EmberWings

2024/4

Firebird

December 2024

www.firebirdsql.org

emberwings@firebirdsql.org

The Firebird project was created at SourceForge on

July 31, 2000

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

Support Firebird

Thank you for your support!

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

Firebird Foundation z.s.

In This Issue:

A New Chapter for *EmberWings*

Dear Readers,

Welcome to the December issue of *EmberWings*. This release marks an important transition, as the magazine now operates under the care of the **Firebird Foundation**. While the general concept of *EmberWings* remains the same, this shift brings new opportunities to serve the Firebird community more effectively.

One of the key changes we've introduced is a more balanced focus on content. Moving forward, *EmberWings* will aim to equally reflect the interests of both **Firebird administrators** and **application developers**. By doing so, we hope to provide practical value to all who rely on Firebird in their work.

This issue also marks the debut of **Planet Firebird**, a new regular section dedicated to news and stories from the Firebird user community. This feature will highlight events, real-world use cases, community achievements, and other updates, offering readers insight into how Firebird is being utilized around the globe.

Looking ahead, future issues may include contributions from members of the **Firebird Project**, now that *EmberWings* is the official publication of the project. These articles will provide an inside perspective on development, features, and the direction of Firebird, keeping readers informed about the database's evolution.

Finally, each issue of *EmberWings* will now have a unique theme. For December, we've embraced the **Christmas season**, while the **March issue** will take on an **Easter theme**. These themes will guide the tone and focus of our content, providing a sense of variety and seasonal relevance.

Thank you for reading and for your continued support of the Firebird community. We look forward to bringing you more useful, balanced, and engaging content in the months to come.

Warm regards,
The EmberWings Team

A young apprentice, eager to perfect his craft, approached the Master Developer.

“Master,” the apprentice said, “I have built an application that uses the Firebird, but I feel blind. How can I see the Firebird’s heart? How can I know its strength and wisdom?”

The Master Developer closed his editor and turned to the apprentice. “The Firebird is a quiet and patient creature,” he said. “It will not burden you with more than you need. But if you ask it the right questions, it will tell you all.”

The Master then shared this tale:

Once, a traveler came to the forest where the Firebird dwelled. Seeing the radiant bird perched upon the data tree, the traveler said, “O Firebird, tell me of your world. How can I trust you to carry my dreams?”

The Firebird looked at the traveler and spoke. “My world is vast, but I reveal only what is asked. To hear me, you must speak through the proper voices: the attachment, the transaction, the statement, or the result set.”

The traveler was puzzled. “But why so many voices?

The Firebird replied, "Each voice knows a different truth:

- *The Attachment knows the whole of my being. Ask its getInfo, and it will tell you of my abilities, my state, and the parameters of my current tasks. It sees the broadest view of my nest.*
- *The Transaction knows the flow of work within me. Its getInfo whispers of isolation levels and timeouts.*
- *The Statement holds the plans of queries and their preparation. Its getInfo reveals the strategies I use to turn questions into answers.*
- *The Result Set carries the fruits of your queries. Its getInfo tells of fetched rows.*

To ask all questions of one voice would be like asking the wind to paint. Each must speak in its own domain."

The traveler thought deeply. "How will I know which voice to seek?"

The Firebird smiled. "Seek the heart of your question. When you need to see the forest, call upon the attachment. When you wish to follow the stream, speak to the transaction. When you need the plan of the journey, the statement will answer. And when you wish to gather the fruits, the result set will guide you. Listen well, and you will find your way."

The Master Developer turned to the apprentice. "The Firebird's voices are clear and purposeful. Speak to them with respect, and they will reveal their secrets. But remember: ask only what you need. The Firebird values simplicity in its answers."

The apprentice nodded, his understanding deepened. He returned to his workstation and called upon the Firebird's voices, listening to its whispers with clarity and purpose. And as the Firebird spoke, the apprentice's application grew wiser and stronger.

Christmas Eve disaster

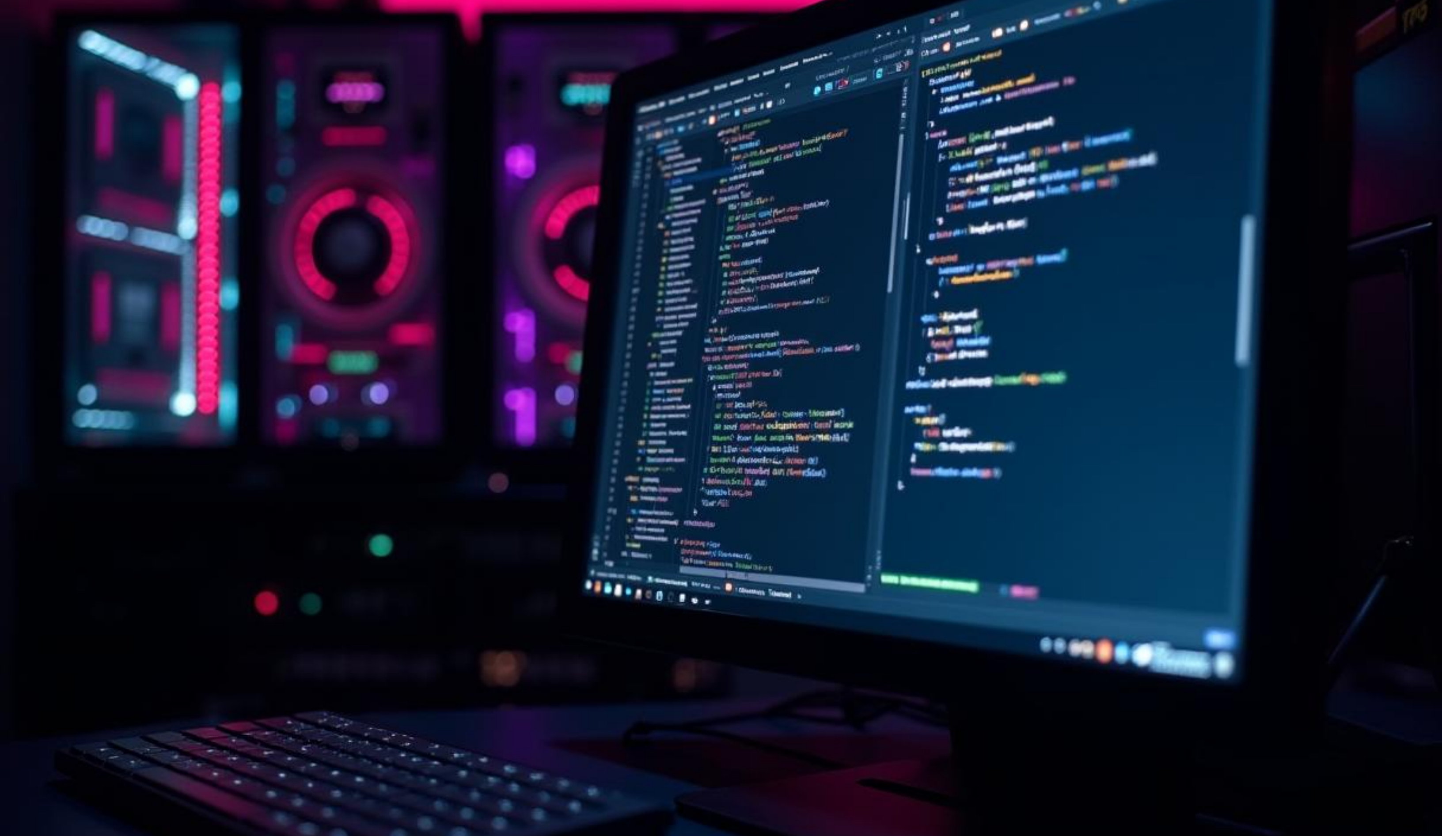
It was a Christmas Eve disaster: an overeager elf accidentally ran a destructive query on the "ShipmentLogs" table without a WHERE clause. The table was wiped clean, and with it, the entire sleigh delivery schedule!

Panicked, Santa called the Firebird support team (consisting of Mrs. Claus and a savvy penguin named Pete).

“Relax,” Pete said. “We’ve been using Firebird’s nbackup with multilevel backups. Let’s restore the latest level 0 backup, then layer the incremental backups on top of it.”

The team quickly restored the database by applying the backups step by step. Within minutes, the "ShipmentLogs" table was fully restored, and Christmas was back on track.

From then on, Santa implemented a strict policy: “Never skip the WHERE clause.”



Firebird Info Calls

By Pavel Císař (IBPhoenix)

When developing applications and libraries working with Firebird, it is often necessary to obtain information about the parameters and current state of the server, connected databases, ongoing transactions and other important elements that make up complex software such as a database server.

This information is available through system tables and dedicated API functions, the so-called "info calls". These basic sources of information are continuously expanded, including the introduction of completely new methods of obtaining specific information such as monitoring tables, security-related tables, or the trace service.

In this article, we will introduce you to the general mechanism of Firebird's info calls, as implemented in the original and new OO API, what information can be obtained through them, and how some Firebird drivers, such as Jaybird, .NET Provider or Python driver, work with the availability of this information.

What Are Info Calls?

Information calls are always tied to a specific active Firebird resource/element, e.g. database connection, transaction, SQL query, query output, BLOB value, service manager, etc. Within the API, one "info" function is then defined for each resource. In the legacy API, these are separate functions with names following the `isc_*_info` pattern (e.g. `isc_database_info`). In the modern OO API, these are methods that are part of the relevant interface (and typically named `getInfo`). All calls have a unified form of structured requests and responses passed as input (request buffer) and output (response buffer) parameters.

Each piece of information that Firebird provides for a given resource has assigned a non-zero numeric identification, the so-called "info code", and a specific binary response format. One or more info codes can be specified in the request buffer, thus obtaining several related pieces of information at the same time. It is therefore a form of a simple language with binary representation.

The great advantages of this mechanism are its simplicity, universality, and extensibility. The weakness is the need to parse the output buffer, and to deal with possible insufficient output buffer size.

Requests and responses

The request for information is transmitted in a buffer as a sequence of one or more info codes of one byte in size. Some info codes may have additional parameters that are stored immediately after the request code, in a pre-specified format. If the buffer is larger than the amount of data stored in it, it is necessary to close the request with the `isc_info_end` (1) code.

It is necessary to reserve a sufficiently large buffer for the response, which is passed to the relevant info function and, upon return, contains the requested data. The response consists of blocks of data for individual info codes in the request. Each block begins with the request info code (with few exceptions), followed by the response data in the format according to the type of request. If the buffer is not large enough, the `isc_info_truncated` (2) value is used instead of the info code. If the response is complete and the entire buffer capacity is not used, the entire sequence is closed with the one-byte `isc_info_end` (1) value. If the request contains an incorrect info code, the engine returns an error code `isc_info_error` (3).

Numbers are stored in little endian, and unless otherwise specified, as signed. They are usually stored directly in the specified length. However, in specific cases they are stored as sized, i.e. first 2 bytes of the number size in bytes, and then the value itself.

Strings are always stored as sized, i.e. first the length of the string (unless otherwise specified, then in two bytes) followed by the characters of the string.

Example call to `isc_database_info` (old API):

```
isc_database_info(status_vector, db_handle, sizeof(request),
                  request, sizeof(response), response);
```

Example call in the OO API:

```
db->getInfo(&status_vector, sizeof(db_info_items), db_info_items,
            sizeof(buffer), buffer);
```

The secret language of info calls

The biggest obstacle to using info calls has always been the lack of comprehensive documentation. Firebird inherited these calls from InterBase 6, and the only documentation available at that time was the "InterBase API Guide". However, it was soon discovered that the source code offers many more info codes than described in the API documentation. Some codes were identified as no longer valid, some never fulfilled a real function, others are intended only for internal use within the implementation of APIs or drivers, but some codes have proven useful. The documentation for them is not comprehensive at all, and in written form exists mainly only in messages that can be found in the `firebird-devel` or `firebird-support` archives. And of course in the Firebird source codes.

Over the years of Firebird development, new info codes have been and are being added. They are documented in the Release Notes of the version that introduced them, and in various text files in the `/doc` subdirectory of the Firebird installation.

Due to the lack of comprehensive documentation, support for info calls in Firebird libraries and drivers is quite uneven. We'll discuss this question later, but first let's take a look at what's on offer.

Firebird info codes

For convenience, we list the names of the info codes as they are listed in the `inf_pub.h` header file supplied with Firebird 5.0.

Database attachment

Info codes are defined via `db_info_types` enum, and are listed in order of their value. Note that the sequence of values is not continuous. This is because some values that are demonstrably no longer functional are not listed. However, some meaningless values are still available.

isc_info_db_id (4): Returns two or more character strings. The first string always contains the database identifier, depending on the `DatabaseAccess` value in `firebird.conf`. When set to "None" it is always the database alias, with other values it is always the full path to the connected database, regardless of whether an alias or file specification was used to connect. The following one or more (typically three) strings contain the server hostname, depending on the connection method. For local protocol - just one string is returned, for network protocols - three strings are returned (like with the isql `SHOW VERSION` command).

The data block consists from its length on two bytes, followed by number of strings on one byte. Strings are stored in Pascal format (one byte of length, then characters).

isc_info_reads (5): Number of reads from disk to page cache. Cumulative value for the database connection since its creation. The same applies to other I/O counters.

isc_info_writes (6): Number of writes from page cache to disk as sized integer.

isc_info_fetches (7): Number of fetches from page cache as sized integer. Formula to calculate the cache hit ratio is: 1 - (reads / fetches).

isc_info_marks (8): Number of writes to page in cache as sized integer.

isc_info_implementation (11): Information about the Firebird implementation serving this connection. This is the original version inherited from InterBase, which was deprecated in Firebird 3 with the much more informative `fb_info_implementation` code. However, it is and will continue to be available.

The response consists from 2 bytes of total data length, followed by byte with the number of two-byte sequences, where the first byte is the implementation code (from `info_db_implementations` enum) and the second byte is the implementation class code (from `info_db_class` enum). Like with `isc_info_db_id`, you may get one or more such sequences in relation to connection type.

isc_info_isc_version (12): Firebird version in text format, compatible with InterBase. Firebird version numbering started from one, but the functionality was based on InterBase v6. Some software checked the version and refused to work with Firebird. Therefore, the answer to this code was modified to follow InterBase v6 versioning. For example, Linux Firebird 5.0.1.1469 returns the string "LI-V6.3.1.1469 Firebird 5.0". The letter "L" here means Linux, for Windows it returns "W". Like with isql `SHOW VERSION`, it may return one or more strings.

The response consists from 2 bytes of total data length, followed by byte with the number of strings. These are then stored in Pascal format.

isc_info_base_level (13): Another historical info code, the exact meaning of which has been lost in the currents of time. According to the remnants of notes in the source code, it should represent the version of the capabilities of the engine itself (unlike ODS which is tied to the database structure). Historically, it corresponded to the InterBase version (with a minor anomaly in version 4.1) and since Firebird is based on InterBase 6, all versions of Firebird return this value. The data block consists from its length on 2 bytes, one byte with number of byte codes, followed by level codes. Once again, you may get one or more codes in relation to connection type.

isc_info_page_size (14): Database page size as a sized integer value.

isc_info_num_buffers (15): Sized integer value of the database page cache size this connection works with. The cache size may differ from the configured size if it has been changed by a connection parameter (does not apply to SuperServer).

isc_info_limbo (16): Ids of transactions stuck in two-phase commit limbo. If there are none, the response is empty, and info code is not present in response at all. Otherwise, the response is made up of one or more blocks consisting of an info code followed by a sized integer value with transaction ID.

isc_info_current_memory (17): Size (in bytes) of memory block currently used by connection, as a sized integer value.

isc_info_max_memory (18): The max. size (in bytes) of memory block that was ever used by connection since creation, as a sized integer value.

isc_info_window_turns (19): Deprecated. For some unknown reason it is still in the header files. On use, the engine returns the error code `isc_info_error` (3).

isc_info_license (20): Deprecated. For some unknown reason it is still in the header files. On use, the engine returns the error code `isc_info_error` (3).

isc_info_allocation (21): Number of pages allocated for the database, as a sized integer value.

isc_info_attachment_id (22): Internal attachment ID as a sized integer value.

isc_info_read_seq_count (23): Number of rows read sequentially. Cumulative value for the database connection since its creation. The output is more complex because it contains values for each table for which a non-zero value is recorded. The first two bytes after the info code contain the total data size in bytes (multiples of 6), followed by the appropriate number of data blocks: a two-byte table identification plus a 4-byte counter value. The table identification correspond to the value of the `RDB$RELATION_ID` column in the `RDB$RELATIONS` system table. The same applies to other table counters.

isc_info_read_idx_count (24): Number of rows read with use of index. Format is the same as with **isc_info_read_seq_count**.

isc_info_insert_count (25): Number of inserted rows. Format is the same as with **isc_info_read_seq_count**.

isc_info_update_count (26): Number of updated rows. Format is the same as with **isc_info_read_seq_count**.

isc_info_delete_count (27): Number of deleted rows. Format is the same as with **isc_info_read_seq_count**.

isc_info_backout_count (28): Number of records that were reverted to previous state (after rollback). Format is the same as with **isc_info_read_seq_count**.

isc_info_purge_count (29): Number of previous record versions that were removed as no longer necessary. Format is the same as with **isc_info_read_seq_count**.

isc_info_expunge_count (30): Number of records that were completely erased from database. Format is the same as with **isc_info_read_seq_count**.

isc_info_sweep_interval (31): Sweep interval as a sized integer value.

isc_info_ods_version (32): Major ODS version as a sized integer value.

isc_info_ods_minor_version (33): Minor ODS version as a sized integer value.

isc_info_no_reserve (34): Database NO RESERVE flag as a sized integer value.

Follows block of deprecated WAL and JOURNAL items with values 35-51. For some unknown reason they are still in the header files. On use, the engine returns the error code `isc_info_error` (3).

isc_info_forced_writes (52): Database FORCED WRITES flag as a sized integer value.

isc_info_user_names (53): Returns a sequence of one or more blocks with the names of users connected to this database. Each block is a sequence consisting of an info code, 2 bytes with the size of the following data block, a string of characters in Pascal format (one byte length followed by characters). This list is never empty.

The following info codes are used by the GFIX utility and are of no particular importance to application developers: **isc_info_page_errors** (54), **isc_info_record_errors** (55), **isc_info_bpage_errors** (56), **isc_info_dpage_errors** (57), **isc_info_ipage_errors** (58), **isc_info_ppage_errors** (59) and **isc_info_tpage_errors** (60).

isc_info_set_page_buffers (61): This info code is used by the GBAK utility, and is of no real use to application developers.

isc_info_db_sql_dialect (62): Database SQL Dialect as a sized integer value.

isc_info_db_read_only (63): Database READ ONLY flag as a sized integer value.

isc_info_db_size_in_pages (64): Number of used database pages as a sized integer value. The value is the same or smaller than number of pages allocated for the database.

Values 65 - 100 are unused to avoid conflict with InterBase.

frb_info_att_charset (101): Attachment character set as a sized integer value. The character set ID corresponds to the value of the `RDB$CHARACTER_SET_ID` column in the `RDB$CHARACTER_SETS` system table.

isc_info_db_class (102): The database access class as a sized integer value. The class code corresponds to an item from `info_db_class` enumeration. It's a bit esoteric, but it can help you differentiate between local and remote access.

isc_info_firebird_version (103): Returns one or more strings identifying the Firebird version, similar to the output of the isql `SHOW VERSION` command. Returns one string when accessed locally, three strings when accessed remotely. Theoretically, there could be more if a gateway was involved. The data block (after the info code) consists of the total length of the data stored in 2 bytes, the number of strings in one byte, followed by the specified number of strings in Pascal format. Example of output for remote access:

```
LI-V5.0.1.1469 Firebird 5.0
LI-V5.0.1.1469 Firebird 5.0/tcp (MyHost)/P18:C
LI-V5.0.1.1469 Firebird 5.0/tcp (MyHost)/P18:C
```

isc_info_oldest_transaction (104): Oldest Interesting Transaction (OIT) as a sized integer value.

isc_info_oldest_active (105): Oldest Active Transaction (OAT) as a sized integer value.

isc_info_oldest_snapshot (106): Oldest Snapshot Transaction (OST) as a sized integer value.

isc_info_next_transaction (107): Next Transaction ID as a sized integer value.

isc_info_db_provider (108): Database Provider as a sized integer value. For Firebird it is always 4, InterBase has value 3.

isc_info_active_transactions (109): Ids of active transactions. If there are none, the response is empty, and info code is not present in response at all. Otherwise, the response is made up of one or more blocks consisting of an info code followed by a sized integer value with transaction ID.

isc_info_active_tran_count (110): Number of active transactions as a sized integer value.

isc_info_creation_date (111): Database creation timestamp (without time zone). The data block is formed by its length on 2 bytes, followed by the appropriate number of bytes, in this case it is always 8. The first 4 bytes are the date

(`ISC_DATE` structure), the next 4 bytes are the time (`ISC_TIME` structure). The isc_decode_timestamp function can be used for conversion in the legacy API, IUtil.decodeDate/decodeTime in the OO API.

isc_info_db_file_size (112): A sized integer value used internally by NBACKUP, and is of no real use to application developers.

fb_info_page_contents (113): This info code returns the contents of the specified database page if the user is SYSDBA. For other users, it returns `isc_info_error` (3). In the request, the info code must be followed by a sized integer value with the sequence number of the page to be returned. The response data block contains 2 bytes of length, followed by the appropriate number of bytes of the page contents.

fb_info_implementation (114): More fine-grained implementation information than old `isc_info_implementation`. The response consists 2 bytes of total data length, followed by byte with the number of six-byte sequences: CPU ID, OS ID, compiler ID, flags, implementation class and current depth of implementation stack (when multiple versions are mixed).

Meaning of first four values is defined only in Firebird sources (see `DbImplementation.cpp`) as they are missing from header files supplied with Firebird. Like with `isc_info_db_id`, you may get one or more such sequences in relation to connection type.

The following info codes are used by the GFIX utility and are of no particular importance to application developers: **fb_info_page_warns** (115), **fb_info_record_warns** (116), **fb_info_bpage_warns** (117), **fb_info_dpage_warns** (118), **fb_info_ipage_warns** (119), **fb_info_ppage_warns** (120), **fb_info_tpage_warns** (121), **fb_info_pip_errors** (122) and **fb_info_pip_warns** (123).

fb_info_pages_used (124): Number of used database pages as a sized integer value.

fb_info_pages_free (125): Number of free database pages as a sized integer value.

isc_info_active_tran_count (110): Number of active transactions as a sized integer value.

fb_info_ses_idle_timeout_db (129): Idle connection timeout value as defined in configuration, stored as a sized integer value.

fb_info_ses_idle_timeout_att (130): Idle connection timeout value as defined at connection level, stored as a sized integer value.

fb_info_ses_idle_timeout_run (131): Actual idle connection timeout value for given connection considering values set at configuration and connection levels, stored as a sized integer value. The "effective value" is evaluated every time user API call leaves the engine.

fb_info_conn_flags (132): Connection flags as a sized integer value. Individual flags are encoded as bits: `COMPRESSED = 0x01`,`ENCRYPTED = 0x02`.

fb_info_crypt_key (133): Name of encryption key, as sized string.

fb_info_crypt_state (134): Database encryption flags as a sized integer value. Individual flags are encoded as bits (of `db_info_crypt` enum): `fb_info_crypt_encrypted = 0x01`,`fb_info_crypt_process = 0x02`

fb_info_statement_timeout_db (135): Idle statement timeout value as defined in configuration, stored as a sized integer value.

fb_info_statement_timeout_att (136): Idle statement timeout value as defined at connection level, stored as a sized integer value.

fb_info_protocol_version (137): Network protocol version, as a sized integer value.

fb_info_crypt_plugin (138): Name of encryption plugin, as sized string.

fb_info_creation_timestamp_tz (139): Database creation timestamp with time zone. The data block is formed by its length on 2 bytes, followed by the appropriate number of bytes that are the `ISC_TIMESTAMP_TZ` structure. Function `IUtil.decodeTimeStampTz` in the OO API can be used for conversion.

fb_info_wire_crypt (140): Name of wire encryption plugin, as sized string.

fb_info_wire_crypt (140): Name of wire encryption plugin, as sized string.

fb_info_features (141): List of features supported by current connection provider. The data block is formed by its length on 2 bytes, followed by the appropriate number of bytes. Each byte in array is one value from `info_features` enumeration:

- **fb_feature_multi_statements** (1): Multiple prepared statements in single attachment.
- **fb_feature_multi_transactions** (2): Multiple concurrent transaction in single attachment.
- **fb_feature_named_parameters** (3): Query parameters can be named.
- **fb_feature_session_reset** (4): ALTER SESSION RESET is supported.
- **fb_feature_read_consistency** (5): Read Consistency transaction isolation level is supported.
- **fb_feature_statement_timeout** (6): Statement timeout is supported.
- **fb_feature_statement_long_life** (7): Prepared statements are not dropped on transaction end.

fb_info_next_attachment (142): ID of next attachment, as a sized integer value.

fb_info_next_statement (143): ID of next statement, as a sized integer value.

fb_info_db_guid (144): Database GUID as sized string.

fb_info_db_file_id (145): Unique database file ID as sized string. Usually a 32 character long hexadecimal number. Used internally in the SRP manager.

fb_info_replica_mode (146): Replica mode as sized integer value: none = 1, read only = 1, read write = 2.

fb_info_username (147): User name as sized string.

fb_info_sqlrole (148): User role as sized string.

Transaction

Transaction information could be obtained using `isc_transaction_info` function in legacy API, or `ITransaction.getInfo` method in OO API. Codes are defined as `isc_info_tra_*` and `fb_info_tra_*` constants, and are listed in order of their value.

isc_info_tra_id (4): Transaction ID, as sized integer value.

isc_info_tra_oldest_interesting (5): Oldest interesting transaction (OIT) as seen by this transaction, as sized integer value.

isc_info_tra_oldest_snapshot (6): Oldest snapshot transaction (OST) as seen by this transaction, as sized integer value.

isc_info_tra_oldest_active (7): Oldest active transaction (OAT) as seen by this transaction, as sized integer value.

isc_info_tra_isolation (8): Transaction isolation level. First two bytes is length of following byte array, where each byte encodes the transaction level.

First value is always one from: `isc_info_tra_consistency` (1), `isc_info_tra_concurrency` (2) or `isc_info_tra_read_committed` (3). For read committed isolation, the second byte defines its variant: `isc_info_tra_no_rec_version` (0), `isc_info_tra_rec_version` (1) or `isc_info_tra_read_consistency` (2).

isc_info_tra_access (9): Transaction access as sized integer value, one from: `isc_info_tra_readonly` (0) or `isc_info_tra_readwrite` (1).

isc_info_tra_lock_timeout (10): Transaction lock timeout as sized integer value. Value -1 means WAIT, zero is NOWAIT, non-zero positive value is wait timeout set explicitly.

fb_info_tra_dbpath (11): The DSN of the parent database connection. Are you wondering what good is this at the transaction level when similar information is available at the database connection level? Well, this code is used by the distributed transaction manager.

fb_info_tra_snapshot_number (12): Transaction snapshot number as sized integer value. Important information when creating parallel processes that work with a consistent database state.

Statement

DSQL statement information could be obtained using `isc_dsql_sql_info` function in legacy API, or `IStatement.getInfo` method in OO API. Codes are defined as `isc_info_sql_*` constants, and are listed in order of their value.

Codes 4 - 20, and 25 are used internally to describe input and output query parameters.

isc_info_sql_stmt_type (21): Statement type as sized integer value. The type of command (e.g. INSERT, UPDATE, EXECUTE PROCEDURE, etc.) allows you to identify its properties and behavioral characteristics. Firebird 5 defines a total of 12 types of DSQL commands. Values are defined as `isc_info_sql_stmt_*` constants.

isc_info_sql_get_plan (22): Execution plan in classic form as sized string.

isc_info_sql_records (23): The number of rows affected by the SQL statement. Values are available only after the statement is executed, and for selects it reflects only fetched rows. The database engine's own support for the determination of affected rows is quirky. The database engine only supports the determination of rowcount for INSERT, UPDATE, DELETE, and SELECT statements. When stored procedures become involved, row count figures are usually not available to the client.

First two bytes is length of following data cluster, which consists of several blocks formed from a single-byte operation code followed by the number of rows as sized integer value. Operation codes are: `isc_info_req_select_count` (13), `isc_info_req_insert_count` (14), `isc_info_req_update_count` (15) and `isc_info_req_delete_count` (16).

isc_info_sql_batch_fetch (24): Batch fetch flag as sized integer value. Value `1` means that data are fetched in batches.

isc_info_sql_explain_plan (26): Execution plan in explained form as sized string.

isc_info_sql_stmt_flags (27): Statement flags as sized integer value. The flag values are not defined in the `inf_pub.h` file, but in `IdlFbInterfaces.h` as constants on `IStatement` interface: `FLAG_HAS_CURSOR` = 0x1 and `FLAG_REPEAT_EXECUTE` = 0x2.

isc_info_sql_stmt_timeout_user (28): Timeout value for given statement as sized integer value.

isc_info_sql_stmt_timeout_run (29): Sized integer value with actual timeout value for given statement evaluated considering values set at config, attachment and statement levels. It's valid only when timeout timer is running, i.e. for currently executed statements.

isc_info_sql_stmt_blob_align (30): Blob stream alignment as sized integer value. Used internally with network protocols older than version 17 (Firebird 5 uses version 18). Not useful for application developers.

Codes **isc_info_sql_exec_path_blr_bytes** (31) and **isc_info_sql_exec_path_blr_text** (32): BLR representation of the statement in binary and text format, as sized byte array or string. This information can only be obtained with the appropriate SET DEBUG OPTIONS setting (SQL command available since Firebird 4.0.1). It is of little value to regular application developers, but it is important to developers of tools and libraries.

Blobs

Information about fetched BLOB could be obtained using `isc_blob_info` function in legacy API, or `IBlob.getInfo` method in OO API. Codes are defined as `isc_info_blob_*` constants, and are listed in order of their value.

isc_info_blob_num_segments (4): The total number of segments that make up this segmented blob, as sized integer value.

isc_info_blob_max_segment (5): The maximum size of a segment that creates a segmented BLOB object, as sized integer value. It is used to determine the size of the buffer for working with a BLOB value.

isc_info_blob_total_length (6): Total BLOB length in bytes, as sized integer value.

isc_info_blob_type (7): BLOB type as sized integer value. Values are defined as `isc_bpb_type_segmented` (0) and `isc_bpb_type_stream` (1) constants.

Result set

Firebird 5 added `getInfo` method to `IResultSet` interface. So far it supports only one info code `INF_RECORD_COUNT` (10) defined on the interface, which returns the number of cached rows for scrollable cursors, as sized integer value. For non-scrollable cursors returns -1.

All others

There are additional `getInfo` methods defined on `IBatch`, `IRequest` and `IExternalContext` interfaces. Info codes for `IBatch` are defined as `INF_*` constants on interface, and might be interesting only to those that construct batches at such low level (i.e. driver developers). Interface `IRequest` is used only to work with BLR requests, which are not used by application developers as they use the higher-level DSQL interface. And finally the `IExternalContext` interface, which interests only Firebird extension developers, already provides direct methods to obtain all essential information they need.

Info calls in Firebird drivers and access libraries

Jaybird Java driver

Jaybird API generally handles the info calls itself, and presents results to the user. As such, users generally don't need to execute or parse info calls themselves.

For example:

- Basic information about the database (ODS, dialect) is accessible through `FirebirdDatabaseMetaData`.
- Transaction information (oldest, oldest active, oldest snapshot, etc) is available through `FBStatisticsManager.getDatabaseTransactionInfo`
- Table statistics are available through `FBTableStatisticsManager`
- Execution/explained plans can be obtained from `FirebirdStatement`/`FirebirdPreparedStatement` (depends on Jaybird version).
- Statement counts can be obtained from `FirebirdStatement.getSqlCounts()`
- Parameter/column information is exposed through standard JDBC APIs `ResultSetMetaData` and `ParameterMetaData`.

However, some of those options are not (or only high-level) documented in the Jaybird Manual, so that is something that needs some work.

If you really want to, you can dive into the GDS-ng API, and use the internal/low-level API:

- `FbDatabase.getDatabaseInfo`
- `FbStatement.getSqlInfo`
- `FbStatement.getCursorInfo`, not supported for native, nor for Firebird 4.0 or lower for pure Java (wire protocol)
- `FbTransaction.getTransactionInfo`
- `FbBlob.getBlobInfo`

There are two variants of this API, one for native client library and one for direct network protocol access. These are basically thin wrappers around opcodes for the wire protocol, or legacy API functions for the native implementation.

These methods accept a byte-array with the info items, and a buffer length, and - for the second variant - an "InfoProcessor". The first variant returns a byte array with the server response, the second variant returns an object produced by feeding the response of the server to the InfoProcessor. The InfoProcessor is an interface where you can implement parsing of the information response to produce an object.

However, this is considered an internal Jaybird API, and the maintainer’s stance is that if you need to delve into this level, you should file a feature request in issue tracker at GitHub.

.NET Provider

The situation with the .NET provider is similar to Jaybird. All the primary important information is provided directly at the object level, but it is not a complete set. Access to lower levels is then more complicated than with Jaybird, and if you are missing access to some information, you should file a feature request in issue tracker at GitHub

Python drivers

Because Python is used for testing in Firebird development, support for info calls in official Python drivers is traditionally above standard, and if possible complete. Because the concept of information access differs between the two drivers, we will focus only on the new `firebird-driver` in latest version, and omit the legacy `FDB` driver.

Since the repertoire of info codes expands with each Firebird version, the handling of these calls is encapsulated in a separate hierarchy of classes derived from the `InfoProvider` class. The "info" property is then defined on the `Connection`, `Transaction`, and `Statement` classes, which provides access to the `InforProvider` instance corresponding to the connected Firebird version.

Info codes for individual areas are defined as separate enum types, e.g. `DbInfoCode` for connections or `TraInfoCode` for transactions. Each `InfoProvider` provides two functions: `supports()` and `get_info()`, which are passed the required info code. The first returns a boolean value indicating whether the given info code is supported, and the second returns the return value of the info call.

For convenience, InfoProviders also provide the most frequently used information as separate properties or methods that perform the corresponding `get_info()` call in the background. For information that is static in nature, such as ODS version, database page size, etc., the returned value is cached.

The driver currently does not support InfoProviders at the IResultSet and IBlob levels because important information is made available in a different way.

Firebird access libraries

Support in Firebird libraries varies considerably. The most important factor is specialization. Libraries designed to work with multiple database types (such as FireDAC, ODBC) offer only limited access to information available through info calls. Libraries specialized for working with Firebird (such as IBX for Lazarus) typically have a much better level of support.

The Christmas Temp Table Magic

With millions of presents to track, the elves needed a quick way to calculate delivery priorities. One clever elf suggested using a temporary table.

They created a Firebird TEMPORARY TABLE to store real-time data for each delivery zone. This allowed them to run lightning-fast calculations without impacting the main database.

Santa was so impressed with how quickly the sleigh's schedule was optimized that he renamed the process "Temp Table Tactics" and made it part of the official Christmas workflow.

Interview with Adriano dos Santos Fernandes

Adriano dos Santos Fernandes is a software developer from Brazil and a core member of the Firebird SQL development team. Over 20 years, Adriano has contributed significantly to areas such as internationalization, multi-byte character support, advanced SQL features, user-defined routines, and Java integration. His work also includes reviewing and improving the Firebird codebase.

To begin, could you share a bit of background about yourself? Where do you live, and what other interests do you have besides Firebird?

I live in a small city in São Paulo state in Brazil since I was born. I became interested in computers and programming when I was 9 years old (currently 41). My brother had contact with dBase III Plus and Clipper Summer 87 in a course (before the instructors disappeared) and in his work, and started to teach me as he was learning. At that time, we didn't have a computer at home and I had never used one . So I was writing small programs like calculator and tic-tac-toe game with a pencil in a paper. Something like a year later, he bought a computer and we were able to make my programs actually work after a few corrections. He was going to work in the afternoons, and allowed me to use his computer. I split my time on it to play games and learning more technologies, like Access, Visual Basic, Delphi, C/C++ and later Java.

While programming has become a profession for me, I still consider it a hobby too. I like to learn and do different things with technology.

Another of my hobbies is to drive go-karts. There is a good go-kart track in my city, and I participate in a championship there.

Although you weren't with Firebird from the very beginning, you joined fairly early—if I remember correctly, it was around 2004, when version 2.0 was in development. Could you share what led you to Firebird and how you became involved with the project?

Around 2000, I discovered Harbour, an open-source Clipper clone in development, and was immediately drawn to the open-source. I played a lot with its source code as I became very interested in compiler development, and also created some incomplete Clipper clones on my own. I never tried to contribute anything to Harbour directly, but I was using MinGW (GCC for Windows) which had an annoying problem that caused C++ exceptions to fail across DLL boundaries unless the MinGW DLL was used instead of the static library. So I learned things from GCC and MinGW, and was able to made the necessary modifications to make a fix, which was subsequently incorporated there.

A bit later I remember seeing some news about Interbase becoming open source which caught my attention, though I didn't dive into it right away. But later I started to use InterBase with Delphi, replacing the Paradox engine, and specially its transaction control impressed me. So I remembered about the open source news, switched to Firebird and became interested in its code.

I think it was around 2002 when I started to send emails to fb-devel regarding building the code, minor bugs and their potential fixes.

Your first major contribution to Firebird was a complete overhaul of international character-set handling. Since then, with numerous developments in this area, you've become the leading expert in internationalization and localization support for Firebird. Could you share some insights from this journey? What led you to choose this area initially, and what have been the biggest challenges over the years?

Some Brazilians reached out to me after noticing that I'm also Brazilian and active in the fb-devel group. They wanted to discuss their project for a PT_BR collation (Portuguese-Brazilian case-insensitive). However, their proposal wasn't accepted by the Firebird project because it was implemented in the simplest way possible, essentially as a quick hack. Although I was aware of their project and understood how crucial such a collation is for Brazil, I had never considered getting involved until then.

My knowledge in this area was basic, but it guided my involvement with Firebird, leading to many discussions, particularly with Nickolay Samofatov.

It was a very challenging project, as it required integrating a large external library (ICU) and making extensive changes to the core engine, as well as to existing character sets and collations. The process was time-consuming, and managing the integration using CVS branches added another layer of complexity.

The last major feature you developed in this area was the introduction of TIME/TIMESTAMP WITH TIME ZONE data types in Firebird 4. Do you feel that Firebird now has all the necessary tools for full internationalization, or is there something still missing that you'd like to implement?

I think it's now one of the most complete among DBMS, and there has never been significant demand for missing features like wide character sets (UTF-16).

While other core developers primarily work on Firebird's internal structures, services, protocols, and execution paths, your focus has been on adding and enhancing user-facing features, such as improving SQL standard support, introducing new built-in functions, and developing the SQL profiler. What led you to choose this direction?

I think this was driven by my interest in compiler technologies and the fact that others were focused on different areas. Interestingly, when I was a child and just starting to learn, compression and indexing technologies were my biggest interests. Yet, I never ended up working in those areas within Firebird.

I'm sure Firebird users appreciate your choice, as the number of features you've developed over the years is truly impressive. It's hard to say which one is the most important since each user has their own favorite. But which feature is your personal favorite, and why? Feel free to mention more than one if it's hard to choose just one.

Without going deep on all features I have developed, I'd say packages and subroutines. I consider isolated routines and its dependency management a nightmare.

Another favorite of mine isn't immediately visible: the extensive refactoring of the previous dsql_nod and jrd_nod structures. This effort transformed all internal SQL/PSQL-related structures into object-oriented and almost self-contained entities, significantly simplifying maintenance and enabling easier development of new features.

For Firebird 6, you've already completed several smaller features, such as the CALL statement, named arguments for function calls, and DROP [IF EXISTS]. The major new feature you're currently working on is SQL standard-compliant Schema support, which has been one of the most requested features by Firebird users. How is that progressing?

This is undoubtedly the most complex task I've undertaken. It affects every aspect where an object name is used: permissions, interfaces with external code and users, query aliases, execution plans, arrays, and both forward and backward compatibility. While many aspects are already complete, some tricky issues remain unresolved. Additionally, everything related to ISQL is still pending.

Most users have little idea of how Firebird development actually unfolds. Since you've been dedicated to long-term, full-time development, could you give them a glimpse behind the scenes? What does a typical day or week look like for you working on Firebird?

I'd say it depends on the task at hand. When working on smaller tasks, new valid issues and features tend to grab attention and get resolved quickly. However, when working on larger tasks, such as schemas, I tend to be much more selective about switching tasks.

You've been developing Firebird for twenty years now, which is an impressive career by any measure. Looking back on all those years, how do you feel about your journey with Firebird? Is there anything you would change, or has it been a completely satisfying experience?

It's satisfying but sometimes complicated. The code is complex and that is challenging. But people have their taste, ego and agenda and this is much more difficult to deal with in an open source project than in a company.

Brazil is undoubtedly the country with the largest number of Firebird users, and it has a very active community. As the only local Firebird core developer, I imagine your involvement in the community is highly sought after. Are you active in the local community, contributing to support forums, or attending conferences?

I was more involved in some mailing lists in the past, but that has decreased to almost zero. Written questions rarely provide all the necessary details to answer them well in one reply, which leads to protracted conversations. Nowadays, people are more interested in videos and conferences, which are not things I enjoy. Fortunately, there are others, like Alexey Kovyazin, Edson Gregório, and Carlos Cantu, who are doing a great job in that area.

Thanks for you time!

The Elf Who Loved Joins

Elf Dexter was a fan of complex SQL queries. For every task, he crafted intricate JOIN statements involving half a dozen tables. But one day, his query to find the "Top 10 Most Difficult Toys to Build" took hours to run.

Mrs. Claus stepped in to help. She examined Dexter’s query and noticed he hadn’t indexed the "ToyParts" table. After adding an index and simplifying some conditions, the query ran in seconds.

Dexter learned that while Firebird could handle his love for joins, optimization was key. "Indexes are a database’s best friend!" he said, now a wiser query writer.

Development update: 2024/Q4

A regular overview of new developments and releases in Firebird Project

Releases:

- Jaybird 6.0.0-beta-1, released 11.11.2024
- Jaybird 5.0.6, released 16.10.2024
- FirebirdClient 10.3.2.0, released 10.12.2024
- EF provider - 10.1.0.0, released 10.12.2024
- firebird-driver for Python 1.10.7, released 15.12.2024
- firebird-testcontainers-java 1.5.0, release 16.12.2024

The ROW type

Alexey Mochalov is finalizing the development of the SQL-compliant ROW data type and has presented a proposal describing the syntax and logic of this feature. The discussion is still ongoing, so it is too early to give a more detailed report.

Package constants

Red Soft recently implemented Package Constants and submitted a feature specification for discussion and approval before a pull request is created for Firebird 6. The discussion is still ongoing, so it is too early to give a more detailed report.

Tablespaces

Since the last discussion, it has been decided to add next functionality:

- Add the main database file to the `RDB$TABLESPACES` table, designated as `PRIMARY`.
- Add option to transfer tablespace to main database on restore.
- Add `TEMPORARY` predefined tablespaces.

The Pull Request is now in review.

Schema support (preliminary documentation)

Firebird 6.0 introduces support for schemas in the database. Schemas are not an optional feature, so every Firebird 6 database has at least a `SYSTEM` schema, reserved for Firebird system objects (**`RDB$` and `MON$`**).

User's objects live in different schemas, which may be the automatically created `PUBLIC` schema or user-defined ones. It's not allowed (with an exception about indexes) to create or change objects in the `SYSTEM` schema.

This documentation explains how schemas work in Firebird, how to use them and what may be different when migrating a database from previous versions to Firebird 6.

Why schemas

Schemas allow the logical grouping of database objects (such as tables, views and indexes), providing a clear structure to the database. Mostly frequently, they are used for two different purposes.

Schemas for database object organization

This usage of schemas helps in organizing database objects in a modular way, making the database easier to manage and maintain. By dividing the database into different schemas, developers and administrators can focus on specific areas of the database, improving teams scalability and reducing complexity.

For example, the `SYSTEM` schema is used with this purpose, separating objects created by two different groups (the Firebird DBMS core team and the Firebird users).

Firebird users may also want to organize objects in different schemas, for example, creating schemas like `FINANCE` and `MARKETPLACE` in the same database.

Schemas for data isolation

In multi-tenant applications, schemas can be leveraged to provide data isolation for different clients or tenants. By assigning a unique schema to each tenant, tables and others objects can be created with the same names in different schemas, making it more difficult to leak data and sometimes increasing the performance. Applications can then set the schema search path for the current selected customer.

This also simplifies database management and scaling, as each tenant's data is isolated within its schema, making maintenance, updates, and backups more straightforward.

Example schemas names could be `CUSTOMER_1` and `CUSTOMER_2`.

Schema-less and schema-bound objects

There are two categories of database objects: schema-less and schema-bound objects.

Some database objects live outside of schemas, and for them, everything works as before. They are: Users, Roles, Blob filters and Schemas.

The others are now always contained in a schema: Tables, Views, Triggers, Procedures, Exceptions, Domains, Indexes, Character sets, Sequences / generators, Functions, Collations and Packages

Some objects are highly dependent on their parents, like table-based triggers and indexes depending on the table. In this case the child object always lives in the same schema of its parent.

Search path

Every Firebird session has a search path, which is a list of schemas that is used to resolve not-qualified object names. By default, the initial search path is `PUBLIC, SYSTEM`.

This default can be changed using `isc_dpb_search_path` in the API or later changed using the `SET SEARCH_PATH TO` statement.

`ALTER SESSION RESET` resets the search path to the initial search path, i.e. the one passed to `isc_dpb_search_path` or the default (`PUBLIC, SYSTEM`).

Non-existing schemas may be present in the search path, and in this case, they are not considered.

The first existing schema in the search path is called the `current schema` and is exclusively used in some operations.

Bind of unqualified objects to a schema generally happens at statement prepare time. An exception to this rule is `MAKE_DBKEY` function with expression (not simple literal) as first argument, making it resolve the table at execution time.

Object names may be now qualified with the schema name, for example `SCHEMA_NAME.TABLE_NAME`, `SCHEMA_NAME.TABLE_NAME.COLUMN_NAME`, SCHEMA_NAME.PACKAGE_NAME.PROCEDURE_NAME. But the schema qualifier is optional. And here is where the search path is used, and it is used in different ways depending on where the unqualified name appears.

With CREATE, CREATE OR ALTER and RECREATE statements, an existing object is searched only in the current schema (the first valid schema present in the search path) and the new object is created in this same schema. That same rule is also used with GRANT and REVOKE statements related to DDL operations without using ON SCHEMA sub clause. If there is no current schema (no valid schema in the search path), an error will be raised.

Examples using this rule:

```
create table TABLE1 (ID integer);
recreate table TABLE1 (ID integer);
create or alter function F1 returns integer as begin end;
grant create table to user USER1;
```

With ALTER, DROP and others statements, an existing object is searched in all schemas present in the search path, and the reference is bound to the first one found, or an error is raised.

Examples using this rule:

```
alter table TABLE1 add X integer;
alter function FUNCTION1 returns integer as begin end;
select * from TABLE1;
```

There is also a difference in relation to search paths in DML vs DDL statements.

With DML statements, the search path is used to find all the referenced unqualified objects. For example:

```
insert into TABLE1 values (1);

execute block returns (out DOMAIN1)
as
begin    select val from TABLE2 into out;
end;
```

In this case, the search path is used to find `TABLE1`, `DOMAIN1` and `TABLE2`.

With DDL statements, it is actually used to search in the same way, but the search path is implicitly temporarily changed just after the object being created/changed is bound to a schema when preparing the statement. The change makes the search path equal to the schema of the object followed by the `SYSTEM` schema.

For example:

```
create schema SCHEMA1;
create schema SCHEMA2;

create domain SCHEMA1.DOMAIN1 integer;

-- DOMAIN1 is bound to SCHEMA1 even without it being in the search path,
-- as the table being created is bound to SCHEMA1
create table SCHEMA1.TABLE1 (id DOMAIN1);

set search_path to SCHEMA2, SCHEMA1;
-- Error: even if SCHEMA1 is in the search path,
-- TABLE2 is bound to SCHEMA2, so DOMAIN1 is searched in
-- SCHEMA2 and SYSTEM schemas
create table TABLE2 (id DOMAIN1);

set search_path to SYSTEM;

create procedure SCHEMA1.PROC1
as
begin
    -- TABLE1 is bound to SCHEMA1 as PROC1
    insert into TABLE1 values (1);
end;
```

Resolving between PACKAGE.OBJECT and SCHEMA.OBJECT

There is now an ambiguity in the syntax `<name>.<name>` between `<package>.<object>` and `<schema>.<object>` when referring to procedures and functions.

In this case it first looks for a package using the search path, and if it exists, bound its name treating the expression as `<package>.<object>` in the found `<schema>`.

If the package is not found, then the name is treated as an already qualified name (`<schema>.<object>`).

Permissions

Permissions to control and use schema-bound objects are now influenced by the schema permissions.

A schema, like other objects, has an owner. Its owner can manipulate and use any object in the schema, even the objects created by others users in that schema.

To manipulate objects in a schema from another user, a user needs DDL permissions. DDL permissions already existed in previous versions, but now they are more fine-grained, like in these examples:

```
grant create table on schema SCHEMA1 to user USER1;
grant alter any procedure on schema SCHEMA1 to PUBLIC;
```

`ON SCHEMA <name>` clause is optional, and if not present, it is implicitly assumed to be the `current schema`.

To use objects it was already necessary to have permissions like `EXECUTE` or `USAGE` granted for the object. Now, in addition to that, it is necessary to have granted the `USAGE` permission for the schema where the object is contained, like in this example:

```
-- Connected as USER1
create schema SCHEMA1;
create table SCHEMA1.TABLE1 (ID integer);

grant usage on schema SCHEMA1 to user USER2;
grant select on table SCHEMA1.TABLE1 to user USER2;
```

The SYSTEM schema

All system schema-bound objects (**`RDB$` and `MON$`**) are now created in a special schema called `SYSTEM`. As the `SYSTEM` schema has a default `USAGE` permission granted to `PUBLIC` and by default is present in the search path, its usage is backward compatible with previous versions.

With the exception of index creation and manipulation of these created indexes, the `SYSTEM` schema is locked for changes. However, it is not recommended to manipulate objects there.

The PUBLIC schema

A schema called `PUBLIC` is automatically created in new databases, with a default `USAGE` permission granted to `PUBLIC`. Only the database/schema owner has default permissions to manipulate objects in that schema.

The `PUBLIC` schema is not a system object and it can even be dropped by the database owner or by a user with `DROP ANY SCHEMA` permission. When restoring a backup of Firebird >= 6 with `gbak` where the `PUBLIC` schema was not present, the restored database will be created without it.

New statements and expressions

CREATE SCHEMA

```
{CREATE [IF NOT EXISTS] | CREATE OR ALTER | RECREATE} SCHEMA <schema name>
    [DEFAULT CHARACTER SET <character set name>]
```

If `DEFAULT CHARACTER SET` is not specified, new schemas are created with the default character set equal to the database default character set.

The default character set is fine-grained, so each schema can have a different default character set. When previous Firebird versions used the default database character set, now it uses the default schema character set of the contained object.

Different than the automatically created `PUBLIC` schema, the newly created schema has `USAGE` permission granted only to its owner and not to `PUBLIC`.

Schema names `INFORMATION_SCHEMA` and `DEFINITION_SCHEMA` are reserved and cannot be created.

ALTER SCHEMA

```
ALTER SCHEMA <schema name>
    [SET DEFAULT CHARACTER SET <character set name>]
```

DROP SCHEMA

```
DROP SCHEMA [IF EXISTS] <schema name>
```

It is currently allowed to drop only empty schemas. In the future the `CASCADE` sub clause will be added allowing to drop schema and all its contained objects.

CURRENT_SCHEMA

`CURRENT_SCHEMA` returns the first valid schema name present in the search path of the current session. If there is none, it returns `NULL`.

SET SEARCH_PATH TO

```
SET SEARCH_PATH TO <schema name> [, <schema name>]...
```

RDB$GET_CONTEXT

CURRENT_SCHEMA (SYSTEM)

`RDB$GET_CONTEXT('SYSTEM', 'CURRENT_SCHEMA')` returns the same value as the `CURRENT_SCHEMA` expression.

SEARCH_PATH (SYSTEM)

`RDB$GET_CONTEXT('SYSTEM', 'SEARCH_PATH')` returns the current session search path, including invalid schemas present in the list.

SCHEMA_NAME (DDL_TRIGGER)

`RDB$GET_CONTEXT('DDL_TRIGGER', 'SCHEMA_NAME')` returns the schema name of the affected object in a DDL TRIGGER.

Monitoring

Monitoring tables has now these information related to schemas:

`MON$ATTACHMENTS`

- `MON$SEARCH_PATH`: search path of the attachment

`MON$TABLE_STATS`

- `MON$SCHEMA_NAME`: table schema

`MON$CALL_STACK`

- `MON$SCHEMA_NAME`: routine schema

`MON$COMPILED_STATEMENTS`

- `MON$SCHEMA_NAME`: routine schema

Queries

Field aliases can now be qualified not only with the table name, but with the schema too, even in the case of implicit deduced schema name from the search path. It is also possible to qualify table name with schema and use the column only with the table name. For example:

```
create schema SCHEMA1;

create table SCHEMA1.TABLE1 (ID integer);
```

```
set search_path to SCHEMA1;

select TABLE1.ID from SCHEMA1.TABLE1;
select SCHEMA1.TABLE1.ID from TABLE1;
select SCHEMA1.TABLE1.ID from SCHEMA1.TABLE1;
```

If the same table name is used from different schemas, fields should be qualified with the schema names or aliases:

```
create schema SCHEMA1;
create schema SCHEMA2;

create table SCHEMA1.TABLE1 (ID integer);
create table SCHEMA2.TABLE1 (ID integer);

select SCHEMA1.TABLE1.ID, SCHEMA2.TABLE1.ID from SCHEMA1.TABLE1,
SCHEMA2.TABLE1;
select S1.ID, S2.ID from SCHEMA1.TABLE1 S1, SCHEMA2.TABLE1 S2;
```

Plans

To be specified.

New DPB items

isc_dpb_search_path

`isc_dpb_search_path` is a string DPB (like `isc_dpb_user_name`) that sets the initial schema search path.

Array support

isc_sdl_schema

When dealing with arrays using SDL (Array Slice Description Language), there is now `isc_sdl_schema` to explicitly qualify the schema. Its format is the same as used with `isc_sdl_relation`.

Utilities

isql

Option -(SE)ARCH_PATH

This option makes ISQL pass the search path (as received by the OS) as `isc_dpb_search_path` in every attachment stablished.

```
isql -search_path x,y t1.fdb
select RDB$GET_CONTEXT('SYSTEM', 'SEARCH_PATH') from system.rdb$database;
-- Result: "X", "Y"
set search_path to y;
select RDB$GET_CONTEXT('SYSTEM', 'SEARCH_PATH') from system.rdb$database;
-- Result: "Y"

connect 't2.fdb';
select RDB$GET_CONTEXT('SYSTEM', 'SEARCH_PATH') from system.rdb$database;
-- Result: "X", "Y"
```

```
isql -search_path '"x", "y"' t1.fdb
select RDB$GET_CONTEXT('SYSTEM', 'SEARCH_PATH') from system.rdb$database;
-- Result: "x", "y"
```

gbak

To use databases created in previous Firebird versions with Firebird 6, it is necessary to restore a backup in the new version using the Firebird 6 `gbak`. The restored database will have all users objects in the `PUBLIC` schema.

NOT DECIDED YET: Should we add an option to give customized name to the `PUBLIC` schema?

System metadata changes

The following fields were added to system tables. It is important that applications and tools that read metadata starts to use them when appropriate, for example considering `RDB$SCHEMA_NAME` when joining the tables.

Table: Column(s)

- MON$ATTACHMENTS: MON$SEARCH_PATH
- MON$CALL_STACK: MON$SCHEMA_NAME
- MON$COMPILED_STATEMENTS: MON$SCHEMA_NAME
- MON$TABLE_STATS: MON$SCHEMA_NAME
- RDB$CHARACTER_SETS: RDB$DEFAULT_COLLATE_SCHEMA_NAME, RDB$SCHEMA_NAME
- RDB$CHECK_CONSTRAINTS: RDB$SCHEMA_NAME
- RDB$COLLATIONS: RDB$SCHEMA_NAME
- RDB$DATABASE: RDB$CHARACTER_SET_SCHEMA_NAME
- RDB$DEPENDENCIES: RDB$DEPENDED_ON_SCHEMA_NAME, RDB$DEPENDENT_SCHEMA_NAME
- RDB$EXCEPTIONS: RDB$SCHEMA_NAME
- RDB$FIELDS: RDB$SCHEMA_NAME
- RDB$FIELD_DIMENSIONS: RDB$SCHEMA_NAME
- RDB$FUNCTIONS: RDB$SCHEMA_NAME
- RDB$FUNCTION_ARGUMENTS: RDB$FIELD_SOURCE_SCHEMA_NAME, RDB$RELATION_SCHEMA_NAME, RDB$SCHEMA_NAME
- RDB$GENERATORS: RDB$SCHEMA_NAME
- RDB$INDEX_SEGMENTS: RDB$SCHEMA_NAME
- RDB$INDICES: RDB$FOREIGN_KEY_SCHEMA_NAME, RDB$SCHEMA_NAME
- RDB$PACKAGES: RDB$SCHEMA_NAME
- RDB$PROCEDURES: RDB$SCHEMA_NAME
- RDB$PROCEDURE_PARAMETERS: RDB$FIELD_SOURCE_SCHEMA_NAME, RDB$RELATION_SCHEMA_NAME, RDB$SCHEMA_NAME
- RDB$PUBLICATION_TABLES: RDB$TABLE_SCHEMA_NAME
- RDB$REF_CONSTRAINTS: RDB$CONST_SCHEMA_NAME_UQ, RDB$SCHEMA_NAME
- RDB$RELATIONS: RDB$SCHEMA_NAME
- RDB$RELATION_CONSTRAINTS: RDB$SCHEMA_NAME
- RDB$RELATION_FIELDS: RDB$FIELD_SOURCE_SCHEMA_NAME, RDB$SCHEMA_NAME
- RDB$SCHEMAS: RDB$CHARACTER_SET_NAME, RDB$CHARACTER_SET_SCHEMA_NAME, RDB$DESCRIPTION, RDB$OWNER_NAME, RDB$SCHEMA_NAME, RDB$SECURITY_CLASS, RDB$SYSTEM_FLAG
- RDB$TRIGGERS: RDB$SCHEMA_NAME
- RDB$TRIGGER_MESSAGES: RDB$SCHEMA_NAME
- RDB$USER_PRIVILEGES: RDB$RELATION_SCHEMA_NAME, RDB$USER_SCHEMA_NAME
- RDB$VIEW_RELATIONS: RDB$RELATION_SCHEMA_NAME, RDB$SCHEMA_NAME

Differences with previous versions

CREATE SCHEMA in IAttachment::executeCreateDatabase and isc_create_database

It was possible to use `CREATE SCHEMA` using the APIs functions `IAttachment::executeCreateDatabase` and `isc_create_database` to create databases. Not this is not allowed. The only valid syntax is `CREATE DATABASE`.

Object names in error messages

Object names present in error or informative messages are now qualified and quoted in the parameters of the messages, even for DIALECT 1 databases. For example:

```
SQL> create table TABLE1 (ID integer);
SQL> create table TABLE1 (ID integer);
Statement failed, SQLSTATE = 42S01
unsuccessful metadata update
-CREATE TABLE "PUBLIC"."TABLE1" failed
-Table "PUBLIC"."TABLE1" already exists

SQL> create schema "Weird ""Schema""";
SQL> create schema "Weird ""Schema""";
Statement failed, SQLSTATE = 42000
unsuccessful metadata update
-CREATE SCHEMA "Weird ""Schema""" failed
-Schema "Weird ""Schema""" already exists
```

Object name parsing outside SQL

When dealing with object names in `isc_dpb_search_path`, `isc_sdl_schema` and `MAKE_DBKEY`, object names follow the same rules as when using in SQL, requiring quotes for names with special or lower case characters. For `MAKE_DBKEY`, unqualified names use the search path.

Previously, `MAKE_DBKEY` used exact table name inside its first parameter and do not allowed usage of double-quotes for special characters.

Minimum page size

The minimal database page size was increased from 4096 to 8192 because the old minimum was not enough to fit changes in system indexes.

Bult-in plugins

To be specified.

Downgrade compatibility

It’s expected that Firebird 6 databases not using multiple users schemas (for example, a Firebird 5 database just migrated to Firebird 6 by `gbak`) would not be always downgradable to previous Firebird versions using `gbak`.

The Firebird team will backport to Firebird 5 essential internal changes to make that possible. This documentation will be updated when this is ready.

The Snowflake Database Feud

One day, a Snowflake database strutted into the North Pole's IT room, boasting about its scalability and cloud-first design. Firebird, sitting quietly in the corner, chuckled.

"Impressive," Firebird said, "but can you deliver a seamless Christmas with minimal hardware and decades of reliability?"

Snowflake stammered, "Well, I... uh, need an internet connection..."

Santa overheard and chimed in, "Firebird has never let us down. Christmas is too important to trust the clouds alone!"

Firebird winked at Snowflake, saying, "Welcome to the North Pole. We've got room for everyone, but some traditions never fail."

Toolbox: IBQuery

The IBQuery from MiTec is a simple tool for working with InterBase/Firebird databases on the Windows platform. It has been around since 2002, and has been quite popular in its category due to its simplicity and user-friendly design.

IBQuery is a tool primarily intended for database administrators. In addition to standard functions such as Database Object Explorer for metadata visualization, SQL Editor for executing commands and scripts, it also offers specific administrator functions such as User and Grant Manager, Event Watcher or Performance Monitor. It is built in Delphi and uses IBX components to work with Firebird. It's distributed as single 7MB .zip file that contains 32-bit and 64-bit executables, and language files for English, German, French, Spanish, Hungarian and Czech localization.

IBQuery is free to use for private, educational and non-commercial purposes, and for other usage you should buy commercial license. Also, some functionality is available only in commercial version.

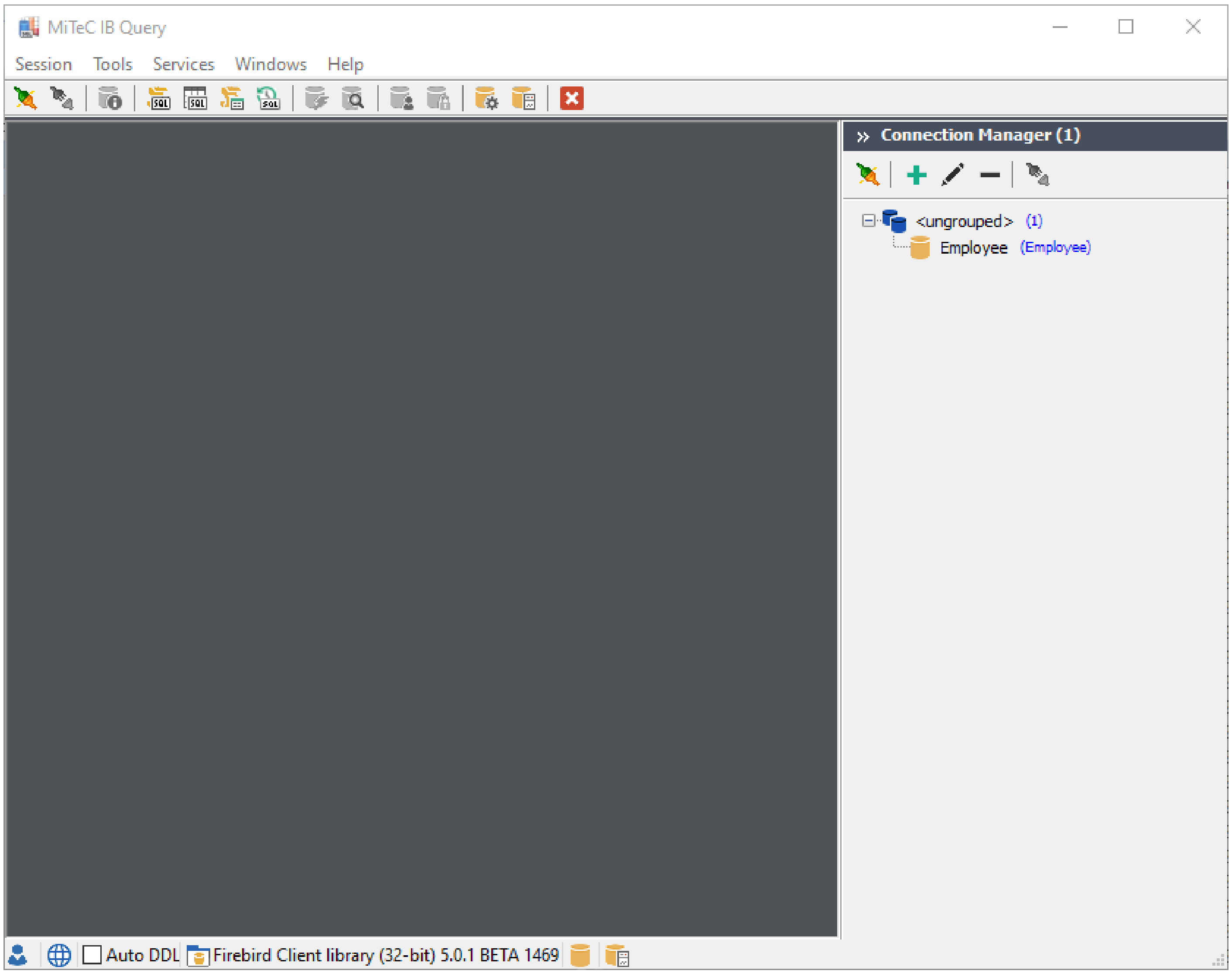
For this review, we used the latest commercial version 9.5.0, provided by MiTec.

IBQuery features a no-frills, single-window layout that prioritizes functionality and ease of use. The interface is divided into three main sections:

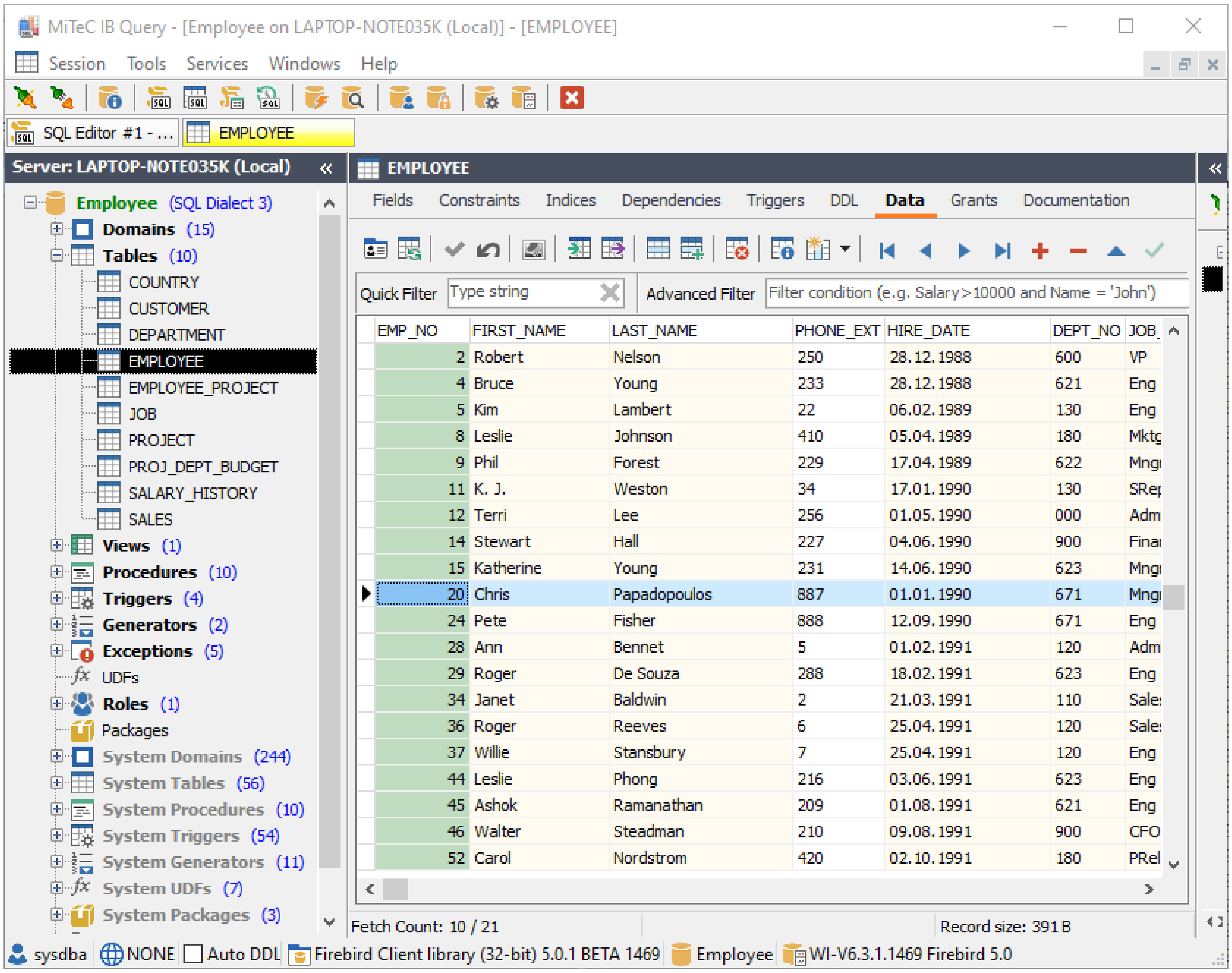
- Collapsible panels on either side, which house the Object Explorer (left) and Connection Manager (right).
- A central workspace dedicated to tool windows, such as query editors and data views.
- A switching bar below the toolbar that lists open windows for easy navigation.

The design is utilitarian, which longtime users may appreciate, but it can feel dated compared to the polished interfaces of contemporary database tools. However, its simplicity ensures that new users can get up to speed quickly without needing extensive documentation or training.

At startup, the right panel is displayed, initially empty on the first launch, showing the defined connections.



Upon connecting to the database, the Database Object Explorer is displayed in a standard tree structure. However, it lacks filtering and search capabilities, which can make navigating databases with a large number of objects less efficient. Selecting an object opens a dedicated working window that provides detailed information about the selected item. For tables, a standard data view is available, offering options for data modification, as well as export and import functionalities.


MiTeC IB Query - [Employee on LAPTOP-NOTE035K (Local)] - [EMPLOYEE]
Session Tools Services Windows Help
SQL Editor #1 - ...
EMPLOYEE
Server: LAPTOP-NOTE035K (Local)
EMPLOYEE
Fields Constraints Indices Dependencies Triggers DDL Data Grants Documentation
Quick Filter
Type string
Advanced Filter
Filter condition (e.g. Salary>10000 and Name = 'John')

Employee (SQL Dialect 3)
- Domains (15)
- Tables (10)
  - COUNTRY
  - CUSTOMER
  - DEPARTMENT
  - EMPLOYEE
  - EMPLOYEE_PROJECT
  - JOB
  - PROJECT
  - PROJ_DEPT_BUDGET
  - SALARY_HISTORY
  - SALES
- Views (1)
- Procedures (10)
- Triggers (4)
- Generators (2)
- Exceptions (5)
- UDFs
- Roles (1)
- Packages
- System Domains (244)
- System Tables (56)
- System Procedures (10)
- System Triggers (54)
- System Generators (11)
- System UDFs (7)
- System Packages (3)

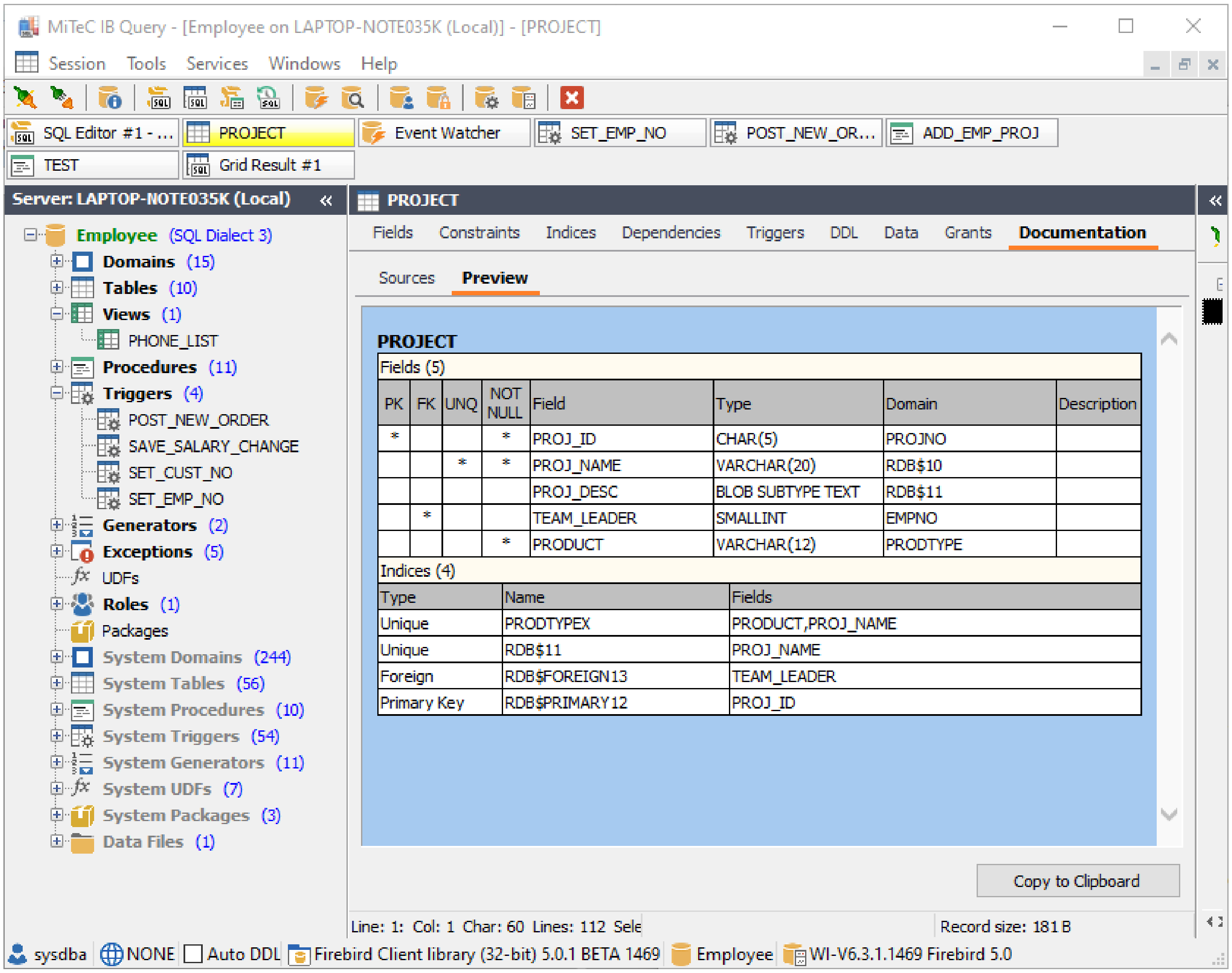
EMP_NO	FIRST_NAME	LAST_NAME	PHONE_EXT	HIRE_DATE	DEPT_NO	JOB_
2	Robert	Nelson	250	28.12.1988	600	VP
4	Bruce	Young	233	28.12.1988	621	Eng
5	Kim	Lambert	22	06.02.1989	130	Eng
8	Leslie	Johnson	410	05.04.1989	180	Mktg
9	Phil	Forest	229	17.04.1989	622	Mng
11	K. J.	Weston	34	17.01.1990	130	SRep
12	Terri	Lee	256	01.05.1990	000	Adm
14	Stewart	Hall	227	04.06.1990	900	Fina
15	Katherine	Young	231	14.06.1990	623	Mng
20	Chris	Papadopoulos	887	01.01.1990	671	Mng
24	Pete	Fisher	888	12.09.1990	671	Eng
28	Ann	Bennet	5	01.02.1991	120	Adm
29	Roger	De Souza	288	18.02.1991	623	Eng
34	Janet	Baldwin	2	21.03.1991	110	Sale
36	Roger	Reeves	6	25.04.1991	120	Sale
37	Willie	Stansbury	7	25.04.1991	120	Eng
44	Leslie	Phong	216	03.06.1991	623	Eng
45	Ashok	Ramanathan	209	01.08.1991	621	Eng
46	Walter	Steadman	210	09.08.1991	900	CFO
52	Carol	Nordstrom	420	02.10.1991	180	PRel

Fetch Count: 10 / 21   Record size: 391 B

sysdba | NONE | Auto DDL | Firebird Client library (32-bit) 5.0.1 BETA 1469 | Employee | WI-V6.3.1.1469 Firebird 5.0

The data view, implemented as a standard Delphi dataset, has a minor drawback: the transaction control icons only activate after data in the grid has been modified. This makes it less convenient for inspecting data from monitoring tables, as their contents cannot be easily refreshed. For monitoring tables, it is recommended to use the SQL Editor, which does not have this limitation.

The object documentation panel offers an overview of key properties that are otherwise spread across individual panels. It also provides access to the HTML source code of a table. Unfortunately, this documentation is limited to individual objects and does not extend to the entire database, object categories, or selected groups of objects.



PROJECT

Fields (5)							
PK	FK	UNQ	NOT NULL	Field	Type	Domain	Description
*			*	PROJ_ID	CHAR(5)	PROJNO	
		*	*	PROJ_NAME	VARCHAR(20)	RDB$10	
				PROJ_DESC	BLOB SUBTYPE TEXT	RDB$11	
	*			TEAM_LEADER	SMALLINT	EMPNO	
			*	PRODUCT	VARCHAR(12)	PRODTYPE	

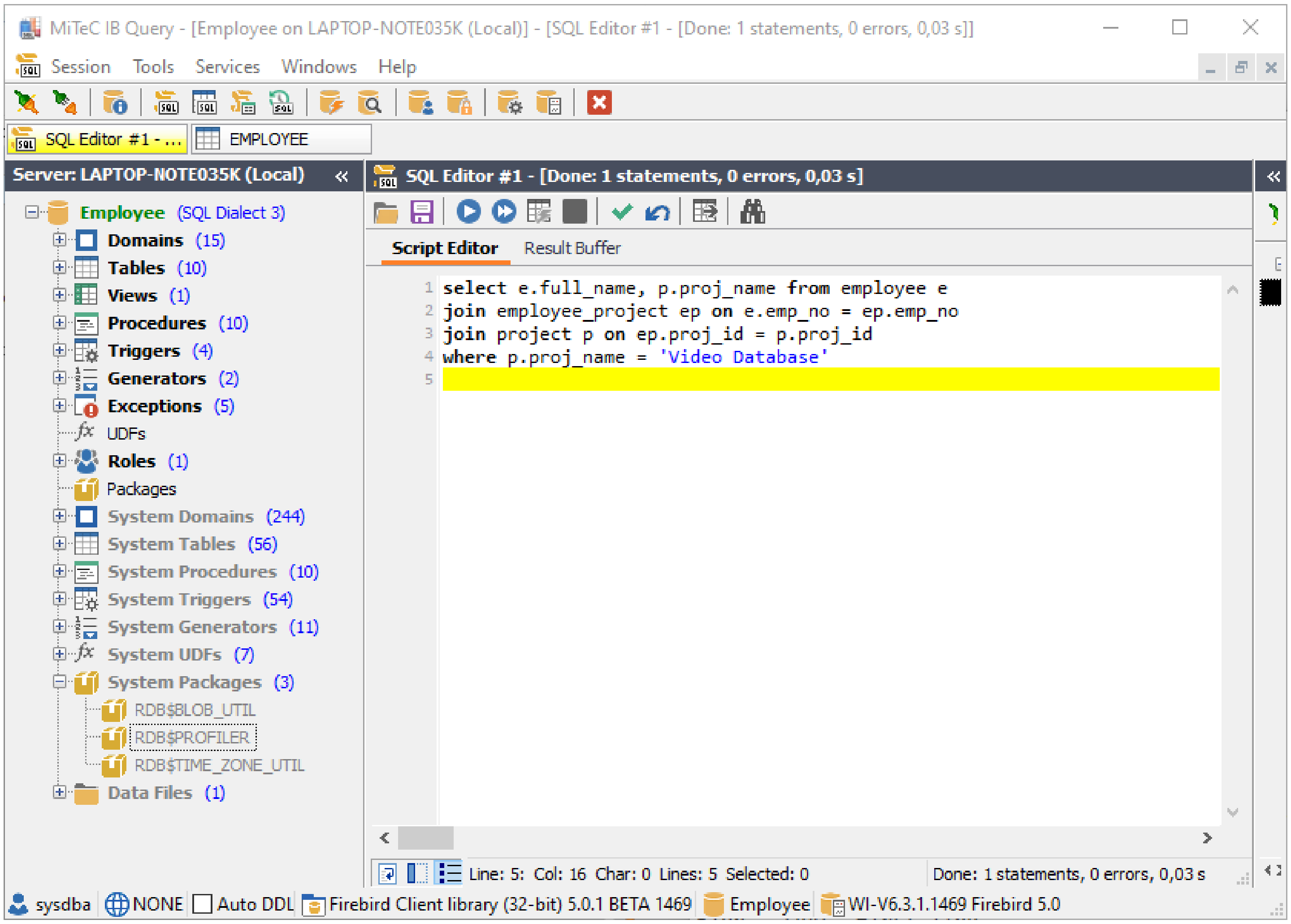
Indices (4)		
Type	Name	Fields
Unique	PRODTYPEX	PRODUCT,PROJ_NAME
Unique	RDB$11	PROJ_NAME
Foreign	RDB$FOREIGN13	TEAM_LEADER
Primary Key	RDB$PRIMARY12	PROJ_ID

Line: 1: Col: 1 Char: 60 Lines: 112 Sele

Record size: 181 B

The SQL editor allows the execution of both individual commands and entire scripts, offering two modes: one that generates query output and another for running scripts without output.

While functional, the editor is very basic, lacking advanced features such as command completion, even for object names. Syntax highlighting is also minimal, which is disappointing given that both features are now standard in most modern editors.

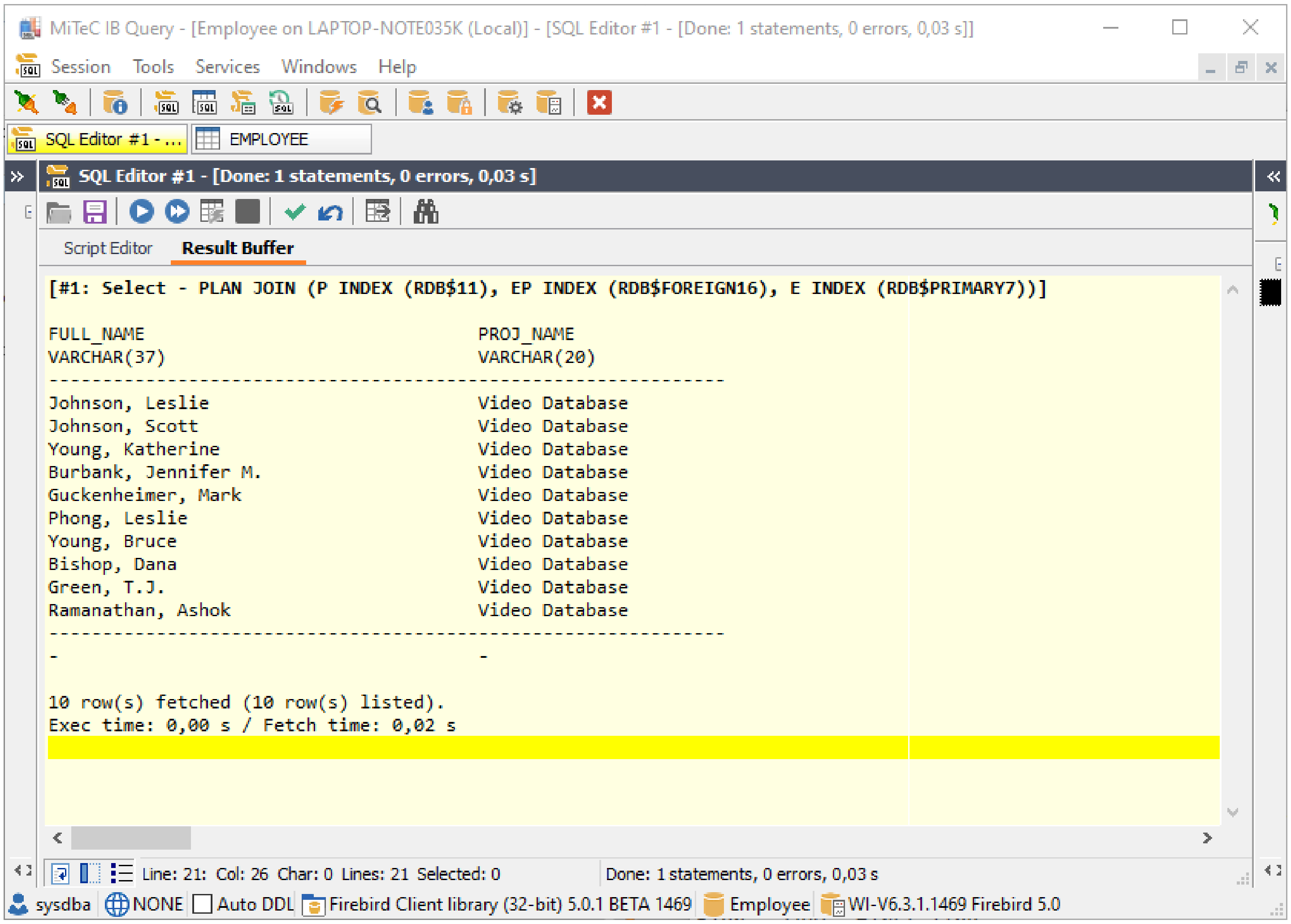


```
select e.full_name, p.proj_name from employee e
join employee_project ep on e.emp_no = ep.emp_no
join project p on ep.proj_id = p.proj_id
where p.proj_name = 'Video Database'
```

By default, query results are displayed as text output rather than in a grid, with the grid view available as a separate function. This solution has its undeniable advantages. Although detailed statistics are not displayed, essential information such as the execution plan, execution time, and result retrieval time is provided. Similar to ISQL, both row-by-row and column-by-column views are supported. Additionally, results can be exported in various formats.

Regarding data handling, a significant limitation of IBQuery must be noted. Even in its latest version, it does not natively support the new data types introduced in Firebird 4. To work with these new types without encountering errors, users must rely on the SET BIND command to configure data type coercion rules.

Resources: IBQuery download

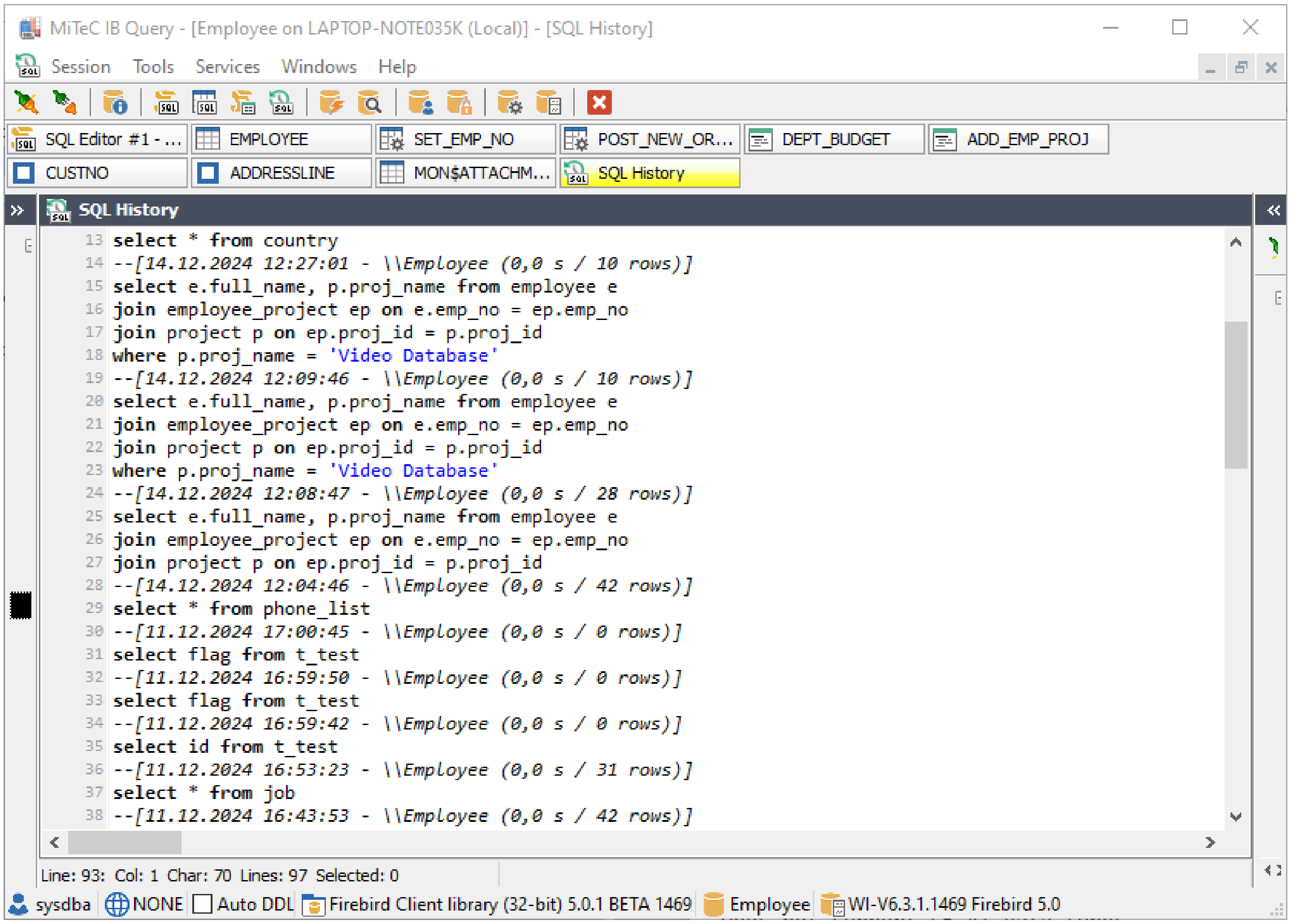

MiTeC IB Query - [Employee on LAPTOP-NOTE035K (Local)] - [SQL Editor #1 - [Done: 1 statements, 0 errors, 0,03 s]]
Session Tools Services Windows Help
SQL Editor #1 - ...
EMPLOYEE
SQL Editor #1 - [Done: 1 statements, 0 errors, 0,03 s]
Script Editor
Result Buffer

```
[#1: Select - PLAN JOIN (P INDEX (RDB$11), EP INDEX (RDB$FOREIGN16), E INDEX (RDB$PRIMARY7))]

FULL_NAME                               PROJ_NAME
VARCHAR(37)                             VARCHAR(20)
---------------------------------------------------------------
Johnson, Leslie                         Video Database
Johnson, Scott                          Video Database
Young, Katherine                        Video Database
Burbank, Jennifer M.                    Video Database
Guckenheimer, Mark                      Video Database
Phong, Leslie                           Video Database
Young, Bruce                            Video Database
Bishop, Dana                            Video Database
Green, T.J.                             Video Database
Ramanathan, Ashok                       Video Database
---------------------------------------------------------------
-                                       -

10 row(s) fetched (10 row(s) listed).
Exec time: 0,00 s / Fetch time: 0,02 s
```

Line: 21: Col: 26 Char: 0 Lines: 21 Selected: 0
Done: 1 statements, 0 errors, 0,03 s
sysdba
NONE
Auto DDL
Firebird Client library (32-bit) 5.0.1 BETA 1469
Employee
WI-V6.3.1.1469 Firebird 5.0

Efficient SQL editing hinges on a robust command history function. In contrast to many other tools, this feature exists independently as a commented text file. Users can select specific commands within this file and transfer them to the editor with a dedicated function. However, the history currently logs all executed commands sequentially, irrespective of duplicates. This approach can diminish the history's clarity and usability.

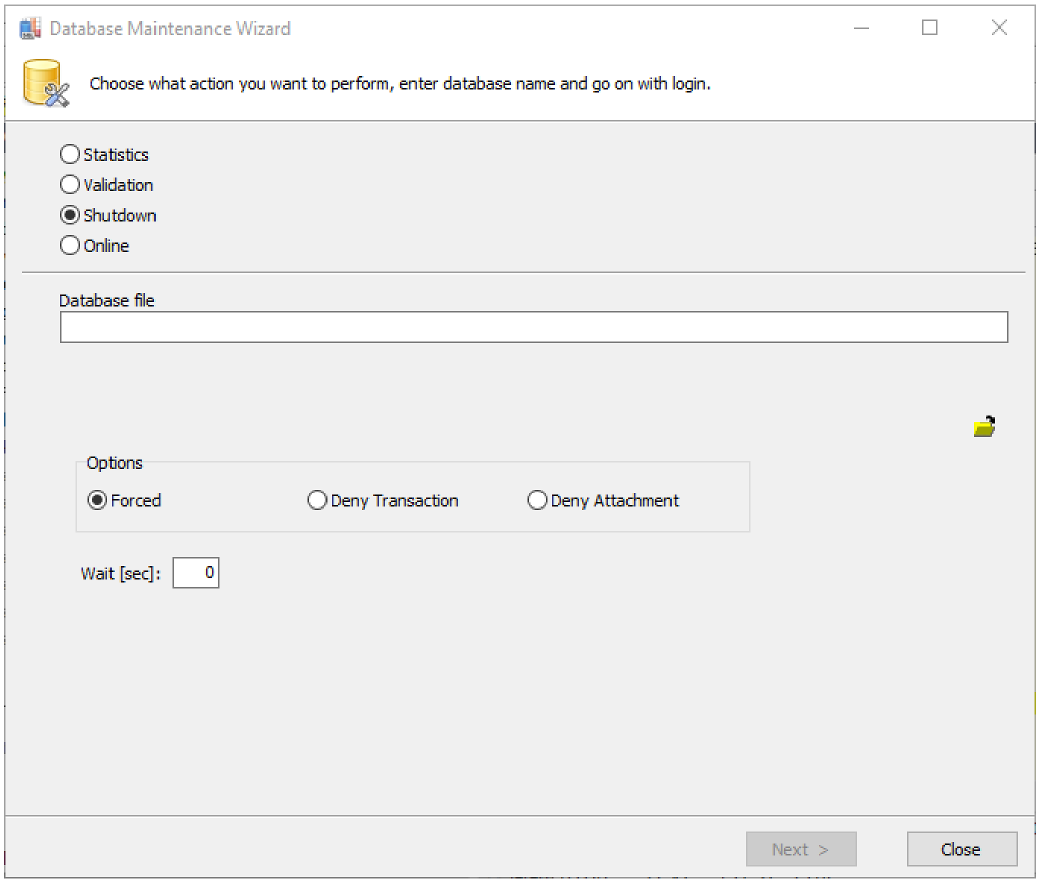
In addition to the above, the free version also offers User and Grant Manager functionality and access to some server Services. The User Manager uses the Services interface, allowing only basic user management without more advanced options available via SQL user-management commands. The Grant manager is nice, but supports only standard privileges, and not new DDL ones. Supported services include access to the server log, viewing basic server information, detailed gstat statistics as plain text output, online validation, and the ability to shut down the database for maintenance purposes and then make it online again.


MiTeC IB Query - [Employee on LAPTOP-NOTE035K (Local)] - [SQL History]
Session Tools Services Windows Help
SQL Editor #1 - ... EMPLOYEE SET_EMP_NO POST_NEW_OR... DEPT_BUDGET ADD_EMP_PROJ
CUSTNO ADDRESSLINE MON$ATTACHM... SQL History
SQL History

```
select * from country
--[14.12.2024 12:27:01 - \\Employee (0,0 s / 10 rows)]
select e.full_name, p.proj_name from employee e
join employee_project ep on e.emp_no = ep.emp_no
join project p on ep.proj_id = p.proj_id
where p.proj_name = 'Video Database'
--[14.12.2024 12:09:46 - \\Employee (0,0 s / 10 rows)]
select e.full_name, p.proj_name from employee e
join employee_project ep on e.emp_no = ep.emp_no
join project p on ep.proj_id = p.proj_id
where p.proj_name = 'Video Database'
--[14.12.2024 12:08:47 - \\Employee (0,0 s / 28 rows)]
select e.full_name, p.proj_name from employee e
join employee_project ep on e.emp_no = ep.emp_no
join project p on ep.proj_id = p.proj_id
--[14.12.2024 12:04:46 - \\Employee (0,0 s / 42 rows)]
select * from phone_list
--[11.12.2024 17:00:45 - \\Employee (0,0 s / 0 rows)]
select flag from t_test
--[11.12.2024 16:59:50 - \\Employee (0,0 s / 0 rows)]
select flag from t_test
--[11.12.2024 16:59:42 - \\Employee (0,0 s / 0 rows)]
select id from t_test
--[11.12.2024 16:53:23 - \\Employee (0,0 s / 31 rows)]
select * from job
--[11.12.2024 16:43:53 - \\Employee (0,0 s / 42 rows)]
```

Line: 93: Col: 1 Char: 70 Lines: 97 Selected: 0

sysdba NONE Auto DDL Firebird Client library (32-bit) 5.0.1 BETA 1469 Employee WI-V6.3.1.1469 Firebird 5.0



Choose what action you want to perform, enter database name and go on with login.

Statistics
Validation
[✓ Shutdown]
Online

Database file

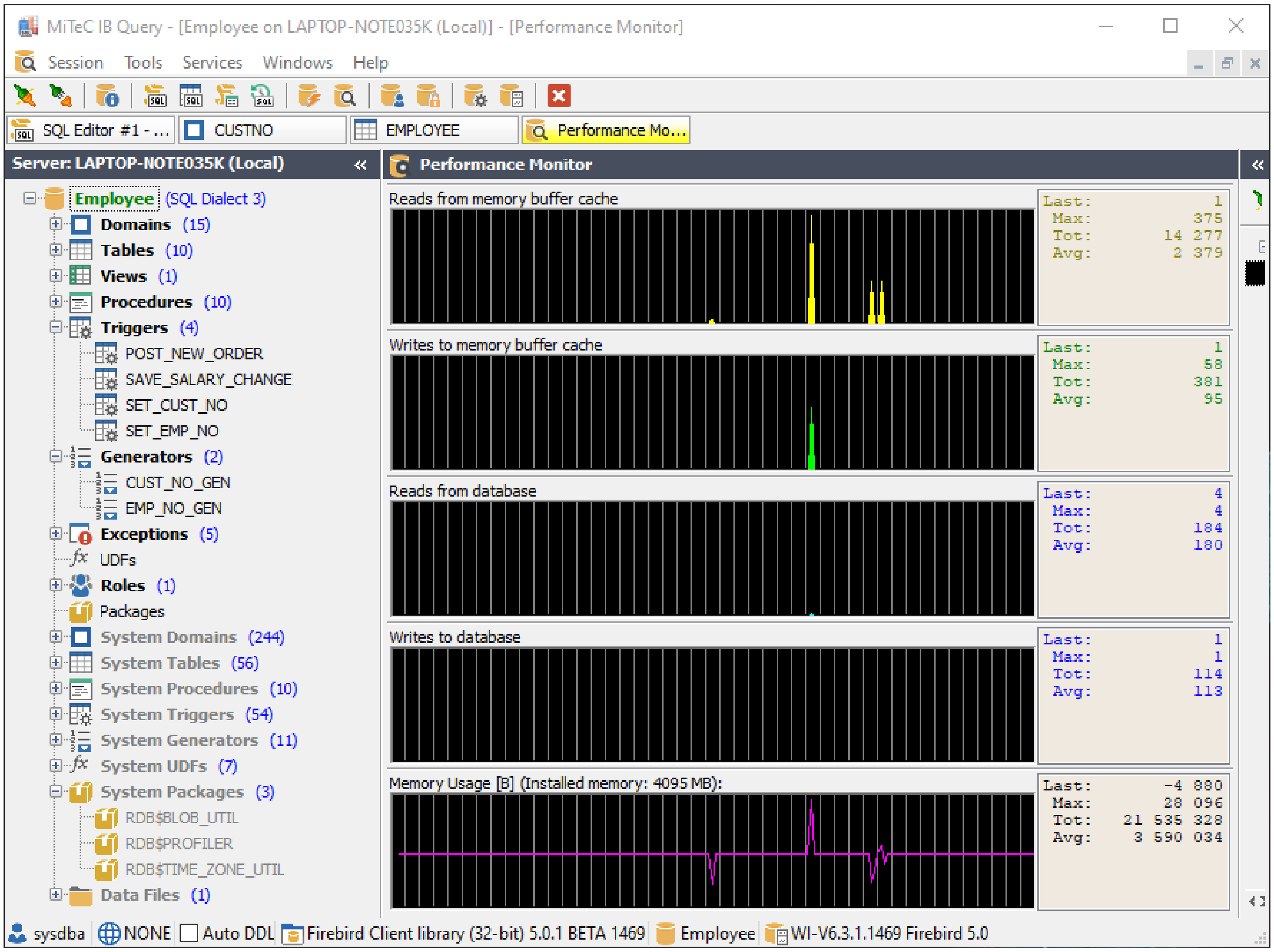
Options

[✓ Forced] Deny Transaction Deny Attachment

Wait [sec]: 0

The commercial version provides enhanced capabilities, including a Performance Monitor, Event Watcher, Index Rebuilders, data import/export functionality, and a script repository known as Codebase.

The Performance Monitor visualizes I/O statistics and memory usage over time through intuitive trend graphs. It gathers data using lightweight info calls, minimizing the performance impact on the server compared to systems that rely on monitoring tables. However, this approach also restricts its scope, as it is limited to data within a single database.


MiTeC IB Query - [Employee on LAPTOP-NOTE035K (Local)] - [Performance Monitor]
Session
Tools
Services
Windows
Help
SQL Editor #1 - ...
CUSTNO
EMPLOYEE
Performance Mo...
Server: LAPTOP-NOTE035K (Local)
Performance Monitor

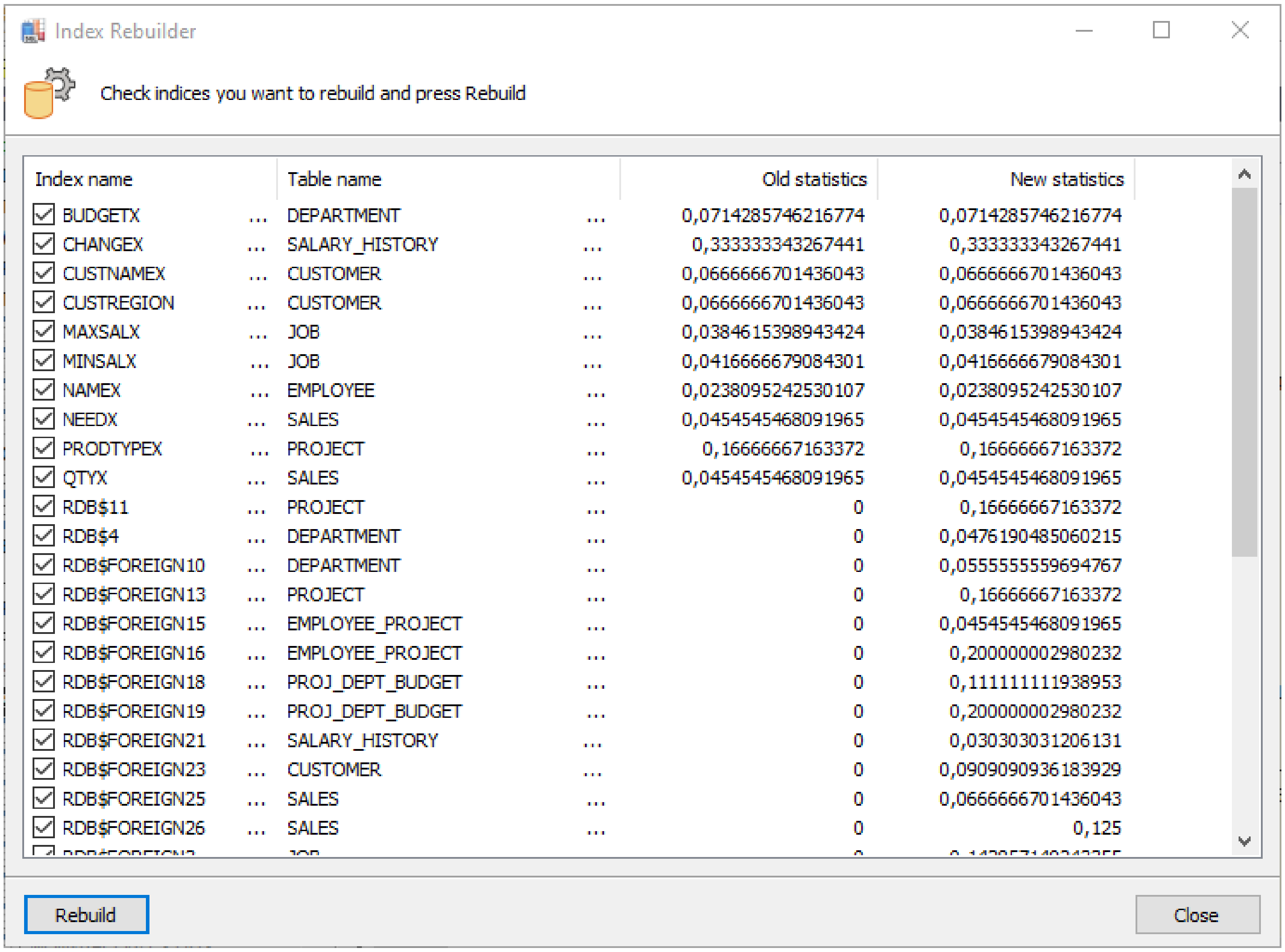
```
Employee (SQL Dialect 3)
  Domains (15)
  Tables (10)
  Views (1)
  Procedures (10)
  Triggers (4)
    POST_NEW_ORDER
    SAVE_SALARY_CHANGE
    SET_CUST_NO
    SET_EMP_NO
  Generators (2)
    CUST_NO_GEN
    EMP_NO_GEN
  Exceptions (5)
  UDFs
  Roles (1)
  Packages
  System Domains (244)
  System Tables (56)
  System Procedures (10)
  System Triggers (54)
  System Generators (11)
  System UDFs (7)
  System Packages (3)
    RDB$BLOB_UTIL
    RDB$PROFILER
    RDB$TIME_ZONE_UTIL
  Data Files (1)
```

```
Reads from memory buffer cache
Last:           1
 Max:         375
 Tot:      14 277
 Avg:       2 379

Writes to memory buffer cache
Last:           1
 Max:          58
 Tot:         381
 Avg:          95

Reads from database
Last:           4
 Max:           4
 Tot:         184
 Avg:         180

Writes to database
Last:           1
 Max:           1
 Tot:         114
 Avg:         113

Memory Usage [B] (Installed memory: 4095 MB):
Last:      -4 880
 Max:      28 096
 Tot:  21 535 328
 Avg:   3 590 034
```

sysdba | NONE | Auto DDL | Firebird Client library (32-bit) 5.0.1 BETA 1469 | Employee | WI-V6.3.1.1469 Firebird 5.0

If you use event notifications, you will appreciate Event Watcher to verify that they are working properly.

While the name might suggest otherwise, Index Rebuilder does not physically rebuild indexes; instead, it efficiently updates their statistics, a crucial task for maintaining optimal query performance.


Index Rebuilder

Check indices you want to rebuild and press Rebuild

	Index name		Table name		Old statistics	New statistics
✓	BUDGETX	...	DEPARTMENT	...	0,0714285746216774	0,0714285746216774
✓	CHANGEX	...	SALARY_HISTORY	...	0,333333343267441	0,333333343267441
✓	CUSTNAMEX	...	CUSTOMER	...	0,0666666701436043	0,0666666701436043
✓	CUSTREGION	...	CUSTOMER	...	0,0666666701436043	0,0666666701436043
✓	MAXSALX	...	JOB	...	0,0384615398943424	0,0384615398943424
✓	MINSALX	...	JOB	...	0,0416666679084301	0,0416666679084301
✓	NAMEX	...	EMPLOYEE	...	0,0238095242530107	0,0238095242530107
✓	NEEDX	...	SALES	...	0,0454545468091965	0,0454545468091965
✓	PRODTYPEX	...	PROJECT	...	0,16666667163372	0,16666667163372
✓	QTYX	...	SALES	...	0,0454545468091965	0,0454545468091965
✓	RDB$11	...	PROJECT	...	0	0,16666667163372
✓	RDB$4	...	DEPARTMENT	...	0	0,0476190485060215
✓	RDB$FOREIGN10	...	DEPARTMENT	...	0	0,0555555559694767
✓	RDB$FOREIGN13	...	PROJECT	...	0	0,16666667163372
✓	RDB$FOREIGN15	...	EMPLOYEE_PROJECT	...	0	0,0454545468091965
✓	RDB$FOREIGN16	...	EMPLOYEE_PROJECT	...	0	0,200000002980232
✓	RDB$FOREIGN18	...	PROJ_DEPT_BUDGET	...	0	0,111111111938953
✓	RDB$FOREIGN19	...	PROJ_DEPT_BUDGET	...	0	0,200000002980232
✓	RDB$FOREIGN21	...	SALARY_HISTORY	...	0	0,030303031206131
✓	RDB$FOREIGN23	...	CUSTOMER	...	0	0,0909090936183929
✓	RDB$FOREIGN25	...	SALES	...	0	0,0666666701436043
✓	RDB$FOREIGN26	...	SALES	...	0	0,125

Codebase serves as a straightforward repository for scripts, enabling users to edit them within a basic editor and seamlessly transfer them to the SQL editor for execution.

Export and import functions support CSV, XML, and binary RAW formats.

Summary

While IBQuery may not be suitable for demanding production environments or for development of database applications, it remains a valuable tool for field technicians. Its user-friendly interface makes it an efficient aid for minor interventions and maintenance tasks performed directly at client sites.

The primary limitation of IBQuery is its lack of support for the newer data types introduced in Firebird 4. However, it remains a viable option for those still working with Firebird 3 or earlier versions.

The Festive Stored Procedure Cleanup

Santa tasked the elves with reviewing old stored procedures in Firebird, searching for bad code or outdated logic. With the help of Firebird's PSQL profiler, they analyzed execution times and pinpointed several poorly written procedures that were consuming unnecessary resources.

The team diligently optimized and consolidated these procedures, resulting in significantly improved performance. Santa was delighted. “Even stored procedures deserve a holiday cleanup!” he chuckled.

Answers to your questions

Documentation is said to be a collection of answers to unspoken questions. If you ask a search engine, it will answer you with a link to a document that (hopefully) contains the answer. There are documents, forums and entire systems like Stack Overflow that consisting only of questions and answers. And now an army of AIs is starting to chase us to answer our questions. Questions and answers cannot be avoided, there is no hiding place.

However, amidst the sea of routine questions and responses, there lie truly captivating inquiries and answers, like hidden treasures. Our commitment is to regularly present you with a curated collection of these precious gems.

Firebird indices 101

Kjell Rilbe asked:

Apparently it’s not possible to deactivate a primary key index. Seems reasonable. So, how do I rebuild it?

Sean Leyne answers:

Why do you need to rebuild it?

Unlike user indexes, primary key indexes have a fixed selectivity which means that they never become "unbalanced" — so they never need to be re-indexed.

Ann W. Harrison answers:

Sean, I agree with your conclusion, but have trouble with the reasons. Firebird indexes do not get unbalanced at all. Selectivity has little or nothing to do with index balance.

Primary key indexes - and unique indexes - do not suffer from bad selectivity values because their selectivity is known a priori. Other indexes created on an empty table will have incorrect selectivities after data is loaded and should at a minimum have the selectivity set after a large load or other operation so the stored selectivity matches the actual data.

Now for my contention that Firebird indexes don’t get unbalanced. The classic Database 101 b-tree adds layers going downward so the path to one record from the top of the tree is say 5 layers while the path to another record is 15 layers. That’s why databases don’t use simple b-trees. In the description below, I’m ignoring jump nodes which are important but not significant to the discussion.

A Firebird index starts as a single page containing pairs of values and record identifiers. Eventually, someone wants to add another pair and it doesn’t fit on the page. Firebird allocates a new page, copies half the pairs into the new page, then allocates a second page with two entries, each consisting of a value, a record identifier, and a page number. The page numbers are the original index page and the new index page. The values and record numbers are the first pairs from the lower pages. Then the system goes along, happily filling bottom level pages, splitting them as necessary, and adding new pages to the top page. Eventually, the top page fills, splits, and creates another top page with two entries, one for the first half of the former top page and one for the second.

If you think about that for a second, you'll realize that the length from the top to every lowest level entry is exactly the same.

Oh, but what about deleted records. Won't I have an unbalanced index if my key is always increasing with time and I delete the old records. The answer is no, because Firebird does almost exactly the reverse when pages get below 1/3 full - it combines them and removes the pointer from the next level up. That turns out to be very hard and has taken years of Vlad's life to make work under load as pages are added to and removed from indexes.

Kjell Rilbe wrote:

So, when IS it relevant to rebuild a Firebird index, which I understand happens when an index is deactivated and then reactivated?

Ann W. Harrison answers:

Index pages are recombined when two adjacent pages are each less than (roughly) 30% full. Sorry, I don't remember the exact number and am not going to look it up. That reduces - but does not eliminate - the problem of releasing and index page and instantly needing it back again because the recombined page will only be 60% full. If you've got a table from which records are regularly deleted across the range of the index, gstat may tell you that the index pages are less than 50% full on average. That might be a reason to rebuild the index - unless you expect to create new records with key values all across the range of the index.

In fact, the only reason I can think of for rebuilding an index is that you want a better fill level - whether the source of the problem is deleted records or creating the index before loading a large amount of data in random order relative to the index.

And some people just like doing maintenance and sleep better knowing their indexes have been scrubbed.

Kjell Rilbe wrote:

We’re preparing a database for an online system and have imported a large amount of data. The DB file is currently 42 gigabyte. There are two tables with over 110 million records and then some with about 1-2 million records. Most other tables only have a few hundred or a few thousand records. No records are deleted.

Page size is 4096.

Now, after the batch import finishes, I assume the indexes could use with some "conditioning". But what’s my best option?

1. Just set statistics on all of them?
2. Rebuild all indexes with deactivate/reactivate?
3. Do a backup/restore cycle with gbak?
4. Other options?

Would you handle indexes differently depending on selectivity? For example, the "largest" table, with over 110 million records, has a primary key where key values are incremental (not always +1, but always +something). Is it better to leave that index without a rebuild?

Also, what considerations should I make regarding page size? Maybe I should bump up the page size?

Downtime is no problem. More important is to maximize index and query performance before the system is put online.

Ann W. Harrison answers:

If you had asked before you did the load, I would have said, import with indexes off, then create the indexes.

Ad 1. Absolutely.

Ad 2. Will produce dense index pages - which may be good or bad, depending on what you do next. If the database continues to grow significantly across all index ranges, then having some space in the index reduces future splits. If you’ve got most of the data you expect to have, maximize packing density to reduce the number of index reads.

Ad 3. Overkill.

Ad 4. Don’t activate the indexes until the data is stored.

The normal index loading does have an optimization when the page to be split is the last one in the index. Normally, a split puts half the data in each page. If the split is the last page in the index, the new page has only the record that didn’t fit. That means that if you load in key order, the index will be created dense.

About page size, it depends on how deep are your indexes.

Kjell Rilbe wrote:

Seems that on the largest table, they all have depth 4. Are there any figures in this report that would warrant a config change or anything, e.g. different page size?

Ann W. Harrison answers:

Yes, I’d double the page size. Every new index access reads the full depth of the index tree, so a four level index is going to be slower than a three level index. With luck, a range retrieval will read across the bottom of the index, but if it conflicts with an index update, it starts again from the top.

Looking at a couple of indexes:

```
Analyzing database pages ...
Uppgift (567)
    Index IX_PK_Uppgift (0)
        Depth: 4, leaf buckets: 588728, nodes: 131418408
        Average data length: 5.08, total dup: 0, max dup: 0
        Fill distribution:
             0 - 19% = 2968
            20 - 39% = 15
            40 - 59% = 279494
            60 - 79% = 185021
            80 - 99% = 121230
```

Hmm. That one could certainly be more dense. Is it possible to sort your input data by primary key before you store it? In case it’s not clear to everybody a "leaf bucket" is a bottom level index page. Index pages are called buckets for reasons lost in the mists of time. And index trees have the leaves on the bottom. "Old Frothingslosh, the pale stale ale with the foam on the bottom". Maybe I need less coffee.

```
Index IX_Uppgift_BorttagsuppA5K (8)
    Depth: 4, leaf buckets: 253733, nodes: 136466214
    Average data length: 0.00, total dup: 136444473, max dup: 119722141
    Fill distribution:
         0 - 19% = 127
        20 - 39% = 7685
        40 - 59% = 131314
        60 - 79% = 28000
        80 - 99% = 86607
```

And this one … hmm again. Once again, medium fill level, but here the average data length is zero to two decimal places. Which is good, but suggests lots of duplicates.

```
nodes:     136,466,214 <- that's the number of entries in the index
total dup: 136,444,473 <- that's the number that are identical
max dup:   119,722,141 <- that's the longest chain of a single value.
```

Amazing that it works at all. Do you know which value has 119 million instances? It was less than 10 years ago when gstat changed its accumulators from 16 to 32 bits.

```
Index IX_Uppgift_Till├ñggsuppYNC (9)
    Depth: 4, leaf buckets: 196830, nodes: 131804550
    Average data length: 0.02, total dup: 131766961, max dup: 11251
    Fill distribution:
         0 - 19% = 9
        20 - 39% = 1
        40 - 59% = 3217
        60 - 79% = 44
        80 - 99% = 193559
```

This index looks pretty good. Yes there are a lot of duplicates, but they’re distributed pretty evenly.

Sergio asked:

Hello, I’ve been reading this FAQ and I undestand that having an index on a boolean field is the worst of the cases. But I’ve also seen that if you have (as in my case) very little "active" records (MyBoolean = 1) and a LOT of inactive records, is good to have an index to select the active records.

Is that True? Should I create an index on my boolean field?

My paricular case is a calendar with events in some days. I need to select the pending events. They will be always, lets say, less than 100, while the table, as the the years pass will grow a lot with the old (not pending) events…

Ann W. Harrison answers:

In very early versions of Firebird, an index like that would have caused enormous problems during garbage collection. Now there’s an algorithm that makes cleaning out old duplicate entries fast enough.

An index will improve performance when you’re looking for the 1% of the records that are not like the rest, if you can convince the optimizer to use it. The optimizer may be smart enough to ignore indexes with terrible selectivity, in which case you may need to add a plan to force use of the index. But if you accidentally run a query that looks for the 99% of records that all have the same value (and the optimizer chooses to use that index) you’ll get poor performance. You can avoid that by adding a nonsense OR clause - (pending = 0 OR 1 = 2) - that will cause the optimizer to ignore the index.

Besides, adding and dropping an index is pretty trivial. Try it, if you like it, keep it, otherwise get rid of it.

Why is CommitRetaining bad?

rodrigogoncalves asked:

What I don’t understand is why there is a performance issue when an CommitRetaining is applied. Since the comands related to the transaction won’t be "rollbacked" anymore, other transaction have no interest on this one anymore, so they should not have a performance impact.

Ann W.Harrison answers:

The problem is that a commit retaining starts a new transaction that keeps the same context as the one that was "commit retained". From the point of view of garbage collection, it’s as if that transaction were still running, and it block garbage collection at the state when the original transaction started.

Pavel Císař adds:

The RETAINING option for commit and rollback was introduced in InterBase 4.0 to address a specific technical challenge faced by Borland. To understand its purpose, some historical context is necessary.

The early 1990s were a golden era for Borland, still under the leadership of its founder and CEO, Philippe Kahn. Among Borland's top products were the legendary Turbo Pascal and development tools for other languages. Another key product was the Paradox database. In 1991, Borland acquired Ashton-Tate, bringing dBase and InterBase (which was bought by Ashton-Tate early before) into its portfolio. This acquisition was part of "brilliant" Borland strategy: to leverage all its assets by creating a RAD tool for rapid application development that could seamlessly work with dBase, Paradox, or InterBase.

A critical piece of this puzzle was the Borland Database Engine (BDE), which provided a unified API for accessing dBase, Paradox, and InterBase. The BDE was first introduced in 1992, bundled with Borland Pascal 7.0 and Paradox for Windows. However, this initial release was primarily a proof of concept. The real game-changer was the "VBK" project, which began in late 1993 and was later renamed Delphi.

When the Delphi team integrated the BDE with InterBase, they encountered a significant challenge. Unlike file-based databases like dBase or Paradox, InterBase required transactions to manage data access. In InterBase, closing a transaction also closed all associated cursors. This behavior caused a major issue: closing a transaction would immediately disconnect data-aware components from their underlying data, effectively "losing" their state.

To address this, the InterBase team was tasked with finding a solution to allow cursors to persist across transaction boundaries. Their solution was the RETAINING option, introduced with InterBase 4.0, released in 1994.

When Delphi 1.0 launched in 1995, it included a free single-user license for InterBase 4, encouraging developers to adopt InterBase for their projects. At first, it was all "rainbow and unicorns", until they discovered that their applications do not scale. Applications that worked flawlessly in single-user or test environments began to fail catastrophically in production, especially for larger deployments. The RETAINING option, while solving the cursor persistence issue, introduced side effects like bloated transaction metadata and stalled garbage collection, which degraded performance over time.

The issue was eventually addressed with the introduction of the IBX (InterBase Express) component library, shipped with Delphi 5 in August 1999. Around the same time, Borland began deprecating the BDE, officially replacing it with dbExpress in the early 2000s.

The final history lesson is this:

RETAINING is suitable for single-user applications using Firebird Embedded, provided the application is closed at the end of the day. However, it should never be used in multi-user server environments, as it can severely compromise scalability and performance.

The Power of CTEs

The elves needed to calculate the total production time for each toy, including its components. Elf Mathy used a common table expression (CTE) to build a recursive query that traversed the "ToyComponents" hierarchy.

With Firebird's CTEs, Mathy generated the report in record time. Santa marveled, "It's like following the star on a Christmas tree—all paths lead to the top!"

The Window to Santa's Metrics

Santa wanted to see which toys were most popular, grouped by country and ranked by request counts. Elf Analyst used Firebird's window functions to create a report that included rankings, cumulative totals, and averages—all in one query.

The results helped Santa prioritize toy production for each region. "Window functions give me a whole new perspective!" Santa said, already planning next year's analytics.

Planet Firebird

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

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

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

The Rebirth of the Firebird Foundation

The Firebird Foundation has recently undergone a significant transformation, marking the end of more than two decades of operations from Australia. A new version of the Foundation, based in the Czech Republic within the European Union, has been established. The goal of this shift was to strengthen the foundation's connection to Project Firebird and provide a more efficient and targeted approach to supporting its development.

The new Firebird Foundation's position allows us to offer more valuable and tailored support to Firebird users and contributors. Because stable funding is essential for the continued development of Firebird, regular financial contributions play a key role in moving the project forward. That's why the Foundation now offers affordable subscription plans for individuals and businesses, with benefits that vary by contribution level. These include early access to EmberWings, partner discounts, exclusive webinars, exclusive technical content or participation in release planning. High-level sponsors can enjoy even greater perks, including priority access and personalized recognition. This new structure allows the Foundation to offer more meaningful, long-term value to its supporters, ensuring Firebird's continued success and innovation.

With the new organization, we aim to adopt a much more proactive approach, far surpassing the level of activity seen in recent years. Our goal is to open the project up more substantially to the user community, strengthening and deepening mutual connections that have somewhat diminished due to the lack of regular international conferences.

Unfortunately, the reorganization has also brought some confusion and mistakes. We hope you will continue to support us despite these temporary challenges and imperfections. We have ambitious plans to implement over the next year, and we would greatly appreciate your help by providing valuable feedback to guide our efforts.

You can already provide feedback on this magazine and share your preferences for organizing webinars. Additionally, we are working to map the needs and challenges of Firebird users, allowing the Firebird Project to better focus its resources and enabling the Foundation to invest where it is most needed.

Links to the relevant questionnaires can be found at the end of this issue. They will also appear later on our website, which will undergo renovation over the next year.

Firebird at ITDevCon 2024

ITDevCon is Europe’s leading conference on Delphi and related technologies. The 13th edition took place on November 14–15 at the Time Group headquarters in Rome. Among sessions valuable for Firebird users—such as those on integrating Delphi with databases, using RESTful APIs, and interfacing with external services—was a standout session titled "Firebird Goes Mobile," presented by Firebird Foundation member Fabio Codebue. With 40 attendees, Fabio reported a buzzing atmosphere, highlighting the strong interest in mobile databases.

This interest is understandable, as Firebird faces little competition in the mobile database space, presenting a unique opportunity to lead this market. The project is now exploring ways to expand its mobile focus to include iOS alongside Android. Additionally, enhancing documentation and offering practical examples for development environments like Delphi, Ionic, and React are priorities.

If you are interested in mobile development, now is the perfect time to partner with the Foundation and support these exciting advancements.

First Firebird Certified Professionals

The Firebird Foundation has begun implementing its certification programs for application developers and database administrators. While an online demo exam has been available for administrators for some time, November 22 marked a significant milestone: eight professionals successfully passed the Firebird Basic DBA exam in São Paulo, Brazil, completing the first official certification.

A special congratulations goes to Alcides Magno Baptista Moreira de Deus, who earned certificate number 0001 with a perfect 100% score and was the first to finish the exam—a remarkable achievement!

The Foundation will soon introduce online exams and announce the next steps in the Firebird Certification program. To learn more about these certifications, visit our website.

A Festive Countdown to a Milestone Year

The holiday season is here, and Firebird is inviting everyone to partake in a unique countdown that promises to bring warmth, joy, and community spirit to winter days. The Firebird Advent Calendar is not just a celebration of the season—it's a lead-up to Firebird's momentous 25th Anniversary in 2025.

Each day of December, a new door on the calendar reveals exciting content, surprises, and engaging activities that highlight the essence of Firebird. From nostalgic reflections on the past to thrilling glimpses into the future, this calendar celebrates the collaboration and innovation that have defined the Firebird journey.

But don't worry if you're late to the party—this special advent calendar will remain open for exploration until the end of January 2025. This extended availability ensures that everyone has a chance to dive into the festive fun and be part of this community celebration.

So, grab a cup of cocoa, and join the countdown as we honor the past and eagerly anticipate an incredible year ahead for Firebird. Together, let's make this season and the journey to the 25th Anniversary unforgettable.

Don't miss a single day—your next surprise is just a click away!

The calendar is available on the Firebird website, or you can click the button below to access the full-page version directly.

Visit Advent Calendar

The Elves' Secret Santa

In Santa's bustling North Pole office, the elves were wrapping up their annual Secret Santa gift exchange. Clara, one of the tech-savvy elves who managed the Naughty-or-Nice database, eagerly pulled her gift from the pile.

“Let’s see if Firebird got me something good this year,” Clara joked as she untied the festive ribbon.

Her coworker David raised an eyebrow. “Firebird? Pretty sure it’s not one of the elves, Clara.”

Clara laughed. “Maybe not, but it handles everything else for us—naughty lists, toy inventories, even sleigh logistics. It’s fast and reliable, so why not gift-giving too?”

The other elves chuckled, shaking their heads at Clara’s unwavering faith in the North Pole’s favorite database engine.

Inside the box, Clara found a shiny new USB drive, engraved with a tiny firebird emblem. Attached was a note that read:

“For the times you need something small but powerful—just like me. Merry Christmas!”

Clara held up the gift with a grin. “Looks like Firebird really *was* my Secret Santa!”

The workshop erupted in laughter and cheers, and from that day on, the elves affectionately nicknamed their database engine “Santa’s Little Helper.”

Notes on the effect of using ALTER INDEX

By Paul Reeves (IBPhoenix)

The documentation for `ALTER INDEX … [IN]ACTIVE` is somewhat ambiguous. At first sight it appears to be obvious - you would use it to activate or de-activate an index. A switch to turn them on or off. De-activating an index can be useful, especially during bulk insert operations. It could also be useful for deactivating indexes with very little selectivity. Such indexes are usually created by foreign keys and they are surprisingly common, require additional page writes to maintain and can never be used in any useful sense. If only there was a way to turn them off! And then the documentation states quite clearly that you can't de-activate indexes that are created by constraints. Hmmm… the command is suddenly not *that* useful after all. At this point most users will just shrug and move on. This is not the solution they were looking for. Dropping constraints just to de-active a bunch of indexes is way too complicated.

But hidden in the documentation for `ALTER INDEX … ACTIVE` is a little nugget. It says executing the command will rebuild the index. That is right -

REBUILD it. It does not say that an index *must* be INactive in order to activate. And with no mention whether indexes created by constraints are excluded or not. So, what is going on here? Does using `ALTER INDEX ... ACTIVE` on any already active index really rebuild it?

Why would you want to rebuild an index?

When a database is restored it recreates each index precisely based on the existing data. Each index page has space for changes, in the same way that tables are filled to 80% so are indexes. This is a reasonable compromise. Pages are reasonably well packed and can still accommodate some changes by writing to existing pages. But when a table sees many inserts, updates and deletes to records then holes start appearing, both in the data pages and in the index pages.

When a table is freshly restored retrieving a record should take around 4 page reads if the index has a depth of three. Unless the index depth changes retrieving a single record will always take 4 page reads. In a table with millions of records and thousands of data pages this is a win. And when pages are well packed an attempt to retrieve, say, 100 records will immediately benefit from this as many of the records in the set will already be in fetched pages. As holes start to appear in pages then obviously more pages need to be read to retrieve the data.

To take a hypothetical example, if most of the index pages are now half full then approximately twice many pages will need to be read in order to retrieve the hypothetical 100 rows. In theory `ALTER INDEX ... ACTIVE` will rebuild the indexes and performance should return to something similar to the state after the last backup/restore cycle.

Well, that’s the theory. Does it work on all indexes, built by a constraint or otherwise? How long does it take? And is performance better afterwards? Let’s see.

The test database

The test database is just a simple table with a primary key, a foreign key and some indexes on two data columns.

```
create table lookup_colour( ID D_INTEGER, COLOUR varchar(12),
  constraint pk_lookup_colour PRIMARY KEY (ID)
  );

create table test_indexes (  id D_INTEGER,  col_id D_INTEGER,
  txt D_GUID,  txt2 D_GUID_TEXT,  constraint pk_test_pk primary key (id)
  );
commit;

create unique index test_uq_idx on test_indexes(id);
create index test_nonuq_idx on test_indexes(id);
create index test_nonfk_idx on test_indexes(col_id);
create asc  index test_txt_idx on test_indexes(txt);
create desc index test_txt__desc_idx on test_indexes(txt);
create unique asc  index test_txt2_idx on test_indexes(txt2)
  where txt2 is not null;
create unique desc index test_txt2_desc_idx on test_indexes(txt2)
  where txt2 is not null;

alter table test_indexes add constraint fk_lkp_col
  foreign key (col_id) references lookup_colour (id);

commit;
```

The tests

10 million rows were inserted in blocks of one million rows. After each insertion block 10% of the table was randomly selected for update and then another 10% was randomly selected for deletion. The final test database had around 7 million rows in the test table.

- For these tests background garbage collection was turned off.
- These tests were carried out using Firebird 5.0.1 SuperServer on Linux. The behaviour of earlier versions will be slightly different but the overall results will be similar.

Result of setting FK_LKP_COL active

To start with let's look at what happens when we run `ALTER INDEX … ACTIVE` on a single index. For this test FK_LKP_COL was chosen.

Here is the gstat output from before and after the index was rebuilt.

BEFORE ACTIVATE

```
TEST_INDEXES (129)
  Index FK_LKP_COL (8)
Root page: 25240, depth: 3,
    leaf buckets: 10053
    nodes: 9274374
Average node length: 4.99,
    total dup: 9274364,
    max dup: 928845
Average key length: 2.00,
    compression ratio: 0.90
Average prefix length: 1.80,
    average data length: 0.00
Clustering factor: 951868,
    ratio: 0.10
Fill distribution:
      0 - 19% = 0
     20 - 39% = 0
     40 - 59% = 9446
     60 - 79% = 7
     80 - 99% = 600
```

AFTER ACTIVATE

```
TEST_INDEXES (129)
  Index FK_LKP_COL (8)
Root page: 25062, depth: 3,
    leaf buckets: 5727 <--
    nodes: 9274374
Average node length: 4.99,
    total dup: 9274364,
    max dup: 928845
Average key length: 2.00,
    compression ratio: 0.90
Average prefix length: 1.80,
    average data length: 0.00
Clustering factor: 951868,
    ratio: 0.10
Fill distribution:
      0 - 19% = 0
     20 - 39% = 0
     40 - 59% = 0
     60 - 79% = 0
     80 - 99% = 5727 <--
```

- Leaf buckets (the actual pages storing index values) have almost been halved. ✅
- Index pages are repacked to high density ✅

Result of setting rebuilding ALL of the indexes

From the above it looks as if rebuilding a single index will improve performance where that index is used. What happens when we rebuild all the indexes?

We will start by taking a look at the primary key index.

BEFORE ACTIVATE

```
Index PK_TEST_PK (0)
Root page: 24634, depth: 3,
    leaf buckets: 7784
nodes: 9274374
Average node length: 6.03,
    total dup: 0, max dup: 0
Average key length: 3.04,
    compression ratio: 1.52
Average prefix length: 3.60,
    average data length: 1.02
Clustering factor: 214796,
    ratio: 0.02
Fill distribution:
     0 - 19% = 1
    20 - 39% = 0
    40 - 59% = 517
    60 - 79% = 1590
    80 - 99% = 5676
```

AFTER ACTIVATE

```
Index PK_TEST_PK (0)
Root page: 23906, depth: 3,
    leaf buckets: 5414 <--
    nodes: 7114411 <--
Average node length: 6.14,
    total dup: 0, max dup: 0
Average key length: 3.15,
    compression ratio: 1.48
Average prefix length: 3.59,
    average data length: 1.07
Clustering factor: 210009,
    ratio: 0.03
Fill distribution:
     0 - 19% = 1
    20 - 39% = 0
    40 - 59% = 0
    60 - 79% = 0
    80 - 99% = 5413 <--
```

- Number of pages (leaf buckets) in the index have now been reduced by ~25% ✅
- Number of index entries (nodes) now reflects the actual number of records in the table. ✅
- Index pages have been cleaned up with more nodes stored per page. ✅

What happens to the other indexes ? The effect upon the other indexes is similar. By way of example we will look at TEST_TXT_IDX in detail.

```
Root page: 2334, depth: 3,
    leaf buckets: 32806
    nodes: 9274374
Average prefix length: 2.22,
    average data length: 13.78
Clustering factor: 9274272,
    ratio: 1.00
Fill distribution:
     0 - 19% = 121
    20 - 39% = 0
    40 - 59% = 9226
    60 - 79% = 16146
    80 - 99% = 7313
```

```
Root page: 2310, depth: 3
    leaf buckets: 17504 <--
    nodes: 7114411 <--
Average prefix length: 2.17,
    average data length: 13.82
Clustering factor: 7114331,
    ratio: 1.00
Fill distribution:
     0 - 19% = 1
    20 - 39% = 0
    40 - 59% = 0
    60 - 79% = 0
    80 - 99% = 17503 <--
```

There is no need to examine the effect of rebuilding the other indexes. In all case they exhibited similar results to the above.

Now that the indexes have all been rebuilt, let’s take a look at the data pages to see the effect of rebuilding all indexes on the data storage of the table itself.

BEFORE

```
TEST_INDEXES (129)
Primary pointer page: 234,
Index root page: 235
Total formats: 1, used formats: 1
Average record length: 33.41,
    total records: 9274374
Average version length: 44.76
    total versions: 2159963
Average fragment length: 34.00
    total fragments: 36809
Average unpacked length: 172.00
    compression ratio: 5.15
Pointer pages: 60,
    data page slots: 96384
Data pages: 96384,
    average fill: 77%
Primary pages: 95191,
    secondary pages: 1193
    swept pages: 0
Empty pages: 0,
    full pages: 89851
Fill distribution:
         0 - 19% = 0
        20 - 39% = 58
        40 - 59% = 688
        60 - 79% = 39157
        80 - 99% = 56481
```

AFTER

```
TEST_INDEXES (129)
Primary pointer page: 234,
Index root page: 235
Total formats: 1, used formats: 1
Average record length: 40.82,
    total records: 7114411 <--
Average version length: 0.00
    total versions: 0, <--
Average fragment length: 34.00
    total fragments: 36809
Average unpacked length: 172.00
    compression ratio: 4.21
Pointer pages: 60,
    data page slots: 95840
Data pages: 95728,
    average fill: 54% <--
Primary pages: 95191,
    secondary pages: 537
    swept pages: 0
Empty pages: 12,
    full pages: 28467
Fill distribution:
         0 - 19% = 14
        20 - 39% = 859
        40 - 59% = 61081 <--
        60 - 79% = 33774
        80 - 99% = 0
```

- The total number of records stored is now correct at the data page level. ✅
- Back record versions have been cleaned out ✅
- Average page fill is significantly worse ✕
- Fill distribution is also worse ✕

Overall data page storage *is* better. And while page fill appears to have deteriorated the reality is that these pages stored back record versions of rows that had either been updated or deleted. So it was inaccessible to new transactions anyway. And now that the data pages have been cleaned up this space is now available for re-use. But this does indicate that we should be wary of the page fill statistics.

What about rebuilding indexes for a table that is in use?

Let's try two tests:

1. Table with an active (uncommitted) select
2. Table with active deletes

Table with an active (uncommitted) select

Executing this statement:

```
select r.ID, r.COL_ID, r.TXT from TEST_INDEXES r
join LOOKUP_COLOUR c on r.COL_ID = c.id
where c.colour = 'White'
order by r.ID
rows 1 to 10
```

and subsequently executing `ALTER INDEX FB_LKP_COL ACTIVE` worked fine, so active selects do not appear to block rebuilding indexes.

Table with active deletes

Running this query:

```
execute block
as
declare anum D_INTEGER;
begin
  execute procedure populate_pk_list ( 0, 9999);
  for select pk_list_id from pk_list into :anum do
    delete from test_indexes where id = :anum;
end ^
```

```
Statement prepared (elapsed time: 0.012s).
-- line 12, column 5
PLAN (TEST_INDEXES INDEX (PK_TEST_PK))
-- line 7, column 3
PLAN (PK_LIST NATURAL)

Executing statement...
Statement executed (elapsed time: 2.217s).
7505801 fetches, 1830 marks, 96397 reads, 6 writes.
711 inserts, 0 updates, 370 deletes, 481 index, 7115125 seq.
Delta memory: 412688 bytes.
TEST_INDEXES: 370 deletes.
PK_LIST: 711 inserts.
0 rows affected directly.
Total execution time: 2.267s
Script execution finished.
```

produces some minor changes to the data pages of the table:

Now let's see if we can rebuild the index for FK_LKP_COL:

```
Starting transaction...
Preparing statement: ALTER INDEX FK_LKP_COL ACTIVE
Statement prepared (elapsed time: 0.000s).
Plan not available.

Executing statement...
Statement executed (elapsed time: 0.000s).
29 fetches, 2 marks, 9 reads, 0 writes.
0 inserts, 1 updates, 0 deletes, 5 index, 0 seq.
Delta memory: 55408 bytes.
RDB$INDICES: 1 updates.
Total execution time: 0.027s
Script execution finished.
```

All looking good, so far. Now let's commit the transaction.

SQL Message : -901

```
Unsuccessful execution caused by system error
Engine Code    : 335544345
Engine Message :
lock conflict on no wait transaction
unsuccessful metadata update
```

Oops.

It is clear that `ALTER INDEXES … ACTIVE` does not work on tables that are actively being changed. This is obvious really but it is worth bearing in mind. The command needs to take an exclusive lock on the table.

And how about performance?

We have established that rebuilding indexes improves the page storage of data pages and index pages but does performance actually improve? Let's see…

We executed this query on a table with seven million records:

```
select r.ID, r.COL_ID, r.TXT, r.TXT2
from TEST_INDEXES r
where TXT starting with x'aa' || x'aa'
--where TXT starting with x'aa'||x'aa'
and r.TXT2 is not null;
```

and it returned twelve rows.

BEFORE REBUILD

```
Current memory = 53128800
Delta memory = 156672
Max memory = 53194240
Elapsed time = 0.165 sec
 Cpu = 0.000 sec
Buffers = 6000
Reads = 16271
Writes = 0
Fetches = 16503
```

AFTER REBUILD

```
Current memory = 57003328
Delta memory = 22864
Max memory = 147186672
Elapsed time = 0.083 sec <--
  Cpu = 0.000 sec
Buffers = 6000
Reads = 7723 <--
Writes = 0
Fetches = 7775 <--
```

- Execution time halved. ✅
- Page reads from disc more than halved. ✅
- Page fetches from cache more than halved. ✅

Summary

As data is changed indexes become unbalanced leading to poor density of index pages. An unbalanced index will require more page reads in order to retrieve a records.

It is clear that `ALTER INDEX … ACTIVE` will rebuild the index whether it was created by a constraint or not.

It is also clear that `ALTER INDEX … ACTIVE` must be executed for all indexes in a table in order to fully gain its benefits.

Performance will improve.

There is an overhead to execution of `ALTER INDEX … ACTIVE` however. The first execution takes the longest if there is any garbage in the table. Subsequent executions to rebuild other indexes on the table complete far more quickly.

Indexes cannot be rebuilt if data in the table is being changed.

It is probable that the entire database needs to be put into single user maintenance mode in order to rebuild all the indexes. This is probably faster and easier than trying to gain exclusive locks, table by table.

The cost of putting the database into maintenance mode for this operation will require downtime, however this *will* be cheaper than a full backup/restore cycle. On the other hand, it is less effective than backup/restore at optimally filling pages. And neither does it recover lost pages.

If there is a large amount of garbage in the table, it still makes sense to first DEACTIVATE indexes that can be deactivated to avoid unnecessary cleaning during activation, and use ACTIVATE only on indexes that are part of the constraint.

Conclusion

Rebuilding indexes can be a definite win for performance if…

- The database is very large
- The database sees many changes.
- The backup/restore cycle is longer than the time required to rebuild indexes.

...And now for something completely different

Finn and the Twelve

Once upon a time in a large corporation, a young administrator named Finn joined the IT department. Eager and dedicated, Finn was appointed to manage the company’s Firebird database server. But he quickly discovered that the system was outdated and riddled with issues after years of neglect. Upper and middle management didn’t care for the database or its inner workings; they saw it only as a tool to serve their needs.

Frustrations over the Firebird server’s performance grew, and management turned their gaze toward Finn. They demanded he deliver impossible improvements, setting him up as a scapegoat for the database’s failings. One cold Monday morning, the CTO, a ruthless and impatient man, summoned Finn to his office.

“Finn, by the end of the week, you need to make this server run three times faster and deploy real-time reporting!” barked the CTO. “Do it, or you’ll be out the door.”

Finn tried to explain that such a transformation would take months, but the CTO dismissed his pleas, demanding immediate results.

Returning to his desk, Finn felt the weight of an impossible task. He poured over technical articles and old manuals, but they seemed written in another language. Desperate, he decided to reach out to the Firebird community, where developers and administrators alike shared their knowledge.

As he posted his questions, he noticed responses trickling in from mysterious accounts with nickname like **January**, **February**, **March**, and so on. Each response carried invaluable advice. As Finn read on, he realized he wasn't speaking to regular users—he was in touch with the legendary **Twelve**. These were figures who had shaped Firebird from the beginning, each with specialized knowledge. They were the keepers of wisdom within the Firebird Project, and Finn was fortunate to be receiving their help.

January was the first to reply: "Finn, start by rebuilding the indices and examining the statistics. The basics are your foundation." Following January's guidance, Finn made adjustments that immediately improved performance.

Next, **February** shared insights into managing transactions to keep the database efficient under high load. Finn followed his advice, and the system became more responsive.

Over the next days, others chimed in. **March**, an expert in memory management, guided Finn through tuning cache settings. **April** introduced clever techniques that allowed him to monitor the system without creating bottlenecks.

As the week wore on, Finn's understanding grew with each piece of wisdom. **May** taught him about effective backup and replication, essential for resilience. **June** offered guidance on data security and access permissions, while **July** showed him ways to streamline code through stored procedures and triggers.

The database was transforming before his eyes. By midweek, **August** helped Finn make sense of query execution plans, while **September** shared advanced indexing strategies that brought a whole new level of optimization. **October** provided insights into Firebird's inner workings, helping Finn connect the dots between each concept he'd learned.

Finally, **November**, a respected figure in the community, hinted at a special setting hidden within Firebird's configuration file. "Look closely, follow the leads, and you might find an option that could unlock performance beyond standard configurations," he said cryptically. "Firebird holds secrets known only to those who delve deeply. Use them wisely, and always keep backup plans ready."

At last, **December**, the most experienced of the **Twelve**, offered a final message: "Remember, Firebird rewards those who respect its strengths and limitations. Mastery comes through understanding, not force. Now, go and let your knowledge carry the system forward."

By the end of the week, Finn had transformed the database. When the CTO reviewed the server's performance, he was astounded. "I don't know how you managed this, but you've saved us from embarrassment," he admitted grudgingly.

However, his praise was short-lived. "But Finn," he added, narrowing his eyes, "this level of performance is our new standard. I expect even better results next quarter—or we'll find someone else who can keep up."

Exhausted but undeterred, Finn returned to his desk. He knew the Firebird community—and the **Twelve** who had guided him—would always be there, ready to help those in need. Just then, he noticed a final message from **December** appear on his screen:

"Firebird is like the seasons, ever-reliable and full of renewal. Those who respect it will always find its hidden strengths. Use what you've learned wisely."

And with that, Finn continued his journey, a humble yet skilled keeper of Firebird's secrets, knowing he would never face his challenges alone.

EmberWings

The official quarterly magazine of the Firebird Project

Do you develop with Firebird?

Are you using Firebird as a database backend for your applications? Share your experience and help shape its future! Your insights on how you develop with Firebird, the tools you rely on, and your wishlist for improvements will directly impact its development. The survey takes just 5-10 minutes—join us in building a better Firebird!

Take the Developer Experience survey

Do you manage Firebird deployment?

Your insights as a Firebird administrator are invaluable! By taking just a few minutes to complete this survey, you'll help shape the future of Firebird, identify key challenges, and improve the tools and features you use every day.

Participate in the Admin survey

We value your opinion! Help us improve **EmberWings** magazine by sharing your thoughts and feedback. Our quick questionnaire will only take a few minutes, and your responses will guide us in making future issues more relevant, engaging, and valuable to the Firebird community.

Help us improve EmberWings!

Published by Firebird Foundation z.s. © 2024