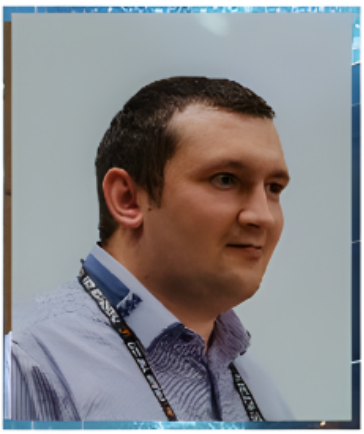
Detailed New Features Of Firebird 5



by D.Simonov

Preface and adaptation
A.Kovyazin

2024

Table of Contents

This material is sponsored and created with the sponsorship and support of IBSurgeon https://www.ib-aid.com, vendor of HQbird (advanced distribution of Firebird) and supplier of performance optimization, migration and technical support services for Firebird.

Preface: Firebird 5.0 - A Game-Changing Release in the World of Relational Databases

Hey there, fellow Firebirders! If you're scrolling through it on your favorite e-reader, I'm guessing you're as excited about Firebird 5 as I am. And let me tell you, you're in for a treat.

Now, I know what some of you might be thinking: "Another Firebird update? What's the big deal?" Well, let me assure you, Firebird 5 is not just another run-of-the-mill update. It's a game-changer, a quantum leap forward in the world of relational databases. And in this book, we're going to dive deep into all the amazing new features that make Firebird 5 stand out from the crowd.

But before we get into the nitty-gritty, let's take a step back and consider why this release is so important. In today's fast-paced digital world, data is king. Businesses of all sizes are collecting and processing more data than ever before, and they need database systems that can keep up with this ever-growing demand. That's where Firebird 5 comes in.

This new version isn't just an incremental improvement – it's a major overhaul that addresses some of the most pressing needs in modern database management. From turbocharging query performance to scaling up to handle massive datasets, Firebird 5 is designed to meet the challenges of today's data-driven world head-on.

So, what can you expect to learn about in this book? Let's take a quick tour of some of the headline features:

SQL Query Optimization: Faster Than Ever

One of the standout features of Firebird 5 is its dramatically improved SQL query optimization. I know that "query optimization" might not sound like the most exciting topic in the world, but trust me, this is where the rubber meets the road in database performance.

The team behind Firebird has completely revamped the query optimizer, implementing many new algorithms and techniques to improve performance of your queries. In this book, we'll dive deep into how the new optimizer works, exploring the magic behind the scenes that makes your queries fly. You'll learn how to take full advantage of these optimizations, and we'll look at real-world examples that showcase just how much of a difference this can make to your applications.

Scalability: Growing with Your Data

Let's face it: data growth is exploding. What seemed like a large dataset a few years ago is now considered small potatoes. That's why scalability is such a crucial feature in modern database systems, and it's an area where Firebird 5 really shines.

The new version introduces a host of features designed to help Firebird scale up to handle truly massive datasets. We're talking about improvements to indexing, better memory management, and smarter data distribution techniques. Whether you're dealing with millions of records or billions, Firebird 5 has got you covered.

In the coming chapters, we'll explore these scalability features in depth.

Parallel Execution: Harnessing the Power of Modern Hardware

Remember when computers only had one core? Yeah, me neither. These days, even your smartphone probably has multiple cores, and server-grade hardware can have dozens or even hundreds of cores. Firebird 5 is designed to take full advantage of all this computing power with its new parallel execution features to execute backup, restore, sweep, and index creation much faster.

We'll dedicate a whole chapter to parallel execution features, looking at how it works under the hood and how you can structure your queries and database schema to make the most of this powerful feature. Also, I would recommend to read the detailed article "Parallel reading of data in Firebird" and take a look on sample implementation of parallel reading in the test application FBCSVExport.

Prepared Statement Cache

Here's a feature that might not sound sexy, but trust me, it's a game-changer for many applications. Firebird 5 introduces a cache for compiled prepared statements, and it's going to make a world of difference for applications that run the same queries over and over again.

Think about it: how many times does your application run essentially the same query, just with different parameters? With previous versions, Firebird would have to compile that query from scratch each time. But with the new prepared statement cache, Firebird can keep the compiled version in memory, ready to go at a moment's notice.

The performance implications of this are huge, especially for applications that handle lots of small, frequent transactions. We'll look at how to make the most of this feature in your own applications, and explore some of the under-the-hood optimizations that make it possible.

Improved Compression of Records

In an ideal world, we'd all have infinite storage and blazing fast I/O. But in the real world, storage is often at a premium, and I/O can be a major bottleneck. That's why Firebird 5's improved compression features are such a big deal.

The new version introduces more efficient compression algorithms that can significantly reduce the size of your database on disk, without sacrificing query performance. In fact, in many cases, the reduced I/O from better compression actually improves overall performance!

SQL Query Profiling: Shining a Light on Performance Of Complex Stored Procedures

Last but certainly not least, let's talk about Firebird 5's new SQL query profiling features. As the old saying goes, you can't improve what you can't measure, and that's especially true when it comes to

database performance.

The new profiling tools in Firebird 5 give you unprecedented visibility into how your stored procedures are executing. You can see exactly where time is being spent, which parts are causing the most I/O, and where you might be missing opportunities for optimization.

We'll spend a good chunk of this book exploring these profiling tools in depth. You'll learn how to use them to identify performance bottlenecks in your queries, how to interpret the results, and how to use that information to optimize your database schema and queries for maximum performance.

Wrapping Up and More Materials

So there you have it – a whirlwind tour of some of the exciting new features in Firebird 5. But trust me, we've only scratched the surface. In the chapters that follow, we're going to dive deep into each of these features and more, exploring how they work, how to use them effectively, and how they can help you build faster, more scalable, and more efficient database applications.

Whether you're a seasoned database administrator, a developer looking to get more out of your data layer, or just someone who's curious about the cutting edge of database technology, I promise you'll find something valuable in this book.

Firebird has always been known for its combination of power, flexibility, and ease of use. With version 5, it's taking all of those qualities to the next level. By the time you finish this book, you'll have all the knowledge you need to harness the full power of Firebird 5 in your own projects.

Practical Migration Guide To Firebird 5

After discovering the exciting features of Firebird 5, you'll likely want to upgrade from your older Firebird version. While the migration process is straightforward, it involves some essential steps and potential pitfalls. To help you navigate this process, there's a free Practical Migration Guide to Firebird 5 available, offering a collection of migration solutions.

So buckle up, grab your favorite beverage, and let's dive in. The world of high-performance, scalable database management is waiting, and Firebird 5 is your ticket to ride. Let's get started!

Alexey Kovyazin, President Of Firebird Foundation

And, let's start!

In Firebird 5.0, Firebird Project had focus on performance improvements in various areas:

1. Optimizer improvements – SQLs runs faster due to better execution plans
2. Scalability in multi-user environments – Firebird can serve more concurrent connections and queries in highly concurrent environments
3. Parallel backup, restore, sweep, index creation – up to 6-10x faster (depends on hardware)
4. Cache of compiled prepared statements – up to 25% of improvement for frequent queries
5. Improved compression of records – Firebird 5.0 works faster with large VARCHARs

6. Profiling plugin – identify bottlenecks and slow code inside complex stored procedures and triggers

In addition, new features have appeared in the SQL and PSQL languages, but there are not many of them in this version.

One highly anticipated feature is the introduction of a built-in SQL and PSQL profiling tool, allowing database administrators and application developers to find bottlenecks.

Databases created in Firebird 5.0 have ODS (On-Disk Structure) version 13.1. Firebird 5.0 allows you to work with databases with ODS 13.0 (created in Firebird 4.0), but some features will not be available.

To make the transition to Firebird 5.0 easier, a new `-upgrade` switch has been added to the `gfix` command line utility, which allows you to update a minor version of ODS without lengthy backup and restore operations.

Below I will list the key improvements made in Firebird 5.0 and a brief description of them. A detailed description of all changes can be found in Firebird 5.0 Release Notes.

Chapter 1. New ODS and upgrade without backup-restore

The traditional way of updating ODS (On-Disk Structure) is to perform backup on the old version of Firebird and restore on the new one. This is a rather lengthy process, especially on large databases.

However, in the case of updating a minor version of ODS (the number after the dot) backup/restore is redundant (it is only necessary to add the missing system tables and fields, as well as some packages). An example of such an update is updating ODS 13.0 (Firebird 4.0) to ODS 13.1 (Firebird 5.0), since the major version of ODS 13 remained the same.

Starting from Firebird 5.0, it became possible to update the minor version of ODS without the lengthy backup and restore operations. For this, the `gfix` utility is used with the `-upgrade` switch.

Key points:

- The update must be performed manually using the command `gfix -upgrade`
- Exclusive access to the database is required, otherwise an error is issued.
- The system privilege `USE_GFIX_UTILITY` is required.
- The update is transactional, all changes are rolled back in case of an error.
- After the update, Firebird 4.0 can no longer open the database.

Usage:

```
gfix -upgrade <dbname> -user <username> -pass <password>
```

- This is a one-way modification, there is no way back. Therefore, before updating, make a copy of the database (using `nbackup b -0`) to have a restore point in case something goes wrong during the process.
- Updating ODS using `gfix -upgrade` does not change the data pages of user tables, so the records will not be repacked using the new RLE compression algorithm. But newly inserted records will be compressed using the improved RLE.

Chapter 2. Improving the data compression algorithm

As you know, in Firebird, table records are stored on data pages (DP) in compressed form. This is done so that as many records as possible can fit on one page, which in turn saves disk input-output. Until Firebird 5.0, the classic Run Length Encoding (RLE) algorithm was used to compress records.

The classic RLE algorithm works as follows. A sequence of repeated characters is reduced to a control byte, which determines the number of repetitions, followed by the actual repeated byte. If the data cannot be compressed, the control byte indicates that "the next n bytes should be output unchanged".

The control byte is used as follows:

- `n > 0` [1 .. 127] - next *n* байт will be stored as is;
- `n < 0` [-3 .. -128] - next byte will be repeated *n* times, but stored only once;
- `n = 0` - end of data.

Mainly, RLE is effective for compressing trailing zeros in fields of type `VARCHAR(N)`, which are not fully filled or are equal to NULL. It is fast enough and does not load the processor much unlike dictionary-based algorithms, such as LHZ, ZIP, GZ.

But the classic RLE algorithm has drawbacks:

- the maximum compression ratio is 64 times: the control byte can encode 128 repeating bytes turning them into 2 bytes. Thus, 32000 identical bytes will take up 500 bytes. This problem has worsened lately with the advent of the UTF8 encoding, where 4 bytes are allocated for each character.
- in some cases, the compressed byte sequence may become longer than the uncompressed one, if the data is not compressible.
- frequent alternation of short compressible and non-compressible sequences additionally loads the processor, thus offsetting the benefit of saving disk input-output.

Therefore, in Firebird 5.0, an improved RLE compression algorithm (with a variable-length counter) was developed. This algorithm is available only in databases with ODS 13.1 and higher.

Updating ODS using `gfix -upgrade` does not change the data pages of user tables, so the records will not be repacked using the new RLE compression algorithm. But newly inserted records will be compressed using the improved RLE.

The improved RLE algorithm works as follows. Two previously unused lengths -1 and -2 are used as special markers for longer compressible sequences:

- `{-1, two-byte counter, byte value}` - repeating sequences of length from 128 bytes to 64 KB;
- `{-2, four-byte counter, byte value}` - repeating sequences of length more than 64 KB.

Compressible sequences of length 3 bytes make no sense if they are located between two non-compressible runs. Compressible sequences of length from 4 to 8 bytes are a borderline case, as they are not very compressed, but increase the total number of runs, which negatively affects the unpacking speed. Starting from Firebird 5.0 fragments shorter than 8 bytes are not compressed.

In addition, in Firebird 5.0 (ODS 13.1) there is another improvement: if as a result of applying the RLE compression algorithm to the record, the byte sequence turned out to be longer (non-compressible data), then the record will be written to the page as is and marked with a special flag as uncompressed.

Now I will show by examples how the new RLE algorithm increases the performance of queries.

First of all, let's note that compressing records is not a free operation in terms of resources (CPU and memory). This can be easily verified by executing two queries:

```
SELECT COUNT(*) FROM BIG_TABLE;

SELECT COUNT(SOME_FIELD) FROM BIG_TABLE;
```

The first query does not use record unpacking, because we are not interested in their content (it is enough to just count the number). The second query has to unpack each record to make sure that the field `SOME_FIELD` is not NULL. First, let's see how this is done in Firebird 4.0.

```
SELECT COUNT(*)
FROM WORD_DICTIONARY;
```

```
                COUNT
=====================
              4079052

Current memory = 2610594912
Delta memory = 0
Max memory = 2610680272
Elapsed time = 0.966 sec
Buffers = 153600
Reads = 0
Writes = 0
Fetches = 4318077
```

```
SELECT COUNT(CODE_DICTIONARY)
FROM WORD_DICTIONARY;
```

```
                COUNT
=====================
              4079052

Current memory = 2610596096
```

```
Delta memory = 1184
Max memory = 2610685616
Elapsed time = 1.770 sec
Buffers = 153600
Reads = 0
Writes = 0
Fetches = 4318083
```

`1.770 - 0.966 = 0.804` - `1.770 - 0.966 = 0.804` - most of this time is just the cost of unpacking records.

Now let's look at the same thing on Firebird 5.0.

```
SELECT COUNT(*)
FROM WORD_DICTIONARY;
```

```
                COUNT
=====================
              4079052

Current memory = 2577478608
Delta memory = 176
Max memory = 2577562528
Elapsed time = 0.877 sec
Buffers = 153600
Reads = 0
Writes = 0
Fetches = 4342385
```

```
SELECT COUNT(CODE_DICTIONARY)
FROM WORD_DICTIONARY;
```

```
                COUNT
=====================
              4079052

Current memory = 2577491280
Delta memory = 12672
Max memory = 2577577520
Elapsed time = 1.267 sec
Buffers = 153600
Reads = 0
Writes = 0
Fetches = 4342393
```

`1.267 - 0.877 = 0.390` - This is twice less than in Firebird 4.0. Let's take a look at the statistics of this table in Firebird 4.0 and Firebird 5.0.

Statistics in Firebird 4.0

```
WORD_DICTIONARY (265)
    Primary pointer page: 855, Index root page: 856
    Total formats: 1, used formats: 1
    Average record length: 191.83, total records: 4079052
    Average version length: 0.00, total versions: 0, max versions: 0
    Average fragment length: 0.00, total fragments: 0, max fragments: 0
    Average unpacked length: 670.00, compression ratio: 3.49
    Pointer pages: 19, data page slots: 59752
    Data pages: 59752, average fill: 87%
    Primary pages: 59752, secondary pages: 0, swept pages: 0
    Empty pages: 1, full pages: 59750
    Fill distribution:
         0 - 19% = 1
        20 - 39% = 0
        40 - 59% = 0
        60 - 79% = 1
        80 - 99% = 59750
```

Statistics in Firebird 5.0

```
WORD_DICTIONARY (265)
    Primary pointer page: 849, Index root page: 850
    Total formats: 1, used formats: 1
    Average record length: 215.83, total records: 4079052
    Average version length: 0.00, total versions: 0, max versions: 0
    Average fragment length: 0.00, total fragments: 0, max fragments: 0
    Average unpacked length: 670.00, compression ratio: 3.10
    Pointer pages: 21, data page slots: 65832
    Data pages: 65832, average fill: 88%
    Primary pages: 65832, secondary pages: 0, swept pages: 0
    Empty pages: 4, full pages: 65824
    Fill distribution:
         0 - 19% = 5
        20 - 39% = 2
        40 - 59% = 0
        60 - 79% = 1
        80 - 99% = 65824
```

From the statistics, it can be seen that the compression ratio is even lower than in Firebird 4.0. So what accounts for such a colossal gain in performance? To understand this, we need to look at the structure of this table:

```
CREATE TABLE WORD_DICTIONARY (
    CODE_DICTIONARY      BIGINT NOT NULL,
    CODE_PART_OF_SPEECH  INTEGER NOT NULL,
    CODE_WORD_GENDER     INTEGER NOT NULL,
    CODE_WORD_AFFIXE     INTEGER NOT NULL,
    CODE_WORD_TENSE      INTEGER DEFAULT -1 NOT NULL,
    NAME                 VARCHAR(50) NOT NULL COLLATE UNICODE_CI,
    PARAMS               VARCHAR(80),
    ANIMATE              D_BOOL DEFAULT 'Нет' NOT NULL /* D_BOOL = VARCHAR(3) CHECK (VALUE
```

```
IN('Да', 'Нет')) */,
    PLURAL              D_BOOL DEFAULT 'Нет' NOT NULL /* D_BOOL = VARCHAR(3) CHECK (VALUE
IN('Да', 'Нет')) */,
    INVARIABLE          D_BOOL DEFAULT 'Нет' NOT NULL /* D_BOOL = VARCHAR(3) CHECK (VALUE
IN('Да', 'Нет')) */,
    TRANSITIVE          D_BOOL DEFAULT 'Нет' NOT NULL /* D_BOOL = VARCHAR(3) CHECK (VALUE
IN('Да', 'Нет')) */,
    IMPERATIVE          D_BOOL DEFAULT 'Нет' NOT NULL /* D_BOOL = VARCHAR(3) CHECK (VALUE
IN('Да', 'Нет')) */,
    PERFECT             D_BOOL DEFAULT 'Нет' NOT NULL /* D_BOOL = VARCHAR(3) CHECK (VALUE
IN('Да', 'Нет')) */,
    CONJUGATION         SMALLINT,
    REFLEXIVE           D_BOOL DEFAULT 'Нет' NOT NULL /* D_BOOL = VARCHAR(3) CHECK (VALUE
IN('Да', 'Нет')) */,
    PROHIBITION         D_BOOL DEFAULT 'Нет' NOT NULL /* D_BOOL = VARCHAR(3) CHECK (VALUE
IN('Да', 'Нет')) */
);
```

In this table, only the fields NAME and PARAMS can be well compressed. Since the fields of type INTEGER have the NOT NULL modifier, and the field takes up 4 bytes, such fields are not compressed in Firebird 5.0. Fields with the D_BOOL domain in UTF8 encoding can be compressed for the value 'Yes' (12 - 4 = 8 bytes) and will not be for the value 'No' (12 - 6 = 6 bytes).

Since the table has many short sequences that could be compressed in Firebird 4.0 and are not compressed in Firebird 5.0, the number of runs processed for unpacking in Firebird 5.0 is less, which gives us a performance gain.

Now I will show an example where the new RLE algorithm greatly wins in compression. For this, we will execute the following script:

```
CREATE TABLE GOOD_ZIP
(
  ID BIGINT NOT NULL,
  NAME VARCHAR(100),
  DESCRIPTION VARCHAR(1000),
  CONSTRAINT PK_GOOD_ZIP PRIMARY KEY(ID)
);

SET TERM ^;

EXECUTE BLOCK
AS
DECLARE I BIGINT = 0;
BEGIN
  WHILE (I < 100000) DO
  BEGIN
    I = I + 1;
    INSERT INTO GOOD_ZIP (
      ID,
      NAME,
      DESCRIPTION
    )
    VALUES (
```

```
      :I,
      'OBJECT_' || :I,
      'OBJECT_' || :I
    );
  END
END^


SET TERM ;^

COMMIT;
```

And now let's look at the statistics of the table GOOD_ZIP in Firebird 4.0 and Firebird 5.0.

Statistics in Firebird 4.0

```
GOOD_ZIP (128)
    Primary pointer page: 222, Index root page: 223
    Total formats: 1, used formats: 1
    Average record length: 111.09, total records: 100000
    Average version length: 0.00, total versions: 0, max versions: 0
    Average fragment length: 0.00, total fragments: 0, max fragments: 0
    Average unpacked length: 4420.00, compression ratio: 39.79
    Pointer pages: 2, data page slots: 1936
    Data pages: 1936, average fill: 81%
    Primary pages: 1936, secondary pages: 0, swept pages: 0
    Empty pages: 0, full pages: 1935
    Fill distribution:
         0 - 19% = 0
        20 - 39% = 0
        40 - 59% = 1
        60 - 79% = 5
        80 - 99% = 1930
```

Statistics in Firebird 5.0

```
GOOD_ZIP (128)
    Primary pointer page: 225, Index root page: 226
    Total formats: 1, used formats: 1
    Average record length: 53.76, total records: 100000
    Average version length: 0.00, total versions: 0, max versions: 0
    Average fragment length: 0.00, total fragments: 0, max fragments: 0
    Average unpacked length: 4420.00, compression ratio: 82.22
    Pointer pages: 1, data page slots: 1232
    Data pages: 1232, average fill: 70%
    Primary pages: 1232, secondary pages: 0, swept pages: 0
    Empty pages: 2, full pages: 1229
    Fill distribution:
         0 - 19% = 3
        20 - 39% = 0
        40 - 59% = 0
        60 - 79% = 1229
        80 - 99% = 0
```

As you can see, in this case the compression ratio in Firebird 5.0 is twice as high!

And finally, let's look at an example with non-compressible data. For this, we will execute the script:

```
CREATE TABLE NON_ZIP
(
  UID BINARY(16) NOT NULL,
  REF_UID_1 BINARY(16) NOT NULL,
  REF_UID_2 BINARY(16) NOT NULL
);

SET TERM ^;

EXECUTE BLOCK
AS
DECLARE I BIGINT = 0;
BEGIN
  WHILE (I < 100000) DO
  BEGIN
    I = I + 1;
    INSERT INTO NON_ZIP (
      UID,
      REF_UID_1,
      REF_UID_2
    )
    VALUES (
      GEN_UUID(),
      GEN_UUID(),
      GEN_UUID()
    );
  END
END^


SET TERM ;^

COMMIT;
```

Let's look at the statistics of the table NON_ZIP in v4 and v5:

Statistics in Firebird 4.0

```
NON_ZIP (129)
    Primary pointer page: 2231, Index root page: 2312
    Total formats: 1, used formats: 1
    Average record length: 53.00, total records: 100000
    Average version length: 0.00, total versions: 0, max versions: 0
    Average fragment length: 0.00, total fragments: 0, max fragments: 0
    Average unpacked length: 52.00, compression ratio: 0.98
    Pointer pages: 1, data page slots: 1240
    Data pages: 1240, average fill: 69%
    Primary pages: 1240, secondary pages: 0, swept pages: 0
    Empty pages: 5, full pages: 1234
    Fill distribution:
```

```
         0 - 19% = 5
        20 - 39% = 1
        40 - 59% = 0
        60 - 79% = 1234
        80 - 99% = 0
```

Statistics in Firebird 5.0

```
NON_ZIP (129)
    Primary pointer page: 1587, Index root page: 1588
    Total formats: 1, used formats: 1
    Average record length: 52.00, total records: 100000
    Average version length: 0.00, total versions: 0, max versions: 0
    Average fragment length: 0.00, total fragments: 0, max fragments: 0
    Average unpacked length: 52.00, compression ratio: 1.00
    Pointer pages: 1, data page slots: 1240
    Data pages: 1240, average fill: 68%
    Primary pages: 1240, secondary pages: 0, swept pages: 0
    Empty pages: 5, full pages: 1234
    Fill distribution:
         0 - 19% = 5
        20 - 39% = 1
        40 - 59% = 0
        60 - 79% = 1234
        80 - 99% = 0
```

In Firebird 4.0, as a result of compression, the record length increased, Firebird 5.0 saw that as a result of compression, the records become longer and saved the record as it is.

Chapter 3. Cache of prepared (compiled) statements

The prepared queries cache is one of the most impressive new features of version 5.0. Simply enabling the parameter in the configuration can speed up the application in some scenarios (with frequent queries) by several times.

3.1. A little theory

Any SQL query goes through two mandatory stages: preparation (compilation) and execution.

During the preparation of the query, its syntactic analysis, allocation of buffers for input and output messages, construction of the query plan and its execution tree are performed.

If the application requires multiple execution of the same query with a different set of input parameters, then prepare is usually called separately, the handle of the prepared query is saved in the application, and then execute is called for this handle. This allows to reduce the costs of re-preparing the same query or each execution.

Starting from Firebird 5.0, a cache of compiled (prepared) queries is supported for each connection. This allows to reduce the costs for re-preparing the same queries, if your application does not use explicit caching of handles of prepared queries (at the global level this is not always easy).

By default, caching is enabled, the caching threshold is determined by the parameter `MaxStatementCacheSize` in `firebird.conf`. It can be disabled by setting `MaxStatementCacheSize` to zero. The cache is maintained automatically: cached statements become invalid when necessary (usually when executing any DDL statement).

A query is considered the same if it matches exactly by character, that is, if you have semantically identical queries, but they differ by a comment, then for the cache of prepared queries these are different queries.

In addition to top-level queries, stored procedures, functions and triggers also fall into the cache of prepared queries. The contents of the compiled queries cache can be viewed using the new monitoring table `MON$COMPILED_STATEMENTS`.

Table 1. Description of the columns of the table `MON$COMPILED_STATEMENTS`

Column name	Datatype	Description
`MON$COMPILED_STATEMENT_ID`	`BIGINT`	Identified of compiled query
`MON$SQL_TEXT`	`BLOB TEXT`	The text of the statement in SQL language. Inside PSQL objects, the text of SQL statements is not displayed.
`MON$EXPLAINED_PLAN`	`BLOB TEXT`	Operator’s plan in 'explain' format.

Column name	Datatype	Description
MON$OBJECT_NAME	CHAR(63)	Name of PSQL object (trigger, stored function or stored procedure), where this SQL operator was compiled.
MON$OBJECT_TYPE	SMALLINT	Тип объекта. 2 — trigger; 5 — stored procedure; 15 — stored function.
MON$PACKAGE_NAME	CHAR(63)	Name of PSQL package
MON$STAT_ID	INTEGER	Identifier of statistics

A new column MON$COMPILED_STATEMENT_ID has appeared in the tables MON$STATEMENTS and MON$CALL_STACK, which refers to the corresponding prepared statement in MON$COMPILED_STATEMENTS.

The monitoring table MON$COMPILED_STATEMENTS allows you to easily get the plans of internal queries in a stored procedure, for example like this:

```
SELECT CS.MON$EXPLAINED_PLAN
FROM MON$COMPILED_STATEMENTS CS
WHERE CS.MON$OBJECT_NAME = 'SP_PEDIGREE'
  AND CS.MON$OBJECT_TYPE = 5
ORDER BY CS.MON$COMPILED_STATEMENT_ID DESC
FETCH FIRST ROW ONLY
```

Note that the same stored procedure can appear in MON$COMPILED_STATEMENTS multiple times. This is because currently the cache of prepared queries is made for each connection. In future versions, it is planned to make the cache of compiled queries and the metadata cache common for all connections in the Super Server architecture.

Chapter 4. Support for bidirectional cursors in the network protocol

A cursor in SQL is an object that allows you to move through the records of any result set. It can be used to process a single database record returned by a query. There are unidirectional and bidirectional (scrollable) cursors.

A unidirectional cursor does not support scrolling, that is, retrieving records from such a cursor is possible only sequentially, from the beginning to the end of the cursor. This type of cursor is available in Firebird from the earliest versions, both in PSQL (explicitly declared and implicit cursors) and through the API.

A scrollable or bidirectional cursor allows you to move through the cursor in any direction, jump around and even move to a given position. Support for bidirectional (scrollable) cursors first appeared in Firebird 3.0. They are also available in PSQL and through the API interface.

However, until Firebird 5.0, scrollable cursors were not supported at the network protocol level. This means that you could use the API of bidirectional cursors in your application, only if your connection occurs in embedded mode. Starting from Firebird 5.0 you can use the API of scrollable cursors even if you connect to the database over the network protocol, while the client library fbclient must be no lower than version 5.0.

If your application does not use fbclient, for example written in Java or .NET, then the corresponding driver must support the network protocol Firebird 5.0. For example, Jaybird 5 supports bidirectional cursors in the network protocol.

Chapter 5. Tracing the COMPILE event

In Firebird 5.0, it became possible to track a new tracing event: parsing stored modules. It allows you to track the moments of parsing stored modules, the corresponding time spent and most importantly - the plans of queries inside these PSQL modules. Tracking the plan is also possible if the PSQL module was already loaded before the start of the tracing session; in this case, the plan will be reported during the first execution noticed by the tracing session.

The following parameters have appeared in the tracing configuration to track the module parsing event:

- `log_procedure_compile` - enables tracing of procedure parsing events;
- `log_function_compile` - enables tracing of function parsing events;
- `log_trigger_compile` - enables tracing of trigger parsing events.

Suppose we have the following query:

```
SELECT * FROM SP_PEDIGREE(7435, 8, 1);
```

To track the plan of a stored procedure in a tracing session, you need to set the parameter `log_procedure_compile = true`. In this case, when preparing this query or executing it, a procedure parsing event will appear in the tracing log, which looks like this:

```
2023-10-18T20:40:51.7620 (3920:00000000073A17C0) COMPILE_PROCEDURE
    horses (ATT_30, SYSDBA:NONE, UTF8, TCPv6:::1/54464)
    C:\Firebird\5.0\isql.exe:10960

Procedure SP_PEDIGREE:
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Cursor "V" (scrollable) (line 19, column 3)
    -> Record Buffer (record length: 132)
        -> Nested Loop Join (inner)
            -> Window
                -> Window Partition
                    -> Record Buffer (record length: 82)
                        -> Sort (record length: 84, key length: 12)
                            -> Window Partition
                                -> Window Buffer
                                    -> Record Buffer (record length: 41)
                                        -> Procedure "SP_HORSE_INBRIDS" as "V H_INB
SP_HORSE_INBRIDS" Scan
            -> Filter
                -> Table "HUE" as "V HUE" Access By ID
                    -> Bitmap
                        -> Index "HUE_IDX_ORDER" Range Scan (full match)
Select Expression (line 44, column 3)
    -> Recursion
        -> Filter
            -> Table "HORSE" as "PEDIGREE HORSE" Access By ID
                -> Bitmap
```

```
                -> Index "PK_HORSE" Unique Scan
    -> Union
        -> Filter (preliminary)
            -> Filter
                -> Table "HORSE" as "PEDIGREE HORSE" Access By ID
                    -> Bitmap
                        -> Index "PK_HORSE" Unique Scan
        -> Filter (preliminary)
            -> Filter
                -> Table "HORSE" as "PEDIGREE HORSE" Access By ID
                    -> Bitmap
                        -> Index "PK_HORSE" Unique Scan
28 ms
```

Chapter 6. Per-table statistics in isql

Per-table statistics show how many records for each table were read by a full scan, how many using an index, how many inserted, updated or deleted and other counters. The values of these counters have been available for a long time through the API function isc_database_info, which was used by many graphical tools, but not by the console tool isql. The values of these same counters can be obtained by using the monitoring tables MON$RECORD_STATS and MON$TABLE_STATS, or in tracing. Starting from Firebird 5.0, this useful feature appeared in isql.

By default, per-table statistics output is disabled.

To enable it, you need to type the command:

```
SET PER_TAB ON;
```

To disable:

```
SET PER_TAB OFF;
```

The command SET PER_TAB without the words ON or OFF toggles the state of statistics output.

The full syntax of this command can be obtained using the command HELP SET.

Example of per-table statistics output:

```
SQL> SET PER_TAB ON;

SQL> SELECT COUNT(*)
CON> FROM HORSE
CON> JOIN COLOR ON COLOR.CODE_COLOR = HORSE.CODE_COLOR
CON> JOIN BREED ON BREED.CODE_BREED = HORSE.CODE_BREED;

                COUNT
=====================
               519623

Per table statistics:
--------------+---------+---------+---------+---------+---------+---------+---------+---------+
 Table name   | Natural | Index   | Insert  | Update  | Delete  | Backout | Purge   | Expunge |
--------------+---------+---------+---------+---------+---------+---------+---------+---------+
BREED         |      282|         |         |         |         |         |         |         |
COLOR         |      239|         |         |         |         |         |         |         |
HORSE         |         |   519623|         |         |         |         |         |         |
--------------+---------+---------+---------+---------+---------+---------+---------+---------+
```

Chapter 7. Parallel execution of maintenance tasks

Since version 5.0 Firebird can perform main maintenance tasks using multiple threads in parallel. Some of these tasks uses parallelism at the Firebird kernel level, others are implemented directly in the Firebird tool. Currently, only the parallel execution of sweep and index creation tasks is implemented at the kernel level. Parallel execution is supported for both automatic and manual sweeps.

In future versions it is planned to add parallelism when executing SQL queries.

7.1. Parallel execution of tasks in the Firebird kernel

To handle a task with multiple threads, the Firebird engine launches additional worker threads and creates internal worker connections. By default, parallel execution is disabled. There are two ways to enable parallel features for the connection:

- Set the number of parallel workers in DPB using the `isc_dpb_parallel_workers` tag;
- Set the default number of parallel worker processes using the `ParallelWorkers` parameter in `firebird.conf`.

Some utilities (`gfix`, `gbak`) supplied with Firebird have a command line option `-parallel` to set the number of parallel worker processes. Often this switch will simply pass the number of workers through the `isc_dpb_parallel_workers` tag when connecting to the database. The new `ParallelWorkers` parameter in `firebird.conf` sets the default number of parallel worker processes that can be used by any user connection running a parallelizable task. The default value is 1 and means no additional parallel worker processes are used. The value in DPB takes precedence over the value in `firebird.conf`.

To control the number of additional workers the engine can create, there are two new settings in `firebird.conf`:

ParallelWorkers

Sets the default number of parallel worker processes used by user connections. Can be overridden for a connection by using the `isc_dpb_parallel_workers` tag in the DPB.

MaxParallelWorkers

Limits the maximum number of concurrent worker processes for a given database and Firebird process.

Internal workers are created and managed by the Firebird engine itself. The engine maintains worker connection pools for each database. The number of threads in each pool is limited by the value of the `MaxParallelWorkers` parameter. Pools are created independently by each Firebird process. In the SuperServer architecture, worker connections are implemented as lightweight system connections, while in Classic and SuperClassic they look like regular user connections. All workers connections are built into the server creation process. Thus, in Classic architectures there are no additional server processes. Worker connections are present in the monitoring tables. Dead

working connections are destroyed after 60 seconds of inactivity. Additionally, in classic architectures, worker connections are destroyed immediately after the last user connection is disconnected from the database.

7.1.1. Practical recommendations for parameters

What are practical recommendations? Since this feature came from HQbrid (vv2.5-4) where it was used during several years on hundreds of servers, we can use its experience for Firebird 5.0. For SuperServer it is recommended to set parameter `MaxParallelWorkers` to 64, and `ParallelWorkers` to 2. For maintenance tasks (backup/restore/sweep) it is better to set individually with switch `-parallel`, with the value equal to the half of the number of physical cores of your processor (or all processors).

For Classic architecture, `MaxParallelWorkers` should be set to a number smaller than the number of physical cores of your processor it is important since the `MaxParallelWorkers` limit is set on each process.

It all depends on whether it is done under load or not. If I restore the database, then at that moment no one else is working with it. I can ask for the maximum. But if you are doing a backup/sweep/creating an index under load, then you need to moderate your appetites.

Let's look at how parallelism affects the execution time of building or rebuilding an index. The effect of parallelism on automatic sweep will not be shown, since it starts automatically without our participation. Impact of concurrency on manual sweep will be demonstrated when examining how Firebird tools perform maintenance tasks.

7.1.2. Multi-threaded index creation or rebuild

Let's compare the speed of creating an index for the `WORD_DICTIONARY` table for different ParallelWorkers values, containing 4079052 records.

For the purity of the experiment, before the new test, we restart the Firebird service. In addition, in order for the table to be in the page cache, we run the following query:

```
SELECT COUNT(*) FROM WORD_DICTIONARY;
```

The query to create an index looks like this:

```
CREATE INDEX IDX_WORD_DICTIONARY_NAME ON WORD_DICTIONARY (NAME);
```

The execution statistics for this query with `ParallelWorkers = 1` are as follows:

```
Current memory = 2577810256
Delta memory = 310720
Max memory = 3930465024
Elapsed time = 6.798 sec
```

```
Buffers = 153600
Reads = 11
Writes = 2273
Fetches = 4347093
```

Now let's delete this index, set `ParallelWorkers = 4` and `MaxParallelWorkers = 4` in the config and restart the server. Statistics for running the same query look like this:

```
Current memory = 2580355968
Delta memory = 2856432
Max memory = 4157427072
Elapsed time = 3.175 sec
Buffers = 153600
Reads = 11
Writes = 2277
Fetches = 4142838
```

As you can see, the index creation time has decreased by a little over 2 times.

The same thing happens when rebuilding the index with the query:

```
ALTER INDEX IDX_WORD_DICTIONARY_NAME ACTIVE;
```

7.2. Parallel execution of maintenance tasks by Firebird tools

Main utilities (`gfix`, `gbak`) supplied with Firebird also support parallel task execution. They use the number of parallel worker processes set in the `ParallelWorkers` parameter in `firebird.conf`. The number of parallel worker processes can be overridden using the `-parallel` command line switch.

It is recommended to always set the number of parallel processes explicitly using the `-parallel` or `-par` switch.

The following tasks can use parallel execution:

- Creating a backup using the `gbak` utility
- Restoring from a backup using the `gbak` utility
- Manual sweep using the `gfix` utility
- Updating icu using the `gfix` utility

7.2.1. Parallelism when performing backups using the `gbak`

Let's see how parallel execution will affect backup's speed of `gbak` tool.

We will use the fastest backup option - through the service manager and with garbage collection disabled. In order to be able to track the time of each operation during the backup, we will add the

`-stat td` switch.

First, let's run the backup without parallelism:

```
gbak -b -g -par 1 "c:\fbdata\db.fdb" "d:\fbdata\db.fbk" -se localhost/3055:service_mgr -user
SYSDBA
  -pas masterkey -stat td -v -Y "d:\fbdata\5.0\backup.log"
```

The backup completed in 35.810 seconds.

Now let's try to run a backup using 4 threads (on the computer which has 8 cores).

```
gbak -b -g -par 4 "c:\fbdata\db.fdb" "d:\fbdata\db.fbk" -se localhost/3055:service_mgr -user
SYSDBA
  -pas masterkey -stat td -v -Y "d:\fbdata\5.0\backup-4.log"
```

The backup completed in 18.267 seconds!

As you can see, as the number of parallel processors increases, the backup speed increases, although not linearly.

In fact, the effect of parallel threads on backup speed depends on your hardware. The optimal number of parallel threads should be selected experimentally.

Any additional switches can also change the picture. For example, the `-ZIP` switch compresses the backup copy may reduce parallelism to almost nothing, or may still speed up copying. It depends on the speed of the disk drive, whether the copy is made to the same disk where the database is located and other factors. Therefore, it is necessary to conduct experiments on your hardware to find the ideal value.

7.2.2. Parallelism when performing restore using the `gbak`

Now let's look at how parallelism affects the speed of restoring from a backup. Restoring from a backup consists of the following steps:

- creating a database with the corresponding ODS;
- restoring metadata from a backup copy;
- inserting data to user tables;
- build indices.

Parallelism will only be involved in the last two stages.

In order to be able to track the time of each operation during restoration from a backup, we will add the `-stat td` switch.

First, let's start restoring from a backup without parallelism:

```
gbak -c -par 1 "d:\fbdata\db.fbk" "c:\fbdata\db.fdb" -se localhost/3055:service_mgr -user
SYSDBA
  -pas masterkey -v -stat td -Y "d:\fbdata\restore.log"
```

Restore from backup completed in 201.590 seconds. Of these, 70.73 seconds were spent on restoring table data and 121.142 seconds on building indexes.

Now let's try to start restoring from a backup using 4 threads.

```
gbak -c -par 4 "d:\fbdata\db.fbk" "c:\fbdata\db.fdb" -se localhost/3055:service_mgr -user
SYSDBA
  -pas masterkey -v -stat td -Y "d:\fbdata\restore-4.log"
```

Restore from backup completed in 116.718 seconds. Of these, 26.748 seconds were spent on restoring table data and 86.075 seconds on building indexes.

With the help of 4 parallel workers, we were able to almost double the recovery speed. At the same time, the speed of data recovery has increased by almost 3 times, and the construction of indexes has accelerated by 1.5 times.

Why? The explanation is simple: parallelism is used only when engine builds large indexes. Many tables in the database taken as an example are small, the indexes on them are small too, and the number of such tables is large. Therefore, the numbers in your database may be different.

Note that the MaxParallelWorkers parameter limits the use of parallel threads to the Firebird kernel only. When restoring a database using the gbak utility, you can observe the following picture: data in tables is restored quickly (parallelism is noticeable), and building indexes is slower. The point is that indexes are always built by the Firebird kernel. And if MaxParallelWorkers has a value less than that specified in -parallel, then only MaxParallelWorkers of threads will be used to build indexes. However, gbak inserts data to the tables, using -parallel worker threads.

7.2.3. Parallel manual sweep using the `gfix` tool

Sweep (cleaning) is the important maintenance process: Firebird scans specified database, and if there are "garbage" records versions, remove them from data pages and from indices. By default, Firebird database is created with autosweep setting, but for the medium and large databases (30+Gb) with high number of transactions per second it could be necessary to disable automatic sweep use manual (usually, scheduled) sweep instead.

Before Firebird 3.0 sweep always scanned all data pages. However, starting with Firebird 3.0 (ODS 12.0), data pages (DP) and pointer pages to data pages (PP) have a special swept flag that is set to 1 if sweep has already scanned the data page and cleared garbage from it. When records in this table are modified for the first time, the flag is reset to 0 again. Starting with Firebird 3.0, automatic and manual sweep skips pages that have the swept flag set to 1. Therefore, a repeated sweep will go

> much faster, unless, of course, since the previous sweep you have not managed to change records on all pages of the database data. New data pages are always created with swept flag = 0. When restoring the database and backup, all DP and PP pages will be with swept flag = 0.

How to test correctly? An idle sweep after restoring from the database did not show any difference in single-threaded and multi-threaded mode. Therefore, I first ran a sweep on the restored database so that the next sweep would not check uncluttered pages, and then I made a request like this:

```
update bigtable set field=field;
rollback;
exit;
```

The purpose of this request was to create 'garbage' in the database. Now you can run the sweep to test its execution speed.

First, let's run sweep without parallelism:

```
gfix -user SYSDBA -password masterkey -sweep -par 1 inet://localhost:3055/mydb
```

```
DESKTOP-E3INAFT Sun Oct 22 16:24:21 2023
    Sweep is started by SYSDBA
    Database "mydb"
    OIT 694, OAT 695, OST 695, Next 696


DESKTOP-E3INAFT Sun Oct 22 16:24:42 2023
    Sweep is finished
    Database "mydb"
    1 workers, time 20.642 sec
    OIT 696, OAT 695, OST 695, Next 697
```

Now we will update the large table and rollback again, and run a sweep with 4 parallel workers.

```
gfix -user SYSDBA -password masterkey -sweep -par 4 inet://localhost:3055/mydb
```

```
DESKTOP-E3INAFT Sun Oct 22 16:26:56 2023
    Sweep is started by SYSDBA
    Database "mydb"
    OIT 697, OAT 698, OST 698, Next 699


DESKTOP-E3INAFT Sun Oct 22 16:27:06 2023
    Sweep is finished
    Database "mydb"
    4 workers, time 9.406 sec
    OIT 699, OAT 702, OST 701, Next 703
```

As you can see, the speed of sweep execution has increased more than 2 times.

7.2.4. Parallel icu update using the `gfix` utility

The `-icu` switch allows you to rebuild the indexes in the database using the new ICU.

The ICU library is used by Firebird to support COLLATION for multibyte encodings like UTF8. On Windows, ICU is always bundled with Firebird. In Linux, ICU is usually a system library and depends on Linux version. When moving a database file from one Linux distribution to another, the ICU installed on the system may have a different version. This may result in a database on an OS running a different version of ICU being binary incompatible for indexes character data types.

Since rebuilding indexes can be done using parallelism, this is also supported for `gfix -icu`.

Chapter 8. Improvements in Optimizer

The optimizer is a part of Firebird database engine which is responsible for decision: how to execute the SQL on the specific database in the fastest way. In v5.0 Firebird optimizer has received the biggest amount of changes since version 2.0. It is the most practical part which directly improves the performance of SQLs for databases of any size, from 1Gb to 1Tb. Let's see in details what was improved, with examples and performance comparisons.

8.1. Cost estimation of HASH vs NESTED LOOP JOIN

HASH JOIN appeared in Firebird 3.0.

The simplified idea of HASH JOIN is to cache the small dataset (e.g., a small table) into the memory, calculate hashes of keys, and use the hash of the key to join with larger dataset (e.g., a large table). If there will be the same hash for the several records, they will be processed one by one, to find the exactly the same key. HASH JOIN works only with strict key equality (i.e., =), and allows expressions with keys.

Until version 5.0, Firebird optimizer uses HASH JOIN only in a case of absence of indices for the join condition. If there is index, optimizer of pre-v5 version will use NESTED LOOP JOIN (usually it is INNER JOIN) with index. However, it is not always the fastest way: for example, if very large table is joined with small table using the primary key using INNER JOIN, each record of small table will be read multiple times (data pages and appropriate index pages). With HASH JOIN, the small table will be read once, cached, and calculated hashes will be used to join with very large table.

The obvious question is when to use HASH JOIN and when NESTED LOOP JOIN (INNER JOIN in majority cases)? It is relatively easy to decide to use HASH when small table like `CLIENT_COUNTRY` is joining with large table like INVOICES, but not all situations are clear. Until Firebird 5, the decision was the sole responsibility of the developer, since it required change of the text of SQL query, now it is done by Firebird optimizer using cost estimation algorithm.

To better understand the effect of HASH JOIN, let's consider how the same query will be executed in Firebird 4.0 and in Firebird 5.0. To analyze the difference, we have EXPLAIN PLAN, query execution statistics and extended statistics from the `isql` with set per-tab option (new in 5.0).

In the example below, we the large table named `HORSE`, where we have list of horses, and small tables with a few records: `SEX`, `COLOR`, `BREED`, `FARM`. The query selects all records from the large table.

We use `COUNT(*)` to read all records, and to exclude time to transfer records from the server to the client.

```
SELECT
  COUNT(*)
FROM
  HORSE
  JOIN SEX ON SEX.CODE_SEX = HORSE.CODE_SEX
  JOIN COLOR ON COLOR.CODE_COLOR = HORSE.CODE_COLOR
  JOIN BREED ON BREED.CODE_BREED = HORSE.CODE_BREED
```

```
  JOIN FARM ON FARM.CODE_FARM = HORSE.CODE_FARM
```

In Firebird 4.0:

```
Select Expression
    -> Aggregate
        -> Nested Loop Join (inner)
            -> Table "COLOR" Full Scan
            -> Filter
                -> Table "HORSE" Access By ID
                    -> Bitmap
                        -> Index "FK_HORSE_COLOR" Range Scan (full match)
            -> Filter
                -> Table "SEX" Access By ID
                    -> Bitmap
                        -> Index "PK_SEX" Unique Scan
            -> Filter
                -> Table "BREED" Access By ID
                    -> Bitmap
                        -> Index "PK_BREED" Unique Scan
            -> Filter
                -> Table "FARM" Access By ID
                    -> Bitmap
                        -> Index "PK_FARM" Unique Scan

                COUNT
=====================
               519623

Current memory = 2614108752
Delta memory = 438016
Max memory = 2614392048
Elapsed time = 2.642 sec
Buffers = 153600
Reads = 0
Writes = 0
Fetches = 5857109
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  |
--------------------------------+---------+---------+---------+---------+---------+
BREED                           |         |   519623|         |         |         |
COLOR                           |      239|         |         |         |         |
FARM                            |         |   519623|         |         |         |
HORSE                           |         |   519623|         |         |         |
SEX                             |         |   519623|         |         |         |
--------------------------------+---------+---------+---------+---------+---------+
```

And in Firebird 5.0. The optimizer of v5 uses cardinality of table to decide when to use HASH JOIN.

```
Select Expression
    -> Aggregate
        -> Filter
```

```
            -> Hash Join (inner)
                -> Hash Join (inner)
                    -> Hash Join (inner)
                        -> Nested Loop Join (inner)
                            -> Table "COLOR" Full Scan
                            -> Filter
                                -> Table "HORSE" Access By ID
                                    -> Bitmap
                                        -> Index "FK_HORSE_COLOR" Range Scan (full match)
                        -> Record Buffer (record length: 25)
                            -> Table "SEX" Full Scan
                    -> Record Buffer (record length: 25)
                        -> Table "BREED" Full Scan
                -> Record Buffer (record length: 33)
                    -> Table "FARM" Full Scan

                COUNT
=====================
               519623

Current memory = 2579749376
Delta memory = 352
Max memory = 2582802608
Elapsed time = 0.702 sec
Buffers = 153600
Reads = 0
Writes = 0
Fetches = 645256
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  |
--------------------------------+---------+---------+---------+---------+---------+
BREED                           |      282|         |         |         |         |
COLOR                           |      239|         |         |         |         |
FARM                            |    36805|         |         |         |         |
HORSE                           |         |   519623|         |         |         |
SEX                             |        4|         |         |         |         |
--------------------------------+---------+---------+---------+---------+---------+
```

As you can see, HASH JOIN in this situation is 3.5 times faster!

8.2. Cost estimation of HASH vs MERGE JOIN

In Firebird 3.0 the algorithm MERGE JOIN was temporary disabled in favor of HASH JOIN. Usually it was used when NESTED LOOP JOIN was non-optimal (when there were no indices for join condition, or when joining datasets were not related).

And, in the majority of situations, HASH JOIN is more effective than MERGE JOIN, due to the fact that it does not require to sort joining datasets with keys before the merge, however, there are a few cases when MERGE JOIN is better than HASH JOIN:

- Merged datasets are already sorted with the join key, for example, merge of 2 subselects with keys specified in GROUP BY:

```
select count(*)
from
(
    select code_father+0 as code_father, count(*) as cnt
    from horse group by 1
) h
join (
    select code_father+0 as code_father, count(*) as cnt
    from cover group by 1
) c on h.code_father = c.code_father
```

In this example, merged datasets are already sorted by the key code_father, and we don't need to sort them again, in this case MERGE JOIN will be the most effective.

Unfortunately, Firebird 5.0 cannot recognize this situation, hopefully it will appear in the next version.

- Merged datasets are very large. In this case hash-table will become very large and will not fit into memory. The optimizer of Firebird v5 checks cardinalities of merged datasets (tables, for example), and if the smallest is more than 1009000 records, v5 will choose MERGE JOIN instead of HASH JOIN. In the explain plan we will see it in the following manner:

```
SELECT
  *
FROM
  BIG_1
  JOIN BIG_2 ON BIG_2.F_2 = BIG_1.F_1
```

```
Select Expression
    -> Filter
        -> Merge Join (inner)
            -> Sort (record length: 44, key length: 12)
                -> Table "BIG_2" Full Scan
            -> Sort (record length: 44, key length: 12)
                -> Table "BIG_1" Full Scan
```

8.3. Transforming OUTER JOIN into INNER JOIN

In current versions of Firebird (including v5), OUTER JOIN (LEFT JOIN, as the most often example), can be executed only with algorithm NESTED LOOP JOIN, with index for the key of joining table (if possible). The biggest restriction of the LEFT JOIN is the strict order of joining, so optimizer cannot change it to keep the result set exactly the same as it was designed by the developer.

However, if there is non-NULL condition in the "right" (joining) table, the OUTER JOIN works as INNER, with the exception of join order – it is still locked. However, it is true only for previous versions: in v5 Firebird will transform the OUTER join to the INNER, and optimize it.

For example, we have the following query, where we join the large table `HORSES` with small table

FARM, using LEFT JOIN. As you can see, the query has condition on the “right” table FARM, which effectively excludes NULLified records which could produce LEFT JOIN, it means that it is implicit INNER JOIN, but with forced order of join, which prevents optimizer of pre-v5 to filter records on FARM first.

```
SELECT
  COUNT(*)
FROM
  HORSE
  LEFT JOIN FARM ON FARM.CODE_FARM = HORSE.CODE_FARM
WHERE FARM.CODE_COUNTRY = 1
```

The result in Firebird 4.0:

```
Select Expression
    -> Aggregate
        -> Filter
            -> Nested Loop Join (outer)
                -> Table "HORSE" Full Scan
                -> Filter
                    -> Table "FARM" Access By ID
                        -> Bitmap
                            -> Index "PK_FARM" Unique Scan

                COUNT
=====================
               345525

Current memory = 2612613792
Delta memory = 0
Max memory = 2614392048
Elapsed time = 1.524 sec
Buffers = 153600
Reads = 0
Writes = 0
Fetches = 2671475
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  |
--------------------------------+---------+---------+---------+---------+---------+
FARM                            |         |   519623|         |         |         |
HORSE                           |   519623|         |         |         |         |
--------------------------------+---------+---------+---------+---------+---------+
```

```
Select Expression
    -> Aggregate
        -> Nested Loop Join (inner)
            -> Filter
                -> Table "FARM" Access By ID
                    -> Bitmap
                        -> Index "FK_FARM_COUNTRY" Range Scan (full match)
            -> Filter
```

```
                -> Table "HORSE" Access By ID
                    -> Bitmap
                        -> Index "FK_HORSE_FARMBORN" Range Scan (full match)

                COUNT
=====================
               345525

Current memory = 2580089760
Delta memory = 240
Max memory = 2582802608
Elapsed time = 0.294 sec
Buffers = 153600
Reads = 0
Writes = 0
Fetches = 563801
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  |
--------------------------------+---------+---------+---------+---------+---------+
FARM                            |         |    32787|         |         |         |
HORSE                           |         |   345525|         |         |         |
--------------------------------+---------+---------+---------+---------+---------+
```

As you can see, Firebird v4 follows the specified order of joining – first, it reads the whole table `HORSE` without an index (there is not condition on `HORSE`), then join `FARM` using the join condition index.

In Firebird 5.0 the plan is different:

```
SELECT
  COUNT(*)
FROM
  HORSE
  LEFT JOIN FARM ON FARM.CODE_FARM = HORSE.CODE_FARM
WHERE FARM.CODE_FARM IS NOT NULL
```

```
Select Expression
    -> Aggregate
        -> Filter
            -> Nested Loop Join (outer)
                -> Table "HORSE" Full Scan
                -> Filter
                    -> Table "FARM" Access By ID
                        -> Bitmap
                            -> Index "PK_FARM" Unique Scan

                COUNT
=====================
               519623

Current memory = 2580315664
Delta memory = 240
```

```
Max memory = 2582802608
Elapsed time = 1.151 sec
Buffers = 153600
Reads = 0
Writes = 0
Fetches = 2676533
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  |
--------------------------------+---------+---------+---------+---------+---------+
FARM                            |         |  519623|         |         |         |
HORSE                           |  519623|         |         |         |         |
--------------------------------+---------+---------+---------+---------+---------+
```

As you can see, Firebird 5.0 recognized that LEFT JOIN can be transformed to INNER due to the condition on the right side, and applied the index for `FARM`.

As a result, the same query in Firebird 5.0 is executed 4x times faster.

This transformation is very important for dynamically built queries, with custom conditions (various reports, or ORM-generated queries).

8.4. Converting subqueries to ANY/SOME/IN/EXISTS in semi-join

A semi-join is an operation that joins two relations, returning rows from only one of the relations without performing the entire join. Unlike other join operators, there is no explicit syntax for specifying whether to perform a semi-join. However, you can perform a semi-join using subqueries in ANY/SOME/IN/EXISTS.

Traditionally, Firebird transforms subqueries in ANY/SOME/IN predicates into correlated subqueries in the EXISTS predicate, and executes the subquery in EXISTS for each record of the outer query. When executing a subquery inside an EXISTS predicate, the FIRST ROWS strategy is used, and its execution stops immediately after the first record is returned.

Starting with **Firebird 5.0.1**, subqueries in ANY/SOME/IN/EXISTS predicates can be converted to semi-joins. This feature is disabled by default, and can be enabled by setting the `SubQueryConversion` configuration parameter to `true` in the `firebird.conf` or `database.conf` file.

This feature is experimental, so it is disabled by default. You can enable it and test your queries with subqueries in ANY/SOME/IN/EXISTS predicates, and if the performance is better, leave it enabled, otherwise set the `SubQueryConversion` parameter back to the default (`false`).

The default value for the `SubQueryConversion` configuration parameter may be changed in the future, or the parameter may be removed altogether. This will happen once the new way of doing things is proven to be more optimal in most cases.

Unlike performing ANY/SOME/IN/EXISTS on subqueries directly, i.e. as correlated subqueries, performing them as semi-joins gives more room for optimization. Semi-joins can be performed by various `Hash Join (semi)` or `Nested Loop Join (semi)` algorithms, while correlated subqueries are always performed for each record of the outer query.

Let's try to enable this feature by setting the `SubQueryConversion` parameter to `true` in the `firebird.conf` file. Now let's do some experiments.

Let's execute the following query:

```
SELECT
  COUNT(*)
FROM
  HORSE H
WHERE H.CODE_DEPARTURE = 1
  AND H.CODE_SEX = 2
  AND H.CODE_HORSE IN (
    SELECT COVER.CODE_FATHER
    FROM COVER
    WHERE COVER.CODE_DEPARTURE = 1
      AND EXTRACT(YEAR FROM COVER.BYDATE) = 2023
  )
```

```
Select Expression
    -> Aggregate
        -> Filter
            -> Hash Join (semi)
                -> Filter
                    -> Table "HORSE" as "H" Access By ID
                        -> Bitmap And
                            -> Bitmap
                                -> Index "FK_HORSE_DEPARTURE" Range Scan (full match)
                            -> Bitmap
                                -> Index "FK_HORSE_SEX" Range Scan (full match)
                -> Record Buffer (record length: 41)
                    -> Filter
                        -> Table "COVER" Access By ID
                            -> Bitmap And
                                -> Bitmap
                                    -> Index "IDX_COVER_BYYEAR" Range Scan (full match)
                                -> Bitmap
                                    -> Index "FK_COVER_DEPARTURE" Range Scan (full match)

                COUNT
=====================
                  297

Current memory = 552356752
Delta memory = 352
Max memory = 552567920
Elapsed time = 0.045 sec
Buffers = 32768
Reads = 0
```

```
Writes = 0
Fetches = 43984
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  |
--------------------------------+---------+---------+---------+---------+---------+
COVER                           |         |     1516|         |         |         |
HORSE                           |         |    37069|         |         |         |
--------------------------------+---------+---------+---------+---------+---------+
```

In the execution plan we see a new join method `Hash Join (semi)`. The result of the subquery in `IN` was buffered, which is visible in the plan as `Record Buffer (record length: 41)`. That is, in this case the subquery in IN was executed once, its result was saved in the hash table memory, and then the outer query simply searched in this hash table.

For comparison, let's run the same query with subquery-to-semijoin conversion disabled.

```
Sub-query
    -> Filter
        -> Filter
            -> Table "COVER" Access By ID
                -> Bitmap And
                    -> Bitmap
                        -> Index "FK_COVER_FATHER" Range Scan (full match)
                    -> Bitmap
                        -> Index "IDX_COVER_BYYEAR" Range Scan (full match)
Select Expression
    -> Aggregate
        -> Filter
            -> Table "HORSE" as "H" Access By ID
                -> Bitmap And
                    -> Bitmap
                        -> Index "FK_HORSE_DEPARTURE" Range Scan (full match)
                    -> Bitmap
                        -> Index "FK_HORSE_SEX" Range Scan (full match)

                COUNT
=====================
                  297

Current memory = 552046496
Delta memory = 352
Max memory = 552135600
Elapsed time = 0.395 sec
Buffers = 32768
Reads = 0
Writes = 0
Fetches = 186891
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  |
--------------------------------+---------+---------+---------+---------+---------+
COVER                           |         |      297|         |         |         |
HORSE                           |         |    37069|         |         |         |
```

```
------------------------------+---------+---------+---------+---------+---------+
```

The execution plan shows that the subquery is executed for each record of the main query, but uses an additional index FK_COVER_FATHER. This is also visible in the execution statistics: the number of Fetches is 4 times greater, the execution time is almost 4 times worse.

The reader may ask: why does hash semi-join show 5 times more index reads of the COVER table, but otherwise it is better? The fact is that index reads in statistics show the number of records read using the index, they do not show the total number of index accesses, some of which do not result in retrieving records at all, but these accesses are not free.

What happened? To better understand the transformation of subqueries, let's introduce an imaginary semi-join operator "SEMI JOIN". As I already said, this type of join is not represented in the SQL language. Our query with the IN operator was transformed into an equivalent form, which can be written as follows:

```
SELECT
  COUNT(*)
FROM
  HORSE H
  SEMI JOIN (
    SELECT COVER.CODE_FATHER
    FROM COVER
    WHERE COVER.CODE_DEPARTURE = 1
      AND EXTRACT(YEAR FROM COVER.BYDATE) = 2023
  ) TMP ON TMP.CODE_FATHER = H.CODE_HORSE
WHERE H.CODE_DEPARTURE = 1
  AND H.CODE_SEX = 2
```

Now it's clearer. The same thing happens for subqueries using EXISTS. Let's look at another example:

```
SELECT
  COUNT(*)
FROM
  HORSE H
WHERE H.CODE_DEPARTURE = 1
  AND EXISTS (
    SELECT *
    FROM COVER
    WHERE COVER.CODE_DEPARTURE = 1
      AND COVER.CODE_FATHER = H.CODE_FATHER
      AND COVER.CODE_MOTHER = H.CODE_MOTHER
  )
```

Currently, it is not possible to write such an EXISTS using IN. Let's see how it is implemented without transforming it into a semi-join.

```
Sub-query
    -> Filter
        -> Table "COVER" Access By ID
            -> Bitmap And
                -> Bitmap
                    -> Index "FK_COVER_MOTHER" Range Scan (full match)
                -> Bitmap
                    -> Index "FK_COVER_FATHER" Range Scan (full match)
Select Expression
    -> Aggregate
        -> Filter
            -> Table "HORSE" as "H" Access By ID
                -> Bitmap
                    -> Index "FK_HORSE_DEPARTURE" Range Scan (full match)

                COUNT
=====================
                91908

Current memory = 552240400
Delta memory = 352
Max memory = 554680016
Elapsed time = 19.083 sec
Buffers = 32768
Reads = 0
Writes = 0
Fetches = 935679
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  |
--------------------------------+---------+---------+---------+---------+---------+
COVER                           |         |    91908|         |         |         |
HORSE                           |         |    96021|         |         |         |
--------------------------------+---------+---------+---------+---------+---------+
```

Very slow. Now let's set `SubQueryConversion = true` and run the query again.

```
Select Expression
    -> Aggregate
        -> Filter
            -> Hash Join (semi)
                -> Filter
                    -> Table "HORSE" as "H" Access By ID
                        -> Bitmap
                            -> Index "FK_HORSE_DEPARTURE" Range Scan (full match)
                -> Record Buffer (record length: 49)
                    -> Filter
                        -> Table "COVER" Access By ID
                            -> Bitmap
                                -> Index "FK_COVER_DEPARTURE" Range Scan (full match)

                COUNT
=====================
                91908
```

```
Current memory = 552102000
Delta memory = 352
Max memory = 561520736
Elapsed time = 0.208 sec
Buffers = 32768
Reads = 0
Writes = 0
Fetches = 248009
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  |
--------------------------------+---------+---------+---------+---------+---------+
COVER                           |         |   140254|         |         |         |
HORSE                           |         |    96021|         |         |         |
--------------------------------+---------+---------+---------+---------+---------+
```

The query was executed 100 times faster! If we rewrite it using our fictitious SEMI JOIN operator, the query will look like this:

```
SELECT
  COUNT(*)
FROM
  HORSE H
  SEMI JOIN (
    SELECT
      COVER.CODE_FATHER,
      COVER.CODE_MOTHER
    FROM COVER
  ) TMP ON TMP.CODE_FATHER = H.CODE_FATHER AND TMP.CODE_MOTHER = H.CODE_MOTHER
WHERE H.CODE_DEPARTURE = 1
```

Can any correlated subquery in IN/EXISTS be converted to a semi-join? No, not any, for example, if the subquery contains FETCH/FIRST/SKIP/ROWS filters, then the subquery cannot be converted to a semi-join and it will be executed as a correlated subquery. Here is an example of such a query:

```
SELECT
  COUNT(*)
FROM
  HORSE H
WHERE H.CODE_DEPARTURE = 1
  AND EXISTS (
    SELECT *
    FROM COVER
    WHERE COVER.CODE_FATHER = H.CODE_HORSE
    OFFSET 0 ROWS
  )
```

Here the phrase `OFFSET 0 ROWS` does not change the semantics of the query, and the result of its execution will be the same as without it. Let's look at the plan and statistics of this query.

```
Sub-query
    -> Skip N Records
        -> Filter
            -> Table "COVER" Access By ID
                -> Bitmap
                    -> Index "FK_COVER_FATHER" Range Scan (full match)
Select Expression
    -> Aggregate
        -> Filter
            -> Table "HORSE" as "H" Access By ID
                -> Bitmap
                    -> Index "FK_HORSE_DEPARTURE" Range Scan (full match)

                COUNT
=====================
                10971

Current memory = 551912944
Delta memory = 288
Max memory = 552002112
Elapsed time = 0.201 sec
Buffers = 32768
Reads = 0
Writes = 0
Fetches = 408988
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  |
--------------------------------+---------+---------+---------+---------+---------+
COVER                           |         |    10971|         |         |         |
HORSE                           |         |    96021|         |         |         |
--------------------------------+---------+---------+---------+---------+---------+
```

As you can see, the transformation to a semi-join did not occur. Now let's remove `OFFSET 0 ROWS` and take statistics again.

```
Select Expression
    -> Aggregate
        -> Filter
            -> Hash Join (semi)
                -> Filter
                    -> Table "HORSE" as "H" Access By ID
                        -> Bitmap
                            -> Index "FK_HORSE_DEPARTURE" Range Scan (full match)
                -> Record Buffer (record length: 33)
                    -> Table "COVER" Full Scan

                COUNT
=====================
                10971

Current memory = 552112128
Delta memory = 288
Max memory = 585044592
```

```
Elapsed time = 0.405 sec
Buffers = 32768
Reads = 0
Writes = 0
Fetches = 854841
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  |
--------------------------------+---------+---------+---------+---------+---------+
COVER                           |   722465|         |         |         |         |
HORSE                           |         |    96021|         |         |         |
--------------------------------+---------+---------+---------+---------+---------+
```

Here the conversion to semi-join has happened, and as we can see the execution time has become worse. The reason is that currently the optimizer does not have a cost estimate between the `Hash Join (semi)` and `Nested Loop Join (semi)` join algorithms using an index, so the rule is: if the join condition contains **only** equality, then the `Hash Join (semi)` algorithm is chosen, otherwise the IN/EXISTS subqueries are executed as usual.

Now let's disable the semi-join conversion and look at the execution statistics.

```
Sub-query
    -> Filter
        -> Table "COVER" Access By ID
            -> Bitmap
                -> Index "FK_COVER_FATHER" Range Scan (full match)
Select Expression
    -> Aggregate
        -> Filter
            -> Table "HORSE" as "H" Access By ID
                -> Bitmap
                    -> Index "FK_HORSE_DEPARTURE" Range Scan (full match)

                COUNT
=====================
                10971

Current memory = 551912752
Delta memory = 288
Max memory = 552001920
Elapsed time = 0.193 sec
Buffers = 32768
Reads = 0
Writes = 0
Fetches = 408988
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  |
--------------------------------+---------+---------+---------+---------+---------+
COVER                           |         |    10971|         |         |         |
HORSE                           |         |    96021|         |         |         |
--------------------------------+---------+---------+---------+---------+---------+
```

As you can see, Fetches is exactly equal to the case when the subquery contained the clause `OFFSET 0 ROWS`, and the execution time differs within the margin of error. This means that you can use the clause `OFFSET 0 ROWS` as a hint to disable the semi-join conversion.

Now let's look at cases where any correlated condition other than equality and `IS NOT DISTINCT FROM` is used in subqueries.

```
SELECT
  COUNT(*)
FROM
  HORSE H
WHERE H.CODE_DEPARTURE = 1
  AND EXISTS (
    SELECT *
    FROM COVER
    WHERE COVER.BYDATE > H.BIRTHDAY
  )
```

```
Sub-query
    -> Filter
        -> Table "COVER" Access By ID
            -> Bitmap
                -> Index "COVER_IDX_BYDATE" Range Scan (lower bound: 1/1)
Select Expression
    -> Aggregate
        -> Filter
            -> Table "HORSE" as "H" Access By ID
                -> Bitmap
                    -> Index "FK_HORSE_DEPARTURE" Range Scan (full match)
```

As I said above, no transformation to a semi-join occurred, the subquery is executed for each record of the main query.

Let's continue the experiments, write a query using equality and one more predicate except equality.

```
SELECT
  COUNT(*)
FROM
  HORSE H
WHERE H.CODE_DEPARTURE = 1
  AND EXISTS (
    SELECT *
    FROM COVER
    WHERE COVER.CODE_FATHER = H.CODE_FATHER
      AND COVER.BYDATE > H.BIRTHDAY
  )
```

```
Select Expression
    -> Aggregate
```

```
        -> Nested Loop Join (semi)
            -> Filter
                -> Table "HORSE" as "H" Access By ID
                    -> Bitmap
                        -> Index "FK_HORSE_DEPARTURE" Range Scan (full match)
            -> Filter
                -> Filter
                    -> Table "COVER" Access By ID
                        -> Bitmap
                            -> Index "COVER_IDX_BYDATE" Range Scan (lower bound: 1/1)
```

Here in the plan we see the first use of the `Nested Loop Join (semi)` join method, but unfortunately this plan is bad, because the `FK_COVER_FATHER` index is not used. You will not get any results from such a query. This can be fixed using the `OFFSET 0 ROWS` hint.

```
SELECT
  COUNT(*)
FROM
  HORSE H
WHERE H.CODE_DEPARTURE = 1
  AND EXISTS (
    SELECT *
    FROM COVER
    WHERE COVER.CODE_FATHER = H.CODE_FATHER
      AND COVER.BYDATE > H.BIRTHDAY
    OFFSET 0 ROWS
  )
```

```
Sub-query
    -> Skip N Records
        -> Filter
            -> Table "COVER" Access By ID
                -> Bitmap
                    -> Index "FK_COVER_FATHER" Range Scan (full match)
Select Expression
    -> Aggregate
        -> Filter
            -> Table "HORSE" as "H" Access By ID
                -> Bitmap
                    -> Index "FK_HORSE_DEPARTURE" Range Scan (full match)

                COUNT
=====================
                72199

Current memory = 554017824
Delta memory = 320
Max memory = 554284480
Elapsed time = 45.548 sec
Buffers = 32768
Reads = 0
Writes = 0
Fetches = 84145713
```

```
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  |
--------------------------------+---------+---------+---------+---------+---------+
COVER                           |         | 75894621|         |         |         |
HORSE                           |         |    96021|         |         |         |
--------------------------------+---------+---------+---------+---------+---------+
```

Not the best execution time, but in this case we at least got the result.

Thus, converting subqueries to ANY/SOME/IN/EXISTS into semi-join allows in some cases to significantly speed up query execution, but at present this feature is still imperfect and therefore disabled by default. In Firebird 6.0, they will try to add cost estimation for this feature, as well as fix a number of other shortcomings. In addition, Firebird 6.0 plans to add conversion of subqueries ALL/NOT IN/NOT EXISTS into anti-join.

In conclusion of the review of the execution of subqueries in IN/EXISTS, I would like to note that if you have a query of the form

```
SELECT ...
FROM T1
WHERE <primary key> IN (SELECT field FROM T2 ...)
```

or

```
SELECT ...
FROM T1
WHERE EXISTS (SELECT ... FROM T2 WHERE T1.<primary key> = T2.field)
```

then such queries are almost always more efficient to execute as

```
SELECT ...
FROM
  T1
  JOIN (SELECT DISTINCT field FROM T2) tmp ON tmp.field = T1.<primary key>
```

Let me give you a clear example:

```
SELECT
  COUNT(*)
FROM
  HORSE H
WHERE H.CODE_HORSE IN (
  SELECT
    CODE_FATHER
  FROM COVER
  WHERE EXTRACT(YEAR FROM COVER.BYDATE) = 2022
)
```

Execution plan and statistics using `Hash Join (semi)`

```
Select Expression
    -> Aggregate
        -> Filter
            -> Hash Join (semi)
                -> Table "HORSE" as "H" Full Scan
                -> Record Buffer (record length: 41)
                    -> Filter
                        -> Table "COVER" Access By ID
                            -> Bitmap
                                -> Index "IDX_COVER_BYYEAR" Range Scan (full match)

                COUNT
=====================
                 1616

Current memory = 554176768
Delta memory = 288
Max memory = 555531328
Elapsed time = 0.229 sec
Buffers = 32768
Reads = 0
Writes = 0
Fetches = 569683
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  |
--------------------------------+---------+---------+---------+---------+---------+
COVER                           |         |     6695|         |         |         |
HORSE                           |   525875|         |         |         |         |
--------------------------------+---------+---------+---------+---------+---------+
```

Quite fast, but the HORSE table is read in full.

Execution plan and statistics with classic subquery execution

```
Sub-query
    -> Filter
        -> Filter
            -> Table "COVER" Access By ID
                -> Bitmap And
                    -> Bitmap
                        -> Index "FK_COVER_FATHER" Range Scan (full match)
                    -> Bitmap
                        -> Index "IDX_COVER_BYYEAR" Range Scan (full match)
Select Expression
    -> Aggregate
        -> Filter
            -> Table "HORSE" as "H" Full Scan

                COUNT
=====================
                 1616
```

```
Current memory = 553472512
Delta memory = 288
Max memory = 553966592
Elapsed time = 6.862 sec
Buffers = 32768
Reads = 0
Writes = 0
Fetches = 2462726
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  |
--------------------------------+---------+---------+---------+---------+---------+
COVER                           |         |    1616|         |         |         |
HORSE                           |  525875|         |         |         |         |
--------------------------------+---------+---------+---------+---------+---------+
```

Very slow. The HORSE table is full scan, and the subquery is executed multiple times — for each record in the HORSE table.

And now a quick option with DISTINCT

```
SELECT
  COUNT(*)
FROM
  HORSE H
  JOIN (
      SELECT
        DISTINCT
        CODE_FATHER
      FROM COVER
      WHERE EXTRACT(YEAR FROM COVER.BYDATE) = 2022
  ) TMP ON TMP.CODE_FATHER = H.CODE_HORSE
```

```
Select Expression
    -> Aggregate
        -> Nested Loop Join (inner)
            -> Unique Sort (record length: 44, key length: 12)
                -> Filter
                    -> Table "COVER" as "TMP COVER" Access By ID
                        -> Bitmap
                            -> Index "IDX_COVER_BYYEAR" Range Scan (full match)
            -> Filter
                -> Table "HORSE" as "H" Access By ID
                    -> Bitmap
                        -> Index "PK_HORSE" Unique Scan

                COUNT
=====================
                 1616

Current memory = 554349728
Delta memory = 320
```

```
Max memory = 555531328
Elapsed time = 0.011 sec
Buffers = 32768
Reads = 0
Writes = 0
Fetches = 14954
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  |
--------------------------------+---------+---------+---------+---------+---------+
COVER                           |         |    6695|         |         |         |
HORSE                           |         |    1616|         |         |         |
--------------------------------+---------+---------+---------+---------+---------+
```

No unnecessary readings, the query is executed very quickly. Hence the conclusion - always look at the execution plan of subqueries in IN/EXISTS/ANY/SOME, and check alternative variants of writing queries.

8.5. Preliminary evaluation of invariant predicates

In Firebird 5.0, if the predicate in WHERE is invariant (i.e., it does not depend on the fields of datasets/tables), and it is FALSE, the optimizer will not read data from the dataset.

The simplest example is the always false condition 1=0. The idea of such condition is usually to return 0 records from the query.

```
SELECT COUNT(*) FROM HORSE
WHERE 1=0;
```

In Firebird 4.0

```
Select Expression
    -> Aggregate
        -> Filter
            -> Table "HORSE" Full Scan

                COUNT
=====================
                    0

Current memory = 2612572768
Delta memory = 0
Max memory = 2614392048
Elapsed time = 0.137 sec
Buffers = 153600
Reads = 0
Writes = 0
Fetches = 552573
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  |
```

```
------------------------------+---------+---------+---------+---------+---------+
HORSE                         |   519623|         |         |         |         |
------------------------------+---------+---------+---------+---------+---------+
```

As you can see, Firebird 4.0 has read all records in table `HORSE`, to check that there is no record which corresponds to the condition `1=0`. Not very intelligent, right?

In Firebird 5.0

```
Select Expression
    -> Aggregate
        -> Filter (preliminary)
            -> Table "HORSE" Full Scan

                COUNT
=====================
                    0

Current memory = 2580339248
Delta memory = 176
Max memory = 2582802608
Elapsed time = 0.005 sec
Buffers = 153600
Reads = 0
Writes = 0
Fetches = 0
```

As you can see, the statistics of the query shows 0 reads and 0 fetches, it means that Firebird 5.0 did not read anything from disk and from cache – the optimizer of v5 has calculated the value of the invariant predicate (`1=0`) before accessing table `HORSE` and excluded it.

The preliminary evaluation of invariant predicates is shown in the explain plan as `Filter (preliminary)`.

Practically, this feature is very useful for dynamically built queries. For example, we have the following query with parameter `:A`.

```
SELECT * FROM HORSE
WHERE :A=1;
```

The parameter `:A` does not depend on the fields of dataset (table `HORSE`), so it can be preliminary calculated, so we can “turn on” and “turn off” this query with this parameter.

Let’s consider more practical example: we need CTE to recursively find epy pedigree of the horse up to the 5th generation.

```
WITH RECURSIVE
  R AS (
    SELECT
      CODE_HORSE,
```

```
      CODE_FATHER,
      CODE_MOTHER,
      0 AS DEPTH
    FROM HORSE
    WHERE CODE_HORSE = ?
    UNION ALL
    SELECT
      HORSE.CODE_HORSE,
      HORSE.CODE_MOTHER,
      HORSE.CODE_FATHER,
      R.DEPTH + 1
    FROM R
      JOIN HORSE ON HORSE.CODE_HORSE = R.CODE_FATHER
    WHERE R.DEPTH < 5
    UNION ALL
    SELECT
      HORSE.CODE_HORSE,
      HORSE.CODE_MOTHER,
      HORSE.CODE_FATHER,
      R.DEPTH + 1
    FROM R
      JOIN HORSE ON HORSE.CODE_HORSE = R.CODE_MOTHER
    WHERE R.DEPTH < 5
  )
SELECT *
FROM R
```

Query's execution statistics in Firebird 4.0 looks as below (plan is not shown purposely):

```
Current memory = 2612639872
Delta memory = 0
Max memory = 2614392048
Elapsed time = 0.027 sec
Buffers = 153600
Reads = 0
Writes = 0
Fetches = 610
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  |
--------------------------------+---------+---------+---------+---------+---------+
HORSE                           |         |      127|         |         |         |
--------------------------------+---------+---------+---------+---------+---------+
```

Let's compare with Firebird 5.0:

```
Select Expression
    -> Recursion
        -> Filter
            -> Table "HORSE" as "R HORSE" Access By ID
                -> Bitmap
                    -> Index "PK_HORSE" Unique Scan
        -> Union
```

```
            -> Filter (preliminary)
                -> Filter
                    -> Table "HORSE" as "R HORSE" Access By ID
                        -> Bitmap
                            -> Index "PK_HORSE" Unique Scan
            -> Filter (preliminary)
                -> Filter
                    -> Table "HORSE" as "R HORSE" Access By ID
                        -> Bitmap
                            -> Index "PK_HORSE" Unique Scan

Current memory = 2580444832
Delta memory = 768
Max memory = 2582802608
Elapsed time = 0.024 sec
Buffers = 153600
Reads = 0
Writes = 0
Fetches = 252
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  |
--------------------------------+---------+---------+---------+---------+---------+
HORSE                           |         |      63|         |         |         |
--------------------------------+---------+---------+---------+---------+---------+
```

As you can see, Firebird used 2x less reads of table `HORSE`, die to the fact that condition `R.DEPTH < 5` is invariant for the each step of recursive query.

8.6. Faster IN with list of constants

Until Firebird 5.0, the predicate IN with a list of constants is limited by 1500 elements, and it was processed recursively, with transformation to the list of OR conditions.

It means that in pre-v5,

```
F IN (V1, V2, ... VN)
```

is actually transformed to

```
(F = V1) OR (F = V2) OR .... (F = VN)
```

Starting with Firebird 5.0 the processing of `IN` is linear, the of 1500 elements is increased to 65535.

In Firebird 5.0, list of constants in `IN` is cached as binary search tree to speed up the comparison

Let's see the following example:

```
SELECT
  COUNT(*)
```

```
FROM COVER
WHERE CODE_COVERRESULT+0 IN (151, 152, 156, 158, 159, 168, 170, 200, 202)
```

In this case we added `CODE_COVERRESULT+0` purposely to disable usage of index.

In Firebird 4.0

```
Select Expression
    -> Aggregate
        -> Filter
            -> Table "COVER" Full Scan

                COUNT
=====================
                45231

Current memory = 2612795072
Delta memory = -288
Max memory = 2614392048
Elapsed time = 0.877 sec
Buffers = 153600
Reads = 0
Writes = 0
Fetches = 738452
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  |
--------------------------------+---------+---------+---------+---------+---------+
COVER                           |   713407|         |         |         |         |
--------------------------------+---------+---------+---------+---------+---------+
```

In Firebird 5.0

```
Select Expression
    -> Aggregate
        -> Filter
            -> Table "COVER" Full Scan

                COUNT
=====================
                45231

Current memory = 2580573216
Delta memory = 224
Max memory = 2582802608
Elapsed time = 0.332 sec
Buffers = 153600
Reads = 0
Writes = 0
Fetches = 743126
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  |
```

```
-------------------------------+---------+---------+---------+---------+---------+
COVER                          |   713407|         |         |         |         |
-------------------------------+---------+---------+---------+---------+---------+
```

As you can see, even for small list in this example, and despite the fact that number of reads of table `COVER` did not change, the query is 2.5x faster.

If list is very long, or if predicate IN is not selective, index scanning will use search of groups with the pointer of the same level (i.e., horizontal), and not the search of each group from the root (i.e., vertical) — it means that it will use the single index scan for all values in the `IN` list.

See the following example:

```
SELECT
  COUNT(*)
FROM LAB_LINE
WHERE CODE_LABTYPE IN (4, 5)
```

The result in Firebird 4.0:

```
Select Expression
    -> Aggregate
        -> Filter
            -> Table "LAB_LINE" Access By ID
                -> Bitmap Or
                    -> Bitmap
                        -> Index "FK_LAB_LINE_LABTYPE" Range Scan (full match)
                    -> Bitmap
                        -> Index "FK_LAB_LINE_LABTYPE" Range Scan (full match)

                COUNT
=====================
               985594

Current memory = 2614023968
Delta memory = 0
Max memory = 2614392048
Elapsed time = 0.361 sec
Buffers = 153600
Reads = 0
Writes = 0
Fetches = 992519
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  |
--------------------------------+---------+---------+---------+---------+---------+
LAB_LINE                        |         |   985594|         |         |         |
--------------------------------+---------+---------+---------+---------+---------+
```

In Firebird 5.0:

```
Select Expression
    -> Aggregate
        -> Filter
            -> Table "LAB_LINE" Access By ID
                -> Bitmap
                    -> Index "FK_LAB_LINE_LABTYPE" List Scan (full match)

                COUNT
====================
               985594

Current memory = 2582983152
Delta memory = 176
Max memory = 2583119072
Elapsed time = 0.306 sec
Buffers = 153600
Reads = 0
Writes = 0
Fetches = 993103
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  |
--------------------------------+---------+---------+---------+---------+---------+
LAB_LINE                        |         |   985594|         |         |         |
--------------------------------+---------+---------+---------+---------+---------+
```

Though the time of query is not changed, the plan is different: instead of 2 Range Scan and bitmap join through OR, Firebird 5 uses the new access method with one-step index list scan, namely `List Scan` in the explained plan.

8.7. Optimizer strategy ALL ROWS vs FIRST ROWS

There are 2 strategies to optimize queries:

- FIRST ROWS — when optimizer builds query plan to return as soon as possible the first rows of the resulting dataset;
- ALL ROWS — when optimizer builds query plan to return all rows of the resulting dataset as soon as possible.

Until Firebird 5.0 these strategies also existed, but there was no way to control which one to use.

The default strategy was `ALL ROWS`, however, in case of clause `FIRST …`, `ROWS …` or `FETCH FIRST n ROWS`, the optimizer had used the strategy to `FIRST ROWS`. Also, for subselects in `IN` and in `EXISTS` it also used strategy `FIRST ROWS`.

Starting with Firebird 5.0, by default will be used the optimization strategy specified in the parameter OptimizeForFirstRows of firebird.conf or database.conf.

OptimizeForFirstRows = false means strategy ALL ROWS, OptimizeForFirstRows = true means FIRST ROWS.

It is possible to change the optimization strategy for the current session (connection) with the following command:

```
SET OPTIMIZE FOR {FIRST | ALL} ROWS
```

It can be useful for reporting and BI applications.

Also, the strategy can be set on the level of the SQL command with clause `OPTIMIZE FOR`.

`SELECT` query with clause `OPTIMIZE FOR` has the following syntax:

```
SELECT ...
FROM [...]
[WHERE ...]
[...]
[OPTIMIZE FOR {FIRST | ALL} ROWS]
```

Clause `OPTIMIZE FOR` should be specified in the end of `SELECT` query. In PSQL it should be set right before the clause `INTO`.

A bit of internals behind optimization strategies

Datasets in the query can be conveyors or buffered:

- Conveyor dataset returns records during the reading process of its input,
- Buffered dataset need to read all records from input, and only after competition it can return the first row.

If active strategy is `FIRST ROWS`, the optimizer will try to avoid buffered datasets (i.e., SORT or HASH JOIN).

Let's see how the choice of the optimization strategy changes the query plan — for this let's use clause `OPTIMIZE FOR`.

The example of query and plan with optimizer strategy `ALL ROWS`:

```
SELECT
  HORSE.NAME AS HORSENAME,
  SEX.NAME AS SEXNAME,
  COLOR.NAME AS COLORNAME
FROM
  HORSE
  JOIN SEX ON SEX.CODE_SEX = HORSE.CODE_SEX
  JOIN COLOR ON COLOR.CODE_COLOR = HORSE.CODE_COLOR
ORDER BY HORSE.NAME
OPTIMIZE FOR ALL ROWS
```

```
Select Expression
```

```
    -> Sort (record length: 876, key length: 304)
        -> Filter
            -> Hash Join (inner)
                -> Nested Loop Join (inner)
                    -> Table "COLOR" Full Scan
                    -> Filter
                        -> Table "HORSE" Access By ID
                            -> Bitmap
                                -> Index "FK_HORSE_COLOR" Range Scan (full match)
                -> Record Buffer (record length: 113)
                    -> Table "SEX" Full Scan
```

The example of query and plan with optimizer strategy `FIRST ROWS`:

```
SELECT
  HORSE.NAME AS HORSENAME,
  SEX.NAME AS SEXNAME,
  COLOR.NAME AS COLORNAME
FROM
  HORSE
  JOIN SEX ON SEX.CODE_SEX = HORSE.CODE_SEX
  JOIN COLOR ON COLOR.CODE_COLOR = HORSE.CODE_COLOR
ORDER BY HORSE.NAME
OPTIMIZE FOR FIRST ROWS
```

```
Select Expression
    -> Nested Loop Join (inner)
        -> Table "HORSE" Access By ID
            -> Index "HORSE_IDX_NAME" Full Scan
        -> Filter
            -> Table "SEX" Access By ID
                -> Bitmap
                    -> Index "PK_SEX" Unique Scan
        -> Filter
            -> Table "COLOR" Access By ID
                -> Bitmap
                    -> Index "PK_COLOR" Unique Scan
```

As you can see, in the first case, the optimizer has chosen the HASH JOIN and SORT to return all records of the query as soon as possible.

In the second case, the optimizer has chosen the index (`ORDER index`) and join with NESTED LOOP, because this plan will return the first rows as fast as possible.

8.8. Improved plan output

Queries plans are important for the understanding of the performance of queries, and in Firebird 5 we have better representation of explain plan's elements.

As you know, there are 2 types of plans: legacy and explained.

Now in the output of explain plan we can see user's SELECTs (shown as "select expressions"), declared PSQL cursors and sub-queries.

Both legacy and explain plans now includes the information of cursor position (line/column) inside PSQL module.

To see the difference, let's compare plan's outputs for several Firebird 4.0 and Firebird 5.0.

Let's start with the query with the subquery:

```
SELECT *
FROM HORSE
WHERE EXISTS(SELECT * FROM COVER
             WHERE COVER.CODE_FATHER = HORSE.CODE_HORSE)
```

The explained plan in Firebird 4.0 will look like this:

```
Select Expression
    -> Filter
        -> Table "COVER" Access By ID
            -> Bitmap
                -> Index "FK_COVER_FATHER" Range Scan (full match)
Select Expression
    -> Filter
        -> Table "HORSE" Full Scan
```

In Firebird the plan will look like the following:

```
Sub-query
    -> Filter
        -> Table "COVER" Access By ID
            -> Bitmap
                -> Index "FK_COVER_FATHER" Range Scan (full match)
Select Expression
    -> Filter
        -> Table "HORSE" Full Scan
```

Now in the plan you can clearly see where the main query is and where the subquery is.

Now let's compare how the plan is displayed for PSQL, for example in the statement `EXECUTE BLOCK`:

```
EXECUTE BLOCK
RETURNS (
   CODE_COLOR INT,
   CODE_BREED INT
)
AS
BEGIN
  FOR
    SELECT CODE_COLOR
```

```
    FROM COLOR
    INTO CODE_COLOR
  DO
    SUSPEND;

  FOR
    SELECT CODE_BREED
    FROM BREED
    INTO CODE_BREED
  DO
    SUSPEND;
END
```

In Firebird 4.0, both legacy and explain plans will be printed for each cursor within a block, without additional details, just one after the other.

```
PLAN (COLOR NATURAL)
PLAN (BREED NATURAL)
```

```
Select Expression
    -> Table "COLOR" Full Scan
Select Expression
    -> Table "BREED" Full Scan
```

In Firebird 5.0, each cursor plan will be preceded by the number of the column and row where the cursor is declared.

```
-- line 8, column 3
PLAN (COLOR NATURAL)
-- line 15, column 3
PLAN (BREED NATURAL)
```

```
Select Expression (line 8, column 3)
    -> Table "COLOR" Full Scan
Select Expression (line 15, column 3)
    -> Table "BREED" Full Scan
```

Now let's compare the output of explain plans if the cursor is declared explicitly.

```
EXECUTE BLOCK
RETURNS (
   CODE_COLOR INT
)
AS
  DECLARE C1 CURSOR FOR (
    SELECT CODE_COLOR
    FROM COLOR
  );
```

```
  DECLARE C2 SCROLL CURSOR FOR (
    SELECT CODE_COLOR
    FROM COLOR
  );
BEGIN
  SUSPEND;
END
```

For Firebird 4.0 the plan will be like this:

```
Select Expression
    -> Table "COLOR" as "C1 COLOR" Full Scan
Select Expression
    -> Record Buffer (record length: 25)
        -> Table "COLOR" as "C2 COLOR" Full Scan
```

The plan gives the impression that the `COLOR` table has an alias of `C1`, although this is not the case.

In Firebird 5.0 the plan will be much clearer:

```
Cursor "C1" (line 6, column 3)
    -> Table "COLOR" as "C1 COLOR" Full Scan
Cursor "C2" (scrollable) (line 11, column 3)
    -> Record Buffer (record length: 25)
        -> Table "COLOR" as "C2 COLOR" Full Scan
```

Firstly, it is clear that we have cursors C1 and C2 declared in the block.

Secondly, an additional “scrollable” attribute has been introduced for a bidirectional cursor.

8.9. How to get stored procedure plans

Until Firebird 3.0, the engine had shown plans for stored procedures as a compilation of plans of internal queries, and it was often misleading. In Firebird 3.0, the plan for stored procedures is always shown as NATURAL, and it is also not the best solution.

In Firebird 5 there is better option.

If we will try to see the plan for the stored procedure for the following query, it will not return desired details:

```
SELECT *
FROM SP_PEDIGREE(?, 5, 1)
```

```
Select Expression
    -> Procedure "SP_PEDIGREE" Scan
```

As expected, the top-level query plan is displayed without the details of the cursor plans inside the stored procedure, like in Firebird 3-4.

Firebird 5.0 has cache of the compiled queries, and monitoring table MON$COMPILED_STATEMENTS can show the plan for compiled queries, including stored procedures.

Once we have prepared query with stored procedure, this procedure will be cached, and its plan can be viewed with the following query:

```
SELECT CS.MON$EXPLAINED_PLAN
FROM MON$COMPILED_STATEMENTS CS
WHERE CS.MON$OBJECT_NAME = 'SP_PEDIGREE'
  AND CS.MON$OBJECT_TYPE = 5
ORDER BY CS.MON$COMPILED_STATEMENT_ID DESC
FETCH FIRST ROW ONLY
```

```
Cursor "V" (scrollable) (line 19, column 3)
    -> Record Buffer (record length: 132)
        -> Nested Loop Join (inner)
            -> Window
                -> Window Partition
                    -> Record Buffer (record length: 82)
                        -> Sort (record length: 84, key length: 12)
                            -> Window Partition
                                -> Window Buffer
                                    -> Record Buffer (record length: 41)
                                        -> Procedure "SP_HORSE_INBRIDS" as "V H_INB SP_HORSE_INBRIDS" Scan
            -> Filter
                -> Table "HUE" as "V HUE" Access By ID
                    -> Bitmap
                        -> Index "HUE_IDX_ORDER" Range Scan (full match)
Select Expression (line 44, column 3)
    -> Recursion
        -> Filter
            -> Table "HORSE" as "PEDIGREE HORSE" Access By ID
                -> Bitmap
                    -> Index "PK_HORSE" Unique Scan
        -> Union
            -> Filter (preliminary)
                -> Filter
                    -> Table "HORSE" as "PEDIGREE HORSE" Access By ID
                        -> Bitmap
                            -> Index "PK_HORSE" Unique Scan
            -> Filter (preliminary)
                -> Filter
                    -> Table "HORSE" as "PEDIGREE HORSE" Access By ID
                        -> Bitmap
                            -> Index "PK_HORSE" Unique Scan
```

Also, plans for stored procedures will be shown in the trace if configuration contains the line `log_procedure_compile = true`.

Chapter 9. New features in SQL language

9.1. Support for `WHEN NOT MATCHED BY SOURCE` clause in `MERGE` statement

The `MERGE` statement merges records from the source table and the target table (or updatable view).

During execution of a `MERGE` statement, the source records are read and then `INSERT`, `UPDATE` or `DELETE` is performed on the target table depending on the conditions.

Syntax of MERGE statement

```
MERGE
  INTO target [[AS] target_alias]
  USING <source> [[AS] source_alias]
  ON <join condition>
  <merge when> [<merge when> ...]
  [<plan clause>]
  [<order by clause>]
  [<returning clause>]

<source> ::= tablename | (<select_stmt>)

<merge when> ::=
    <merge when matched>
  | <merge when not matched by target>
  | <merge when not matched by source>

<merge when matched> ::=
  WHEN MATCHED [ AND <condition> ]
  THEN { UPDATE SET <assignment_list> | DELETE }

<merge when not matched by target> ::=
  WHEN NOT MATCHED [ BY TARGET ] [ AND <condition> ]
  THEN INSERT [ <left paren> <column_list> <right paren> ]
  VALUES <left paren> <value_list> <right paren>

<merge when not matched by source> ::=
  WHEN NOT MATCHED BY SOURCE [ AND <condition> ] THEN
  { UPDATE SET <assignment list> | DELETE }
```

Firebird 5.0 introduced conditional branches `<merge when not matched by source>`, which allow you to update or delete records from the target table if they are not present in the data source.

Now the `MERGE` statement is a truly universal tool for any modifications of the target table for a certain set of data.

The data source can be a table, view, stored procedure, or derived table. When a `MERGE` statement is executed, a join is made between the source (`USING`) and the target table. The join type depends on the presence of `WHEN NOT MATCHED` clause:

- <merge when not matched by target> and <merge when not matched by source> — FULL JOIN
- <merge when not matched by source> — RIGHT JOIN
- <merge when not matched by target> — LEFT JOIN
- only <merge when matched> — INNER JOIN

The action on the target table, as well as the condition under which it is performed, is described in the WHEN clause.

It is possible to have several clauses WHEN MATCHED, WHEN NOT MATCHED [BY TARGET] and WHEN NOT MATCHED BY SOURCE.

If the condition in the WHEN clause is not met, then Firebird skips it and moves on to the next clause.

This will continue until the condition for one of the WHEN clauses is met. In this case, the action associated with the WHEN clause is performed and the next record of the join result between the source (USING) and the target table is moved to. Only one action is performed for each result record of the join.

9.1.1. WHEN MATCHED

Specifies that all *target* rows that match the rows returned by the <source> ON <join condition> expression and satisfy additional search conditions are updated (UPDATE clause) or deleted (DELETE clause) according to the clause <merge when matched>.

Multiple WHEN MATCHED clauses are allowed. If more than one WHEN MATCHED clause is specified, all of them should be supplemented with additional search terms except the last one.

A MERGE statement cannot update the same row more than once, or it cannot update and delete the same row at the same time.

9.1.2. WHEN NOT MATCHED [BY TARGET]

Specifies that all *target* rows that do not match the rows returned by the <source> ON <join condition> expression and satisfy additional search conditions are inserted into the target table (INSERT clause) according to the clause <merge when not matched by target>.

Multiple WHEN NOT MATCHED [BY TARGET] clauses are allowed. If more than one WHEN NOT MATCHED [BY TARGET] clause is specified, then all of them should be supplemented with additional search terms, except for the last one.

9.1.3. WHEN NOT MATCHED BY SOURCE

Specifies that all *target* rows that do not match the rows returned by the <source> ON <join condition> expression and satisfy additional search conditions (UPDATE clause) or are deleted (DELETE clause) according to the clause <merge when not matched by source>.

The WHEN NOT MATCHED BY SOURCE clause became available in Firebird 5.0.

Multiple WHEN NOT MATCHED BY SOURCE clauses are allowed. If more than one WHEN NOT MATCHED BY

SOURCE clause is specified, all of them should be supplemented with additional search terms except the last one.

In the SET list of an UPDATE clause, it makes no sense to use expressions that refer to <source>, since no entries from match *target* entries.

9.1.4. Example of using MERGE with clause WHEN NOT MATCHED BY SOURCE

Let's say you have a price list in the tmp_price temporary table and you need to update the current price so that:

- if the product is not in the current price list, then add it;
- if the product is in the current price list, then update the price for it;
- if the product is included in the current price list. but it is not in the new one, then delete this price line.

All these actions can be done in the single SQL command:

```
MERGE INTO price
USING tmp_price
ON price.good_id = tmp_price.good_id
WHEN NOT MATCHED
  -- add if it wasn't there
  THEN INSERT (good_id, name, cost)
  VALUES (tmp_price.good_id, tmp_price.name, tmp_price.cost)
WHEN MATCHED AND price.cost <> tmp_price.cost THEN
  -- update the price if the product is in the new price list and the price is different
  UPDATE SET cost = tmp_price.cost
WHEN NOT MATCHED BY SOURCE
  -- if there is no product in the new price list, then we remove it from the current price
list
  DELETE;
```

In this example, instead of the temporary table tmp_price, there can be an arbitrarily complex SELECT query or stored procedure. Please note, that since both the WHEN NOT MATCHED [BY TARGET] and WHEN NOT MATCHED BY SOURCE clauses are present, the join between the target table and the data source will be done using a FULL JOIN. In the current version of Firebird FULL JOIN will not use indices on both the right and left, and will be slow.

9.2. Clause SKIP LOCKED

Firebird 5.0 introduced the SKIP LOCKED clause, which can be used in SELECT .. WITH LOCK, UPDATE, and DELETE statements.

Using this clause causes the engine to skip records locked by other transactions instead of waiting for them, or cause update conflict errors.

Using SKIP LOCKED is useful for implementing work queues, in which one or more processes submit work to a table and emit an event, while worker (executor) threads listen for events and read/remove items from the table. Using SKIP LOCKED, multiple workers can receive exclusive jobs from a table without conflicts.

```
SELECT
  [FIRST ...]
  [SKIP ...]
  FROM <sometable>
  [WHERE ...]
  [PLAN ...]
  [ORDER BY ...]
  [{ ROWS ... } | {OFFSET ...} | {FETCH ...}]
  [FOR UPDATE [OF ...]]
  [WITH LOCK [SKIP LOCKED]]
```

```
UPDATE <sometable>
  SET ...
  [WHERE ...]
  [PLAN ...]
  [ORDER BY ...]
  [ROWS ...]
  [SKIP LOCKED]
  [RETURNING ...]
```

```
DELETE FROM <sometable>
  [WHERE ...]
  [PLAN ...]
  [ORDER BY ...]
  [ROWS ...]
  [SKIP LOCKED]
  [RETURNING ...]
```

When using the SKIP LOCKED clause, locked records are first skipped and then FIRST/SKIP/ROWS/OFFSET/FETCH restrictsions are applied to the remaining records.

Example:

- Create table and trigger:

```
create table emails_queue (
  subject varchar(60) not null,
  text blob sub_type text not null
);

set term !;

create trigger emails_queue_ins after insert on emails_queue
```

```
as
begin
  post_event('EMAILS_QUEUE');
end!

set term ;!
```

- Sending a message by an application

```
insert into emails_queue (subject, text)
values ('E-mail subject', 'E-mail text...');

commit;
```

- Client application

```
-- The client application can check table to the EMAILS_QUEUE event,
-- to send emails using this command:

delete from emails_queue
  rows 10
  skip locked
  returning subject, text;
```

More than one instance of an application can be running, for example for load balancing.

The use of `SKIP LOCKED` for organizing queues will be discussed in in the separate article.

9.3. Support for returning multiple records by operators with clause `RETURNING`

Since Firebird 5.0, client-side modification statements `INSERT .. SELECT`, `UPDATE`, `DELETE`, `UPDATE OR INSERT` and `MERGE`, with clause `RETURNING` will return a cursor: it means that they are able to return multiple rows instead of throwing the "multiple rows in singleton select" error as previously.

These queries are now described as `isc_info_sql_stmt_select`, whereas in previous versions they were described as `isc_info_sql_stmt_exec_procedure`.

Singleton statements `INSERT .. VALUES`, and positioned statements `UPDATE` and `DELETE` (those containing a `WHERE CURRENT OF` clause) retain the existing behavior and are described as `isc_info_sql_stmt_exec_procedure`.

However, all of these statements, if used in PSQL, and if the `RETURNING` clause is used, are still treated as singletons.

Examples of modifying statements containing `RETURNING` and returning a cursor:

```
INSERT INTO dest(name, val)
SELECT desc, num + 1 FROM src WHERE id_parent = 5
RETURNING id, name, val;

UPDATE dest
SET a = a + 1
RETURNING id, a;

DELETE FROM dest
WHERE price < 0.52
RETURNING id;

MERGE INTO PRODUCT_INVENTORY AS TARGET
USING (
  SELECT
    SL.ID_PRODUCT,
    SUM(SL.QUANTITY)
  FROM
    SALES_ORDER_LINE SL
    JOIN SALES_ORDER S ON S.ID = SL.ID_SALES_ORDER
  WHERE S.BYDATE = CURRENT_DATE
    AND SL.ID_PRODUCT = :ID_PRODUCT
  GROUP BY 1
) AS SRC(ID_PRODUCT, QUANTITY)
ON TARGET.ID_PRODUCT = SRC.ID_PRODUCT
WHEN MATCHED AND TARGET.QUANTITY - SRC.QUANTITY <= 0 THEN
  DELETE
WHEN MATCHED THEN
  UPDATE SET
    TARGET.QUANTITY = TARGET.QUANTITY - SRC.QUANTITY,
    TARGET.BYDATE = CURRENT_DATE
RETURNING OLD.QUANTITY, NEW.QUANTITY, SRC.QUANTITY;
```

9.4. Partial indices

In Firebird 5.0, when creating an index, it became possible to specify an optional `WHERE` clause, which specifies a search condition that limits the subset of table records to be indexed. Such indices are called partial indices. The search condition must contain one or more table columns.

The partial index definition may include a `UNIQUE` specification. In this case, each key in the index must be unique. This allows you to ensure uniqueness for a certain subset of table rows.

The definition of a partial index can also include a `COMPUTED BY` clause so that the partial index can be computed.

So the complete syntax for creating an index is as follows:

```
CREATE [UNIQUE] [ASC[ENDING] | DESC[ENDING]]
INDEX indexname ON tablename
{(<column_list>) | COMPUTED [BY] (<value_expression>)}
[WHERE <search_condition>]
```

```
<column_list> ::= col [, col ...]
```

The optimizer can only use a partial index in the following cases:

- the WHERE clause includes exactly the same logical expression as the one defined for the index;
- the search condition defined for the index contains Boolean expressions combined with an OR, and one of them is explicitly included in the WHERE clause;
- The search condition defined for the index specifies IS NOT NULL, and the WHERE clause includes an expression for the same field that is known to ignore NULL.

If a regular index and a partial index exist for the same set of fields, the optimizer will choose the regular index even if the WHERE clause includes the same expression as defined in the partial index.

The reason for this behavior is that the regular index has better selectivity than the partial index.

But there are exception to this rule: using predicates with poor selectivity on indexed fields, such as <>, IS DISTINCT FROM, or IS NOT NULL, provided that the predicate is used in a partial index.

Partial indices cannot be used to constrain a primary key or a foreign key: USING INDEX clause cannot specify a partial index definition.

Let's see when partial indices are useful.

Example 1. Partial uniqueness

Let's say we have a table storing a person's email address.

```
CREATE TABLE MAN_EMAILS (
  CODE_MAN_EMAIL BIGINT GENERATED BY DEFAULT AS IDENTITY,
  CODE_MAN BIGINT NOT NULL,
  EMAIL VARCHAR(50) NOT NULL,
  DEFAULT_FLAG BOOLEAN DEFAULT FALSE NOT NULL,
  CONSTRAINT PK_MAN_EMAILS PRIMARY KEY(CODE_MAN_EMAIL),
  CONSTRAINT FK_EMAILS_REF_MAN FOREIGN KEY(CODE_MAN) REFERENCES MAN(CODE_MAN)
);
```

One person can have many email addresses, but only one can be the default address. A regular unique index or restriction will not work in this case, since in this case we will be limited to only two addresses.

Here we can use the partial unique index:

```
CREATE UNIQUE INDEX IDX_UNIQUE_DEFAULT_MAN_EMAIL
ON MAN_EMAILS(CODE_MAN) WHERE DEFAULT_FLAG IS TRUE;
```

Thus, for one person we allow as many addresses as desired with DEFAULT_FLAG=FALSE and only

one address with DEFAULT_FLAG=TRUE.

Partial indices can be used simply to make the index more compact.

Example 2. Reducing the index size

Suppose you have a horse table HORSE in your database and it has the IS_ANCESTOR field, which is used to indicate whether the horse is the ancestor of a line or family. Obviously, there are hundreds of times fewer ancestors than other horses — see the result of the query below:

```
SELECT
  COUNT(*) FILTER(WHERE IS_ANCESTOR IS TRUE) AS CNT_ANCESTOR,
  COUNT(*) FILTER(WHERE IS_ANCESTOR IS FALSE) AS CNT_OTHER
FROM HORSE
```

```
         CNT_ANCESTOR             CNT_OTHER
==================== ====================
                1426                518197
```

The goal is to quickly obtain a list of ancestors. From the above statistics it is also obvious that for the IS_ANCESTOR IS FALSE option, the use of index is practically useless.

Let’s try to create a regular index:

```
CREATE INDEX IDX_HORSE_ANCESTOR ON HORSE(IS_ANCESTOR);
```

But in this case, such an index will be redundant. Let’s look at its statistics using gstat tool:

```
    Index IDX_HORSE_ANCESTOR (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
```

Instead of a regular index, we can create a partial index (the previous one must be deleted):

```
CREATE INDEX IDX_HORSE_ANCESTOR ON HORSE(IS_ANCESTOR) WHERE IS_ANCESTOR IS TRUE;
```

Let’s compare the statistics using gstat tool:

```
    Index IDX_HORSE_ANCESTOR (26)
        Root page: 163417, depth: 1, leaf buckets: 1, nodes: 1426
        Average node length: 4.75, total dup: 1425, max dup: 1425
        Average key length: 2.00, compression ratio: 0.50
        Average prefix length: 1.00, average data length: 0.00
        Clustering factor: 764, ratio: 0.54
        Fill distribution:
             0 - 19% = 0
            20 - 39% = 0
            40 - 59% = 1
            60 - 79% = 0
            80 - 99% = 0
```

As you can see, the partial index is much more compact — there are 1426 nodes in partial index instead 519623 in regular. Let's check that it can be used to obtain ancestors:

```
SELECT COUNT(*)
FROM HORSE
WHERE IS_ANCESTOR IS TRUE;
```

```
Select Expression
    -> Aggregate
        -> Filter
            -> Table "HORSE" Access By ID
                -> Bitmap
                    -> Index "IDX_HORSE_ANCESTOR" Full Scan

                COUNT
=====================
                 1426

Current memory = 556868928
Delta memory = 176
Max memory = 575376064
Elapsed time = 0.007 sec
Buffers = 32768
Reads = 0
Writes = 0
Fetches = 2192
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  | Backout | Purge   | Expunge |
--------------------------------+---------+---------+---------+---------+---------+---------+---------+---------+
HORSE                           |         |    1426|         |         |         |         |         |         |
--------------------------------+---------+---------+---------+---------+---------+---------+---------+---------+
```

Please note that if you specify `WHERE IS_ANCESTOR` or `WHERE IS_ANCESTOR = TRUE` in the query, the index will not be used. It is necessary that the expression specified to filter the index completely matches the expression in the `WHERE` of your query.

Another case when partial indices can be useful is when using them with non-selective predicates.

Example 3. Using partial indices with non-selective predicates

Suppose we need to get all dead horses for which the date of death is known. A horse is definitely dead if it has a date of death, but it often happens that it is not listed or is simply unknown. Moreover, the number of unknown dates of death is much greater than the known ones. To do this, we will write the following query:

```
SELECT COUNT(*)
FROM HORSE
WHERE DEATHDATE IS NOT NULL;
```

We want to get this list as quickly as possible, so we'll try to create an index on the `DEATHDATE` field.

```
CREATE INDEX IDX_HORSE_DEATHDATE
ON HORSE(DEATHDATE);
```

Now let's try to run the query above and look at its plan and statistics:

```
Select Expression
    -> Aggregate
        -> Filter
            -> Table "HORSE" Full Scan

                COUNT
=====================
                16234

Current memory = 2579550800
Delta memory = 176
Max memory = 2596993840
Elapsed time = 0.196 sec
Buffers = 153600
Reads = 0
Writes = 0
Fetches = 555810
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  | Backout | Purge   | Expunge |
--------------------------------+---------+---------+---------+---------+---------+---------+---------+---------+
HORSE                           |   519623|         |         |         |         |         |         |         |
--------------------------------+---------+---------+---------+---------+---------+---------+---------+---------+
```

As you can see, it was not possible to use the index.

The reason is that the predicates `IS NOT NULL`, `<>`, `IS DISTINCT FROM` are low-selective.

Currently, Firebird does not have histograms with the distribution of index key values, and therefore the distribution is assumed to be uniform. With a uniform distribution, there is no point in using an index for such predicates, which is what is done.

Now let's try to delete the previously created index and create a partial index instead:

```
DROP INDEX IDX_HORSE_DEATHDATE;

CREATE INDEX IDX_HORSE_DEATHDATE
ON HORSE(DEATHDATE) WHERE DEATHDATE IS NOT NULL;
```

And let's try to repeat the request above:

```
Select Expression
    -> Aggregate
        -> Filter
            -> Table "HORSE" Access By ID
                -> Bitmap
                    -> Index "IDX_HORSE_DEATHDATE" Full Scan

                COUNT
=====================
                16234

Current memory = 2579766848
Delta memory = 176
Max memory = 2596993840
Elapsed time = 0.017 sec
Buffers = 153600
Reads = 0
Writes = 0
Fetches = 21525
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  | Backout | Purge   | Expunge |
--------------------------------+---------+---------+---------+---------+---------+---------+---------+---------+
HORSE                           |         |    16234|         |         |         |         |         |         |
--------------------------------+---------+---------+---------+---------+---------+---------+---------+---------+
```

As you can see, the optimizer managed to use our index. But the most interesting thing is that our index will continue to work with other date comparison predicates (but it will not work for `IS NULL`).

See example below:

```
SELECT COUNT(*)
FROM HORSE
WHERE DEATHDATE = DATE '2005-01-01';
```

```
Select Expression
    -> Aggregate
        -> Filter
            -> Table "HORSE" Access By ID
                -> Bitmap
                    -> Index "IDX_HORSE_DEATHDATE" Range Scan (full match)

                COUNT
=====================
                  190

Current memory = 2579872992
Delta memory = 192
```

```
Max memory = 2596993840
Elapsed time = 0.004 sec
Buffers = 153600
Reads = 0
Writes = 0
Fetches = 376
Per table statistics:
--------------------------------+---------+---------+---------+---------+---------+---------+---------+---------+
 Table name                     | Natural | Index   | Insert  | Update  | Delete  | Backout | Purge   | Expunge |
--------------------------------+---------+---------+---------+---------+---------+---------+---------+---------+
HORSE                           |         |     190|         |         |         |         |         |         |
--------------------------------+---------+---------+---------+---------+---------+---------+---------+---------+
```

The optimizer in this case realized that the `IS NOT NULL` filter condition in the partial index covers any other predicates that do not compare to `NULL`.

It is important to note that if you specify the condition `FIELD > 2` in the partial index, and the query contains the search condition `FIELD > 1`, then despite the fact that any number greater than 2 is also greater than 1, the partial index will not be used. The optimizer is not smart enough to derive this equivalence condition.

9.5. Functions `UNICODE_CHAR` and `UNICODE_VAL`

Firebird 2.1 introduced a pair of functions `ASCII_CHAR` — returning a character by its code in the ASCII table, and `ASCII_VAL` — returning the code in the ASCII table by character. These functions only apply to single-byte encodings; there is nothing similar for UTF-8. Firebird 5.0 added two more functions that work with multibyte encodings:

```
UNICODE_CHAR (number)

UNICODE_VAL (string)
```

The `UNICODE_CHAR` function returns the UNICODE character for the given code point.

The `UNICODE_VAL` function returns the UTF-32 code point for the first character in a string. For an empty string, 0 is returned.

```
SELECT
  UNICODE_VAL(UNICODE_CHAR(0x1F601)) AS CP_VAL,
  UNICODE_CHAR(0x1F601) AS CH
FROM RDB$DATABASE
```

9.6. Query expressions in parentheses

In 5.0, the DML syntax was expanded to allow the use of a query expression within parentheses (SELECT, including `order by`, `offset` and `fetch` clauses, but without `with` clause), where previously only the query specification was allowed (SELECT without clauses `with`, `order by`, `offset` and `fetch`).

This allows you to write clearer queries, especially in `UNION` statements, and provides greater

compatibility with statements generated by some ORMs.

Using query expressions in parentheses is not free from the Firebird engine's point of view, since they require additional query context compared to a simple query specification. The maximum number of request contexts in a statement is limited to 255.

Example:

```
(
  select emp_no, salary, 'lowest' as type
  from employee
  order by salary asc
  fetch first row only
)
union all
(
  select emp_no, salary, 'highest' as type
  from employee
  order by salary desc
  fetch first row only
);
```

9.7. Improved Literals

9.7.1. Full syntax of string literals

The character string literal syntax has been changed to support full standard SQL syntax. This means that the literal can be "interrupted" by spaces or comments. This can be used, for example, to split a long literal across multiple lines or to provide inline comments.

String literal syntax according to ISO/IEC 9075-2:2016 SQL - Part 2: Foundation

```
<character string literal> ::=
  [ <introducer> <character set specification> ]
    <quote> [ <character representation>... ] <quote>
    [ { <separator> <quote> [ <character representation>... ] <quote> }... ]

<separator> ::=
  { <comment> | <white space> }...
```

Example:

```
-- spaces between literals
select 'ab'
       'cd'
from RDB$DATABASE;
-- output: 'abcd'
```

```
-- comment and spaces between literals
select 'ab' /* comment */ 'cd'
from RDB$DATABASE;
-- output: 'abcd'
```

9.7.2. Complete syntax for binary literals

The syntax for binary string literals has been changed to support full standard SQL syntax. This means that the literal can contain spaces to separate hexadecimal characters and can be “interrupted” by spaces or comments. This can be used, for example, to make a hexadecimal string more readable by grouping characters, or to split a long literal over multiple lines, or to provide inline comments.

The binary literal syntax is as per ISO/IEC 9075-2:2016 SQL - Part 2: Foundation

```
<binary string literal> ::=
  {X|x} <quote> [ <space>... ] [ { <hexit> [ <space>... ] <hexit> [ <space>... ] }...
      ] <quote>
    [ { <separator> <quote> [ <space>... ] [ { <hexit> [ <space>... ]
    <hexit> [ <space>... ] }... ] <quote> }... ]
```

Examples:

```
-- Группировка по байтам (пробелы внутри литерала)
select _win1252 x'42 49 4e 41 52 59'
from RDB$DATABASE;
-- output: BINARY

-- пробелы между литералами
select _win1252 x'42494e'
                 '415259'
from RDB$DATABASE;
-- output: BINARY
```

9.8. Improved predicate IN

Prior to Firebird 5.0, the `IN` predicate with a list of constants was limited to 1500 elements because it was processed by recursively converting the original expression into an equivalent form.

This

```
F IN (V1, V2, ... VN)
```

will be transformed into

```
F = V1 OR F = V2 OR .... F = VN
```

Since Firebird 5.0, processing of IN predicates is linear. The 1500 item limit has been increased to 65535 items. In addition, queries using the IN predicate with a list of constants are processed much faster. This was discussed in detail in the first part.

9.9. Package RDB$BLOB_UTIL

The operations with BLOBs inside PSQL were not fast, because any modification to a BLOB always creates a new temporary BLOB, which leads to additional memory consumption and, in some cases, to a larger database file for storing temporary BLOBs.

In Firebird 4.0.2, a built-in function, BLOB_APPEND, was added to solve BLOB concatenation problems. In Firebird 5.0, was added a built-in RDB$BLOB_UTIL package with procedures and functions for more efficient BLOB manipulation.

We will show several practical examples how to use functions from the package RDB$BLOB_UTIL. The full description can be found in the Firebird 5.0 Release Notes and in the Firebird 5.0 SQL Language Reference.

9.9.1. Using the function RDB$BLOB_UTIL.NEW_BLOB

The RDB$BLOB_UTIL.NEW_BLOB function creates a new BLOB SUB_TYPE BINARY. It returns a BLOB suitable for appending data, similar to BLOB_APPEND.

The difference over BLOB_APPEND is that you can set parameters SEGMENTED and TEMP_STORAGE.

The BLOB_APPEND function always creates blobs in temporary storage, which may not always be the best approach if the created blob will be stored in a permanent table because it would require a copy operation.

The BLOB returned by RDB$BLOB_UTIL.NEW_BLOB can be used with BLOB_APPEND to append data, even if TEMP_STORAGE = FALSE.

Table 2. Input parameters for function RDB$BLOB_UTIL.NEW_BLOB

Parameter	Type	Description
SEGMENTED	BOOLEAN NOT NULL	Type of BLOB. If TRUE - a segmented BLOB will be created, FALSE - a streaming one.
TEMP_STORAGE	BOOLEAN NOT NULL	In what storage is the BLOB created? TRUE - in temporary, FALSE - in permanent (for writing to a regular table).

Return type

BLOB SUB_TYPE BINARY

Example:

```
execute block
```

```
declare b blob sub_type text;
as
begin
  -- create a streaming non-temporary BLOB, since it will be added to the table later
  b = rdb$blob_util.new_blob(false, false);

  b = blob_append(b, 'abcde');
  b = blob_append(b, 'fghikj');

  update t
  set some_field = :b
  where id = 1;
end
```

9.9.2. Reading BLOBs in chunks

When you needed to read part of a BLOB, you used the SUBSTRING function, but this function has one significant drawback: it always returns a new temporary BLOB.

Since Firebird 5.0 you can use the RDB$BLOB_UTIL.READ_DATA function for this purpose.

Table 3. Input parameters for function RDB$BLOB_UTIL.READ_DATA

Parameter	Type	Description
HANDLE	INTEGER NOT NULL	Handle of opened BLOB.
LENGTH	INTEGER	Quantity of bytes to read.

Return type
VARBINARY(32765)

The RDB$BLOB_UTIL.READ_DATA function is used to read pieces of data from a BLOB handle opened with RDB$BLOB_UTIL.OPEN_BLOB. When the BLOB has been completely read and there is no more data, it returns NULL.

If LENGTH parameter value is a positive number, a VARBINARY of maximum length LENGTH is returned.

If NULL is passed to LENGTH, a BLOB segment with a maximum length of 32765 is returned.

When you are done with a BLOB handle, you must close it using the RDB$BLOB_UTIL.CLOSE_HANDLE procedure.

Example 4. Opening a BLOB and returning it piece by piece to EXECUTE BLOCK

```
execute block returns (s varchar(10))
as
    declare b blob = '1234567';
    declare bhandle integer;
begin
    -- opens a BLOB for reading and returns its handle.
    bhandle = rdb$blob_util.open_blob(b);
```

```
    -- Getting the blob in parts
    s = rdb$blob_util.read_data(bhandle, 3);
    suspend;

    s = rdb$blob_util.read_data(bhandle, 3);
    suspend;

    s = rdb$blob_util.read_data(bhandle, 3);
    suspend;

    -- When there is no more data, NULL is returned.
    s = rdb$blob_util.read_data(bhandle, 3);
    suspend;

    -- Close the BLOB handle.
    execute procedure rdb$blob_util.close_handle(bhandle);
end
```

By passing the `NULL` value as the `LENGTH` parameter, you can read a BLOB segment by segment, if the segments do not exceed 32765 bytes.

Let's write a procedure to return a BLOB segment by segment

```
CREATE OR ALTER PROCEDURE SP_GET_BLOB_SEGEMENTS (
  TXT BLOB SUB_TYPE TEXT CHARACTER SET NONE
)
RETURNS (
  SEG VARCHAR(32765) CHARACTER SET NONE
)
AS
  DECLARE H INTEGER;
BEGIN
  H = RDB$BLOB_UTIL.OPEN_BLOB(TXT);
  SEG = RDB$BLOB_UTIL.READ_DATA(H, NULL);
  WHILE (SEG IS NOT NULL) DO
  BEGIN
    SUSPEND;
    SEG = RDB$BLOB_UTIL.READ_DATA(H, NULL);
  END
  EXECUTE PROCEDURE RDB$BLOB_UTIL.CLOSE_HANDLE(H);
END
```

It can be used, for example, like this:

```
WITH
  T AS (
    SELECT LIST(CODE_HORSE) AS B
    FROM HORSE
  )
SELECT
  S.SEG
FROM T
```

```
  LEFT JOIN SP_GET_BLOB_SEGEMENTS(T.B) S ON TRUE
```

Chapter 10. Why SKIP LOCKED was developed?

The one of the often tasks in the development of applications is to organize processing of queue "task manager - executor". For example, one or more task managers put jobs into a queue, and executors take a outstanding job from the queue and execute it, then they update the status of the job. If there is only one executor, then there are no problems. As the number of executors increases, competition for the task and conflicts between executors arise. The clause SKIP LOCKED helps developers to implement this type of queue processing in easy way without conflicts.

10.1. Preparing the Database

Let's try to implement a task processing queue. To do this, let's create a test database and create the QUEUE_TASK table in it. Task managers will add tasks to this table, and executors will take free tasks and complete them. The database creation script with comments is given below:

```
CREATE DATABASE 'inet://localhost:3055/c:\fbdata\5.0\queue.fdb'
USER SYSDBA password 'masterkey'
DEFAULT CHARACTER SET UTF8;

CREATE DOMAIN D_QUEUE_TASK_STATUS
AS SMALLINT CHECK(VALUE IN (0, 1));

COMMENT ON DOMAIN D_QUEUE_TASK_STATUS IS 'Task completion status';

CREATE TABLE QUEUE_TASK (
  ID BIGINT GENERATED BY DEFAULT AS IDENTITY NOT NULL,
  NAME VARCHAR(50) NOT NULL,
  STARTED BOOLEAN DEFAULT FALSE NOT NULL,
  WORKER_ID BIGINT,
  START_TIME TIMESTAMP,
  FINISH_TIME TIMESTAMP,
  FINISH_STATUS D_QUEUE_TASK_STATUS,
  STATUS_TEXT VARCHAR(100),
  CONSTRAINT PK_QUEUE_TASK PRIMARY KEY(ID)
);

COMMENT ON TABLE QUEUE_TASK IS 'Task queue';
COMMENT ON COLUMN QUEUE_TASK.ID IS 'Task Identifier';
COMMENT ON COLUMN QUEUE_TASK.NAME IS 'Task Name';
COMMENT ON COLUMN QUEUE_TASK.STARTED IS 'Flag that the task has been accepted for processing';
COMMENT ON COLUMN QUEUE_TASK.WORKER_ID IS 'ID of Executor';
COMMENT ON COLUMN QUEUE_TASK.START_TIME IS 'Task execution time start';
COMMENT ON COLUMN QUEUE_TASK.FINISH_TIME IS 'Task execution time finish';
COMMENT ON COLUMN QUEUE_TASK.FINISH_STATUS IS 'The status with which the task completed 0 - successfully, 1 - with
error';
COMMENT ON COLUMN QUEUE_TASK.STATUS_TEXT IS 'Status text. If the task is completed without errors, then "OK", otherwise
the error text';
```

To add a new task, execute the command

```
INSERT INTO QUEUE_TASK(NAME) VALUES (?)
```

In this example, we pass only the task name; in practice, there may be more parameters.

Each executor must select one outstanding task and set its flag to "Taken for processing".

An executor can get a free task using the following request:

```
SELECT ID, NAME
FROM QUEUE_TASK
WHERE STARTED IS FALSE
ORDER BY ID
FETCH FIRST ROW ONLY
```

Next, the executor marks the task as "Taken for processing", sets the task start time and the executor identifier. This is done with the command:

```
UPDATE QUEUE_TASK
SET
    STARTED = TRUE,
    WORKER_ID = ?,
    START_TIME = CURRENT_TIMESTAMP
WHERE ID = ?
```

After the task is accepted for processing, the actual execution of the task begins. When a task is completed, it is necessary to set the completion time of the task and its status. The task may complete with an error; in this case, the appropriate status is set and the error text is saved.

```
UPDATE QUEUE_TASK
SET
    FINISH_STATUS = ?,
    STATUS_TEXT = ?,
    FINISH_TIME = CURRENT_TIMESTAMP
WHERE ID = ?
```

10.2. Script simulating a job queue

Let’s try to test our idea. To do this, let’s write a simple script in Python.

To write a script, we will need to install two libraries:

```
pip install firebird-driver
pip install prettytable
```

Now you can start writing the script. The script is written to run under Windows, however it can also be run under Linux by changing some constants and the path to the `fbclient` library. Let’s save the written script to the file `queue_exec.py`:

```
#!/usr/bin/python3

import concurrent.futures as pool
```

```
import logging
import random
import time

from firebird.driver import connect, DatabaseError
from firebird.driver import driver_config
from firebird.driver import tpb, Isolation, TraAccessMode
from firebird.driver.core import TransactionManager
from prettytable import PrettyTable

driver_config.fb_client_library.value = "c:\\firebird\\5.0\\fbclient.dll"

DB_URI = 'inet://localhost:3055/d:\\fbdata\\5.0\\queue.fdb'
DB_USER = 'SYSDBA'
DB_PASSWORD = 'masterkey'
DB_CHARSET = 'UTF8'

WORKERS_COUNT = 4  # Number of Executors
WORKS_COUNT = 40   # Number of Tasks

# set up logging to console
stream_handler = logging.StreamHandler()
stream_handler.setLevel(logging.INFO)

logging.basicConfig(level=logging.DEBUG,
                    handlers=[stream_handler])

class Worker:
    """Class Worker is am executor"""

    def __init__(self, worker_id: int):
        self.worker_id = worker_id

    @staticmethod
    def __next_task(tnx: TransactionManager):
        """Retrieves the next task from the queue.

        Arguments:
            tnx: The transaction in which the request is executed
        """
        cur = tnx.cursor()

        cur.execute("""
            SELECT ID, NAME
            FROM QUEUE_TASK
            WHERE STARTED IS FALSE
            ORDER BY ID
            FETCH FIRST ROW ONLY
        """)

        row = cur.fetchone()
        cur.close()
        return row

    def __on_start_task(self, tnx: TransactionManager, task_id: int) -> None:
        """Fires when task execution starts.

        Sets the flag to the task to indicate that it is running, and sets the start time of the task.

        Arguments:
            tnx: The transaction in which the request is executed
            task_id: Task ID
        """
        cur = tnx.cursor()
        cur.execute(
```

```
            """
            UPDATE QUEUE_TASK
            SET
                STARTED = TRUE,
                WORKER_ID = ?,
                START_TIME = CURRENT_TIMESTAMP
            WHERE ID = ?
            """,
            (self.worker_id, task_id,)
        )

    @staticmethod
    def __on_finish_task(tnx: TransactionManager, task_id: int, status: int, status_text: str) -> None:
        """Fires when a task completes.

        Sets the task completion time and the status with which the task completed.

        Arguments:
            tnx: The transaction in which the request is executed
            task_id: Task ID
            status: Completion status code. 0 - successful, 1 - completed with error
            status_text: Completion status text. If successful, write "OK",
                otherwise the error text.
        """
        cur = tnx.cursor()
        cur.execute(
            """
                UPDATE QUEUE_TASK
                SET
                    FINISH_STATUS = ?,
                    STATUS_TEXT = ?,
                    FINISH_TIME = CURRENT_TIMESTAMP
                WHERE ID = ?
            """,
            (status, status_text, task_id,)
        )

    def on_task_execute(self, task_id: int, name: str) -> None:
        """This method is given as an example of a function to perform some task.

        In real problems it could be different and with a different set of parameters.

        Arguments:
            task_id: Task ID
            name: Task Name
        """
        # let get random delay
        t = random.randint(1, 4)
        time.sleep(t * 0.01)
        # to demonstrate that a task can be performed with errors,
        # let's generate an exception for two of the random numbers.
        if t == 3:
            raise Exception("Some error")

    def run(self) -> int:
        """Task Execution"""
        conflict_counter = 0
        # For parallel execution, each thread must have its own connection to the database.
        with connect(DB_URI, user=DB_USER, password=DB_PASSWORD, charset=DB_CHARSET) as con:
            tnx = con.transaction_manager(tpb(Isolation.SNAPSHOT, lock_timeout=0, access_mode=TraAccessMode.WRITE))
            while True:
                # We extract the next outstanding task and give it a sign that it is being executed.
                # Since the task may be executed with an error, the task start sign
                # is set in the separate transaction.
                tnx.begin()
```

```
                try:
                    task_row = self.__next_task(tnx)
                    # If the tasks are finished, we terminate the thread
                    if task_row is None:
                        tnx.commit()
                        break
                    (task_id, name,) = task_row
                    self.__on_start_task(tnx, task_id)
                    tnx.commit()
                except DatabaseError as err:
                    if err.sqlstate == "40001":
                        conflict_counter = conflict_counter + 1
                        logging.error(f"Worker: {self.worker_id}, Task: {self.worker_id}, Error: {err}")
                    else:
                        logging.exception('')
                    tnx.rollback()
                    continue

                # Execute task
                status = 0
                status_text = "OK"
                try:
                    self.on_task_execute(task_id, name)
                except Exception as err:
                    # If an error occurs during execution,
                    # then set the appropriate status code and save the error text.
                    status = 1
                    status_text = f"{err}"
                    # logging.error(status_text)

                # We save the task completion time and record its completion status.
                tnx.begin()
                try:
                    self.__on_finish_task(tnx, task_id, status, status_text)
                    tnx.commit()
                except DatabaseError:
                    if err.sqlstate == "40001":
                        conflict_counter = conflict_counter + 1
                        logging.error(f"Worker: {self.worker_id}, Task: {self.worker_id}, Error: {err}")
                    else:
                        logging.exception('')
                    tnx.rollback()

        return conflict_counter

def main():
    print(f"Start execute script. Works: {WORKS_COUNT}, workers: {WORKERS_COUNT}\n")

    with connect(DB_URI, user=DB_USER, password=DB_PASSWORD, charset=DB_CHARSET) as con:
        # Clean previous tasks from the queue
        con.begin()
        with con.cursor() as cur:
            cur.execute("DELETE FROM QUEUE_TASK")
        con.commit()
        # Task Manager sets 40 tasks
        con.begin()
        with con.cursor() as cur:
            cur.execute(
                """
                EXECUTE BLOCK (CNT INTEGER = ?)
                AS
                DECLARE I INTEGER;
                BEGIN
                  I = 0;
                  WHILE (I < CNT) DO
```

```
              BEGIN
                I = I + 1;
                INSERT INTO QUEUE_TASK(NAME)
                VALUES ('Task ' || :I);
              END
            END
            """,
            (WORKS_COUNT,)
        )
    con.commit()

# Let's create executors
workers = map(lambda worker_id: Worker(worker_id), range(WORKERS_COUNT))
with pool.ProcessPoolExecutor(max_workers=WORKERS_COUNT) as executer:
    features = map(lambda worker: executer.submit(worker.run), workers)
    conflicts = map(lambda feature: feature.result(), pool.as_completed(features))
    conflict_count = sum(conflicts)

# read statistics
with connect(DB_URI, user=DB_USER, password=DB_PASSWORD, charset=DB_CHARSET) as con:
    cur = con.cursor()
    cur.execute("""
        SELECT
          COUNT(*) AS CNT_TASK,
          COUNT(*) FILTER(WHERE STARTED IS TRUE AND FINISH_TIME IS NULL) AS CNT_ACTIVE_TASK,
          COUNT(*) FILTER(WHERE FINISH_TIME IS NOT NULL) AS CNT_FINISHED_TASK,
          COUNT(*) FILTER(WHERE FINISH_STATUS = 0) AS CNT_SUCCESS,
          COUNT(*) FILTER(WHERE FINISH_STATUS = 1) AS CNT_ERROR,
          AVG(DATEDIFF(MILLISECOND FROM START_TIME TO FINISH_TIME)) AS AVG_ELAPSED_TIME,
          DATEDIFF(MILLISECOND FROM MIN(START_TIME) TO MAX(FINISH_TIME)) AS SUM_ELAPSED_TIME,
          CAST(? AS BIGINT) AS CONFLICTS
        FROM QUEUE_TASK
    """, (conflict_count,))
    row = cur.fetchone()
    cur.close()

    stat_columns = ["TASKS", "ACTIVE_TASKS", "FINISHED_TASKS", "SUCCESS", "ERROR", "AVG_ELAPSED_TIME",
                    "SUM_ELAPSED_TIME", "CONFLICTS"]

    stat_table = PrettyTable(stat_columns)
    stat_table.add_row(row)
    print("\nStatistics:")
    print(stat_table)

    cur = con.cursor()
    cur.execute("""
        SELECT
          ID,
          NAME,
          STARTED,
          WORKER_ID,
          START_TIME,
          FINISH_TIME,
          FINISH_STATUS,
          STATUS_TEXT
        FROM QUEUE_TASK
    """)
    rows = cur.fetchall()
    cur.close()

    columns = ["ID", "NAME", "STARTED", "WORKER", "START_TIME", "FINISH_TIME",
               "STATUS", "STATUS_TEXT"]

    table = PrettyTable(columns)
    table.add_rows(rows)
```

```
        print("\nTasks:")
        print(table)

if __name__ == "__main__":
    main()
```

In this script, the task manager creates 40 tasks that must be completed by 4 executors. Each executor runs in its own thread. Based on the results of the script, task execution statistics are displayed, as well as the number of conflicts and the tasks themselves.

Let's run the script:

```
python ./queue_exec.py
```

```
Start execute script. Works: 40, workers: 4

ERROR:root:Worker: 2, Task: 2, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95695
ERROR:root:Worker: 2, Task: 2, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95697
ERROR:root:Worker: 2, Task: 2, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95703
ERROR:root:Worker: 2, Task: 2, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95706
ERROR:root:Worker: 0, Task: 0, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95713
ERROR:root:Worker: 2, Task: 2, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95722
ERROR:root:Worker: 3, Task: 3, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95722
ERROR:root:Worker: 1, Task: 1, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95722
ERROR:root:Worker: 1, Task: 1, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95728
ERROR:root:Worker: 0, Task: 0, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95734
ERROR:root:Worker: 0, Task: 0, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95736
ERROR:root:Worker: 1, Task: 1, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95741
ERROR:root:Worker: 1, Task: 1, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95744
ERROR:root:Worker: 0, Task: 0, Error: deadlock
-update conflicts with concurrent update
-concurrent transaction number is 95749
```

```
Statistics:
+-------+--------------+----------------+---------+-------+-----------------+-----------------+-----------+
| TASKS | ACTIVE_TASKS | FINISHED_TASKS | SUCCESS | ERROR | AVG_ELAPSED_TIME | SUM_ELAPSED_TIME | CONFLICTS |
+-------+--------------+----------------+---------+-------+-----------------+-----------------+-----------+
|   40  |      0       |       40       |    28   |   12  |       43.1       |       1353       |     14    |
+-------+--------------+----------------+---------+-------+-----------------+-----------------+-----------+

Tasks:
+------+---------+---------+--------+-------------------------+-------------------------+--------+-------------+
|  ID  |   NAME  | STARTED | WORKER |        START_TIME       |        FINISH_TIME      | STATUS | STATUS_TEXT |
+------+---------+---------+--------+-------------------------+-------------------------+--------+-------------+
| 1341 |  Task 1 |   True  |   0    | 2023-07-06 15:35:29.9800 | 2023-07-06 15:35:30.0320 |    1   |  Some error |
| 1342 |  Task 2 |   True  |   0    | 2023-07-06 15:35:30.0420 | 2023-07-06 15:35:30.0800 |    1   |  Some error |
| 1343 |  Task 3 |   True  |   0    | 2023-07-06 15:35:30.0900 | 2023-07-06 15:35:30.1130 |    0   |      OK     |
| 1344 |  Task 4 |   True  |   0    | 2023-07-06 15:35:30.1220 | 2023-07-06 15:35:30.1450 |    0   |      OK     |
...
```

From the results of the script execution it is clear that 4 executors are constantly conflicting over the task. The faster the task is completed and the more performers there are, the higher the likelihood of conflicts.

10.3. Clause SKIP LOCKED

How can we change our solution so that it works efficiently and without conflicts? Here comes the new clause `SKIP LOCKED` from Firebird 5.0.

Clause `SKIP LOCKED` allows you to skip already locked entries, thereby allowing you to work without conflicts. It can be used in queries where there is a possibility of an update conflict, that is, in `SELECT ⋯ WITH LOCK`, `UPDATE` and `DELETE` queries. Let's look at its syntax:

```
SELECT
  [FIRST ...]
  [SKIP ...]
  FROM <sometable>
  [WHERE ...]
  [PLAN ...]
  [ORDER BY ...]
  [{ ROWS ... } | {OFFSET ...} | {FETCH ...}]
  [FOR UPDATE [OF ...]]
  [WITH LOCK [SKIP LOCKED]]
```

```
UPDATE <sometable>
  SET ...
  [WHERE ...]
  [PLAN ...]
  [ORDER BY ...]
  [ROWS ...]
  [SKIP LOCKED]
  [RETURNING ...]
```

```
DELETE FROM <sometable>
  [WHERE ...]
```

```
  [PLAN ...]
  [ORDER BY ...]
  [ROWS ...]
  [SKIP LOCKED]
  [RETURNING ...]
```

10.4. Job queue without conflicts

Let's try to fix our script so that executors do not conflict over tasks.

To do this, we need to slightly rewrite the request in the `__next_task` method of the `Worker` class.

```
    @staticmethod
    def __next_task(tnx: TransactionManager):
        """Retrieves the next task from the queue.

        Arguments:
            tnx: The transaction in which the request is executed
        """
        cur = tnx.cursor()

        cur.execute("""
            SELECT ID, NAME
            FROM QUEUE_TASK
            WHERE STARTED IS FALSE
            ORDER BY ID
            FETCH FIRST ROW ONLY
            FOR UPDATE WITH LOCK SKIP LOCKED
        """)

        row = cur.fetchone()
        cur.close()
        return row
```

Let's run the script:

```
python ./queue_exec.py
```

```
Start execute script. Works: 40, workers: 4

Statistics:
+-------+--------------+----------------+---------+-------+------------------+------------------+-----------+
| TASKS | ACTIVE_TASKS | FINISHED_TASKS | SUCCESS | ERROR | AVG_ELAPSED_TIME | SUM_ELAPSED_TIME | CONFLICTS |
+-------+--------------+----------------+---------+-------+------------------+------------------+-----------+
|   40  |      0       |       40       |    32   |   8   |       39.1       |       1048       |     0     |
+-------+--------------+----------------+---------+-------+------------------+------------------+-----------+

Tasks:
+------+---------+---------+--------+-------------------------+-------------------------+--------+-------------+
|  ID  |   NAME  | STARTED | WORKER |         START_TIME      |        FINISH_TIME      | STATUS | STATUS_TEXT |
+------+---------+---------+--------+-------------------------+-------------------------+--------+-------------+
| 1381 |  Task 1 |   True  |   0    | 2023-07-06 15:57:22.0360 | 2023-07-06 15:57:22.0740 |   0    |      OK     |
```

```
| 1382 |  Task 2 |   True  |   0    | 2023-07-06 15:57:22.0840 | 2023-07-06 15:57:22.1130 |   0    |      OK     |
| 1383 |  Task 3 |   True  |   0    | 2023-07-06 15:57:22.1220 | 2023-07-06 15:57:22.1630 |   0    |      OK     |
| 1384 |  Task 4 |   True  |   0    | 2023-07-06 15:57:22.1720 | 2023-07-06 15:57:22.1910 |   0    |      OK     |
| 1385 |  Task 5 |   True  |   0    | 2023-07-06 15:57:22.2020 | 2023-07-06 15:57:22.2540 |   0    |      OK     |
| 1386 |  Task 6 |   True  |   0    | 2023-07-06 15:57:22.2620 | 2023-07-06 15:57:22.3220 |   0    |      OK     |
| 1387 |  Task 7 |   True  |   0    | 2023-07-06 15:57:22.3300 | 2023-07-06 15:57:22.3790 |   1    |  Some error |
...
```

This time there are no conflicts. Thus, in Firebird 5.0 you can use the `SKIP LOCKED` phrase to avoid unnecessary update conflicts.

10.5. Next steps

Our job queue could be improved even more. Let's look at the query execution plan

```
SELECT
  ID, NAME
FROM QUEUE_TASK
WHERE STARTED IS FALSE
ORDER BY ID
FETCH FIRST ROW ONLY
FOR UPDATE WITH LOCK SKIP LOCKED
```

```
Select Expression
    -> First N Records
        -> Write Lock
            -> Filter
                -> Table "QUEUE_TASK" Access By ID
                    -> Index "PK_QUEUE_TASK" Full Scan
```

This execution plan is not very good. A record from the `QUEUE_TASK` table is retrieved using index navigation, however, it reads the whole table with the complete index scan. If the `QUEUE_TASK` table is not cleared as we did in our script, then over time, the selection of unprocessed tasks will become slower and slower.

You can create an index on the `STARTED` field. If the task manager constantly adds new tasks, and the executors perform them, then the number of unstarted tasks is always less than the number of completed ones, thus this index will effectively filter tasks. Let's check it:

```
CREATE INDEX IDX_QUEUE_TASK_INACTIVE ON QUEUE_TASK(STARTED);

SELECT
  ID, NAME
FROM QUEUE_TASK
WHERE STARTED IS FALSE
ORDER BY ID
FETCH FIRST ROW ONLY
FOR UPDATE WITH LOCK SKIP LOCKED;
```

```
Select Expression
    -> First N Records
        -> Write Lock
            -> Filter
                -> Table "QUEUE_TASK" Access By ID
                    -> Index "PK_QUEUE_TASK" Full Scan
                        -> Bitmap
                            -> Index "IDX_QUEUE_TASK_INACTIVE" Range Scan (full match)
```

This is true, but now there are two indexes, one for filtering and one for navigation.

We can go further and create a composite index:

```
DROP INDEX IDX_QUEUE_TASK_INACTIVE;

CREATE INDEX IDX_QUEUE_TASK_INACTIVE ON QUEUE_TASK(STARTED, ID);
```

```
Select Expression
    -> First N Records
        -> Write Lock
            -> Filter
                -> Table "QUEUE_TASK" Access By ID
                    -> Index "IDX_QUEUE_TASK_INACTIVE" Range Scan (partial match: 1/2)
```

This will be more efficient since only one index is used for navigation, and it is partially scanned. However, such an index has a significant drawback: it will not be compact (and not be very fast).

To solve this problem, you can use another new feature from Firebird 5.0: partial indices.

> A partial index is an index that is built on a subset of table rows defined by a conditional expression (this is called a partial index predicate). Such an index contains entries only for rows satisfying the predicate.

Let's create partial index:

```
DROP INDEX IDX_QUEUE_TASK_INACTIVE;

CREATE INDEX IDX_QUEUE_TASK_INACTIVE ON QUEUE_TASK (STARTED, ID) WHERE (STARTED IS FALSE);

SELECT
  ID, NAME
FROM QUEUE_TASK
WHERE STARTED IS FALSE
ORDER BY STARTED, ID
FETCH FIRST ROW ONLY
FOR UPDATE WITH LOCK SKIP LOCKED
```

```
Select Expression
    -> First N Records
        -> Write Lock
            -> Filter
                -> Table "QUEUE_TASK" Access By ID
                    -> Index "IDX_QUEUE_TASK_INACTIVE" Full Scan
```

A record from the QUEUE_TASK table is retrieved by navigating the IDX_QUEUE_TASK_INACTIVE index. Despite, that the index scan is complete, the index itself is very compact, since it contains only the keys for which the condition STARTED IS FALSE is satisfied. There are always much fewer such entries in the normal task queue, than records with completed tasks.

10.6. Summary

In this material we demonstrated how to use the new SKIP LOCKED functionality that appeared in Firebird 5.0, and also have shown example of PARTIAL indices, which also appeared in Firebird 5.0.

A DDL script for creating a database, as well as a Python script with emulation of a task queue can be downloaded from the following links:

- ddl.sql
- queue_exec.py

Chapter 11. SQL and PSQL Profiling

One of the tasks of a database developer or administrator is to determine the causes of "slowdowns" in the information system.

Starting from Firebird 2.5, a powerful tracing tool has been added to their arsenal. Tracing is an indispensable tool for finding application bottlenecks, evaluating resources consumed during query execution, and determining the frequency of certain actions. Tracing shows statistics in the most detailed form (unlike the statistics available in ISQL, for example). The statistics do not take into account the costs of query preparation and data transmission over the network, which makes it "cleaner" than the data shown by ISQL. At the same time, tracing has a very insignificant impact on performance. Even with intensive logging, it usually results in no more than a 2-3% drop in the speed of executed queries.

After slow queries are "caught" by tracing, you can start optimizing them. However, such queries can be quite complex, and sometimes even call stored procedures, so a profiling tool is needed to help identify bottlenecks in the query itself or in the called PSQL module. Starting from Firebird 5.0, such a tool has appeared.

The profiler allows users to measure the performance costs of SQL and PSQL code. This is implemented using a system package in the engine that transmits data to the profiler plugin.

In this document, the engine and plugin are considered as a whole. Additionally, it is assumed that the default profiler plugin (Default_Profiler) is used.

The `RDB$PROFILER` package can profile the execution of PSQL code, collecting statistics on how many times each line was executed, as well as its minimum, maximum, and total execution time (accurate to nanoseconds), as well as statistics on opening and fetching records from implicit and explicit SQL cursors. In addition, you can get statistics on SQL cursors in terms of data sources (access methods) of the expanded query plan.

Although the execution time is measured with nanosecond accuracy, this result should not be trusted. The process of measuring execution time has certain overhead costs, which the profiler tries to compensate for. Accordingly, the measured time cannot be accurate, but it allows for a comparative analysis of the costs of executing individual sections of PSQL code among themselves, and identifying bottlenecks.

To collect profiling data, the user must first start a profiling session using the `RDB$PROFILER.START_SESSION` function. This function returns a profiling session identifier, which is later saved in the profiler snapshot tables. Later, you can execute queries to these tables for user analysis. A profiler session can be local (same connection) or remote (another connection).

Remote profiling simply redirects session control commands to the remote connection. Thus, it is possible for a client to profile multiple connections simultaneously. It is also possible that a locally or remotely started profiling session contains commands issued by another connection.

Remotely executed session control commands require the target connection to be in a waiting state, i.e., not executing other queries. When the target connection is not in waiting mode, the call is

blocked waiting for this state.

If the remote connection comes from another user, the calling user must have the PROFILE_ANY_ATTACHMENT system privilege.

After starting a session, statistics for PSQL and SQL statements are collected in memory. The profiling session only collects data about statements executed in the connection associated with the session. Data is aggregated and saved for each query (i.e., executed statement). When querying snapshot tables, the user can perform additional aggregation for each statement or use auxiliary views that do this automatically.

The session can be paused to temporarily disable statistics collection. Later, it can be resumed to return statistics collection in the same session.

To analyze the collected data, the user must flush the data to snapshot tables, which can be done by finishing or pausing the session (with the FLUSH parameter set to TRUE), or by calling RDB$PROFILER.FLUSH. Data is flushed using an autonomous transaction (a transaction starts and ends with the specific purpose of updating profiler data).

All procedures and functions of the RDB$PROFILER package contain an ATTACHMENT_ID parameter, which should be specified if you want to manage a remote profiling session. If this parameter is NULL or not specified, the procedures and functions manage the local profiling session.

11.1. Starting a Profiling Session

To start a profiling session, you need to call the RDB$PROFILER.START_SESSION function, which returns the profiling session identifier.

This function has the following parameters:

- DESCRIPTION of type VARCHAR(255) CHARACTER SET UTF8, default is NULL;
- FLUSH_INTERVAL of type INTEGER, default is NULL;
- ATTACHMENT_ID of type BIGINT, default is NULL (which means CURRENT_CONNECTION);
- PLUGIN_NAME of type VARCHAR(255) CHARACTER SET UTF8, default is NULL;
- PLUGIN_OPTIONS of type VARCHAR(255) CHARACTER SET UTF8, default is NULL.

You can pass an arbitrary text description of the session to the DESCRIPTION parameter.

If the FLUSH_INTERVAL parameter is not NULL, it sets the interval for automatic flushing of statistics to snapshot tables, as when manually calling the RDB$PROFILER.SET_FLUSH_INTERVAL procedure. If FLUSH_INTERVAL is greater than zero, automatic statistics flushing is enabled; otherwise, it is disabled. The FLUSH_INTERVAL parameter is measured in seconds.

If ATTACHMENT_ID is not NULL, the profiling session is started for a remote connection; otherwise, the session starts for the current connection.

The PLUGIN_NAME parameter is used to specify which profiling plugin is used for the profiling session. If it is NULL, the profiling plugin specified in the DefaultProfilerPlugin configuration parameter is

used.

Each profiling plugin may have its own options, which can be passed to the PLUGIN_OPTIONS parameter. For the Default_Profiler plugin included in the standard Firebird 5.0 distribution, the following values are allowed: NULL or 'DETAILED_REQUESTS'.

When DETAILED_REQUESTS is used, the PLG$PROF_REQUESTS table will store detailed query data, i.e., one record for each SQL statement call. This can result in the creation of a large number of records, which will lead to slow operation of RDB$PROFILER.FLUSH.

When DETAILED_REQUESTS is not used (default), the PLG$PROF_REQUESTS table saves an aggregated record for each SQL statement, using REQUEST_ID = 0.

Here, an SQL statement refers to a prepared SQL query stored in the prepared query cache. Queries are considered the same if they match exactly, character by character. So if you have semantically identical queries that differ in comments, they are different queries for the prepared query cache. Prepared queries can be executed multiple times with different sets of input parameters.

11.2. Pausing a Profiling Session

The RDB$PROFILER.PAUSE_SESSION procedure pauses the current profiler session (with the given ATTACHMENT_ID). For a paused session, execution statistics for subsequent SQL statements are not collected.

If the FLUSH parameter is TRUE, the snapshot tables are updated with profiling data up to the current moment; otherwise, the data remains only in memory for subsequent updating.

Calling RDB$PROFILER.PAUSE_SESSION(TRUE) has the same effect as calling RDB$PROFILER.PAUSE_SESSION(FALSE) followed by a call to RDB$PROFILER.FLUSH (using the same ATTACHMENT_ID).

Input parameters:

- FLUSH of type BOOLEAN NOT NULL, default is FALSE;
- ATTACHMENT_ID of type BIGINT, default is NULL (which means CURRENT_CONNECTION).

11.3. Resuming a Profiling Session

The RDB$PROFILER.RESUME_SESSION procedure resumes the current profiler session (with the given ATTACHMENT_ID) if it was paused. After resuming the session, execution statistics for subsequent SQL statements are collected again.

Input parameters:

- ATTACHMENT_ID of type BIGINT, default is NULL (which means CURRENT_CONNECTION).

11.4. Finishing a Profiling Session

The RDB$PROFILER.FINISH_SESSION procedure finishes the current profiler session (with the given ATTACHMENT_ID).

If the FLUSH parameter is TRUE, the snapshot tables are updated with data from the finished session (and old finished sessions not yet present in the snapshot); otherwise, the data remains only in memory for subsequent updating.

Calling RDB$PROFILER.FINISH_SESSION(TRUE) has the same effect as calling RDB$PROFILER.FINISH_SESSION(FALSE) followed by a call to RDB$PROFILER.FLUSH (using the same ATTACHMENT_ID).

Input parameters:

- FLUSH of type BOOLEAN NOT NULL, default is TRUE;
- ATTACHMENT_ID of type BIGINT, default is NULL (which means CURRENT_CONNECTION).

11.5. Canceling a Profiling Session

The RDB$PROFILER.PAUSE_SESSION procedure pauses the current profiler session (with the given ATTACHMENT_ID). For a paused session, execution statistics for subsequent SQL statements are not collected.

If the FLUSH parameter is TRUE, the snapshot tables are updated with profiling data up to the current moment; otherwise, the data remains only in memory for subsequent updating.

Calling RDB$PROFILER.PAUSE_SESSION(TRUE) has the same effect as calling RDB$PROFILER.PAUSE_SESSION(FALSE) followed by a call to RDB$PROFILER.FLUSH (using the same ATTACHMENT_ID).

Input parameters:

- FLUSH of type BOOLEAN NOT NULL, default is FALSE;
- ATTACHMENT_ID of type BIGINT, default is NULL (which means CURRENT_CONNECTION).

11.6. Resuming a Profiling Session

The RDB$PROFILER.RESUME_SESSION procedure resumes the current profiler session (with the given ATTACHMENT_ID) if it was paused. After resuming the session, execution statistics for subsequent SQL statements are collected again.

Input parameters:

- ATTACHMENT_ID of type BIGINT, default is NULL (which means CURRENT_CONNECTION).

11.7. Finishing a Profiling Session

The RDB$PROFILER.FINISH_SESSION procedure finishes the current profiler session (with the given ATTACHMENT_ID).

If the FLUSH parameter is TRUE, the snapshot tables are updated with data from the finished session (and old finished sessions not yet present in the snapshot); otherwise, the data remains only in memory for subsequent updating.

Calling RDB$PROFILER.FINISH_SESSION(TRUE) has the same effect as calling RDB$PROFILER.FINISH_SESSION(FALSE) followed by a call to RDB$PROFILER.FLUSH (using the same ATTACHMENT_ID).

Input parameters:

- FLUSH of type BOOLEAN NOT NULL, default is TRUE;
- ATTACHMENT_ID of type BIGINT, default is NULL (which means CURRENT_CONNECTION).

11.8. Canceling a Profiling Session

The RDB$PROFILER.CANCEL_SESSION procedure cancels the current profiling session (with the given ATTACHMENT_ID).

All session data present in the profiler plugin's memory is destroyed and not flushed to the snapshot tables.

Already flushed data is not automatically deleted.

Input parameters:

- ATTACHMENT_ID of type BIGINT, default is NULL (which means CURRENT_CONNECTION).

11.9. Discarding Profiling Sessions

The RDB$PROFILER.DISCARD procedure removes all sessions (with the given ATTACHMENT_ID) from memory without flushing them to the snapshot tables.

If there is an active profiling session, it is canceled.

Input parameters:

- ATTACHMENT_ID of type BIGINT, default is NULL (which means CURRENT_CONNECTION).

11.10. Flushing Profiling Session Statistics to Snapshot Tables

The RDB$PROFILER.FLUSH procedure updates the snapshot tables with data from profiling sessions (with the given ATTACHMENT_ID).

After flushing the statistics, the data is saved in the tables PLG$PROF_SESSIONS, PLG$PROF_STATEMENTS, PLG$PROF_RECORD_SOURCES, PLG$PROF_REQUESTS, PLG$PROF_PSQL_STATS, and PLG$PROF_RECORD_SOURCE_STATS and can be read and analyzed by the user.

Data is updated using an autonomous transaction, so if the procedure is called in a snapshot transaction, the data will not be available for immediate reading in the same transaction.

After flushing the statistics, completed profiling sessions are removed from memory.

Input parameters:

- ATTACHMENT_ID of type BIGINT, default is NULL (which means CURRENT_CONNECTION).

11.11. Setting the Statistics Flush Interval

The RDB$PROFILER.SET_FLUSH_INTERVAL procedure enables periodic automatic flushing of statistics (when FLUSH_INTERVAL is greater than 0) or disables it (when FLUSH_INTERVAL is 0).

The FLUSH_INTERVAL parameter is interpreted as the number of seconds.

Input parameters:

- FLUSH_INTERVAL of type INTEGER NOT NULL;
- ATTACHMENT_ID of type BIGINT, default is NULL (which means CURRENT_CONNECTION).

11.12. Snapshot Tables

Snapshot tables (as well as views and sequences) are created automatically when the profiler is first used. They belong to the database owner with read/write permissions for PUBLIC.

When a profiling session is deleted, the associated data in other profiler snapshot tables is automatically deleted using foreign keys with the DELETE CASCADE option.

Below is a list of tables that store profiling data.

11.12.1. Table PLG$PROF_SESSIONS

The PLG$PROF_SESSIONS table contains information about profiling sessions.

Table 4. Description of columns in the PLG$PROF_SESSIONS table

Column Name	Data Type	Description
PROFILE_ID	BIGINT	Profiling session identifier.
ATTACHMENT_ID	BIGINT	Connection identifier for which the profiling session was started.
USER_NAME	CHAR(63)	Name of the user who started the profiling session.

Column Name	Data Type	Description
DESCRIPTION	VARCHAR(255)	Description passed in the DESCRIPTION parameter when calling RDB$PROFILER.START_SESSION.
START_TIMESTAMP	TIMESTAMP WITH TIME ZONE	Start time of the profiling session.
FINISH_TIMESTAMP	TIMESTAMP WITH TIME ZONE	End time of the profiling session (NULL if the session is not finished).

Primary key: PROFILE_ID.

11.12.2. Table PLG$PROF_STATEMENTS

The PLG$PROF_STATEMENTS table contains information about statements.

Table 5. Description of columns in the PLG$PROF_STATEMENTS *table*

Column Name	Data Type	Description
PROFILE_ID	BIGINT	Profiling session identifier.
STATEMENT_ID	BIGINT	Statement identifier.
PARENT_STATEMENT_ID	BIGINT	Parent statement identifier. Relates to subprograms.
STATEMENT_TYPE	VARCHAR(20)	Statement type: BLOCK, FUNCTION, PROCEDURE, or TRIGGER.
PACKAGE_NAME	CHAR(63)	Package name.
ROUTINE_NAME	CHAR(63)	Function, procedure, or trigger name.
SQL_TEXT	BLOB SUB_TYPE TEXT	SQL text for BLOCK type statements.

Primary key: PROFILE_ID, STATEMENT_ID.

11.12.3. Table PLG$PROF_REQUESTS

The PLG$PROF_REQUESTS table contains statistics on SQL query execution.

If the profiler is started with the DETAILED_REQUESTS option, the PLG$PROF_REQUESTS table will store detailed query data, i.e., one record for each statement call. This can result in the creation of a large number of records, which will lead to slow operation of RDB$PROFILER.FLUSH.

When DETAILED_REQUESTS is not used (default), the PLG$PROF_REQUESTS table saves an aggregated record for each statement, using REQUEST_ID = 0.

Table 6. Description of columns in the PLG$PROF_REQUESTS *table*

Column Name	Data Type	Description
PROFILE_ID	BIGINT	Profiling session identifier.

Column Name	Data Type	Description
STATEMENT_ID	BIGINT	Statement identifier.
REQUEST_ID	BIGINT	Request identifier.
CALLER_STATEMENT_ID	BIGINT	Caller statement identifier.
CALLER_REQUEST_ID	BIGINT	Caller request identifier.
START_TIMESTAMP	TIMESTAMP WITH TIME ZONE	Request start time.
FINISH_TIMESTAMP	TIMESTAMP WITH TIME ZONE	Request finish time.
TOTAL_ELAPSED_TIME	BIGINT	Accumulated request execution time (in nanoseconds).

Primary key: PROFILE_ID, STATEMENT_ID, REQUEST_ID.

11.12.4. Table PLG$PROF_CURSORS

The PLG$PROF_CURSORS table contains information about cursors.

Table 7. Description of columns in the PLG$PROF_CURSORS *table*

Column Name	Data Type	Description
PROFILE_ID	BIGINT	Profiling session identifier.
STATEMENT_ID	BIGINT	Statement identifier.
CURSOR_ID	BIGINT	Cursor identifier.
NAME	CHAR(63)	Name of the explicitly declared cursor.
LINE_NUM	INTEGER	PSQL line number where the cursor is defined.
COLUMN_NUM	INTEGER	PSQL column number where the cursor is defined.

Primary key: PROFILE_ID, STATEMENT_ID, CURSOR_ID.

11.12.5. Table PLG$PROF_RECORD_SOURCES

The PLG$PROF_RECORD_SOURCES table contains information about data sources.

Table 8. Description of columns in the PLG$PROF_RECORD_SOURCES *table*

Column Name	Data Type	Description
PROFILE_ID	BIGINT	Profiling session identifier.
STATEMENT_ID	BIGINT	Statement identifier.
CURSOR_ID	BIGINT	Cursor identifier.
RECORD_SOURCE_ID	BIGINT	Data source identifier.

Column Name	Data Type	Description
PARENT_RECORD_SOURCE_ID	BIGINT	Parent data source identifier.
LEVEL	INTEGER	Indent level for the data source. Necessary when constructing a detailed plan.
ACCESS_PATH	BLOB SUB_TYPE TEXT	Description of the access method for the data source.

Primary key: PROFILE_ID, STATEMENT_ID, CURSOR_ID, RECORD_SOURCE_ID.

11.12.6. Table PLG$PROF_RECORD_SOURCE_STATS

The PLG$PROF_RECORD_SOURCES table contains statistics on data sources.

Table 9. Description of columns in the PLG$PROF_RECORD_SOURCE_STATS *table*

Column Name	Data Type	Description
PROFILE_ID	BIGINT	Profiling session identifier.
STATEMENT_ID	BIGINT	Statement identifier.
REQUEST_ID	BIGINT	Request identifier.
CURSOR_ID	BIGINT	Cursor identifier.
RECORD_SOURCE_ID	BIGINT	Data source identifier.
OPEN_COUNTER	BIGINT	Number of times the data source was opened.
OPEN_MIN_ELAPSED_TIME	BIGINT	Minimum time to open the data source (in nanoseconds).
OPEN_MAX_ELAPSED_TIME	BIGINT	Maximum time to open the data source (in nanoseconds).
OPEN_TOTAL_ELAPSED_TIME	BIGINT	Accumulated time to open the data source (in nanoseconds).
FETCH_COUNTER	BIGINT	Number of fetches from the data source.
FETCH_MIN_ELAPSED_TIME	BIGINT	Minimum time to fetch a record from the data source (in nanoseconds).
FETCH_MAX_ELAPSED_TIME	BIGINT	Maximum time to fetch a record from the data source (in nanoseconds).
FETCH_TOTAL_ELAPSED_TIME	BIGINT	Accumulated time to fetch records from the data source (in nanoseconds).

Primary key: PROFILE_ID, STATEMENT_ID, REQUEST_ID, CURSOR_ID, RECORD_SOURCE_ID.

11.12.7. Table PLG$PROF_PSQL_STATS

The PLG$PROF_PSQL_STATS table contains PSQL statistics.

Table 10. Description of columns in the PLG$PROF_PSQL_STATS *table*

Column Name	Data Type	Description
PROFILE_ID	BIGINT	Profiling session identifier.
STATEMENT_ID	BIGINT	Statement identifier.
REQUEST_ID	BIGINT	Request identifier.
LINE_NUM	INTEGER	PSQL line number for the statement.
COLUMN_NUM	INTEGER	PSQL column number for the statement.
COUNTER	BIGINT	Number of executions for the line/column number.
MIN_ELAPSED_TIME	BIGINT	Minimum execution time (in nanoseconds) for the line/column.
MAX_ELAPSED_TIME	BIGINT	Maximum execution time (in nanoseconds) for the line/column.
TOTAL_ELAPSED_TIME	BIGINT	Accumulated execution time (in nanoseconds) for the line/column.

Primary key: PROFILE_ID, STATEMENT_ID, REQUEST_ID, LINE_NUM, COLUMN_NUM.

11.13. Auxiliary Views

In addition to snapshot tables, the profiling plugin creates auxiliary views. These views help extract profiling data aggregated at the statement level.

Auxiliary views are the preferred way to analyze profiling data for quickly finding "hot spots". They can also be used in conjunction with snapshot tables. Once "hot spots" are found, you can drill down into the data at the query level using the tables.

Below is a list of views for the Default_Profiler.

PLG$PROF_PSQL_STATS_VIEW

aggregated PSQL statistics in a profiling session.

PLG$PROF_RECORD_SOURCE_STATS_VIEW

aggregated statistics on data sources in a profiling session.

PLG$PROF_STATEMENT_STATS_VIEW

aggregated statistics of SQL statements in a profiling session.

In this document, I will not provide the source code for these views and a description of their columns; you can always view the text of these views yourself. A description of the columns can be found in the "Firebird 5.0 SQL Language Reference".

11.14. Profiler Launch Modes

Before we move on to real-world examples of using the profiler, I'll demonstrate the difference between the different modes of running the profiling plugin.

11.14.1. Option DETAILED_REQUESTS

The profiling statistics also include the `SELECT RDB$PROFILER.START_SESSION() …` query and the `RDB$PROFILER.START_SESSION` function startup statistics.

In order to limit the statistics output, I add a filter by the query text, in this case I output statistics only for those queries that contain 'FROM HORSE'.

```
SELECT RDB$PROFILER.START_SESSION('Profile without "DETAILED_REQUESTS"')
FROM RDB$DATABASE;

SELECT COUNT(*) FROM HORSE;

SELECT COUNT(*) FROM HORSE;

SELECT RDB$PROFILER.START_SESSION('Profile with "DETAILED_REQUESTS"',
  NULL, NULL, NULL, 'DETAILED_REQUESTS')
FROM RDB$DATABASE;

SELECT COUNT(*) FROM HORSE;

SELECT COUNT(*) FROM HORSE;

EXECUTE PROCEDURE RDB$PROFILER.FINISH_SESSION;

COMMIT;

SELECT
  S.DESCRIPTION,
  V.*
FROM PLG$PROF_STATEMENT_STATS_VIEW V
JOIN PLG$PROF_SESSIONS S ON S.PROFILE_ID = V.PROFILE_ID
WHERE V.SQL_TEXT CONTAINING 'FROM HORSE';
```

```
DESCRIPTION                     Profile without "DETAILED_REQUESTS"
PROFILE_ID                      12
STATEMENT_ID                    2149
STATEMENT_TYPE                  BLOCK
PACKAGE_NAME                    <null>
ROUTINE_NAME                    <null>
PARENT_STATEMENT_ID             <null>
PARENT_STATEMENT_TYPE           <null>
PARENT_ROUTINE_NAME             <null>
SQL_TEXT                        13e:9
SELECT COUNT(*) FROM HORSE
COUNTER                         1
```

```
MIN_ELAPSED_TIME                <null>
MAX_ELAPSED_TIME                <null>
TOTAL_ELAPSED_TIME              <null>
AVG_ELAPSED_TIME                <null>

DESCRIPTION                     Profile with "DETAILED_REQUESTS"
PROFILE_ID                      13
STATEMENT_ID                    2149
STATEMENT_TYPE                  BLOCK
PACKAGE_NAME                    <null>
ROUTINE_NAME                    <null>
PARENT_STATEMENT_ID             <null>
PARENT_STATEMENT_TYPE           <null>
PARENT_ROUTINE_NAME             <null>
SQL_TEXT                        13e:b
SELECT COUNT(*) FROM HORSE
COUNTER                         2
MIN_ELAPSED_TIME                165498198
MAX_ELAPSED_TIME                235246029
TOTAL_ELAPSED_TIME              400744227
AVG_ELAPSED_TIME                200372113
```

As you can see, if we run a profiler session without the 'DETAILED_REQUESTS' option, the view gives us less detail. At a minimum, there are no query execution statistics, and it appears as if the query was executed once. Let's try to detail this data using a query to the snapshot tables.

First, let's look at the session details without the 'DETAILED_REQUESTS' option.

```
SELECT
  S.PROFILE_ID,
  S.DESCRIPTION,
  R.REQUEST_ID,
  STMT.STATEMENT_ID,
  STMT.STATEMENT_TYPE,
  STMT.PACKAGE_NAME,
  STMT.ROUTINE_NAME,
  STMT.SQL_TEXT,
  R.CALLER_STATEMENT_ID,
  R.CALLER_REQUEST_ID,
  R.START_TIMESTAMP,
  R.FINISH_TIMESTAMP,
  R.TOTAL_ELAPSED_TIME
FROM PLG$PROF_SESSIONS S
JOIN PLG$PROF_STATEMENTS STMT ON STMT.PROFILE_ID = S.PROFILE_ID
JOIN PLG$PROF_REQUESTS R ON R.PROFILE_ID = S.PROFILE_ID AND R.STATEMENT_ID = STMT.STATEMENT_ID
WHERE S.PROFILE_ID = 12
  AND STMT.SQL_TEXT CONTAINING 'FROM HORSE';
```

```
PROFILE_ID                      12
DESCRIPTION                     Profile without "DETAILED_REQUESTS"
REQUEST_ID                      0
STATEMENT_ID                    2149
STATEMENT_TYPE                  BLOCK
```

```
PACKAGE_NAME                    <null>
ROUTINE_NAME                    <null>
SQL_TEXT                        13e:9
SELECT COUNT(*) FROM HORSE
CALLER_STATEMENT_ID             <null>
CALLER_REQUEST_ID               <null>
START_TIMESTAMP                 2023-11-09 15:48:59.2250
FINISH_TIMESTAMP                <null>
TOTAL_ELAPSED_TIME              <null>
```

Now let's compare it with a session with the 'DETAILED_REQUESTS' option.

```
SELECT
  S.PROFILE_ID,
  S.DESCRIPTION,
  R.REQUEST_ID,
  STMT.STATEMENT_ID,
  STMT.STATEMENT_TYPE,
  STMT.PACKAGE_NAME,
  STMT.ROUTINE_NAME,
  STMT.SQL_TEXT,
  R.CALLER_STATEMENT_ID,
  R.CALLER_REQUEST_ID,
  R.START_TIMESTAMP,
  R.FINISH_TIMESTAMP,
  R.TOTAL_ELAPSED_TIME
FROM PLG$PROF_SESSIONS S
JOIN PLG$PROF_STATEMENTS STMT ON STMT.PROFILE_ID = S.PROFILE_ID
JOIN PLG$PROF_REQUESTS R ON R.PROFILE_ID = S.PROFILE_ID AND R.STATEMENT_ID = STMT.STATEMENT_ID
WHERE S.PROFILE_ID = 13
  AND STMT.SQL_TEXT CONTAINING 'FROM HORSE';
```

```
PROFILE_ID                      13
DESCRIPTION                     Profile with "DETAILED_REQUESTS"
REQUEST_ID                      2474
STATEMENT_ID                    2149
STATEMENT_TYPE                  BLOCK
PACKAGE_NAME                    <null>
ROUTINE_NAME                    <null>
SQL_TEXT                        13e:b
SELECT COUNT(*) FROM HORSE
CALLER_STATEMENT_ID             <null>
CALLER_REQUEST_ID               <null>
START_TIMESTAMP                 2023-11-09 15:49:01.6540
FINISH_TIMESTAMP                2023-11-09 15:49:02.8360
TOTAL_ELAPSED_TIME              165498198

PROFILE_ID                      13
DESCRIPTION                     Profile with "DETAILED_REQUESTS"
REQUEST_ID                      2475
STATEMENT_ID                    2149
STATEMENT_TYPE                  BLOCK
PACKAGE_NAME                    <null>
```

```
ROUTINE_NAME                    <null>
SQL_TEXT                        13e:b
SELECT COUNT(*) FROM HORSE
CALLER_STATEMENT_ID             <null>
CALLER_REQUEST_ID               <null>
START_TIMESTAMP                 2023-11-09 15:49:02.8470
FINISH_TIMESTAMP                2023-11-09 15:49:04.0980
TOTAL_ELAPSED_TIME              235246029
```

Thus, if the profiler is run without the 'DETAILED_REQUESTS' option, all runs of the same SQL statement will appear as a single run, in which statistics are accumulated. Directly in the PLG$PROF_REQUESTS table, statistics are not taken into account at all without the 'DETAILED_REQUESTS' option, but they are aggregated in other tables.

```
SELECT
  R.PROFILE_ID,
  R.STATEMENT_ID,
  RS.CURSOR_ID,
  RS.RECORD_SOURCE_ID,
  RS.PARENT_RECORD_SOURCE_ID,
  RS."LEVEL",
  RS.ACCESS_PATH,
  RSS.OPEN_COUNTER,
  RSS.OPEN_MIN_ELAPSED_TIME,
  RSS.OPEN_MAX_ELAPSED_TIME,
  RSS.OPEN_TOTAL_ELAPSED_TIME,
  RSS.FETCH_COUNTER,
  RSS.FETCH_MIN_ELAPSED_TIME,
  RSS.FETCH_MAX_ELAPSED_TIME,
  RSS.FETCH_TOTAL_ELAPSED_TIME
FROM
  PLG$PROF_REQUESTS R
  JOIN PLG$PROF_RECORD_SOURCES RS
    ON RS.PROFILE_ID = R.PROFILE_ID AND
       RS.STATEMENT_ID = R.STATEMENT_ID
  JOIN PLG$PROF_RECORD_SOURCE_STATS RSS
    ON RSS.PROFILE_ID = R.PROFILE_ID AND
       RSS.STATEMENT_ID = R.STATEMENT_ID AND
       RSS.REQUEST_ID = R.REQUEST_ID AND
       RSS.CURSOR_ID = RS.CURSOR_ID AND
       RSS.RECORD_SOURCE_ID = RS.RECORD_SOURCE_ID
WHERE R.PROFILE_ID = 12
  AND R.STATEMENT_ID = 2149
ORDER BY RSS.REQUEST_ID, RSS.RECORD_SOURCE_ID
```

```
PROFILE_ID                      12
STATEMENT_ID                    2149
CURSOR_ID                       1
RECORD_SOURCE_ID                1
PARENT_RECORD_SOURCE_ID         <null>
LEVEL                           0
ACCESS_PATH                     140:f
```

```
Select Expression
OPEN_COUNTER                    2
OPEN_MIN_ELAPSED_TIME           10266
OPEN_MAX_ELAPSED_TIME           10755
OPEN_TOTAL_ELAPSED_TIME         21022
FETCH_COUNTER                   4
FETCH_MIN_ELAPSED_TIME          0
FETCH_MAX_ELAPSED_TIME          191538868
FETCH_TOTAL_ELAPSED_TIME        356557956

PROFILE_ID                      12
STATEMENT_ID                    2149
CURSOR_ID                       1
RECORD_SOURCE_ID                2
PARENT_RECORD_SOURCE_ID         1
LEVEL                           1
ACCESS_PATH                     140:10
-> Aggregate
OPEN_COUNTER                    2
OPEN_MIN_ELAPSED_TIME           9777
OPEN_MAX_ELAPSED_TIME           9777
OPEN_TOTAL_ELAPSED_TIME         19555
FETCH_COUNTER                   4
FETCH_MIN_ELAPSED_TIME          0
FETCH_MAX_ELAPSED_TIME          191538379
FETCH_TOTAL_ELAPSED_TIME        356556489

PROFILE_ID                      12
STATEMENT_ID                    2149
CURSOR_ID                       1
RECORD_SOURCE_ID                3
PARENT_RECORD_SOURCE_ID         2
LEVEL                           2
ACCESS_PATH                     140:11
-> Table "HORSE" Full Scan
OPEN_COUNTER                    2
OPEN_MIN_ELAPSED_TIME           2444
OPEN_MAX_ELAPSED_TIME           3911
OPEN_TOTAL_ELAPSED_TIME         6355
FETCH_COUNTER                   1039248
FETCH_MIN_ELAPSED_TIME          0
FETCH_MAX_ELAPSED_TIME          905422
FETCH_TOTAL_ELAPSED_TIME        330562264
```

Here you can see that all statistics are "doubled" because the request was run twice. Now let’s look at the statistics with 'DETAILED_REQUESTS'.

```
SELECT
  R.PROFILE_ID,
  R.STATEMENT_ID,
  RS.CURSOR_ID,
  RS.RECORD_SOURCE_ID,
  RS.PARENT_RECORD_SOURCE_ID,
  RS."LEVEL",
```

```
  RS.ACCESS_PATH,
  RSS.OPEN_COUNTER,
  RSS.OPEN_MIN_ELAPSED_TIME,
  RSS.OPEN_MAX_ELAPSED_TIME,
  RSS.OPEN_TOTAL_ELAPSED_TIME,
  RSS.FETCH_COUNTER,
  RSS.FETCH_MIN_ELAPSED_TIME,
  RSS.FETCH_MAX_ELAPSED_TIME,
  RSS.FETCH_TOTAL_ELAPSED_TIME
FROM
  PLG$PROF_REQUESTS R
  JOIN PLG$PROF_RECORD_SOURCES RS
    ON RS.PROFILE_ID = R.PROFILE_ID AND
       RS.STATEMENT_ID = R.STATEMENT_ID
  JOIN PLG$PROF_RECORD_SOURCE_STATS RSS
    ON RSS.PROFILE_ID = R.PROFILE_ID AND
       RSS.STATEMENT_ID = R.STATEMENT_ID AND
       RSS.REQUEST_ID = R.REQUEST_ID AND
       RSS.CURSOR_ID = RS.CURSOR_ID AND
       RSS.RECORD_SOURCE_ID = RS.RECORD_SOURCE_ID
WHERE R.PROFILE_ID = 13
  AND R.STATEMENT_ID = 2149
ORDER BY RSS.REQUEST_ID, RSS.RECORD_SOURCE_ID
```

```
PROFILE_ID                    13
STATEMENT_ID                  2149
CURSOR_ID                     1
RECORD_SOURCE_ID              1
PARENT_RECORD_SOURCE_ID       <null>
LEVEL                         0
ACCESS_PATH                   140:14
Select Expression
OPEN_COUNTER                  1
OPEN_MIN_ELAPSED_TIME         20044
OPEN_MAX_ELAPSED_TIME         20044
OPEN_TOTAL_ELAPSED_TIME       20044
FETCH_COUNTER                 2
FETCH_MIN_ELAPSED_TIME        0
FETCH_MAX_ELAPSED_TIME        165438065
FETCH_TOTAL_ELAPSED_TIME      165438065

PROFILE_ID                    13
STATEMENT_ID                  2149
CURSOR_ID                     1
RECORD_SOURCE_ID              2
PARENT_RECORD_SOURCE_ID       1
LEVEL                         1
ACCESS_PATH                   140:15
-> Aggregate
OPEN_COUNTER                  1
OPEN_MIN_ELAPSED_TIME         19066
OPEN_MAX_ELAPSED_TIME         19066
OPEN_TOTAL_ELAPSED_TIME       19066
FETCH_COUNTER                 2
FETCH_MIN_ELAPSED_TIME        0
```

```
FETCH_MAX_ELAPSED_TIME          165437576
FETCH_TOTAL_ELAPSED_TIME        165437576

PROFILE_ID                      13
STATEMENT_ID                    2149
CURSOR_ID                       1
RECORD_SOURCE_ID                3
PARENT_RECORD_SOURCE_ID         2
LEVEL                           2
ACCESS_PATH                     140:16
-> Table "HORSE" Full Scan
OPEN_COUNTER                    1
OPEN_MIN_ELAPSED_TIME           2444
OPEN_MAX_ELAPSED_TIME           2444
OPEN_TOTAL_ELAPSED_TIME         2444
FETCH_COUNTER                   519624
FETCH_MIN_ELAPSED_TIME          0
FETCH_MAX_ELAPSED_TIME          892222
FETCH_TOTAL_ELAPSED_TIME        161990420

PROFILE_ID                      13
STATEMENT_ID                    2149
CURSOR_ID                       1
RECORD_SOURCE_ID                1
PARENT_RECORD_SOURCE_ID         <null>
LEVEL                           0
ACCESS_PATH                     140:14
Select Expression
OPEN_COUNTER                    1
OPEN_MIN_ELAPSED_TIME           12711
OPEN_MAX_ELAPSED_TIME           12711
OPEN_TOTAL_ELAPSED_TIME         12711
FETCH_COUNTER                   2
FETCH_MIN_ELAPSED_TIME          488
FETCH_MAX_ELAPSED_TIME          235217674
FETCH_TOTAL_ELAPSED_TIME        235218163

PROFILE_ID                      13
STATEMENT_ID                    2149
CURSOR_ID                       1
RECORD_SOURCE_ID                2
PARENT_RECORD_SOURCE_ID         1
LEVEL                           1
ACCESS_PATH                     140:15
-> Aggregate
OPEN_COUNTER                    1
OPEN_MIN_ELAPSED_TIME           11244
OPEN_MAX_ELAPSED_TIME           11244
OPEN_TOTAL_ELAPSED_TIME         11244
FETCH_COUNTER                   2
FETCH_MIN_ELAPSED_TIME          0
FETCH_MAX_ELAPSED_TIME          235217674
FETCH_TOTAL_ELAPSED_TIME        235217674

PROFILE_ID                      13
STATEMENT_ID                    2149
```

```
CURSOR_ID                       1
RECORD_SOURCE_ID                3
PARENT_RECORD_SOURCE_ID         2
LEVEL                           2
ACCESS_PATH                     140:16
-> Table "HORSE" Full Scan
OPEN_COUNTER                    1
OPEN_MIN_ELAPSED_TIME           2444
OPEN_MAX_ELAPSED_TIME           2444
OPEN_TOTAL_ELAPSED_TIME         2444
FETCH_COUNTER                   519624
FETCH_MIN_ELAPSED_TIME          0
FETCH_MAX_ELAPSED_TIME          675155
FETCH_TOTAL_ELAPSED_TIME        196082602
```

For a session with the 'DETAILED_REQUESTS' option, we see that statistics are collected separately for each SQL statement run.

The 'DETAILED_REQUESTS' option creates one record in the PLG$PROF_REQUESTS table not only for each top-level SQL query, but also for each stored procedure or function call. Thus, if you have a PSQL function called on some field in the SELECT clause, then PLG$PROF_REQUESTS will have as many records with the call to this function as there were records snapped. This can significantly slow down the reset of profiling statistics. Therefore, you should not use the 'DETAILED_REQUESTS' option for any profiling session. To find "bottlenecks" in a particular query, it will be sufficient to leave the PLUGIN_OPTIONS parameter at the default value.

11.14.2. Running the profiler in a remote connection

To start a profiling session on a remote connection, you need to know the ID of that connection. You can do this using the MON$ATTACHMENTS monitoring table or by running a query with the CURRENT_CONNECTION context variable in the remote session, and set this ID as the value of the ATTACHMENT_ID parameter of the RDB$PROFILER.START_SESSION function.

Requesting connection ID in session 1

```
select current_connection from rdb$database;
```

```
   CURRENT_CONNECTION
======================
                    29
```

Starting a profiling session on a remote computer in session 2

```
SELECT RDB$PROFILER.START_SESSION('Profile with "DETAILED_REQUESTS"',
  NULL, 29, NULL, 'DETAILED_REQUESTS')
FROM RDB$DATABASE;
```

The request or requests that we profile in session 1

```
select current_connection from rdb$database;
```

Stopping remote profiling in session 2

```
EXECUTE PROCEDURE RDB$PROFILER.FINISH_SESSION(TRUE, 29);

COMMIT;
```

Now we can see the profiling result in any connection.

```
SELECT
  S.PROFILE_ID,
  S.ATTACHMENT_ID,
  S.START_TIMESTAMP AS SESSION_START,
  S.FINISH_TIMESTAMP AS SESSION_FINISH,
  R.REQUEST_ID,
  STMT.STATEMENT_ID,
  STMT.STATEMENT_TYPE,
  STMT.PACKAGE_NAME,
  STMT.ROUTINE_NAME,
  STMT.SQL_TEXT,
  R.CALLER_STATEMENT_ID,
  R.CALLER_REQUEST_ID,
  R.START_TIMESTAMP,
  R.FINISH_TIMESTAMP,
  R.TOTAL_ELAPSED_TIME
FROM PLG$PROF_SESSIONS S
JOIN PLG$PROF_STATEMENTS STMT ON STMT.PROFILE_ID = S.PROFILE_ID
JOIN PLG$PROF_REQUESTS R ON R.PROFILE_ID = S.PROFILE_ID AND R.STATEMENT_ID = STMT.STATEMENT_ID
WHERE S.ATTACHMENT_ID = 29
  AND STMT.SQL_TEXT CONTAINING 'FROM HORSE';
```

```
PROFILE_ID                      14
ATTACHMENT_ID                   29
SESSION_START                   2023-11-09 16:56:39.1640
SESSION_FINISH                  2023-11-09 16:57:41.0010
REQUEST_ID                      3506
STATEMENT_ID                    3506
STATEMENT_TYPE                  BLOCK
PACKAGE_NAME                    <null>
ROUTINE_NAME                    <null>
SQL_TEXT                        13e:1
SELECT COUNT(*) FROM HORSE
CALLER_STATEMENT_ID             <null>
CALLER_REQUEST_ID               <null>
START_TIMESTAMP                 2023-11-09 16:57:29.1010
FINISH_TIMESTAMP                2023-11-09 16:57:30.4800
TOTAL_ELAPSED_TIME              82622
```

11.15. Examples of using the profiler to find "bottlenecks"

Now that you've become familiar with the different modes for launching a profiling session, it's time to show how the profiler can help you find "bottlenecks" in your SQL queries and PSQL modules.

Let's say using tracing you found such a slow query:

```
SELECT
 CODE_HORSE,
 BYDATE,
 HORSENAME,
 FRISK,
 OWNERNAME
FROM SP_SOME_STAT(58, '2.00,0', 2020, 2023)
```

We need to understand the reasons for the slowdown of the query and fix them. The first thing to do is eliminate the time it takes to fetch records for the client. To do this, you can simply wrap this query in another query that will simply calculate the number of records. Thus, we are guaranteed to read all records, but at the same time we need to send only 1 record to the client.

```
SELECT COUNT(*)
FROM (
  SELECT
    CODE_HORSE,
    BYDATE,
    HORSENAME,
    FRISK,
    OWNERNAME
  FROM SP_SOME_STAT(58, '2.00,0', 2020, 2023)
);
```

The execution statistics for this SQL query look like this:

```
                COUNT
=====================
                  240

Current memory = 554444768
Delta memory = 17584
Max memory = 554469104
Elapsed time = 2.424 sec
Buffers = 32768
Reads = 0
Writes = 0
Fetches = 124985
```

In this case, the list of fields could simply be replaced with `COUNT(*)` without wrapping the query in a derived table, however, in the general case, the `SELECT` clause can contain various expressions, including subqueries, so it is better to do as I showed .

Now you can run the query in the profiler:

```
SELECT RDB$PROFILER.START_SESSION('Profile procedure SP_SOME_STAT')
FROM RDB$DATABASE;

SELECT COUNT(*)
FROM (
  SELECT
    CODE_HORSE,
    BYDATE,
    HORSENAME,
    FRISK,
    OWNERNAME
  FROM SP_SOME_STAT(58, '2.00,0', 2020, 2023)
);

EXECUTE PROCEDURE RDB$PROFILER.FINISH_SESSION;

COMMIT;
```

First, let's look at the PSQL statistics:

```
SELECT
  ROUTINE_NAME,
  LINE_NUM,
  COLUMN_NUM,
  COUNTER,
  TOTAL_ELAPSED_TIME,
  AVG_ELAPSED_TIME
FROM PLG$PROF_PSQL_STATS_VIEW STAT
WHERE STAT.PROFILE_ID = 5;
```

```
ROUTINE_NAME            LINE_NUM   COLUMN_NUM         COUNTER    TOTAL_ELAPSED_TIME      AVG_ELAPSED_TIME
=================== ============ ============ =============== ==================== ====================
SF_RACETIME_TO_SEC            22            5           67801           1582095600                23334
SF_RACETIME_TO_SEC            18            3           67801             90068700                 1328
SF_RACETIME_TO_SEC            27            5           67801             53903300                  795
SF_RACETIME_TO_SEC            31            5           67801             49835400                  735
SP_SOME_STAT                  16            3               1             43414600             43414600
SF_RACETIME_TO_SEC            25            5           67801             42623200                  628
SF_RACETIME_TO_SEC            34            5           67801             37339200                  550
SF_RACETIME_TO_SEC            14            3           67801             35822000                  528
SF_RACETIME_TO_SEC            29            5           67801             34874400                  514
SF_RACETIME_TO_SEC            32            5           67801             24093200                  355
SF_RACETIME_TO_SEC            15            3           67801             23832900                  351
SF_RACETIME_TO_SEC             6            1           67801             15985600                  235
SF_RACETIME_TO_SEC            26            5           67801             15625500                  230
```

```
SP_SOME_STAT                 38           5             240               3454800                14395
SF_SEC_TO_RACETIME           20           3             240                549900                 2291
SF_SEC_TO_RACETIME           31           3             240                304100                 1267
SF_SEC_TO_RACETIME           21           3             240                294200                 1225
SF_SEC_TO_RACETIME           16           3             240                293900                 1224
SF_RACETIME_TO_SEC            7           1           67801                202400                    2
SF_RACETIME_TO_SEC            8           1           67801                186100                    2

ROUTINE_NAME             LINE_NUM   COLUMN_NUM         COUNTER    TOTAL_ELAPSED_TIME      AVG_ELAPSED_TIME
======================== ========== ============ =============== ==================== ====================
SF_RACETIME_TO_SEC             20           3           67801                168400                    2
SF_RACETIME_TO_SEC              9           1           67801                156700                    2
SF_RACETIME_TO_SEC             12           1           67801                153900                    2
SF_RACETIME_TO_SEC             10           1           67801                153300                    2
SF_SEC_TO_RACETIME             18           3             240                148600                  619
SF_RACETIME_TO_SEC             16           3           67801                127100                    1
SF_SEC_TO_RACETIME             17           3             240                 92100                  383
SF_SEC_TO_RACETIME              8           1             240                 89200                  371
SF_RACETIME_TO_SEC             11           1           67801                 69500                    1
SF_SEC_TO_RACETIME             28           3             240                 16600                   69
SF_RACETIME_TO_SEC              5           1           67801                  7800                    0
SF_SEC_TO_RACETIME             11           1             240                  2000                    8
SF_SEC_TO_RACETIME             10           1             240                  1800                    7
SF_SEC_TO_RACETIME              9           1             240                  1200                    5
SP_SOME_STAT                   37           5             240                   500                    2
SF_SEC_TO_RACETIME             13           3             240                   500                    2
SF_SEC_TO_RACETIME              7           1             240                   400                    1
```

From these statistics it is clear that the leader in total execution time is the operator located in line 22 of the SF_RACETIME_TO_SEC function. The average execution time of this statement is low, but it is called 67801 times. There are two options for optimization: either optimize the SF_RACETIME_TO_SEC function itself (operator on line 22), or reduce the number of calls to this function.

Let's look at the contents of the SP_SOME_STAT procedure.

```
CREATE OR ALTER PROCEDURE SP_SOME_STAT (
    A_CODE_BREED INTEGER,
    A_MIN_FRISK  VARCHAR(9),
    A_YEAR_BEGIN SMALLINT,
    A_YEAR_END   SMALLINT
)
RETURNS (
    CODE_HORSE BIGINT,
    BYDATE     DATE,
    HORSENAME  VARCHAR(50),
    FRISK      VARCHAR(9),
    OWNERNAME  VARCHAR(120)
)
AS
BEGIN
  FOR
    SELECT
      TL.CODE_HORSE,
      TRIAL.BYDATE,
      H.NAME,
      SF_SEC_TO_RACETIME(TL.TIME_PASSED_SEC) AS FRISK
```

```
    FROM
      TRIAL_LINE TL
      JOIN TRIAL ON TRIAL.CODE_TRIAL = TL.CODE_TRIAL
      JOIN HORSE H ON H.CODE_HORSE = TL.CODE_HORSE
    WHERE TL.TIME_PASSED_SEC <= SF_RACETIME_TO_SEC(:A_MIN_FRISK)
      AND TRIAL.CODE_TRIALTYPE = 2
      AND H.CODE_BREED = :A_CODE_BREED
      AND EXTRACT(YEAR FROM TRIAL.BYDATE) BETWEEN :A_YEAR_BEGIN AND :A_YEAR_END
    INTO
      CODE_HORSE,
      BYDATE,
      HORSENAME,
      FRISK
  DO
  BEGIN
    OWNERNAME = NULL;
    SELECT
      FARM.NAME
    FROM
      (
        SELECT
          R.CODE_FARM
        FROM REGISTRATION R
        WHERE R.CODE_HORSE = :CODE_HORSE
          AND R.CODE_REGTYPE = 6
          AND R.BYDATE <= :BYDATE
        ORDER BY R.BYDATE DESC
        FETCH FIRST ROW ONLY
      ) OWN
      JOIN FARM ON FARM.CODE_FARM = OWN.CODE_FARM
    INTO OWNERNAME;

    SUSPEND;
  END
END
```

The function SF_RACETIME_TO_SEC calls on line number 21:

```
    WHERE TL.TIME_PASSED_SEC <= SF_RACETIME_TO_SEC(:A_MIN_FRISK)
```

Obviously this condition will be checked multiple times for each record. Repeated execution of the SF_RACETIME_TO_SEC function contributes significantly to the overall execution time. If you look at the function call itself, you will notice that the function arguments do not depend on the data source, that is, the function value is invariant. This means that we can move its calculation outside the query. So we can rewrite our procedure like this:

```
CREATE OR ALTER PROCEDURE SP_SOME_STAT (
    A_CODE_BREED INTEGER,
    A_MIN_FRISK  VARCHAR(9),
    A_YEAR_BEGIN SMALLINT,
    A_YEAR_END   SMALLINT
)
```

```
RETURNS (
    CODE_HORSE BIGINT,
    BYDATE     DATE,
    HORSENAME  VARCHAR(50),
    FRISK      VARCHAR(9),
    OWNERNAME  VARCHAR(120)
)
AS
  DECLARE TIME_PASSED NUMERIC(18, 3);
BEGIN
  TIME_PASSED = SF_RACETIME_TO_SEC(:A_MIN_FRISK);
  FOR
    SELECT
      TL.CODE_HORSE,
      TRIAL.BYDATE,
      H.NAME,
      SF_SEC_TO_RACETIME(TL.TIME_PASSED_SEC) AS FRISK
    FROM
      TRIAL_LINE TL
      JOIN TRIAL ON TRIAL.CODE_TRIAL = TL.CODE_TRIAL
      JOIN HORSE H ON H.CODE_HORSE = TL.CODE_HORSE
    WHERE TL.TIME_PASSED_SEC <= :TIME_PASSED
      AND TRIAL.CODE_TRIALTYPE = 2
      AND H.CODE_BREED = :A_CODE_BREED
      AND EXTRACT(YEAR FROM TRIAL.BYDATE) BETWEEN :A_YEAR_BEGIN AND :A_YEAR_END
    INTO
      CODE_HORSE,
      BYDATE,
      HORSENAME,
      FRISK
  DO
  BEGIN
    OWNERNAME = NULL;
    SELECT
      FARM.NAME
    FROM
      (
        SELECT
          R.CODE_FARM
        FROM REGISTRATION R
        WHERE R.CODE_HORSE = :CODE_HORSE
          AND R.CODE_REGTYPE = 6
          AND R.BYDATE <= :BYDATE
        ORDER BY R.BYDATE DESC
        FETCH FIRST ROW ONLY
      ) OWN
      JOIN FARM ON FARM.CODE_FARM = OWN.CODE_FARM
    INTO OWNERNAME;

    SUSPEND;
  END
END
```

Let’s try to execute an SQL query after changing the `SP_SOME_STAT` procedure:

```
SELECT COUNT(*)
FROM (
  SELECT
    CODE_HORSE,
    BYDATE,
    HORSENAME,
    FRISK,
    OWNERNAME
  FROM SP_SOME_STAT(58, '2.00,0', 2020, 2023)
);
```

```
                COUNT
=====================
                  240

Current memory = 555293472
Delta memory = 288
Max memory = 555359872
Elapsed time = 0.134 sec
Buffers = 32768
Reads = 0
Writes = 0
Fetches = 124992
```

2.424 sec vs 0.134 sec - the acceleration is significant. Is it possible to do better? Let's run the profiling session again. The new session ID is 6.

Let's look at the PSQL statistics:

```
SELECT
  ROUTINE_NAME,
  LINE_NUM,
  COLUMN_NUM,
  COUNTER,
  TOTAL_ELAPSED_TIME,
  AVG_ELAPSED_TIME
FROM PLG$PROF_PSQL_STATS_VIEW STAT
WHERE STAT.PROFILE_ID = 6;
```

```
ROUTINE_NAME             LINE_NUM   COLUMN_NUM         COUNTER    TOTAL_ELAPSED_TIME      AVG_ELAPSED_TIME
======================== ========== ========== =============== ==================== ====================
SP_SOME_STAT                   18           3               1             10955200             10955200
SP_SOME_STAT                   40           5             240              3985800                16607
SF_SEC_TO_RACETIME             20           3             240               508100                 2117
SF_SEC_TO_RACETIME             31           3             240               352700                 1469
SF_SEC_TO_RACETIME             21           3             240               262200                 1092
SF_SEC_TO_RACETIME             16           3             240               257300                 1072
SF_SEC_TO_RACETIME             18           3             240               156400                  651
SP_SOME_STAT                   17           3               1               141500               141500
SF_SEC_TO_RACETIME              8           1             240               125700                  523
SF_SEC_TO_RACETIME             17           3             240                94100                  392
```

```
SF_RACETIME_TO_SEC             22            5              1                83400                83400
SF_SEC_TO_RACETIME             28            3            240                38700                  161
SF_SEC_TO_RACETIME             10            1            240                20800                   86
SF_SEC_TO_RACETIME             11            1            240                20200                   84
SF_SEC_TO_RACETIME              9            1            240                16200                   67
SF_RACETIME_TO_SEC              6            1              1                 7100                 7100
SF_SEC_TO_RACETIME              7            1            240                 6600                   27
SF_RACETIME_TO_SEC             27            5              1                 5800                 5800
SF_RACETIME_TO_SEC             18            3              1                 5700                 5700
SF_SEC_TO_RACETIME             13            3            240                 5700                   23

ROUTINE_NAME            LINE_NUM   COLUMN_NUM        COUNTER    TOTAL_ELAPSED_TIME      AVG_ELAPSED_TIME
======================= ========== ========== =============== ==================== ====================
SF_RACETIME_TO_SEC             32            5              1                 4600                 4600
SP_SOME_STAT                   39            5            240                 4400                   18
SF_RACETIME_TO_SEC             14            3              1                 4300                 4300
SF_RACETIME_TO_SEC             34            5              1                 3500                 3500
SF_RACETIME_TO_SEC             25            5              1                 3300                 3300
SF_RACETIME_TO_SEC             31            5              1                 2900                 2900
SF_RACETIME_TO_SEC             29            5              1                 2800                 2800
SF_RACETIME_TO_SEC             15            3              1                 1600                 1600
SF_RACETIME_TO_SEC             26            5              1                 1000                 1000
SF_RACETIME_TO_SEC              7            1              1                  800                  800
SF_RACETIME_TO_SEC             20            3              1                  800                  800
SF_RACETIME_TO_SEC              5            1              1                  400                  400
SF_RACETIME_TO_SEC              8            1              1                  400                  400
SF_RACETIME_TO_SEC             10            1              1                  400                  400
SF_RACETIME_TO_SEC             11            1              1                  400                  400
SF_RACETIME_TO_SEC             12            1              1                  400                  400
SF_RACETIME_TO_SEC             16            3              1                  400                  400
SP_SOME_STAT                   15            3              1                  300                  300
SF_RACETIME_TO_SEC              9            1              1                  300                  300
```

Line 18 in the SP_SOME_STAT procedure takes the longest time - this is the top-level cursor itself, but this cursor is opened once. It is important to note here that the total time for retrieving all records from the cursor is affected by the operators executed inside the statement block for processing each cursor record, that is

```
  FOR
    SELECT
...
  DO
  BEGIN
    -- everything that is done here affects the time it takes to retrieve all records from the
cursor
    ...
  END
```

Let's look at what makes the most significant contribution within this block. This is line number 40 of the SP_SOME_STAT procedure, which is called 240 times. Here is the content of the statement that is called:

```
    SELECT
      FARM.NAME
    FROM
```

```
      (
        SELECT
          R.CODE_FARM
        FROM REGISTRATION R
        WHERE R.CODE_HORSE = :CODE_HORSE
          AND R.CODE_REGTYPE = 6
          AND R.BYDATE <= :BYDATE
        ORDER BY R.BYDATE DESC
        FETCH FIRST ROW ONLY
      ) OWN
      JOIN FARM ON FARM.CODE_FARM = OWN.CODE_FARM
    INTO OWNERNAME;
```

Now let's look at the plan of the procedure SP_SOME_STAT:

```
SELECT CS.MON$EXPLAINED_PLAN
FROM MON$COMPILED_STATEMENTS CS
JOIN PLG$PROF_STATEMENTS S ON S.STATEMENT_ID = CS.MON$COMPILED_STATEMENT_ID
WHERE CS.MON$OBJECT_NAME = 'SP_SOME_STAT'
  AND S.PROFILE_ID = 6;
```

```
================================================================================
MON$EXPLAINED_PLAN:

Select Expression (line 40, column 5)
    -> Singularity Check
        -> Nested Loop Join (inner)
            -> First N Records
                -> Refetch
                    -> Sort (record length: 28, key length: 8)
                        -> Filter
                            -> Table "REGISTRATION" as "OWN R" Access By ID
                                -> Bitmap
                                    -> Index "REGISTRATION_IDX_HORSE_REGTYPE" Range Scan (full match)
            -> Filter
                -> Table "FARM" Access By ID
                    -> Bitmap
                        -> Index "PK_FARM" Unique Scan
Select Expression (line 18, column 3)
    -> Nested Loop Join (inner)
        -> Filter
            -> Table "TRIAL" Access By ID
                -> Bitmap And
                    -> Bitmap
                        -> Index "IDX_TRIAL_BYYEAR" Range Scan (lower bound: 1/1, upper bound: 1/1)
                    -> Bitmap
                        -> Index "FK_TRIAL_TRIALTYPE" Range Scan (full match)
        -> Filter
            -> Table "TRIAL_LINE" as "TL" Access By ID
                -> Bitmap
                    -> Index "FK_TRIAL_LINE_TRIAL" Range Scan (full match)
        -> Filter
            -> Table "HORSE" as "H" Access By ID
                -> Bitmap
                    -> Index "PK_HORSE" Unique Scan
```

```
================================================================================
```

So the inner query plan is:

```
Select Expression (line 40, column 5)
    -> Singularity Check
        -> Nested Loop Join (inner)
            -> First N Records
                -> Refetch
                    -> Sort (record length: 28, key length: 8)
                        -> Filter
                            -> Table "REGISTRATION" as "OWN R" Access By ID
                                -> Bitmap
                                    -> Index "REGISTRATION_IDX_HORSE_REGTYPE" Range Scan (full match)
            -> Filter
                -> Table "FARM" Access By ID
                    -> Bitmap
                        -> Index "PK_FARM" Unique Scan
```

As you can see, the plan is not the most effective. An index is used to filter the data, and then sort the resulting keys and `Refetch`. Let's look at the `REGISTRATION_IDX_HORSE_REGTYPE` index:

```
SQL> SHOW INDEX REGISTRATION_IDX_HORSE_REGTYPE;
REGISTRATION_IDX_HORSE_REGTYPE INDEX ON REGISTRATION(CODE_HORSE, CODE_REGTYPE)
```

Only the `CODE_HORSE` and `CODE_REGTYPE` fields are included in the index, so index navigation cannot be used to determine the last record on a date. Let's try to create another composite index:

```
CREATE DESCENDING INDEX IDX_REG_HORSE_OWNER ON REGISTRATION(CODE_HORSE, CODE_REGTYPE, BYDATE);
```

Let's run the query again:

```
SELECT COUNT(*)
FROM (
  SELECT
    CODE_HORSE,
    BYDATE,
    HORSENAME,
    FRISK,
    OWNERNAME
  FROM SP_SOME_STAT(58, '2.00,0', 2020, 2023)
);
```

```
                COUNT
=====================
                  240

Current memory = 554429808
```

```
Delta memory = 288
Max memory = 554462400
Elapsed time = 0.125 sec
Buffers = 32768
Reads = 0
Writes = 0
Fetches = 124165
```

Just in case, let's check that the procedure plan has changed.

```
SELECT CS.MON$EXPLAINED_PLAN
FROM MON$COMPILED_STATEMENTS CS
WHERE CS.MON$OBJECT_NAME = 'SP_SOME_STAT'
ORDER BY CS.MON$COMPILED_STATEMENT_ID DESC
FETCH FIRST ROW ONLY;
```

```
================================================================================
MON$EXPLAINED_PLAN:

Select Expression (line 38, column 5)
    -> Singularity Check
        -> Nested Loop Join (inner)
            -> First N Records
                -> Filter
                    -> Table "REGISTRATION" as "OWN R" Access By ID
                        -> Index "IDX_REG_HORSE_OWNER" Range Scan (lower bound: 3/3, upper bound: 2/3)
            -> Filter
                -> Table "FARM" Access By ID
                    -> Bitmap
                        -> Index "PK_FARM" Unique Scan
Select Expression (line 16, column 3)
    -> Nested Loop Join (inner)
        -> Filter
            -> Table "TRIAL" Access By ID
                -> Bitmap And
                    -> Bitmap
                        -> Index "IDX_TRIAL_BYYEAR" Range Scan (lower bound: 1/1, upper bound: 1/1)
                    -> Bitmap
                        -> Index "FK_TRIAL_TRIALTYPE" Range Scan (full match)
        -> Filter
            -> Table "TRIAL_LINE" as "TL" Access By ID
                -> Bitmap
                    -> Index "FK_TRIAL_LINE_TRIAL" Range Scan (full match)
        -> Filter
            -> Table "HORSE" as "H" Access By ID
                -> Bitmap
                    -> Index "PK_HORSE" Unique Scan
================================================================================
```

Yes, the plan got better. Navigation by index IDX_REG_HORSE_OWNER is now used with Range Scan. If we compare the execution time, we get 0.134 seconds vs 0.125 seconds and 124992 vs 124165 fetches. The improvements are very minor. In principle, relative to the original version, our procedure has already become 19 times faster, so optimization can be completed.

Chapter 12. Conclusion

This concludes the review of the new features of Firebird 5.0. The Firebird developers have done a great job, for which we are very grateful. Migrate to Firebird 5.0 and get all the features described above.