0.1 should be released in June. Releases of 4.0.5 and 3.0.12 should follow shortly after.

Update to 5.0.1 and 4.0.5 is highly recommended due to [Security advisory](#).

The official Firebird lifecycle policy is V+2, i.e. only the last two released versions are maintained. Therefore, after the release of Firebird v5.0, maintenance of Firebird v3.0 should be discontinued. But the migration to v4/v5 has been quite slow and many users are still using v3 and would certainly appreciate bug fixes. So the fate of the v3.0 series has not yet been sealed. It will probably be maintained until the end of this year (with a 3.0.12 release coming soon). It has not yet been decided whether it will be preserved after this time.

## Trouble with Schemas

---

Schema support is scheduled for Firebird v6.0, and the design and development was going well so far. However, there are some dragons hiding in the dark corners.

### The ARRAY type

The legacy API function `isc_array_lookup_bounds` is working with database metadata performing a query on system tables. This (client) function, when dealing with database engine that supports schemas, should look for `relationName` using the a search path like for other name resolutions.

But should we also create a new function (`isc_array_lookup_bounds2`) also expecting an explicit `relationSchema`? Relying on user encoding a schema inside `relationName` is not going to work, as currently it's possible to create object names that have dots and double quotes and would make things ambiguous.

Nobody expected that this problem will open a can of worms.

This function is vital for implementing support of ARRAY type in any Firebird driver that uses either Legacy or new OO API (as there is no equivalent in OO API). However, introduction of new schema-aware function will require adjustment in drivers that are almost impossible to implement. They would need to become a schema aware as well to pass the proper value to this new function, with no clear way how to achieve that. It seems that Schema and ARRAY type will become mutually exclusive features.

Well, the number of Firebird users that use the ARRAY type in their databases is likely zero, but nobody can tell for sure. There is only one known application that supports it (Firebird's ISQL tool) and only small number of drivers (official Python drivers for example). For years, the ARRAY type was treated as deprecated, although the Firebird project never officially deprecated it. The neglect went so far, that Firebird 5.0.0 was even released with the ARRAY support broken, see [#8100](#).

As ARRAY is once again popular type due to AI frenzy, it's likely that Firebird will see ARRAYs completely redesigned, rather than deprecated. But it's unlikely that this will happen in Firebird 6.0, as its feature set is already defined and will not be extended with other major features. For next version, we'll have to live with such mutual exclusivity.

*Few words about ARRAYs from Jim Starkey:*

*The impetus for the Interface array feature set was a requirement from the Boeing Aircraft noise group. They collected large number of sound samples from microphone arrays and needed to store and crunch these. They had been using an ad hoc database with array support and didn't like having to use to two different database systems.*

*In retrospect, I'm inclined to think that it was seriously over designed, but it had to work in an environments of tiny main memory (by today's standards) and networks that topped out at 10mbs. Then, a complete transfer of a large array was problem; now, not much.*



*In today's AI obsessed world, arrays, hierarchical navigable small world indexes, and a nearest operator are sine quo non if you want to play with the state of the art.*

*The Amorphous array info call and array descriptors are:*

```
virtual int getArray(Array* descriptor) = 0; // returns length of array in bytes
```

```
struct Array  
{  
int size; // number of elements  
int type; // data type of element  
union  
{  
const double* doubles;  
const float* floats;  
const int* ints;  
const unsigned char* bytes;  
const void* data; // used for generic references  
} array;  
};
```

*But, like I said, I may extend/redefine this for a vector of array bounds. And, incidentally, for AI stuff, 16 bit floats are very important; similarity vectors care a great deal more about magnitude than precision.*

## Replication

Replication may work within different engine versions, but this depends on data types of used fields and even DDL commands. It's certainly not recommended for production, but this setup may help when you update your Firebird server but still want to maintain database of old version until the upgrade is fully validated. However, there is no straightforward way to make this work when schemas are introduced and one of the sides do not support schemas, specially in the case of async replication. The consensus so far is that this case should be initially declared as unsupported.



## Toolbox: DBeaver CE

The DBeaver CE <sup>[1]</sup> is a free cross-platform database tool for developers, database administrators, analysts, and everyone working with data. It supports all popular SQL databases, including Firebird.

Windows users can install it using the executable installer, via Chocolatey, or from the Microsoft Store. Linux users can find it in the community repositories for their distribution, install it via the supplied Debian or RPM packages, or use Flatpack or Snap packages. MacOS users won't be left out either, with packages available for both Intel and Apple Silicon. DBeaver is also available as an Eclipse plugin.

DBeaver is written in Java and uses the JayBird driver to access Firebird. If you don't have this driver installed, DBeaver will automatically download and install it when you configure the first Firebird database connection.

In addition to the Community Edition, you can also purchase commercial DBeaver PRO with NoSQL and Cloud database support, visual database performance tools, and more. However, we will only focus on the CE version in this review.

When you launch DBeaver for the first time, you will encounter several initial setup windows. The first window you may see is the Create Sample Database dialog. If you don't want to play around a bit with the sample SQLite database, you can skip it.

To do anything useful with DBeaver, you must first define a connection to a database. You can proceed via the "Database" menu, or the icon or context menu on the Database Navigator panel. After selecting the Firebird server (and possible automatic installation of the JayBird driver), you will get to setting the connection parameters.

### Connection dialog

#### Generic JDBC Connection Settings



Firebird connection settings

MainDriver propertiesSSH

+ Network configurations...

General

Connect by: ☒ Host ☐ URL

JDBC URL: jdbc:firebirdsql://localhost:3050/employee

Host: localhostPort: 3050

Path: employeeOpen ...

Authentication (Database Native)

Username: SYSDBA

Password: .....☒ Save password

[You can use variables in connection parameters.](#)

Connection details (name, type, ... )

Driver name: Firebird

Driver Settings

Driver license

Test Connection ...

< Back

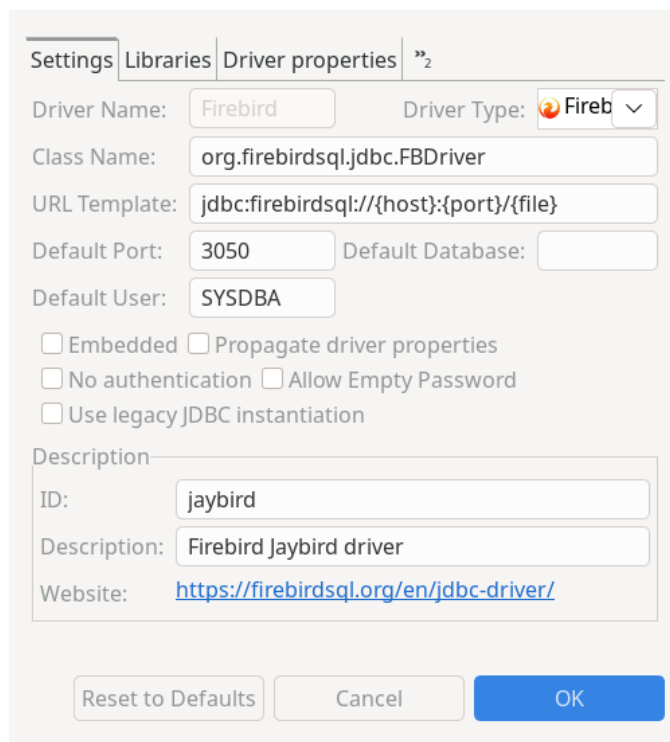
Next >

Cancel

Finish

Note that this is a generic JDBC connection setup where you can set any JayBird driver parameters. If you are not familiar with them, there is a link to the documentation. You also have the option to set up a secure connection via an SSH tunnel.

## Driver settings



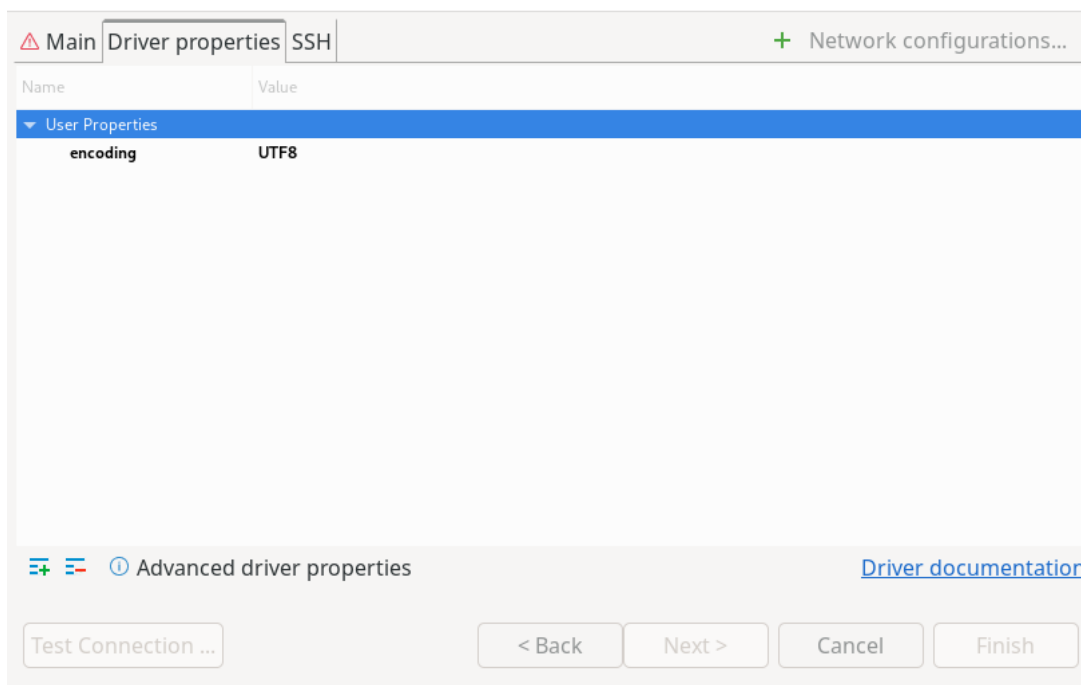
The screenshot shows a dialog box titled "Driver settings" with tabs for "Settings", "Libraries", "Driver properties", and "SSH". The "Settings" tab is active. It contains the following fields and options:

- Driver Name: Firebird
- Driver Type: Firebird (dropdown menu)
- Class Name: org.firebirdsql.jdbc.FBDriver
- URL Template: jdbc:firebirdsql://{host}:{port}/{file}
- Default Port: 3050
- Default Database: (empty field)
- Default User: SYSDBA
- Options:
  - ☐ Embedded
  - ☐ Propagate driver properties
  - ☐ No authentication
  - ☐ Allow Empty Password
  - ☐ Use legacy JDBC instantiation
- Description section:
  - ID: jaybird
  - Description: Firebird Jaybird driver
  - Website: <https://firebirdsql.org/en/jdbc-driver/>

At the bottom are buttons for "Reset to Defaults", "Cancel", and "OK".

### Generic JDBC Connection Settings

Firebird connection settings



The screenshot shows a dialog box titled "Generic JDBC Connection Settings" with tabs for "Main", "Driver properties", and "SSH". The "Driver properties" tab is active. It contains a table with the following data:

Name	Value
User Properties	
encoding	UTF8

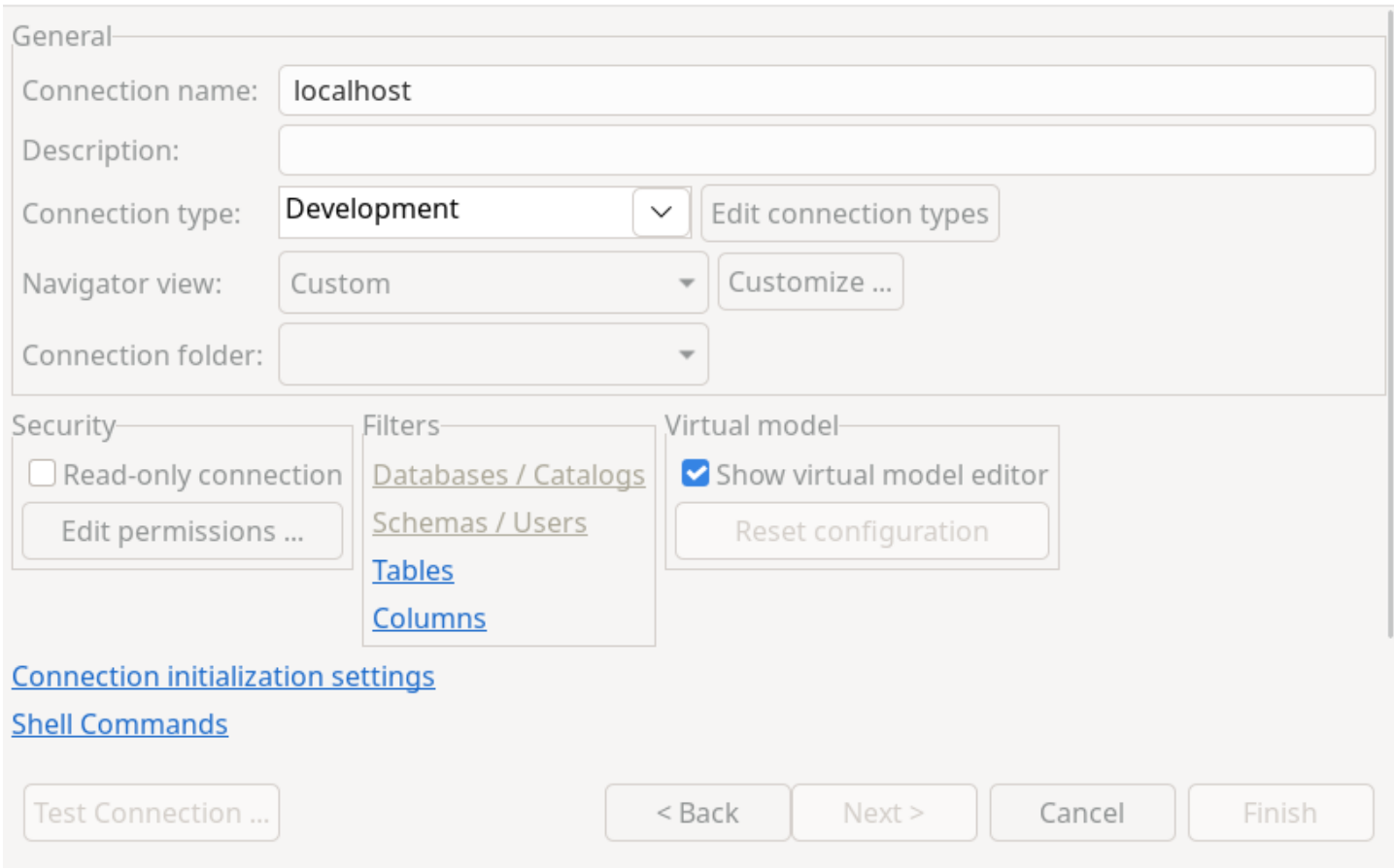
At the bottom are buttons for "Test Connection ...", "< Back", "Next >", "Cancel", and "Finish". There is also a link to "Driver documentation" and a section for "Advanced driver properties".

Configuration parameters for DBeaver can also be set. It mainly concerns display modes, allowed operations, running scripts, etc.

### *DBeaver connection settings*

#### **General**

General connection settings.



The screenshot shows the 'General' tab of the DBeaver connection settings dialog. The 'Connection name' field is set to 'localhost'. The 'Description' field is empty. The 'Connection type' is set to 'Development' with a dropdown arrow and an 'Edit connection types' button. The 'Navigator view' is set to 'Custom' with a dropdown arrow and a 'Customize ...' button. The 'Connection folder' is an empty dropdown menu. Below these fields are three sections: 'Security' with a 'Read-only connection' checkbox and an 'Edit permissions ...' button; 'Filters' with a list of expandable categories: 'Databases / Catalogs', 'Schemas / Users', 'Tables', and 'Columns'; and 'Virtual model' with a checked 'Show virtual model editor' checkbox and a 'Reset configuration' button. At the bottom, there are links for 'Connection initialization settings' and 'Shell Commands', and a row of buttons: 'Test Connection ...', '< Back', 'Next >', 'Cancel', and 'Finish'.

General

Connection name: localhost

Description:

Connection type: Development Edit connection types

Navigator view: Custom Customize ...

Connection folder:

Security

☐ Read-only connection Edit permissions ...

Filters

Databases / Catalogs

Schemas / Users

Tables

Columns

Virtual model

☒ Show virtual model editor Reset configuration

[Connection initialization settings](#)

[Shell Commands](#)

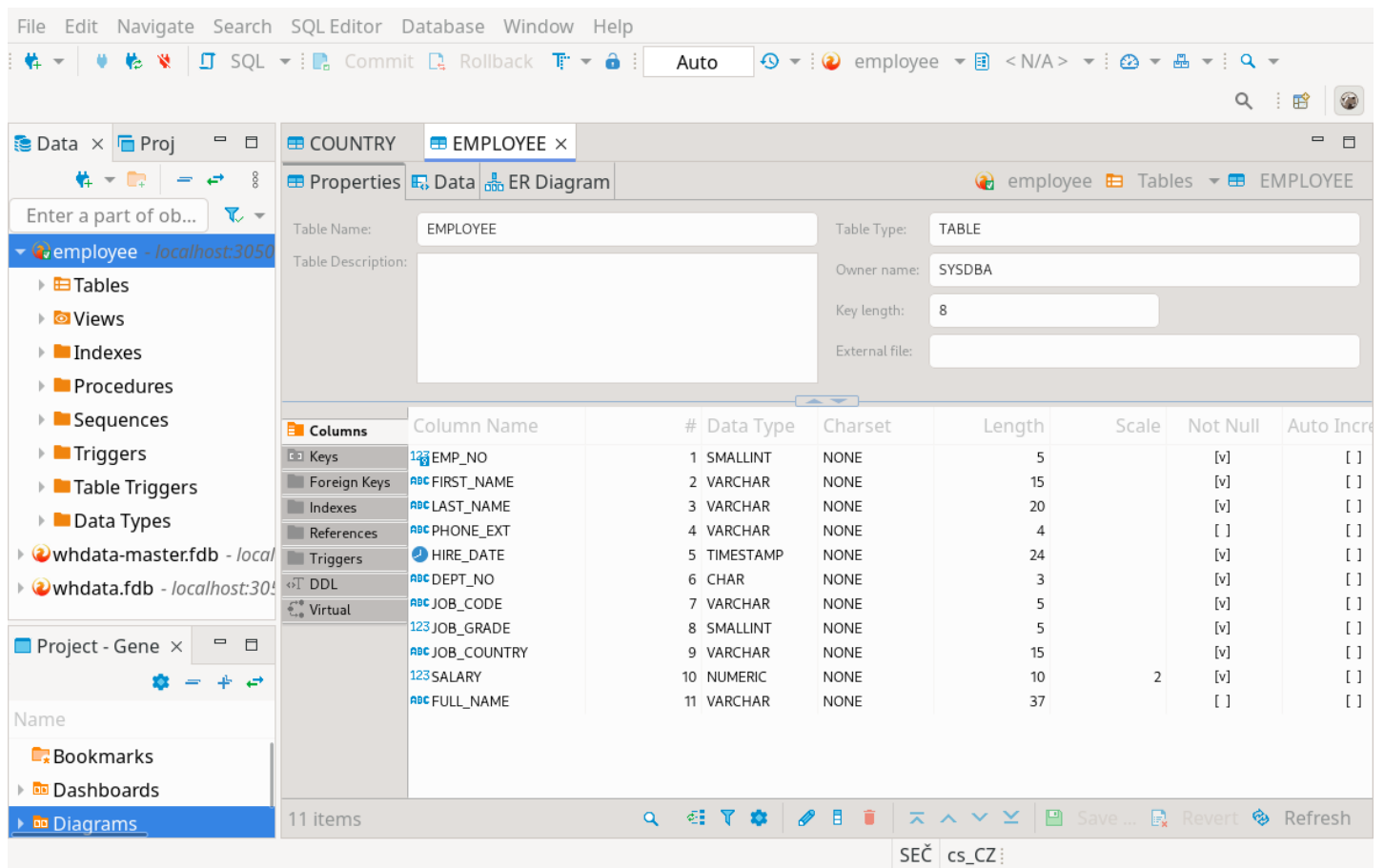
Test Connection ... < Back Next > Cancel Finish

Given the configurability, it makes sense to define multiple connections to the same database for different purposes. Connections can be organized into folders and subfolders.

The user interface consists of one window with standard elements: a menu bar, a toolbar, a workspace with one or more editors and views, and a status bar. It appears simple and straightforward, but could be very confusing. First, almost all panels could be minimized, maximized, resized or detached, and tabs moved around. Once you minimize a panel, it appears on one of two shortcut bars (left or right one) as icon. Sometimes it takes a while to figure out where things are.

But the most annoying thing is that an object once detached to a separate window cannot be reattached back to the main window (this bug was reported in May 2023 and is still open today). If you disconnect a significant panel like the Database Navigator and change your mind later, there is no easy way back because the layout persists between sessions (you have to use the Help→Reset Settings menu item).

### *DBeaver main window*



However, you can get used to the peculiarities of the environment, and its configurability can be an advantage compared to the more static interfaces of other similar tools.

The first thing you'll notice is that DBeaver is designed to work with existing databases, and its options for modifying the database structure are limited. You can use dialogs to modify existing objects, to add tables and views, indexes and referential constraints, but not triggers or stored procedures. However, what cannot be done using dialogs can be done using SQL commands.



The display of data in tables (or queries) is particularly pleasant, where in addition to the data itself you can also display aggregations, function results, data from linked tables, etc.

## Data view

The screenshot displays the DBeaver PRO interface. The main window shows the 'EMPLOYEE' table data in a grid view. The table has columns: EMP\_NO, FIRST\_NAME, LAST\_NAME, PHONE\_EXT, HIRE\_DATE, and DEPARTMENT\_ID. The data is sorted by EMP\_NO. The interface includes a menu bar (File, Edit, Navigate, Search, SQL Editor, Database, Window, Help), a toolbar with various icons, and a sidebar with tabs for Properties, Data, and ER Diagram. The 'Data' tab is active, showing the 'EMPLOYEE' table. The status bar at the bottom indicates '42 row(s) fetched - 0,010s (0,002s fetch), on 2024-05-27 at 16:56:14'. Below the main grid, there is a section for 'JOB\_CC' showing a count of employees by job code.

EMP_NO	FIRST_NAME	LAST_NAME	PHONE_EXT	HIRE_DATE	DEPARTMENT_ID
1	Robert	Nelson	250	1988-12-28 00:00:00.000	60
2	Bruce	Young	233	1988-12-28 00:00:00.000	62
3	Kim	Lambert	22	1989-02-06 00:00:00.000	13
4	Leslie	Johnson	410	1989-04-05 00:00:00.000	18
5	Phil	Forest	229	1989-04-17 00:00:00.000	62
6	K. J.	Weston	34	1990-01-17 00:00:00.000	13
7	Terri	Lee	256	1990-05-01 00:00:00.000	00
8	Stewart	Hall	227	1990-06-04 00:00:00.000	90
9	Katherine	Young	231	1990-06-14 00:00:00.000	62
10	Chris	Papadopoulos	887	1990-01-01 00:00:00.000	67
11	Pete	Fisher	888	1990-09-12 00:00:00.000	67
12	Ann	Bennet	5	1991-02-01 00:00:00.000	12
13	Roger	De Souza	288	1991-02-18 00:00:00.000	62
14	Janet	Baldwin	2	1991-03-21 00:00:00.000	11
15	Roger	Reeves	6	1991-04-25 00:00:00.000	12
16	Willie	Stansbury	7	1991-04-25 00:00:00.000	12
17	Leslie	Phong	216	1991-06-03 00:00:00.000	62
18	Ashok	Ramanathan	209	1991-08-01 00:00:00.000	62
19	Walter	Steadman	210	1991-08-09 00:00:00.000	90
20	Carol	Nordstrom	420	1991-10-02 00:00:00.000	18
21	Luke	Leung	3	1992-02-18 00:00:00.000	11
22	Sue Anne	O'Brien	877	1992-03-23 00:00:00.000	67

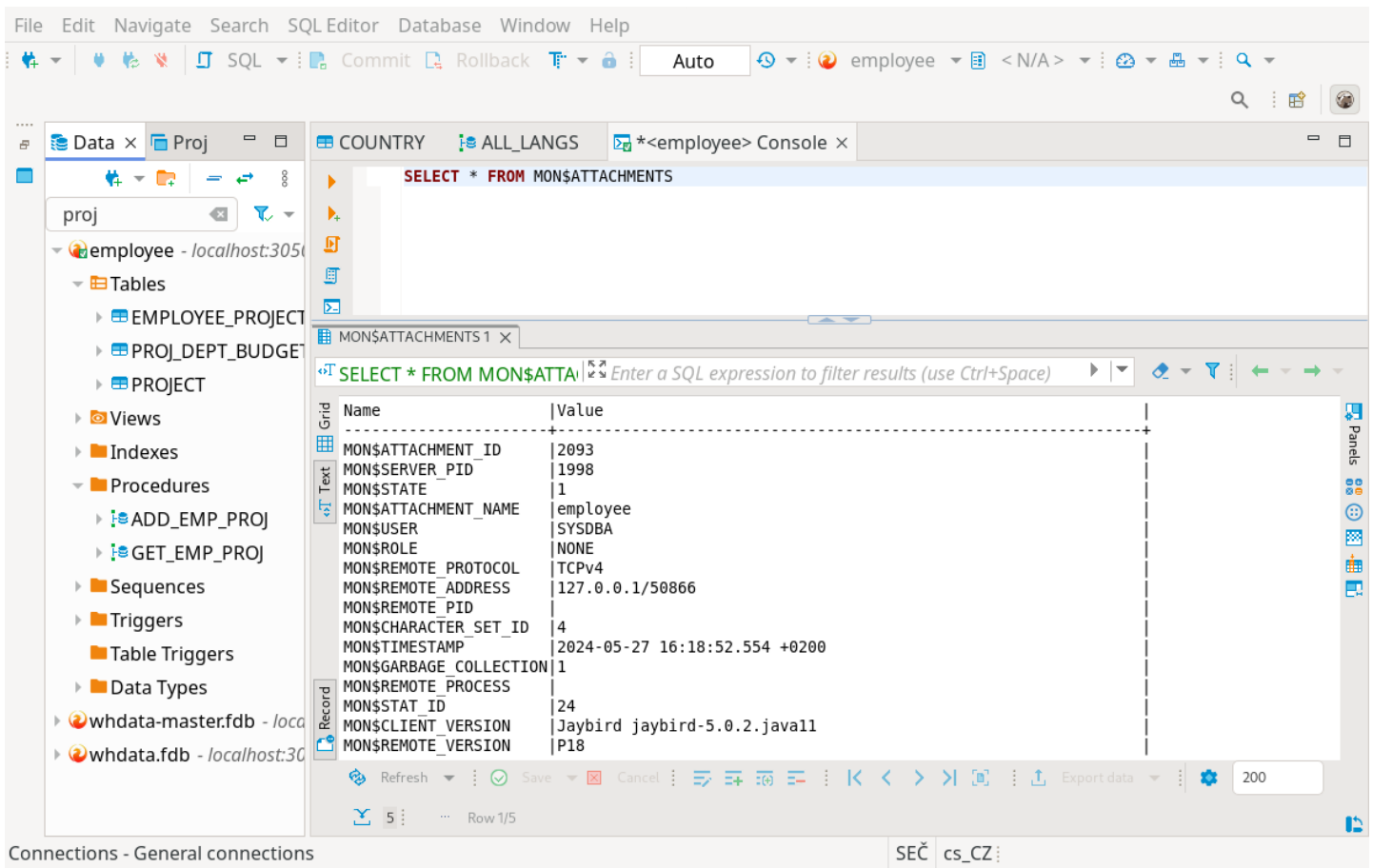
JOB_CODE	COUNT
Admin	4
CEO	1
CFO	1
Dir	1
Doc	1
Eng	15
Finan	1
Mktg	1
Mngr	4
PRel	1
SRep	8
Sales	2
VP	2

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

The SQL console offers the usual functions of editing SQL commands, including syntax highlighting and command completion, and a few less common functions such as SQL templates or searching for marked text using Google. DBeaver PRO users can also use AI (ChatGPT) to create SQL commands based on a text description. Support for SQL command parameters is also nice

In addition to displaying query results in a table (grid or text), an SQL terminal is also available where the results of all commands (ie not only queries) are displayed. The display of the execution plan is limited to the standard format only. What will certainly please you are the wide range of data export options.

## SQL console

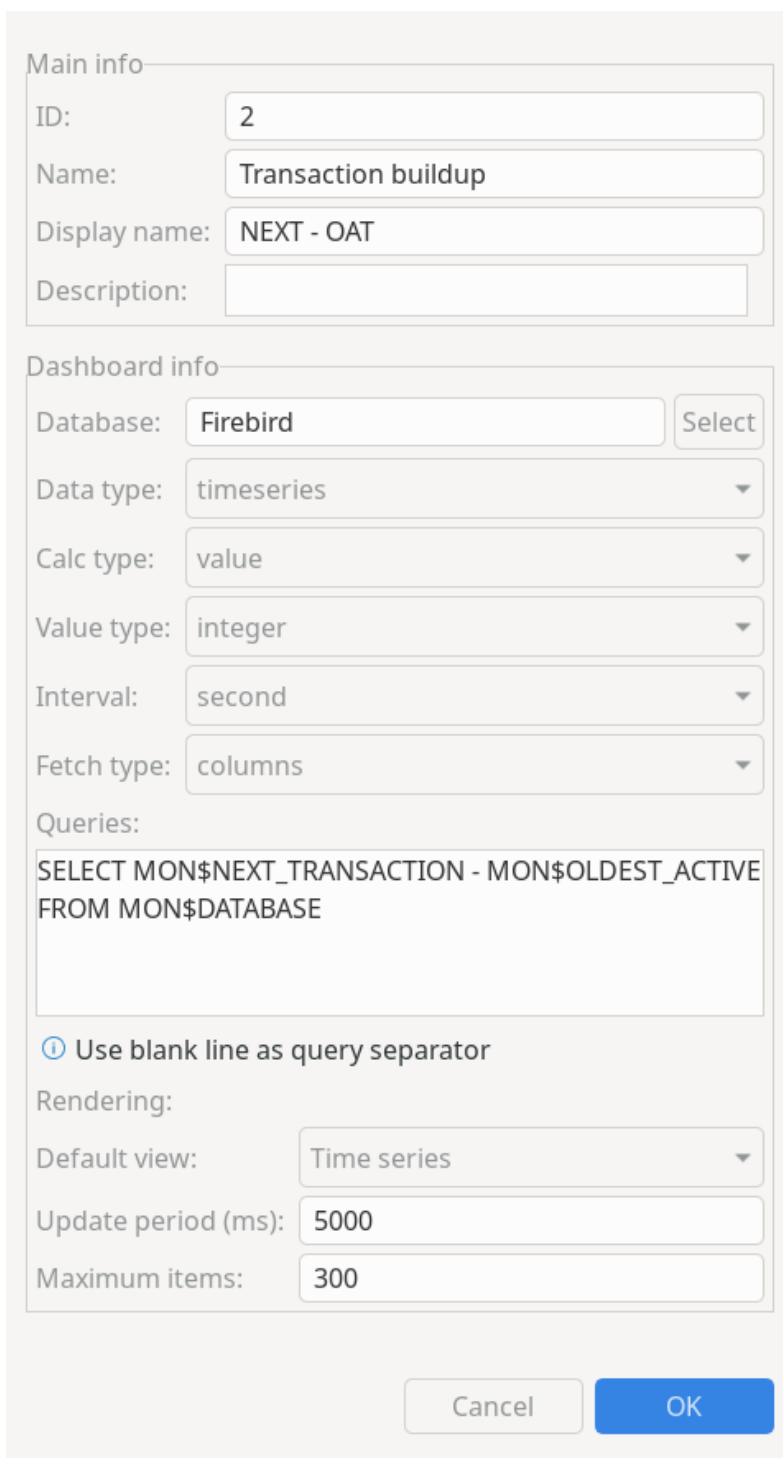


DBeaver also supports the generation of SQL commands, however, it is oriented towards working with data in tables and views. For the table, SQL commands can be generated from the Data grid for SELECT (WHERE = or IN), INSERT, UPDATE, DELETE selected row or CREATE table. From Database navigator, you can generate SELECT, INSERT, UPDATE, DELETE, MERGE or DDL commands for selected table.

Unfortunately, DBeaver is not able to work with Firebird DOMAINS. They will appear in the "Data types" category of Database Navigator, but are not listed for table items or procedure parameters where they are used. Instead DOMAIN reference, you'll get the basic SQL types used by the relevant DOMAIN. This inconvenience is also reflected in the generated DDL commands. It will also do not generate commands to create other related objects like triggers.

Unlike the limited usability of generated SQL scripts, a pleasant surprise is the possibility to create dashboards with visualization of any data or metrics. There are no default elements defined for Firebird, but it is not difficult to design and define your own.

### *Dashboard panel definition*



The image shows a web-based form for defining a dashboard panel. It is divided into several sections: 'Main info', 'Dashboard info', 'Queries', and 'Rendering'. The 'Main info' section contains fields for ID (2), Name (Transaction buildup), Display name (NEXT - OAT), and Description. The 'Dashboard info' section includes Database (Firebird), Data type (timeseries), Calc type (value), Value type (integer), Interval (second), and Fetch type (columns). The 'Queries' section contains a text area with the SQL query: `SELECT MON$NEXT_TRANSACTION - MON$OLDEST_ACTIVE FROM MON$DATABASE`. Below the query is a checkbox labeled 'Use blank line as query separator'. The 'Rendering' section includes Default view (Time series), Update period (ms) (5000), and Maximum items (300). At the bottom right are 'Cancel' and 'OK' buttons.

Main info

ID: 2

Name: Transaction buildup

Display name: NEXT - OAT

Description:

Dashboard info

Database: Firebird Select

Data type: timeseries

Calc type: value

Value type: integer

Interval: second

Fetch type: columns

Queries:

SELECT MON\$NEXT\_TRANSACTION - MON\$OLDEST\_ACTIVE  
FROM MON\$DATABASE

☐ Use blank line as query separator

Rendering:

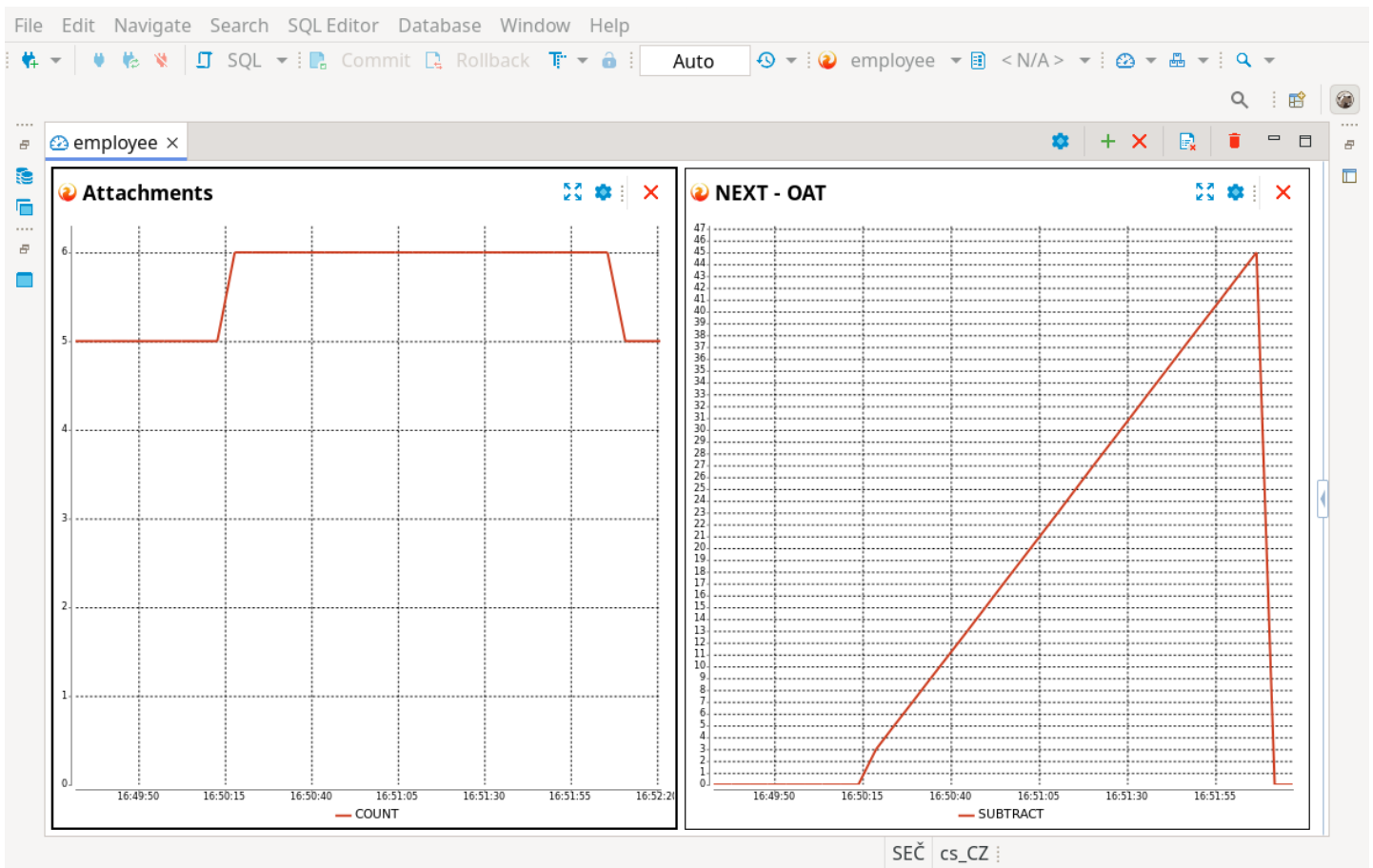
Default view: Time series

Update period (ms): 5000

Maximum items: 300

Cancel OK

## Dashboard with panels



DBeaver also offers other interesting functions, the description of which, however, goes far beyond the scope of this review, because they mainly concern the DBeaver PRO version or are more usable with other types of servers than Firebird. If you are interested in this product, we recommend that you read its (relatively well-crafted) documentation.

## Summary

DBeaver CE is an interesting tool, whose target group is definitely not database developers, but primarily database administrators and data analysts, to whom it offers a number of extremely useful and ergonomic functions. Its biggest advantage is the support of a wide range of different databases and data platforms, and the biggest weakness is the generic, and therefore limited, support for Firebird.

So if you work mainly with data in operational databases, DBeaver is a very interesting choice, especially if you also use systems other than Firebird. If your day job is creating or supporting database applications for Firebird, then DBeaver probably won't satisfy you.

#### **Advantages:**

- Multiplatform
- Freeware
- Dashboards
- ER diagrams
- Support for many database types

#### **Disadvantages:**

- Very generic support for Firebird
- An interface that doesn't suit everyone
- Advanced features not available in Community Edition

## **Our Rating:**

**For Firebird: 5/10**

**For other databases: 8/10**

#### **Resources:**

1. DBeaver CE, <https://dbeaver.io/>



## 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.



# Why table of "key-value" type takes so much disk space?

---

## Bruce Dickinson asked:

I want to keep key-value data associated with some object (ID\_MASTER) in master table. Currently I have 59 such parameters. Most of the time I am using only 2-5, so I thought that this structure will be ideal to save some space on the disk:

---

```
CREATE TABLE DETAILS_DATA (ID_MASTER INTEGER NOT NULL,  
    ID_PARAM INTEGER NOT NULL,  
    PARAM_VALUE VARCHAR(64) NOT NULL );  
  
ALTER TABLE DETAILS_DATA ADD CONSTRAINT FK_DETAILS_DATA_ID_MASTER  
    FOREIGN KEY (ID_MASTER) REFERENCES MASTER_DATA (ID);  
CREATE UNIQUE INDEX UNQ_DETAILS_DATA    ON DETAILS_DATA (ID_MASTER, ID_PARAM);
```

---

However, after running some tests I saw that database grows quickly. So for another test I created another table, this time with all keys in one row (i.e. 59 columns):

---

```
CREATE TABLE FLAT_DATA  
(ID_MASTER INTEGER NOT NULL,  
    P1 VARCHAR(64),  
    P2 VARCHAR(64),  
    P3 VARCHAR(64),  
    -- P4..P57  
    P58 VARCHAR(64),  
    P59 VARCHAR(64) );  
  
ALTER TABLE FLAT_DATA ADD CONSTRAINT FK_FLAT_ID_MASTER  
    FOREIGN KEY (ID_MASTER) REFERENCES MASTER_DATA (ID);
```

---

I've put to this table exactly the same data that was in DETAILS\_DATA table. Obviously, most of the P1-P59 columns were NULL.

Here is the comparison of space taken by both tables:

### DETAILS\_DATA:

```
Size of the table: 9592 MB
FK_DETAILS_DATA_ID_MASTER: 953 MB
UNQ_DETAILS_DATA: 1161 MB
TOTAL SPACE: 11706 MB
```

### FLAT\_DATA:

```
Size of the table: 2084 MB
FK_FLAT_ID_MASTER: 52 MB
TOTAL SPACE: 2136 MB
```

As you can see DETAILS\_DATA takes 5 times more of space. I was completely surprised by this result, after all I am not wasting space for 50+ columns. Could you explain me this phenomenon?

### Dmitry Yemanov answers:

How long are actual strings inside PARAM\_VALUE? It's worth looking at the gstat -r output and compare the average record size in both cases.

With the former (DETAILS\_DATA) approach, the table is very narrow. Storage overhead (record header size, 13 bytes) is nearly the same as the data itself (perhaps even more, considering the data compression). It could explain the wasted space.

### Bruce Dickinson confirms:

Here are the results of gstat:

```
FLAT_DATA (290)
  Primary pointer page: 5239142, Index root page: 5239147
  Average record length: 213.90, total records: 8288203
  Average version length: 0.00, total versions: 0, max versions: 0
  Data pages: 533746, data page slots: 533746, average fill: 88%
  Fill distribution:
    0 - 19% = 0
    20 - 39% = 0
    40 - 59% = 0
    60 - 79% = 30
    80 - 99% = 533716
```

```
DETAILS_DATA (262)
  Primary pointer page: 842, Index root page: 843
  Average record length: 20.19, total records: 164512578
  Average version length: 0.00, total versions: 0, max versions: 0
  Data pages: 2455600, data page slots: 2455600, average fill: 61%
  Fill distribution:
    0 - 19% = 0
    20 - 39% = 1
    40 - 59% = 0
    60 - 79% = 2455599
    80 - 99% = 0
```

I guess this confirms your statement about record header size.

FLAT\_DATA record length is 213.90 which is over 10 times more than DETAILS\_DATA record length 20.19. However in DETAILS\_DATA we have 164512578 records which is 19.8 times more than in FLAT\_DATA.

## The maximum key size limit revisited

---

### Geoff Worboys asked:

Firebird v2.5.3 - Windows 32bit server Database page size is 8192, ODS v11.2.

I'm getting "key size exceeds implementation restriction for index" when I try to create the following index:

```
CREATE INDEX MyIndex1 ON MyTable1 (Field1, Field2);
```

where:

```
Field1 is VARCHAR(80)
Field2 is VARCHAR(255)
```

both use the database default CHARACTER SET WIN1252 with collation WIN1252\_NOCASE as defined like this:

```
CREATE COLLATION WIN1252_UNICODE FOR WIN1252

CREATE COLLATION WIN1252_NOCASE FOR WIN1252
  FROM WIN1252_UNICODE CASE INSENSITIVE
```

As it happens, I'm happy not to have this particular index, but I'd like to understand what's going on.

I've looked at [Index Calculator](#) and I cannot find a combination for the above specification that exceeds 2048 bytes (1/4 page size) - 1260 is the figure it would lead me to expect, I think, though I'm not all that clear about the distinctions between the levels of collation. But even UTF8 at those sizes is reported at only 1675 bytes.

Is the restriction in this case expected, or should I be looking for some other problem? Maybe I have my collation set up incorrectly?

**Dmitry Yemanov answers:**

**IIRC, unicode derived case/accent insensitive collations use six bytes per character encoding.** This gives us 2010 bytes which is pretty near the 1/4 page size. Given that some overhead should be taken into account, the key length may in fact overflow the limit.

**Geoff Worboys confirms:**

That fits. For example I can create an expression index of the same (335 character) size, and that works, so it must be the multi-segment overhead that is killing it.

On a test database with page size 4096 I just tried these:

```
CREATE INDEX myindex ON testtable COMPUTED BY  
(CAST(afield || 'a' AS VARCHAR(169)) COLLATE WIN1252_NOCASE);
```

that one  $((169 * 6) + 9 = 1023)$  works.

```
CREATE INDEX myindex ON testtable COMPUTED BY  
(CAST(afield || 'a' AS VARCHAR(170)) COLLATE WIN1252_NOCASE);
```

that one  $((170 * 6) + 9 = 1029)$  fails with: "key size exceeds implementation restriction for index".

Which seems to confirm what you've said.



## Optimization of the database structure

In general, it is recommended to reoptimize the database once in a while using backup and restore. However, most users have no idea under what conditions or how often to perform this operation, what problems it solves, and which ones it doesn't.

This article explains how the internal structure of the database affects the efficiency of the Firebird engine, how this structure typically changes over time, and how it can be modified in a way that improves Firebird's performance.

A database is made up of blocks of equal length, called pages. The pages are divided into groups according to the type of data they contain, for example there are Data pages in which the data of individual tables are stored, BLOB pages or pages containing only links to other pages. The efficiency of the engine's work is influenced by both the internal organization of data on individual pages and the location of pages within the database.

## Page types

---

Currently (ODS v13.1) there are 10 page types. Their detailed description can be found, for example, in "Firebird Internals" <sup>[1]</sup>. Here we will focus only on the role of individual pages in relation to the efficiency of Firebird's work.

### Header Page

---

This page is always the first page in the database and contains basic information about its structure and parameters. This page is updated very often (e.g. at the start of each transaction.) and is therefore forcibly written to disk. This fact has no major effect on performance. However, you should be aware that even otherwise read only transactions always perform some writes.

### Page Inventory Page

---

This type of page is used to register all other used and free pages in the database in a very economical format (one bit per page). The number of these pages in the database is typically low, and their update frequency is also low. So they have a completely negligible effect on the efficiency of Firebird's work. However, you won't spoil anything if the page size corresponds to the (expected) total size of the database. E.g. at a page size of 8K, one PIP contains information about 65,312 pages, and at a size of 16K, information about 130,848 pages. So in a 200GB database with an 8K page size there will be a total of 401 PIP pages, but using 16K pages there will be only 101 PIP pages.



Pages of this type are distributed evenly in the database, and their placement cannot be influenced in any way.

## Transaction Inventory Page

---

This type of page is used to record the status of transactions. Basically, it is a persistent version of a data structure stored in memory. Information from these pages is used only when Firebird is started, and is an important part of the recovery mechanism after a system crash. In normal operation, these pages are written at the end of each transaction, while only one TIP is ever written, on which the status of the transaction is stored.

The number of TIP pages corresponds to the current size of the transaction map in memory (NEXT - OIT), where the state of one transaction is stored in two bits. Although the number of TIP pages does not have a major impact on Firebird's performance, working with these pages is one of the factors that contributes to the gradual fragmentation of the database over time.

## Pointer Page

---

A pointer page is used internally to hold a list of all data pages (see below) that make up a single table. Large tables may have more than one pointer page but every table, system or user, will have a minimum of one pointer page.

The list of pointer pages is stored in the RDB\$PAGES table, and you can find the pointer pages for any table by running the query like:

```
SELECT P.RDB$PAGE_SEQUENCE, P.RDB$PAGE_NUMBER, P.RDB$RELATION_ID
FROM RDB$PAGES P
JOIN RDB$RELATIONS R ON (R.RDB$RELATION_ID = P.RDB$RELATION_ID)
WHERE R.RDB$RELATION_NAME = 'MYTABLE'
AND P.RDB$PAGE_TYPE = 4
ORDER BY 1;
```

The number of Data Pages that single Pointer Page holds depends on page size (for example 1,632 pages for 8K page, and 3,270 for 16K page). Since Firebird v3.0, the `gstat` output also includes number of Pointer Pages for each listed table.

The list of data pages stored on Pointer pages is a fundamental data structure used to address rows of data, and the well-known `RDB$DB_KEY` refers to, among other things, the sequence number of the data page in this list. A simple calculation is used to identify a specific Pointer Page and the location of the number of the searched data page in the list stored there. The number of pages of this type has a certain influence on the efficiency of Firebird's work. Although a single page of this type is sufficient to find one specific row, the processing of more rows may require the processing of more Pointer pages, and therefore greater demands on the size of the database cache. Therefore, if you do not use Firebird in the SuperServer architecture, it is necessary to take into account the processed amount of these pages when calculating the optimal cache size.

The location of Pointer pages within the database can only be influenced with difficulty, as they are created as needed. Typically, these pages intersperse sequences of Data, BLOB and Index B-tree pages.

## Data Page

---

Each data page belongs exclusively to a single table. Beside the page header it contains an array of pairs (called `slots`) of offset / length unsigned short values that points to actual data contents. This array fills from the lowest address (increasing) while the actual data it points to is stored on the page from the highest address (descending). Part of well-known `RDB$DB_KEY` refers to particular `slot` on Data page.

Individual slots can contain not only data records, but also fragments of (too long) records, versions of individual records, or ARRAY or BLOB type values.

The internal structure and location of individual data pages within the database play a significant role in the efficiency and thus the performance of Firebird.

The most important attribute of each data page is its occupancy. This can be expressed by the absolute amount of stored data or the number of occupied slots. The output from the `gstat` program provides a basic overview of the occupancy of the data pages for the analyzed tables.

AR (140)

```
Primary pointer page: 297, Index root page: 299
Total formats: 1, used formats: 1
Average record length: 2.79, total records: 120
Average version length: 16.61, total versions: 105, max versions: 1
Average fragment length: 0.00, total fragments: 0, max fragments: 0
Average unpacked length: 120.00, compression ratio: 42.99
Pointer pages: 1, data page slots: 3
Data pages: 3, average fill: 86%
Primary pages: 1, secondary pages: 2, swept pages: 0
Empty pages: 0, full pages: 1
Blobs: 125, total length: 11237, blob pages: 0
    Level 0: 125, Level 1: 0, Level 2: 0
Fill distribution:
    0 - 19% = 0
    20 - 39% = 0
    40 - 59% = 0
    60 - 79% = 1
    80 - 99% = 2
```

Page fill factor in absolute amount of stored data is presented only as average page fill and as a histogram of absolute page fill distribution in segments of 20 percent.

The number of used slots is presented only as an aggregate number for all data pages of the given table. In addition, it is divided into individual types of records: primary records, versions, fragments and BLOBs (`gstat` does not distinguish between ARRAY and BLOB values because they are stored in the same way). Therefore, for the total number of slots used, it is necessary to add up all partial values. In the example above, the total number of slots is 350 (the sum of 'total records', 'total versions' and 'Blobs'), and the average occupancy is therefore 116 slots per page.

The occupancy of the pages is double-edged. High fill speeds up data processing. Especially for queries that process a large number of records, the performance gain can be very significant. On the other hand, it degrades performance if data is updated or deleted (not on insert), because the row versions often cannot be placed on the same page as the primary record. This reduces the speed of not only the update itself, but also the subsequent reading of the data, if it is necessary to find the required record in the chain of versions. For that reason, when writing new records, the data pages are not filled to full capacity by default, but a reserve (approx. 20 percent) is left for possible changes.

However, it is not only about the amount of data on the page, but also about the amount of records (occupied slots). The amount of primary records on a page is sometimes referred to as page density. The higher this value, the faster the data processing speed. Page density is affected by the presence of BLOB/ARRAY values or record versions that are stored in separate slots and therefore take up space for primary records. Firebird since version 3.0 tries to alleviate this problem by dividing the data pages into primary and secondary. Primary pages contain only primary records and possibly their versions, secondary pages contain only record versions, fragments and small BLOB (level 0) values. E.g. in the example above, the table has one primary and two secondary data pages. During data processing, the most often processed data are primary records. Moving other types of records to separate pages not only increases the density of primary records on a page, but also reduces the total number of pages that are processed.

But this optimization is not an universal solution. For tables that do not contain BLOB values, its benefit will only be seen in applications that repeatedly update the same data and therefore create longer chains of versions that overflow into separate secondary pages.

An inappropriate database design can then overwhelm any optimization. The amount of data in a record is determined by the structure of the table. A larger amount of data in the record automatically translates into a lower data density. In the worst case, fragmentation of records (forced division into multiple pages) can occur when there is no longer enough space for the entire record when writing to the page.

Another factor affecting the efficiency of data processing is the distribution of data pages within the database. Data pages belonging to individual tables are added as needed. These pages are usually added to the end of the database, but Firebird can also use any free pages inside the database created, for example, by deleting a large amount of data, changing indexes, freeing unnecessary TIP pages, etc. It is therefore common that data pages are scattered in the database in non-contiguous blocks. This so-called fragmentation of the database reduces the efficiency of I/O operations, especially with classic disks. With SSD disks, this problem does not manifest itself in any fundamental way, but the use of SSD has its own disadvantages and limitations.

## Index Root Page

---

Every table in the database has an Index Root Page which holds data that describes the indexes for that table, even if there are no indices defined for the table.

Pages of this type are static, and their internal structure or location within the database has no effect on data processing efficiency.

## Index (B-tree) Page

---

The Index Root Page points to the first page in the index. That page will be a Index B-Tree Page, as will all the other pages that make up this index. Each index has its own range of dedicated pages in the database. Pages are linked to the previous and next pages making up this index.

The content of these pages is quite complex and dynamic. Although Firebird strives for the maximum occupancy of index pages, in practice it can be very different, because when the content of the index changes, not only new pages are added, but also they are divided or merged. It is therefore quite common for indexes to be highly variable in their effectiveness, and to contribute significantly to database fragmentation.

Important properties of the index structure can be found from the output of the ``gstat'` program:

```
Index IDX_OBJ_OWNER (26)
  Root page: 163419, depth: 2, leaf buckets: 159, nodes: 519623
  Average node length: 4.94, total dup: 519621, max dup: 518196
  Average key length: 2.00, compression ratio: 0.50
  Average prefix length: 1.00, average data length: 0.00
  Clustering factor: 9809, ratio: 0.02
  Fill distribution:
    0 - 19% = 0
    20 - 39% = 1
    40 - 59% = 0
    60 - 79% = 0
    80 - 99% = 158
```

The most important values are:

1. **Depth.** Specifies the number of index pages that must be processed to find a specific key. A value of 4 or greater already means a measurable drop in index performance, especially for non-range operations. The depth of the index can be influenced only with difficulty. For indexes with low page occupancy, rebuilding the index (by deactivating and reactivating) may be sufficient. The only other option is to increase the database page size (if possible).
2. **Fill distribution.** If the average occupancy of the pages is 60 percent or less, it is advisable to rebuild the index. However, the ideal value is 80 percent or more.
3. **Clustering factor.** Clustering factor is the relationship between the order of the table and the order of the index. Indexes are always stored in key value sequence, while tables are stored in the order of the inserts. The higher the clustering factor is, the more the index points to different data pages. Indexes are efficient when they point to just a few of the table data pages. It comes into play with Range Scan operations when executing SQL commands, where it translates to more data page reads (roughly: ratio \* number of keys in range). The best clustering factor is equal to the number of primary Data pages.

The remaining data are also useful, but their importance from the point of view of the internal structure of the database is rather supplementary.



## BLOB Page

---

Pages of this type are only used for BLOB values larger than can be stored as a normal record on a data page (i.e. type 1 and 2). Because BLOB values are typically stored along with the rows that reference them, BLOB pages in a database typically interleave sequences of data and index pages. Values that do not fit on one BLOB page usually form a continuous block of BLOB pages.

## Generator Page

---

Every database has at least one Generator Page, even if no generators (sequences) have been defined by the user. A blank database consisting only of system tables and indices already has a number of generators created for use in naming constraints, indices, etc.

This page contains generator values only. Information about the generator (identification number, name, etc.) is stored in the system table `RDB$GENERATORS`. The list of existing Generator pages can be found from the `RDB$PAGES` table by querying:

```
SELECT RDB$PAGE_NUMBER
FROM RDB$PAGES
WHERE RDB$PAGE_TYPE = 9;
```

In normal cases, the effect of Generator pages on performance is absolutely zero, unless you trip over a peculiarity of the implementation of generators: If you delete a generator, the freed space on the Generator page is not reused. So if you get the idea to use temporary generators, e.g. to simulate semaphores to synchronize between stored procedures, triggers, etc., the number of Generator pages in the database will grow continuously with all its unpleasant consequences.

## SCN (Page Scan) Inventory Page

---

These pages contain a global array of SCN numbers (used by NBACKUP) for each database page. They are updated whenever a page is assigned a new SCN and are usually evenly distributed across the database. However, if you are not using NBACKUP, these pages are not present in the database. If present, they have no effect on the efficiency of data processing.

## Evolution of the Database

---

If you are not using the database in read-only mode, the internal structure of the database undergoes gradual evolution. It varies depending on the type of application, the way data is handled and the number of concurrent users. Nevertheless, the way of working with the database shows certain common features in most applications. One of the typical features is the proportional distribution between inserting, changing and deleting data. In most applications, inserting new records makes up the majority of all operations (about 80 percent). The rest of the operations are mostly updates, and data deletion is only a tiny fraction.

This is because most of the data is no longer changed (except for occasional corrections), and in many cases is not even deleted. If the deletion occurs, then typically in batches as part of cleaning the content from historical data that is no longer needed.

A certain exception to this rule are databases whose design includes tables containing aggregated data (for example, the amount of goods in stock), which have a higher percentage of updates than the rest of the data. However, since such a design significantly reduces throughput when multiple users work simultaneously, it has not been used much in the last twenty years.

Most databases begin their journey as mostly empty containers that gradually fill up. First with auxiliary data in the form of code lists, configurations, etc., then comes the main phase of entering and processing data from the main problem domain.

During the main phase of the database's life, individual pages are created according to current needs. The result is automatic fragmentation of the database, when logically related pages are scattered throughout the database, as they were gradually created. But pages don't always have to be arranged in chronological order, because if the data from an entire page is released, that page is reused for a new purpose.

There is also gradual fragmentation of the data stored on the Data and Index pages. Data pages are fragmented only for tables whose data is updated or deleted. Index pages are always fragmented for table indexes that are updated or deleted, however, even indexes for tables that are only added to are not spared from fragmentation, especially if the values of the added keys do not form an ascending sequence.

Data and index page content fragmentation can be evaluated using the `gstat` report. Unfortunately, Firebird does not offer any tool that would allow you to map the level of database fragmentation at the level of individual page types and their logical groups. When assessing the level of fragmentation at the page level, the user is therefore only dependent on indirect indicators.

The basic fact is that any added data is automatically fragmented at the page level. The more users simultaneously participate in inserting new data into different tables, the greater the fragmentation. Thus, for example, a database that stores data from sensors inserted by a single application in batches at regular intervals will have a significantly lower level of page fragmentation than a database that stores data from a CMS system with which dozens of people from different departments work at the same time.

## Revitalization of the database

---

The aim of revitalizing the database is to reduce the fragmentation of the content of both individual pages and the database as a whole. Typically, revitalization is done by restoring from a backup with the `gbak` program (however, there are other methods as well). But when to carry out this revitalization?

There is no simple answer to this question, as it depends on a number of factors. Many Firebird users don't care about this problem at all as long as they are satisfied with the data processing speed. This is a perfectly reasonable approach, especially if your database is less than 80GB (i.e. roughly 5 millions of 16K pages). However, if you have a larger database, or if the largest table contains more than 10 million records, you can often achieve better performance by revitalizing the database. Especially in the case of analytical processing of large amounts of data (various reports, etc.), which normally take minutes or even hours, a substantial acceleration of work can be achieved.

In general, the larger your database (in the order of hundreds of GB) and/or the larger your tables (in the tens of millions of records), the greater the performance gain. Quantitative estimation of this profit is difficult, because it depends on the specific parameters of the database and the way of working with data, but in general an average improvement in the range of 5-15 percent can be achieved, and for specific tasks (reports, etc.) then even in tens of percent. If you're interested in such a performance gain, then your database is a good candidate for revitalization.

The conditions for when to carry out initial revitalization are therefore clear. However, database fragmentation is a continuous process that cannot be completely prevented, and revitalization should therefore be performed repeatedly. But when to repeat it?

The main factors are the increase in database size, the amount of changed/deleted content, and the level of fragmentation of data and index page content over time. In general, it can be said that revitalization should be carried out whenever any of the following conditions are met:

1. The database has grown by 20 percent or more since the last revitalization.
2. At least 15 percent of the database content has been changed or deleted since the last revitalization.
3. The report from the gstat program indicates increased fragmentation of data and/or index pages (see above).

## gbak and others

---

Basic revitalization could be achieved by restoring from a backup with the gbak program.

When restoring from backup, a single transaction is used to gradually insert the data of individual tables, and then individual indexes are created. This achieves minimal fragmentation of data and index pages within the database, so that the pages of one table or index form a more or less continuous block (because they are still sporadically interspersed with infrastructure pages, see Pointer and Page Inventory Pages, but a better result cannot be technically achieved). Also, information on data and index pages is stored with maximum efficiency and density.

However, this revitalization has its limits:

1. The data pages contain a reserve for updates, so the data density is not at its maximum (there are fewer records on the primary pages), regardless of whether in practice the data in the given table is updated or only added (and the existing data is therefore static). Although reserve retention can be turned off at the database level, this practice is not recommended because it will cause problems with updated tables.
2. Tables are inserted into the database in the order in which they were created, and this order cannot be changed in any way. If it were possible to save the data of tables that do not undergo changes or are only extended to the beginning of the database, and only then the tables that change, the extent and speed of fragmentation of the database would be reduced, because changes would only occur in its specific part.
3. Table records are stored in the same order as they were inserted into the original database. The Index Clustering of each index therefore remains exactly the same and cannot be influenced in any way.
4. If you use the parallel processing option when restoring a backup, the index pages are interleaved because multiple indexes are created simultaneously.

Of course, better results can be achieved, but not with gbak. After all, this is a backup and recovery tool, not a database optimization tool. This alternative is the **IBPhoenix FBOpt** program, which solves the above problems and thus allows you to achieve an even better performance gain.

First of all, it allows the user to define or automatically detect tables whose data either does not change or is only added. The data of these tables are then stored at the beginning of the database without a reserve for updates, and therefore with the maximum possible density. Subsequently, the indexes of these tables are stored, and only then the data of the tables, the content of which is updated or deleted, keeping the reserve for updates. Finally, the remaining indexes are stored.

Automatic detection is performed using a timeline analysis of stored data. By heuristic analysis of this timeline, the mode of working with individual tables can be determined with a fairly decent degree of reliability. However, for highly reliable results, it is necessary to compare two snapshots taken with a sufficient time interval (so all possible changes could be seen). FBOpt supports both methods.

FBOpt also allows you to specify one index for each table, which should have the maximum Clustering factor. The data is then inserted sorted by the key of the specified index.

As the icing on the cake, FBOpt can also be used to upgrade or **downgrade** databases between Firebird versions (ODS structures), convert databases to new default character set, or to create regular gbak database backups with **simultaneous** restore.

FBOpt works with Firebird version 3.0 or newer.

## Resources:

### 1. [Firebird Internals](#)





## ...And now for something completely different

A young developer was tasked by his boss to optimize a complex SQL query for Firebird. He struggled with it for several days without success, until finally he began to despair. His older colleague couldn't bear to watch it, so he sent him to work outside in the fresh air to clear his head.

So the developer packed up his laptop and headed to a nearby beach, intending to find a quiet spot and continue working. As he walks along the beach in thought, he trips over a sealed bottle, absentmindedly picks it up and opens it. A genie jumps out of the bottle and says: "For freeing me from my prison, I will grant you one wish".

The young man looks at the genie and asks: "Do you know Firebird? I mean the database server."

The genie smirks and says, "We genies have access to all the information in the world, including the biggest secrets. Of course I know the Firebird database server. I even read its source code when I was in the mood for modern poetry. What do you need?"

And the young man opens the laptop and immediately starts: "You know, I want to ask my boss for a raise, so I need him to see me as very capable and indispensable. My only wish is that you help me with this. He gave me a task to optimize this SQL query for Firebird to make it at least twice as fast, but I still can't crack it. If you help me solve it, I'm sure he won't refuse my request."

The genie just looks at the SQL query with one eye and frowns, "You're kidding, right? This CTE is crazy, and I see more than thirty joins and subqueries. This will take me at least a day and night! We genies know everything, but that doesn't mean we're masters at everything. There has to be another, faster way to get you a raise, right?"

The young man thinks and says, "There would be one more possibility, and since you genies know everything, including the biggest secrets, it should be a small thing for you. You know, we run huge databases on Firebird, and my boss asked me to find out how to verify the validity of the backup without having to restore it to disk, which would save a lot of space. I'll be a star if I can tell him how, but I can't find a way on the Internet. Do you know the answer?"

The genie grimaces and says: "Show me that query again".

---

Chuck Norris and Firebird once had a showdown. The result? Firebird now has a secret setting called 'ChuckMode'. When activated, all queries are automatically optimized to Chuck Norris standards – lightning-fast and unstoppable.

Chuck Norris doesn't need indexes. He just stares at the database, and the rows arrange themselves in fear. Firebird is the only one that can keep up with his gaze.

Chuck Norris doesn't do database optimization; he just whispers encouragement to Firebird, and it runs faster than ever, trying to keep up with his expectations.

When Chuck Norris uses Firebird, the performance is so incredible that physicists argue it defies the laws of relational dynamics. Chuck simply calls it 'database kung fu.'

Chuck Norris doesn't need primary keys; he just points at a record, and Firebird ensures it's the key record for life.



# FBOpt Database Optimiser

**20% OFF with code: EMBERSALE**

Till end of July 2024



Buy on-line

