Firebird 3.0 Release Notes

Helen Borrie (Collator/Editor)

23 July 2013 - Document v.0300-06 - for Firebird 3.0 Alpha 1

Firebird 3.0 Release Notes

23 July 2013 - Document v.0300-06 - for Firebird 3.0 Alpha 1

Helen Borrie (Collator/Editor)

Table of Contents

List of Tables

Chapter 1

General Notes

Thank you for trying out this Alpha release of the forthcoming Firebird 3.0. We cordially invite you to test it hard against your expectations and engage with us in identifying and fixing any bugs you might encounter.

Bug Reporting

- If you think you have discovered a new bug in this release, please make a point of reading the instructions for bug reporting in the article How to Report Bugs Effectively, at the Firebird Project website.
- If you think a bug fix hasn't worked, or has caused a regression, please locate the original bug report in the Tracker, reopen it if necessary, and follow the instructions below.

Follow these guidelines as you attempt to analyse your bug:

1. Write detailed bug reports, supplying the exact build number of your Firebird kit. Also provide details of the OS platform. Include reproducible test data in your report and post it to our Tracker.
2. You are warmly encouraged to make yourself known as a field-tester of this pre-release by subscribing to the field-testers' list and posting the best possible bug description you can.
3. If you want to start a discussion thread about a bug or an implementation, please do so by subscribing to the firebird-devel list. In that forum you might also see feedback about any tracker ticket you post regarding this alpha.

Documentation

You will find all of the README documents referred to in these notes—as well as many others not referred to—in the doc sub-directory of your Firebird 3.0 installation.

An automated "Release Notes" page in the Tracker provides lists and links for all of the Tracker tickets associated with this and other pre-release versions. Use this link.

--*The Firebird Project*

Chapter 2

New About Firebird 3.0

The primary goals for Firebird 3 were to unify the server architecture and to improve support for SMP and multiple-core hardware platforms. Parallel objectives were to improve threading of engine processes and the options for sharing page cache across thread and connection boundaries.

Alongside these aims came new strategies to improve performance, query optimization, monitoring and scalability and to address the demand for more security options. A number of popular features were introduced into the SQL language, including the long-awaited support for the Boolean data type and the associated logical predications.

Summary of Features

The following list summarises the features and changes, with links to the chapters and topics where more detailed information can be found.

Unification of the Firebird executable is complete

With the completion of true SMP support for Superserver, the Firebird core is now a unified library that supports a single ODS, loadable either as an embedded engine or by the “network listener” executable. Choice of server model is determined by settings for two new configuration parameters defining the locking and cache models, respectively: `SharedDatabase` and `SharedCache`. They can be specified at either global level (in `firebird.conf`) or "per database" (in `databases.conf`).

By default, SharedDatabase = false and SharedCache = true, thus meaning SuperServer.

> **Note**
>
> The previous `aliases.conf` is replaced by `databases.conf`, now including not just aliases for databases but also (optionally) configuration parameters to enable configuration of databases and/or alternative security databases individually.

The changes are described in more detail in the chapter Changes in the Firebird Engine.

True SMP support for SuperServer

In Superserver mode, the engine now makes use of multiple CPUs and cores when spawning connections.

Tracker: CORE-775

Implemented by V. Khorsun

New, object-oriented C++ APIs

Object-oriented C++ APIs enable external code routines to plug in and run safely inside Firebird engine space, including (but not limited to):

- Stored procedures, triggers and functions written in Java, C++, ObjectPascal, etc.
- Encryption schemes for data

- User authentication schemes, including secure key exchange
- ..

New Data Type Support
A true BOOLEAN type (True/False/Unknown), complete with support for logical predicates, e.g.,

```
UPDATE ATABLE
  SET MYBOOL = (COLUMN1 IS DISTINCT FROM COLUMN2)
```

See BOOLEAN Type.

IDENTITY type, spawning unique identifiers for the defined column from an internal generator. See IDENTITY-Style Column.

Support for SQL Packages
For details, refer to Packages.

DDL Triggers
Now, triggers can be written to execute when database objects are modified or deleted. A typical use is to block unauthorised users from performing these tasks.
For details, refer to DDL Triggers.

'Window' functions in DML
A whole new series of analytical functions to work with multiple subsets in DML. See Window (Analytical) Functions.

Chapter 3

Changes in the Firebird Engine

In Firebird 3, the remodelling of the architecture that was begun in v.2.5 was completed with the implementation of full SMP support for the Superserver model. In the new scheme, it is possible to configure the execution model individually per database.

Remodelled Architecture

Dmitry Yemanov

The remodelled architecture integrates the core engine for Classic/Superclassic, Superserver and embedded models in a common binary. The cache and lock behaviours that distinguish the execution models are now determined externally by the settings in two new configuration parameters: **SharedDatabase** and **SharedCache** and the connection method in the parameter **Providers**. The parameters for configuring the architecture are specified globally (in `firebird.conf`) and can be overridden specifically for a database (in `databases.conf`).

> **Note**
>
> `databases.conf` is the old `aliases.conf` with a new name. In Firebird 3, the role of this file involves (potentially) much more than being just a lookup for database file paths. For more details about this, refer to the chapter Configuration Additions and Changes.

Working Modes ("Models")

Table 3.1. Matrix of Working Modes (D. Yemanov)

	SharedDatabase=0	SharedDatabase=1
SharedCache=0	Single user	Classic, SuperClassic
SharedCache=1	Superserver	See Note

In `firebird.conf` the defaults make SharedDatabase false (=0) and SharedCache true (=1), i.e., SuperServer.

> **SharedDatabase=1, SharedCache=1**
>
> This mode, although theoretically possible, is not supported currently.

Execution Modes

Classic and SuperClassic

Classic and SuperClassic are set up using the same configuration: SharedDatabase = true and SharedCache = false.

- On Linux, the server startup method determines which will run, i.e., running *xinetd* means Classic, while running the *firebird* means SuperClassic.
- On Windows, the command line options for *firebird.exe* specify the mode, just as they did in v.2.5 for *fb_inet_server.exe*, i.e., switch *-m* (for multi-threaded) means SuperClassic; otherwise Classic is implied.

Single-user Mode

SharedDatabase = false and SharedCache = false means that only one connection is possible, i.e., single user mode.

Whether a "hostless" connection is handled by the embedded engine or by the network listener (e.g. the XNET case), the settings SharedDatabase and SharedCache define the behaviour at the engine level. For an embedded connection, SharedDatabase = false and SharedCache = true would mean the pre-v.2.5 embedded behaviour on Windows (based on Superserver), while SharedDatabase = true and SharedCache = false would mean the v.2.5 embedded behaviour (based on SuperClassic).

How the connection string is processed depends on order specified in the Providers setting. The default setting is `Remote, Engine12, Loopback`. Connection strings to hosts are handled by the **Remote** provider, while "hostless" ones are handled, in turn, by **Engine12** or **Loopback**.

Thus, `libEngine12.so` or `engine12.dll` (as appropriate to platform) is available to the Dispatcher (**y-valve**), a "hostless" connection will be handled by the embedded engine; otherwise it will be handled by the loopback provider (XNET on Windows, TCP via localhost on POSIX).

> **Note**
>
> Of course, technically, XNET is not a local loopback provider (a local connection through a remote interface) and, in previous Firebird versions, it was treated as being in the "remote" space. On Firebird 3, it belongs with the local loopback providers.

Providers

The providers are more or less what we traditionally thought of as the methods used to connect a client to a server, viz., across a network, host-locally, via the local loopback ("localhost") or by a more direct local connection (the old `libfbembed.so` on POSIX, now implemented as the plug-in library `libEngine12.so`; on Windows, `engine12.dll`; on MacOSX, `engine12.dylib`).

In firebird.conf, all are available by default, viz.,

```
#Providers = Remote,Engine12,Loopback
```

> **Note**
>
> In databases.conf, one or more providers can be blocked by pasting the uncommented line and deleting the unwanted provider[s].

The Providers Architecture

Alex Peshkov

Although a key feature of Firebird 3, the Providers architecture is not new. Providers existed historically in Firebird's predecessors and, though well hidden, are present in all previous versions of Firebird. They were introduced originally to deal with a task that has been performed latterly by “interface layers” such as ODBC, ADO, BDE and the like, to enable access to different database engines using a single external interface.

Subsequently, this Providers architecture (known then as Open Systems Relational Interface, OSRI) also showed itself as very efficient for supporting a mix of old and new database formats—different major on-disk structure versions—on a single server having mixed connections to local and remote databases.

The providers implemented in Firebird 3 make it possible to support all these modes (remote connections, databases with differing ODS, foreign engines) as well as *chaining* providers. Chaining is a term for a situation where a provider is using a callback to the standard API when performing an operation on database.

The Components

The main element of the Providers architecture is the **y-valve**. On the initial `attach` or `create database` call **y-valve** scans the list of known providers and calls them one by one until one of them completes the requested operation successfully. For a connection that is already established, the appropriate provider is called at once with almost zero overhead.

Let's take a look at some samples of **y-valve** operation when it selects the appropriate provider at the `attach` stage. These use the default configuration, which contains three providers: **Remote** (establish network connection), **Engine12** (main database engine) and **Loopback** (force network connection to the local server for <database name> without an explicit network protocol being supplied).

The typical client configuration works this way: when one attaches to a database called `RemoteHost:dbname` (TCP syntax) or `\\RemoteHost\dbname` (NetBios) the **Remote** provider detects explicit network protocol syntax and, finding it first in the Provider list, redirects the call to RemoteHost.

When <database name> does not contain a network protocol but just the database name, the **Remote** provider rejects it and the **Engine12** provider comes to the fore and tries to open the named database file. If it succeeds, we get an embedded connection to the database.

> **Note**
>
> A special “embedded library” is no longer required. To make the embedded connection, the standard client loads the appropriate provider and becomes an embedded server.

Failure Response

But what happens if the engine returns an error on an attempt to attach to a database?

- If the database file to be attached to does not exist there is no interest at all.
- An embedded connection may fail if the user attaching to it does not have enough rights to open the database file. That would be the normal case if the database was not created by that user in embedded mode or if he was not explicitly given OS rights for embedded access to databases on that box.

  > **Note**
  >
  > Setting access rights in such a manner is a requirement for correct Superserver operation.

- After a failure of **Engine12** to access the database, the **Loopback** provider is attempted for an attach. It is not very different to **Remote** except that it tries to access the named database `<dbname>` on a server running a TCP/IP local loopback.

  On Windows, the XNET protocol (also known as “Windows local connection”) is used for it. POSIX systems prepend `<dbname>` with `localhost:` and use a TCP connection.

  If the attachment succeeds, a remote-like connection is established with the database even though it is located on the local machine.

Other Providers

Use of providers is not limited to the three standard ones. Firebird 3 does not support pre-ODS 12 databases but Firebird 3 will have an additional provider to access older databases (ODS 8 to 11.x). Removing support for old formats from the engine helps to simplify its code and gain a little speed. Taking into account that this speed gain sometimes takes place in performance-critical places, like searching a key in an index block, avoiding old code and related branches really does make Firebird fly faster.

Nevertheless, the Providers architecture does make it possible to access old databases when changing to a higher version of Firebird.

Custom Providers

A strong feature of the Providers architecture is ability for the deployer to add his own providers to the server, the client, or both.

So what else might be wanted on a client, other than a remote connection? Recall *Provider chaining* that was mentioned earlier. Imagine a case where a database is accessed via very slow network connection, say something like 3G or, worse, GPRS. What comes to mind as a way to speed it up is to cache on the client some big tables that rarely change. Such systems were actually implemented but, to do it, one had to rename `fbclient` to something arbitrary and load it into its own library called `fbclient`, thus making it possible to use standard tools to access the database at the same time as caching required tables. It works but, as a solution, it is clearly not ideal.

With the Providers architecture, instead of renaming libraries, one just adds a local caching provider which can use any method to detect connections to it (something like a `cache@` prefix at the beginning of the database name, or whatever else you choose).

In this example, when the database name `cache@RemoteHost:dbname` is used, the caching provider accepts the connection and invokes the **y-valve** once more with the traditional database name `RemoteHost:dbname`. When the user later performs any call to his database, the caching provider gets control of it before **Remote** does and, for a locally cached table, can avoid making calls to the remote server.

Use of chaining allows a lot of other useful things to be implemented, such as database replication without the need for triggers: just repeat the same calls for the replication host when, for example, a transaction is committed. In this case, the chaining provider is installed on the server, not the client, and no modification of the command line is needed at all.

To avoid cycling when performing a callback to **y-valve** at attach time, such a provider can modify the list of providers using the `isc_dpb_config parameter` in the DPB. The same technique may be used at the client, too.

For details, see the Configuration Additions and Changes chapter.

The ability to access foreign database engines using providers should not be overlooked, either. It might seem strange to consider this, given the number of tools available for this sort of task. Think about the ability to access other Firebird databases using EXECUTE STATEMENT, that became available in Firebird 2.5. With a provider to ODBC or other common tool to access various data sources it is within reach to use EXECUTE STATEMENT to get direct access from procedures and triggers, to data from any database having a driver for the chosen access tool. It is even possible to have a provider to access some particular type of foreign database engine if there is some reason to want to avoid the ODBC layer.

Providers Q & A

Q. Interfaces and providers are probably very good, but I have an old task written using plain API functions and for a lot of reasons I can't rewrite it in the near future. Does it mean I will have problems migrating to Firebird 3?

- A. Definitely no problems. The old API is supported for backward compatibility in Firebird 3 and will be supported in future versions as long as people need it.

And what about performance when using the old API?

- A. The functional API is implemented as a very thin layer over interfaces. Code in most cases is trivial: convert passed handles to pointers to interfaces—a step was always present but referred to as “handle validation”—and invoke the appropriate function from the interface.

  Functions that execute an SQL operator and fetch data from it are a place where it is a little more complex. The SQLDA and the data moves related to it have never been the fastest part of the functional API, anyway. It was one the reasons to have the new API and the logic between the new and old APIs does not add much to that old overhead.

Plug-Ins

Alex Peshkov

From version 3 onward, Firebird's architecture supports plug-ins. For a number of pre-defined points in the Firebird code, a developer can write his own fragment of code for execution when needed.

A plug-in is not necessarily one written by a third party: Firebird has a number of intrinsic plug-ins and, as will be seen, even some core parts of Firebird are implemented as plug-ins.

What is a Plug-In?

The term “plug-in” is often used to name related but different things:

- a dynamic library, containing code to be loaded as a plug-in (often called a *plug-in module*) and stored in the `$FIREBIRD/plugins` directory;
- code *implementing* a plug-in. That is slightly different from the *library*, since a single dynamic library may contain code for more than one plug-in;
- a plug-in's *factory*: an object created by that code (pure virtual C++ class), creating instances of the plug-in at Firebird's request;
- an *instance* of the plug-in, created by its factory.

Plug-In Types

Firebird's plug-in architecture makes it possible to create plug-ins of predefined types. Each version has a fixed set of supported plug-in types. To add a further type, the first requirement is to modify the Firebird code. Our plug-in architecture facilitates both adding new types of plug-ins and simplifying the coding of the plug-in along generic lines.

To be able to implement a plug-in, say, for encrypting a database on the disk, the Firebird code has to be prepared for it: it must have a point from which the plug-in is called.

The set of plug-in types implemented in Firebird 3 comprises:

user authentication related:
- AuthServer (validates user's credentials on server when logins are used)
- AuthClient (prepares credentials to be passed over the wire)
- AuthUserManagement (maintains a list of users on a server in a form, known to AuthServer)

ExternalEngine
: Controls the use of various engines, see External Engines.

Trace
: The Trace plug-in was introduced in Firebird 2.5, but the way it interacts with the engine was changed in Firebird 3 to accord with the new generic rules.

Encryption
: encrypting plug-ins are for
  - network (WireCrypt)
  - disk (DbCrypt)
  - a helper plug-in (KeyHolder), used to help maintain the secret key(s) for DbCrypt

Provider
: Firebird 3 supports providers as a plug-in type.

Technical Details

Plug-ins use a set of special Firebird interfaces. All plug-in-specific interfaces are reference counted, thus putting their lifetime under specific control. Interfaces are declared in the include file plug-in.h. `DbCrypt_example` provides a simple model for writing a plug-in module

> **Note**
>
> The example does not perform any actual encryption, it is just a sample of how to write a plug-in. Complete instructions for writing plug-ins are not in scope for this document.

Features of a Plug-In

A short list of plug-in features:

- You can write a plug-in in any language that supports pure virtual interfaces. Interface declarations will need to be written for your language if they are missing.
- As with UDFs, you are free to add any reasonable code to your plug-in#with emphasis on *reasonable*. For example, asking a question at the server's console from a plug-in is hardly “reasonable”!
- Calling the Firebird API from your plug-in is OK, if needed. For example, the default authentication server and user manager use a Firebird database to store accounts.
- Firebird provides a set of interfaces to help with configuring your plug-ins. It is not obligatory to use them, since the plug-in code is generic and can employ any useful method for capturing configuration information. However, using the standard tools provides commonality with the established configuration style and should save the additional effort of rolling your own and documenting it separately.

Configuring Plug-ins

Configuration of plug-ins has two parts:

1. The engine has to be instructed what plug-ins it should load
2. The plug-ins themselves sometimes need some configuration.

The plug-ins to be loaded for each type of plug-in are defined in the main configuration file, `firebird.conf`, usually with defaults. The ones defined in Firebird 3 are discussed in the chapter entitled “Configuration Additions and Changes”. In summary, the set that provides normal operation in the server, client and embedded cases consists of:

- AuthServer = Srp, Win_Sspi
- AuthClient = Srp, Win_Sspi, Legacy_Auth
- UserManager = Srp
- TracePlugin = fbtrace
- Providers = Remote,Engine12,Loopback
- WireCryptPlugin = Arc4

> **Note**
>
> If you want to add other plug-ins, they must be cited in firebird.conf. Apart from other considerations, this requirement acts as a security measure to avoid loading unknown code.

Taking the entry **TracePlugin = fbtrace** as an example, what does the value **fbtrace** signify? In a trivial case, it can indicate the name of a dynamic library but the precise answer is more complicated.

As mentioned earlier, a single plug-in module may implement more than one plug-in. In addition, a single plug-in may have more than one configuration at once, with a separate plug-in factory created for each configuration. Each of these three object contexts (module | implementation | factory) has its own name:

- The name of a module is the file name of a dynamic library
- The name of a plug-in implementation is the one given to it by the developer of the plug-in. It is hard-coded inside the module.
- The name of a factory is, by default, the same as the name of the plug-in implementation's name. It is the factory name which is actually used in `firebird.conf`.

In a typical trivial case, where a module contains one plug-in that works with just one configuration and all three names are equal, and no more configuration is needed. An example would be libEngine12.so | Engine12.dll | Engine12.dylib, that contains the implementation of the embedded provider Engine12. Nothing other than the record **Providers = Engine12** is needed to load it.

For something more complex a file will help you to set up the plug-in factories precisely.

plugins.conf

The file `$(root)/plugins.conf` has two types of records: **config** and **plugin**.

the **plugin** record is a set of rules for loading land activating the plug-in. Its format is:

```
Plugin = PlugName ## this is the name to be referenced in firebird.conf
  {
    Module = LibName ## name of dynamic library
    RegisterName = RegName ## name given to plug-in by its developer
    Config = ConfName ## name of config record to be used
    ConfigFile = ConfFile ## name of a file that contains plug-in's configuration
  }
```

When plug-in *PlugName* is needed, Firebird loads the library *LibName* and locates the plug-in registered with the name *RegName*. The configuration from the config record *ConfName* or the config file *ConfFile* are passed to the library.

> **Note**
>
> If both *ConfName* and *ConfFile* are given, then the config record will be used.
>
> If both parameters are missing, the default *PlugName* is used; **except that** if the **ConfigFile** is present and its name is the same as the module's dynamic library but with a `.conf` extension, it will be used.

The **ConfigFile** is expected to use the format **Key=Value**, in line with other Firebird configuration files.

For the plug-in record the same format is used:

```
Config = ConfName
```

```
  {
    Key1 = Value1
    Key2 = Value2
    ...
  }
```

A Sample Setup

Suppose you have a server for which some clients trust the wire encryption from one vendor and others prefer a different one. They have different licences for the appropriate client components but both vendors use the name “BestCrypt” for their products.

The situation would require renaming the libraries to, say, WC1 and WC2, since there cannot be two files in the same directory with the same name. Now, the modules stop loading automatically because neither is called “BestCrypt” any longer.

To fix the problem, `plug-ins.conf` should contain something like this:

```
Plugin = WC1
  {
    RegisterName = BestCrypt
  }
Plugin = WC2
  {
    RegisterName = BestCrypt
  }
```

The module names will be automatically set to WC1 and WC2 and found. You can add any configuration info that the plug-ins need.

Remember to modify firebird.conf to enable both plug-ins for **WireCryptPlugin** parameter:

```
WireCryptPlugin = WC1, WC2
```

The server will now select appropriate plug-in automatically to talk to the client.

Another sample is distributed with Firebird, in `$(root)/plugins.conf`, configuring one of the standard plug-ins, UDR. Because it was written to a use non-default configuration, the module name and one configuration parameter are supplied explicitly.

Plug-Ins Q & A

Q. There are plug-ins named Remote, Loopback, Arc4 in the default configuration, but no libraries with such names. How do they work?

- A. They are “built-in” plug-ins, built into fbclient library, and thus always present. Their existence is due to the old ability to distribute the Firebird client for Windows as a single dll. The feature is retained for cases where the standard set of plug-ins is used.

Q. What do the names of Srp and Arc4 plug-ins mean?

- A. Srp implements the Secure Remote Passwords protocol, the default way of authenticating users in Firebird 3. Its effective password length is 20 bytes, resistant to most attacks (including “man in the middle”) and works without requiring any key exchange between client and server to work.

  Arc4 means Alleged RC4 - an implementation of RC4 cypher. Its advantage is that it can generate a unique, cryptographically strong key on both client and server that is impossible to guess by capturing data transferred over the wire during password validation by SRP.

  The key is used after the SRP handshake by Arc4, which makes wire encryption secure without need to exchange any keys between client and server explicitly.

Q. What do Win_Sspi and Legacy_Auth mean?

- A. Windows SSPI has been in use since Firebird 2.1 for Windows trusted user authentication. Legacy_Auth is a compatibility plug-in to enable connection by the Firebird 3 client to older servers. It is enabled by default in the client.

  And Yes, it still transfers almost plain passwords over the wire, for compatibility.

  On the server it works with a security database from Firebird 2.5, and should be avoided except in situations where you understand well what are you doing.

  To use Legacy_Auth on the server you will need to disable network traffic encryption in `firebird.conf`:

```
WireCrypt = Disabled
```

External Engines

Adriano dos Santos Fernandes

The UDR (User Defined Routines) engine adds a layer on top of the FirebirdExternal engine interface with the purpose of

- establishing a way to hook external modules into the server and make them available for use
- creating an API so that external modules can register their available routines
- making instances of routines “per attachment”, rather than dependent on engine the internal implementation details of the engine

External Names

An external name for the UDR engine is defined as

```
'<module name>!<routine name>!<misc info>'
```

The <module name> is used to locate the library, <routine name> is used to locate the routine registered by the given module, and <misc info> is an optional user-defined string that can be passed to the routine to be read by the user.

Module Availability

Modules available to the UDR engine should be in a directory listed by way of the path attribute of the corresponding plugin_config tag. By default, a UDR module should be on <fbroot>/plugins/udr, in accordance with its path attribute in <fbroot>/plugins/udr_engine.conf.

The user library should include FirebirdUdr.h (or FirebirdUdrCpp.h) and link with the udr_engine library. Routines are easily defined and registered, using some macros, but nothing prevents you from doing things manually.

> **Note**
>
> A sample routine library is implemented in `examples/udr`, showing how to write functions, selectable procedures and triggers. It also shows how to interact with the current attachment through the legacy API.

Scope

The state of a UDR routine (i.e., its member variables) is shared among multiple invocations of the same routine until it is unloaded from the metadata cache. However, it should be noted that the instances are isolated “per session”.

Character Set

By default, UDR routines use the character set that was specified by the client.

> **Note**
>
> In future, routines will be able to modify the character set by overriding the **getCharSet** method. The chosen character set will be valid for communication with the ISC library [ADRIANO: which library do you refer to here?] as well as the communications passed through the FirebirdExternal API.

Enabling UDRs in the Database

Enabling an external routine in the database involves a DDL command to “create” it. Of course, it was already created externally and (we hope) well tested.

Syntax Pattern

```
{ CREATE [ OR ALTER ] | RECREATE | ALTER } PROCEDURE <name>
  [ ( <parameter list> ) ]
  [ RETURNS ( <parameter list> ) ]
  EXTERNAL NAME '<external name>' ENGINE <engine>

{ CREATE [ OR ALTER ] | RECREATE | ALTER } FUNCTION <name>
  [ <parameter list> ]
  RETURNS <data type>
  EXTERNAL NAME '<external name>' ENGINE <engine>
```

```
  { CREATE [ OR ALTER ] | RECREATE | ALTER } TRIGGER <name>
    ...
    EXTERNAL NAME '<external name>' ENGINE <engine>
```

Examples

```
create procedure gen_rows (
  start_n integer not null,
  end_n integer not null
) returns (
  n integer not null
) external name 'udrcpp_example!gen_rows'
  engine udr;

create function wait_event (
  event_name varchar(31) character set ascii
) returns integer
  external name 'udrcpp_example!wait_event'
  engine udr;

create trigger persons_replicate
  after insert on persons
  external name 'udrcpp_example!replicate!ds1'
  engine udr;
```

How it Works

The external names are opaque strings to Firebird. They are recognized by specific external engines. External engines are declared in configuration files, possibly in the same file as a plug-in, as in the sample library is implemented in `examples/plugins`.

```
    <external_engine UDR>
        plugin_module UDR_engine
    </external_engine>

    <plugin_module UDR_engine>
        filename $(this)/udr_engine
        plugin_config UDR_config
    </plugin_module>

    <plugin_config UDR_config>
        path $(this)/udr
    </plugin_config>
```

When Firebird wants to load an external routine (function, procedure or trigger) into its metadata cache, it gets (see note below)the external engine through the plug-in external engine factory and asks it for the routine. The plug-in used is the one referenced by the attribute **plugin_module** of the external engine.

> **Note**
>
> Depending on the server architecture (Superserver, Classic, etc) and implementation details, Firebird may get external engine instances “per database” or “per connection”. Currently, it always gets instances “per database”.

Miscellaneous Improvements

Miscellaneous engine improvements include.-

Internal Debug Info Made Human-readable

Vlad Khorsun

A new BLOB filter translates internal debug information into text.

A Silly Message is Replaced

Claudio Valderrama C.

A silly message sent by the parser when a reference to an undefined object was encountered was replaced with one that tells it like it really is.

New Pseudocolumn RDB$RECORD_VERSION

Adriano dos Santos Fernandes

A pseudocolumn named RDB$RECORD_VERSION returns the number of the transaction that created the current record version.

It is retrieved the same way as RDB$DB_KEY, i.e., **select RDB$RECORD_VERSION from aTable where...**

Chapter 4

Changes to the Firebird API and ODS

ODS (On-Disk Structure) Changes

New ODS Number

Firebird 3.0 creates databases with an ODS (On-Disk Structure) version of 12.

Implementation ID is Deprecated

Alex Peshkov

The Implementation ID in the ODS of a database is deprecated in favour of a new field in database headers describing hardware details that need to match in order for the database to be assumed to have been created by a compatible implementation.

The old Implementation ID is replaced with a 4-byte structure consisting of hardware ID, operating system ID, compiler ID and compatibility flags. The three ID fields are just for information: the ODS does not depend upon them directly and they are not checked when opening the database.

The compatibility flags *are checked* for a match between the database and the engine opening it. Currently we have only one flag, for endianness. As previously, Firebird will not open a database on little-endian that was created on big-endian, nor vice versa.

Sample gstat Output

```
# ./gstat -h employee
Database “/usr/home/firebird/trunk/gen/Debug/firebird/examples/empbuild/employee.fdb”
Database header page information:
        ..............
        Implementation          HW=AMD/Intel/x64 little-endian OS=Linux CC=gcc
        ..............
```

The purpose is to make it easier to do ports of Firebird for new platforms.

Maximum Page Size

The maximum page size remains 16 KB (16384 bytes).

Maximum Number of Page Buffers in Cache

The maximum number of pages that can be configured for the database cache depends on whether the database is running under 64-bit or 32-bit Firebird:

- 64-bit :: 2^31 -1 (2,147,483,647) pages
- 32-bit :: 128,000 pages, i.e., unchanged from V.2.5

Changes to System Tables

RDB$SYSTEM_FLAG

Claudio Valderrama C.

RDB$SYSTEM_FLAG has been made NOT NULL in all tables.

CORE-2787.

RDB$TYPES

Dmitry Yemanov

Missing entries were added to RDB$TYPES. They describe the numeric values for these columns:

```
RDB$PARAMETER_TYPE     (table RDB$PROCEDURE_PARAMETERS)
RDB$INDEX_INACTIVE     (table RDB$INDICES)
RDB$UNIQUE_FLAG        (table RDB$INDICES)
RDB$TRIGGER_INACTIVE   (table RDB$TRIGGERS)
RDB$GRANT_OPTION       (table RDB$USER_PRIVILEGES)
RDB$PAGE_TYPE          (table RDB$PAGES)
RDB$PRIVATE_FLAG       (tables RSB$PROCEDURES and RDB$FUNCTIONS)
RDB$LEGACY_FLAG        (table RDB$FUNCTIONS)
RDB$DETERMINISTIC_FLAG (table RDB$FUNCTIONS)
```

Application Programming Interfaces

A new public API replaces the legacy one in new applications, especially object-oriented ones. The interface part can be found in the header file `Provider.h` in the directory `/include/firebird` beneath the installation root directory.

> **Note**
>
> POSIX installations have a symlink pointing to `/usr/include/firebird/Provider.h`

The new public API can be also used inside user-defined routines (UDR, q.v.) for callbacks inside the engine, allowing a UDR to select or modify something in the database, for example.

The main difference between the new API and the legacy one is that UDRs can query and modify data in the same connection or transaction context as the user query that called that UDR. It is now possible to write external triggers and procedures, not just external functions (UDFs).

Interfaces and the New Object-oriented API

Alex Peshkov

Firebird needed a modernised API for a number of compelling reasons.

- High on the list was the limitation of the 16-bit integer pervading the legacy API, encompassing message size, SQL operator length, BLOB data portions, to name a few examples. While 16-bit was probably adequate when that old API came to life, in today's environments it is costly to work around.

  A trivial solution might be to add new functions that support 32-bit variables. The big downside is the obvious need to retain support for the old API by having pairs of functions with the same functionality but differing integer sizes. In fact, we did something like this to support 64-bit performance counters, for no better reason than being pressed to provide for it without having a more elegant way to implement it.

- Another important reason, less obvious, derives from the era when Firebird's predecessor, InterBase, did not support SQL. It used a non-standard query language, GDML, to manage databases. Data requests were transported between client and server using messages whose formats were defined at request compilation time in BLR (binary language representation). In SQL, the operator does not contain the description of the message format so the decision was taken to surround each message with a short BLR sequence describing its format.

  For some reason, that rule was followed so slavishly that every fetch of data from the server now required sending the BLR for it, not just the formatting BLR that was sent at SQL compile time.

  The reason for such apparent strangeness was the SQLDA layer (XSQLDA) that rides on top of the message-based API, invented as an attempt to work around the inefficiency of sending the BLR at every turn.

  The trap with the XSQLDA solution is that it encapsulates both the location of the data and their format, making it possible to change location or format (or both) between fetch calls. Hence, the need for the BLR wrapping in *every* fetch call—notwithstanding, this potential capability to change the data format between fetches was broken in the network layer before Firebird existed.

  This system involving calls processing data through multiple layers is hard to extend and wastes performance; the SQLDA is not simple to use; the desire to fix it was strong.

- Other reasons—numerous but perhaps less demanding—for changing the API included enhancing the status vector and optimizing dynamic library loading. Interfaces also make it so much easier and more comfortable to use the messages API.

The Non-COM Choice

The new interfaces are not compatible with COM, deliberately, and the reasons have to do with future performance enhancement.

At the centre of the Providers architecture in Firebird 3.0 is the **y-valve**, which is directed at dispatching API calls to the correct provider. Amongst the potential providers are older ones with potentially older interfaces. If we used COM, we would have to call the method **IUnknown** for each fetch call, just to ensure that the provider

really had some newer API method. Along with that comes the likelihood of future additions to the catalogue of API calls to optimize performance. A COM-based solution does not play well with that.

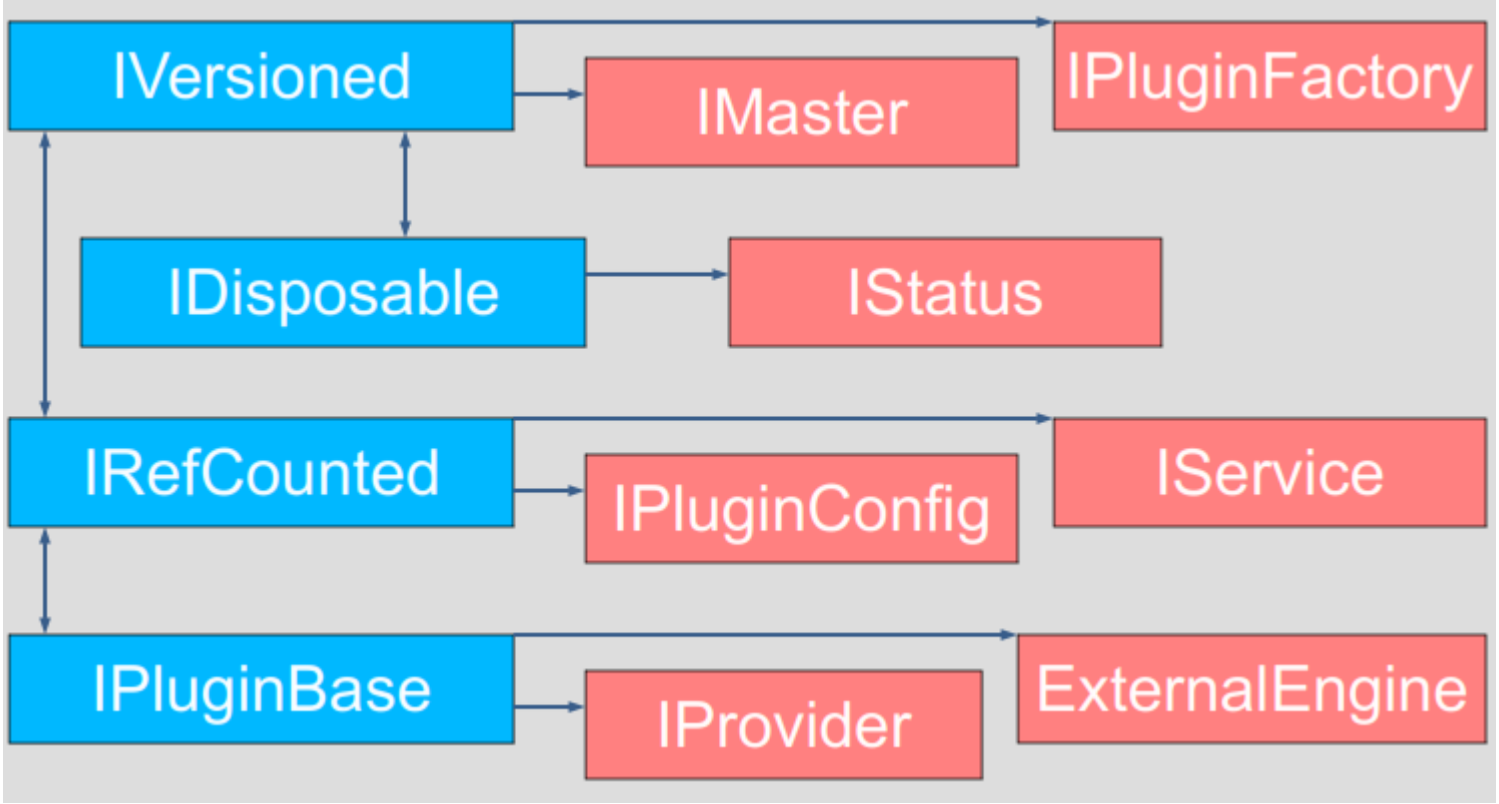
Firebird interfaces, unlike COM, support multiple versions. The interface version is determined by the total number of virtual functions it encompasses and it can be upgraded.

“Upgradable” does not imply that the older interface gets the full functionality of the new, though. After the upgrade, the virtual function table is expanded to include a function defined by the upgrade caller that will reasonably, if minimally, emulate the upgraded behaviour: return an error from a provider, for example, or “place-hold” a specific piece of content with an empty string.

Although the “poor man's upgrade” is no prettier than what would happen with IUnknown without it, the Firebird 3 solution provides the means to incorporate other methods that would be more appropriate.

The Hierarchy of Interfaces

A detailed discussion of all the functions presented by all the interfaces is outside the scope of this overview. The general schematic looks like this:



The base of the structure is **IVersioned**. It is the interface that enables a version upgrade. A lot of interfaces not requiring additional lifetime control are based directly on **IVersioned**. **IMaster** is one example already mentioned. Others include a number of callback interfaces whose lifetimes must match the lifetimes of the objects from which they were to be used for callback.

Two interfaces deal with lifetime control: **IDisposable** and **IRefCounted**. The latter is especially active in the creation of other interfaces: **IPlugin** is reference counted, as are many other interfaces that are used by plug-ins. These include the interfaces that describe database attachment, transaction management and SQL statements.

Not everything needs the extra overhead of a reference-counted interface. For example, IMaster, the main interface that calls functions available to the rest of the API, has unlimited lifetime by definition. For others, the API

is defined strictly by the lifetime of a parent interface; the IStatus interface is non-threaded. For interfaces with limited lifetimes it is of benefit to have a simple way to destroy them, that is, a dispose() function.

Each plug-in has one and only one main interface—**IPlugin**—which is responsible for basic plug-in functionality. In fact, a lot of plugins have only that interface, although that is not a requirement.

Finally, there is **IProvider**, a kind of "main" plug-in in the Firebird API. **IProvider** is derived from **IPlugin** and must be implemented by every provider. If you want to write your own provider you must implement **IProvider**. It is implemented also by the **y-valve**: it is the **y-valve** implementation that is returned to the user when the **getDispatcher()** function from the master interface is called.

IProvider contains functions enabling creation of an attachment to a database (attach and create) or to the Services Manager.

Interfaces Q & A

Q. We access new API using IMaster. But how to get access to IMaster itself?

- A. This is done using the only one new API function **fb_get_master_interface()**. It is exported by the `fbclient` library.

Q. The non-use of COM-based interfaces was said to be to avoid working with IUnknown methods and that this is done due to performance issues. Instead you have to upgrade interfaces. Why is that faster than using IUnknown?

- A. Upgrading an interface requires some work. In a case where the version matches the caller's requirements, it is not so big – just a check. When a real upgrade is needed, more CPU cycles will be spent. The main difference with the COM approach is that an upgrade is performed for the interface only once, after its creation, but IUnknown methods must be called each time we are going to call an interface with unknown version (or that version should be stored separately and later checked). For a Firebird interface, once upgraded, there is absolutely no waste of time when performing calls to it during all its lifetime.

Other New APIs

Other new APIs support various plug-ins by declaring the interfaces between the engine and the plug-in. Besides pluggable authentication and pluggable encryption, Firebird 3 supports "external engines", bridges between the engine and the execution environments that can run UDRs: native code, Java and others. By and large they are intended for use by third-party solution providers, rather than for client application development.

For creating custom plug-ins and bridges, the relevant interface (API) needs to be implemented in the plug-in code.

Available Interface (Header) Files

Authentication
`Auth.h`

Encryption
`Crypt.h`

External engines
`ExternalEngine.h`

API Improvements

Improvements to the legacy API include.-

Better Error Reports for String Overflows

Alex Peshkov

Include expected and actual string length in the error message for string overflows (SQLCODE -802).

More Detail in “Wrong Page Type” Error Reports

Alex Peshkov

More details in the error message "wrong page type", i.e., identifying expected and encountered page types by name instead of numerical type.

Chapter 5

Reserved Words and Changes

New Reserved Words in Firebird 3.0

DETERMINISTIC
OVER
RETURN
SCROLL
SQLSTATE

Chapter 6

Configuration Additions and Changes

The file `aliases.conf` is renamed to `databases.conf`. An old `aliases.conf` from a previous version can simply be renamed and the new engine will just continue to use it as before. However, `databases.conf` can now include some configuration information for individual databases.

Scope of Parameters

Some parameters are marked as configurable *per-database* or *per-connection*.

- Per-database configuration is done in `databases.conf`.
- Per-connection configuration is primarily for client tool use and is done using the DPB parameter **isc_dpb_config** or, for Services, the SPB parameter **isc_spb_config**.
- In the case of Embedded, the DPB can be used to tune per-database entries on first attaching to a database.

Macro Substitution

A number of predefined macros (syntax $(name)) is available for use in the configuration files to substitute for a directory name:

$(root)
: Root directory of Firebird instance

$(install)
: Directory where Firebird is installed

$(this)
: Directory where current configuration file is located

$(dir_conf)
: Directory where firebird.conf and databases.conf are located

$(dir_secdb)
: Directory where the default security database is located

$(dir_plugins)
: Directory where plugins are located

$(dir_udf)
Directory where UDFs are located by default

$(dir_sample)
Directory where samples are located

$(dir_sampledb)
Directory where sample DB (employee.fdb) is located

$(dir_intl)
Directory where international modules are located

$(dir_msg)
Directory where the messages file (firebird.msg) is located

Includes

One configuration file can be included in another by using an “include” directive, e.g.,

```
include some_file.conf
```

A relative path is treated as relative to the enclosing configuration file. So, if our example above is inside `/opt/config/master.conf` then our **include** refers to the file `/opt/config/some_file.conf`.

Wildcards

The standard wildcards ***** and **?** may be used in an **include** directive, to include all matching files in undefined order. For example,

```
include $(dir_plugins)/config/*.conf
```

Expression of Parameter Values

Integer byte values were traditionally specified by default in bytes and other integer values were digital. However, now you can optionally specify them in Kilobytes, Megabytes or Gigabytes, as appropriate, by adding K, M or G (case-insensitive). For example, **24M** is read as 25165824 (24 * 1024 * 1024).

Boolean values are expressed as non-zero (true)|zero (false) by default, but you may now use the quoted strings 'y', 'yes' or 'true' instead of a non-zero digit.

“Per-database” Configuration

Custom configuration at database level is achieved with formal entries in `databases.conf`.

Format of Configuration Entries

To come.

Parameters Available

The following parameters can be copy/pasted to `databases.conf` and used as overrides for specific databases. Asterisk pairs (**) mark parameters that can alternatively be configured at the client connection via the DPB/SPB. Please refer back to Scope of Parameters at the beginning of this chapter if you do not understand these differences.

Table 6.1. Parameters available in databases.conf

ExternalFileAccess	DefaultDbCachePages	DatabaseGrowthIncrement
FileSystemCacheThreshold	** AuthClient	
UserManager	CryptPlugin	** Providers
DeadlockTimeout	MaxUnflushedWrites	MaxUnflushedWriteTime
**only ConnectionTimeout	WireCrypt	**only DummyPacketInterval
**only RemoteServiceName	**only RemoteServicePort	**only RemoteAuxPot
**only TCPNoNagle	LockMemSize	LockAcquireSpins
LockHashSlots	EventMemSize	GCPolicy
SecurityDatabase	**only IpcName	**only RemotePipeName
SharedCache	SharedDatabase	

New Parameters

New parameters added to `firebird.conf` are:

SecurityDatabase

Defines the name and location of the security database that stores login user names and passwords used by the server to validate remote connections. By default, in `firebird.conf`, it is `$(root)/security3.fdb`. It can be overridden for a specific database by a configuration in `databases.conf`.

AuthServer and AuthClient

Two parameters that determine what authentication methods can be used by the network server and the client redirector. The enabled methods are listed as string symbols separated by commas, semicolons or spaces.

- Secure remote passwords (Srp), using the plug-in is the default, using the OS-appropriate plug-in (`libSrp.s0 | Srp.dll | Srp.dylib`)
- On Windows, the Security Support Provider Interface (Sspi) is used when no login credentials are supplied
- Client applications can use legacy authentication (Legacy_Auth) to talk to old servers.

For **AuthServer**, Srp and Win_Sspi are listed; for **AuthClient**, Srp, Win_Sspi and Legacy_Auth.

To disable a method, erase the comment marker (#) and remove the unwanted method from the list.

Both parameters can be used in `databases.conf`. They can both be used in the DPB or the SPB for a connection-specific configuration.

WireCrypt

Sets whether the network connection should be encrypted. It has three possible values: Required (=1) | Enabled (=2) | Disabled (=0). The default is set such that encryption is Required for connections coming in to the server and Enabled for connections outgoing to a client. [Ed.: more comprehension wanted here?]

UserManager

Sets the plug-in that will operate on the security database. It can be a list with blanks, commas or semicolons as separators: the first plug-in from the list is used.

The default plug-in is **Srp** (`libSrp.s0 | Srp.dll | Srp.dylib`).

The UserManager parameter can be used in `databases.conf` for a database-specific override.

TracePlugin

Specifies the plug-in used by Firebird's Trace facility to send trace data to the client app or audit data to the log file.

The default plug-in is **fbtrace** (`libfbtrace.s0 | fbtrace.dll | fbtrace.dylib`).

CryptPlugin

> **Note**
>
> This parameter name is considered confusing and will be changed after the initial Alpha release.

A crypt plug-in is used to encrypt and decrypt data transferred over the network.

The installation default **Arc4** implies use of an **Alleged RC4** plug-in. For Linux, an example plug-in named `libDbCrypt_example.so` can be found in the `/plugins/` sub-directory.

The configured plug-in, which requires a key generated by the configured **Auth?????** plug-in, can be overridden in the API for a specific connection via the DPB or the SPB.

> **Tip**
>
> For information about configuring plug-ins, see Configuring Plug-ins in the Engine chapter.

KeyHolderPlugin

This parameter would represent some form of temporary storage for database encryption keys. Nothing is implemented as a default plug-in but a sample Linux plug-in named `libCryptKeyHolder_example.so` can be found in `/plugins/`.

Providers

List of allowed transports for accessing databases, discussed in the Engine chapter.

SharedCache and SharedDatabase

Two parameters that, together, determine the execution mode of the server (“server model”). Discussed in the Engine chapter.

Parameters Removed or Deprecated

The following parameters have been removed or deprecated:

RootDirectory

In older version, this parameter provided a superfluous option for recording the file system path to Firebird's “root” files (firebird.conf, the security database and so on).

LegacyHash

This parameter used to make it possible to use the old `security.fdb` from Firebird 1.X installations after it had been subjected to an upgrade script and thence to enable or disable use of the obsolete DES hash encrypting algorithm. It is no longer supported.

OldSetClauseSemantics

This parameter enabled temporary support for an implementation fault in certain sequences of SET clauses in versions of Firebird prior to v.2.5. It is no longer available.

OldColumnNaming

This parameter temporarily enabled legacy code support for an old InterBase/Firebird 1.0 bug that generated unnamed columns for computed output which was not explicitly aliased in the SELECT specification. It is no longer available.

LockGrantOrder

This parameter used to allow the option to have Firebird's Lock Manager emulate InterBase v3.3 lock allocation behaviour, whereby locks would be granted in no particular order, as soon as soon as they were available, rather than by the normal order (first-come, first-served). The legacy option is no longer supported.

Obsolete Windows priority settings

UsePriorityScheduler, PrioritySwitchDelay and PriorityBoost, which were marginally relevant to obsolete processors on obsolete Windows versions, are no longer supported.

Chapter 7

Security

Security improvements in Firebird 3 include:

Location of User Lists

Alex Peshkov

CORE-685

Firebird now supports an unlimited number of security databases. Any database may act as a security database and can be a security database for itself.

Use `databases.conf` to configure a non-default security database. This example configures `/mnt/storage/private.security.fdb` as the security database for the first and second databases:

```
first = /mnt/storage/first.fdb
{
   SecurityDatabase = /mnt/storage/private.security.fdb
}

second = /mnt/storage/second.fdb
{
   SecurityDatabase = /mnt/storage/private.security.fdb
}
```

Here we use third database as its own security database:

```
third = /mnt/storage/third.fdb
{
   SecurityDatabase = third
}
```

> **Note**
>
> The value of the SecurityDatabase parameter can be a database alias or the actual database path.

Database Encryption

Alex Peshkov

CORE-657

With Firebird 3 comes the ability to encrypt data stored in database. Not all of the database file is encrypted: just data, index and blob pages.

To make it possible to encrypt a database you need to obtain or write a database crypt plug-in.

> **Note**
>
> The sample crypt plug-in in `examples/dbcrypt` does not perform real encryption, it is merely a sample of how to go about it.

Secret Key

The main problem with database encryption is how to store the secret key. Firebird provides a helper to transfer that key from the client but that does not imply that storing the key on a client is the best way: it is no more than a possible alternative. A very bad option is to keep the key on the same disk as the database.

Tasks

To separate encryption and key access efficiently, a database crypt plug-in is split into two parts: encryption itself and the secret key holder. This may be an efficient approach for third-party plug-ins when you want to use some good encryption algorithm but you have your own secret way to store a key.

Once you have decided on a crypt plug-in and a key, you can enable them with:

```
ALTER DATABASE ENCRYPT WITH <PLUGIN_NAME>
```

Encryption will start right after this statement commits and will be performed in background. Normal database activity is not disturbed during encryption.

> **Tip**
>
> Encryption progress may be monitored using the field `MON$CRYPT_PAGE` in the pseudo-table MON$DATABASE or watching the database header page using `gstat -he`.

To decrypt the database do:

```
ALTER DATABASE DECRYPT
```

New Authentication Method in Firebird 3

Alex Peshkov

All of the code related to authentication is plug-in-enabled. Though Firebird performs the generic work, like extracting authentication data from a network message or putting it into such messages as appropriate, all the

activity related to calculating hashes, storing data in databases or elsewhere, using specific prime numbers and so on is done by plug-ins.

Firebird 3 has new method of user authentication implemented as a default plugin: secure remote password (SRP) protocol. Quoting from Wikipedia:

“The SRP protocol creates a large private key shared between the two parties in a manner similar to Diffie-Hellman key exchange, then verifies to both parties that the two keys are identical and that both sides have the user's password. In cases where encrypted communications as well as authentication are required, the SRP protocol is more secure than the alternative SSH protocol and faster than using Diffie-Hellman key exchange with signed messages. It is also independent of third parties, unlike Kerberos.”

SSH needs key pre-exchange between server and client when placing a public key on the server to make it work. SRP does not need that. All a client needs are login and password. All exchange happens when the connection is established.

Moreover, SRP is resistant to “man-in-the-middle” attacks.

> **Important**
>
> Use of the new authentication method is not compatible with old security databases and passwords from them. There is no way to migrate users from Firebird 2.
>
> Use of an old security database can be supported with the Legacy_Auth authentication plug-in, but this kills the security benefits of Firebird 3.
>
> The Firebird 3 client is built to make it possible to talk to old servers with the default configuration.

SSL/TLS Support

CORE-3251

So, the answer to the question “Does Firebird use SSL/TLS for password validation?” is “yes and no”. The “No” answer comes because, by default, SSL is not used. That is due to a minor licensing incompatibility between Firebird and OpenSSL, the most popular SSL implementation.

The “Yes” applies because anyone is free to write an authentication plug-in that uses SSL and TLS.

Increased Password Length

CORE-1898.

Implementation of SRP in our plugin has increased the password length from 8 bytes. Because of the use of SHA1 for hashes, it is effectively limited to 20 bytes. A custom SRP plug-in can be built quite easily with longer passwords using another hash.

> **Tip**
>
> The increased length limit means the default SYSDBAs password is the full 'masterkey' string (9 chars), no longer 'masterke' (8 chars) as in older versions!

Support for the **LegacyHash** and **Authentication** parameters in firebird.conf has been dropped. **Authentication** is overtaken by an **AuthServer** parameter in firebird.conf or elsewhere.

The Authentication Plug-in

The Authentication plug-in comprises three parts:

- Client—prepares data at the client to be sent to server on client
- Server—validates password for correctness
- User Manager—adds, modifies and deletes users on the server. It is not needed if some external authentication method, such as Windows trusted authentication, is used.

All three parts are actually separate plug-ins which should be configured separately in `firebird.conf`. Let's look at an example of configuring a server to accept connections from old clients. The default setting are:

```
AuthServer = Srp, Win_Sspi
UserManager = Srp
```

To enable access from old clients, AuthServer needs to be changed:

```
AuthServer = Srp, Win_Sspi, Legacy_Auth
```

If we also want to manage the list of users in the old format we must add:

```
UserManager = Legacy_UserManager
```

Multiple User Managers

In the Alpha releases only one User Manager is possible. Later, it is planned to make it possible to enumerate more than one and add *gsec* support to work with all of them.

"Over the wire" Connection Encryption

Alex Peshkov

CORE-672 ...

All network traffic in Firebird 3 may be optionally encrypted. As with authentication, plug-ins are used for encrypting and decrypting network traffic.

The default plug-in is **arc4** (Alleged RC4). It is eminently possible to write your own crypt plug-in to encrypt data travelling over the wire. Whatever you use for your plug-in, it is necessary to use the Firebird 3 version of the `fbclient` library.

The Secret Session Key

The challenge with use of a symmetric cypher is where to get a key for it. Firebird assumes that such a key, also called a *secret session key*, is produced by the authentication plug-in at the connection establishment phase. SRP meets this requirement just fine by producing a cryptographically strong session key.

> **Tip**
>
> If you want to use encryption with an authentication plug-in that does not provide the session key and agree to use some pre-defined key, say, one stored at the client side as a file and on the server in the security database for that specific client, then make that plug-in inform Firebird that it does have a session key.

New Pseudo-Table with List of Users

Alex Peshkov

CORE-2639.

The pseudo-table **SEC$USERS** is much like the MON$ family tables used for monitoring the server. It is created on demand when you run the statement

```
SELECT * FROM SEC$USERS
```

The output is the list of users in the security database that is configured for the current database and available for management to the current user. It is a handy adjunct to the commands `CREATE USER`, `ALTER USER` and `DROP USER`.

New Object Rights

Dmitry Yemanov

Some new object rights have been added to the sets of SQL privileges.

GRANT EXECUTE Privileges for UDFs

CORE-2554: EXECUTE permission is now supported for UDFs (both legacy and PSQL based ones).

Syntax Pattern

```
GRANT EXECUTE ON FUNCTION <name> TO <grantee list>
[<grant option> <granted by clause>]
--
REVOKE EXECUTE ON FUNCTION <name> FROM <grantee list>
[<granted by clause>]
```

Note

The initial EXECUTE permission is granted to the function owner (user who created or declared the function).

Privileges to Protect Other Metadata Objects

New SQL-2008 compliant USAGE permission is introduced to protect metadata objects other than tables, views, procedures and functions.

Syntax Pattern

```
GRANT USAGE ON <object type> <name> TO <grantee list>
[<grant option> <granted by clause>]
--
REVOKE USAGE ON <object type> <name> FROM <grantee list>
[<granted by clause>]
--
<object type> ::= {DOMAIN | EXCEPTION | GENERATOR | SEQUENCE | CHARACTER SET | COLLATION}
```

Notes

The initial USAGE permission is granted to the object owner (user who created the object).

In Firebird 3.0 Alpha 1, only USAGE permissions for exceptions (CORE-2884) and generators/sequences (gen_id, next value for: CORE-2553) are enforced. Permissions for other object types will be validated in subsequent releases.

Chapter 8

Data Definition Language (DDL)

Quick Links

- BOOLEAN Data Type
- IDENTITY-Style Column
- Manage Nullability in Domains and Columns
- Modify Generators (Sequences)
- Alter Default Character Set
- Grant/Revokes GRANTED BY Specified User
- Revoke ALL Privileges on ALL Users

DDL Enhancements

The following enhancements have been added to the SQL data definition language lexicon:

New Data Types

A fully-fledged Boolean type is introduced in this release, along with a surfaced emulation of the Microsoft-style “identity” column.

BOOLEAN Data Type

Adriano dos Santos Fernandes

The SQL-2008 compliant BOOLEAN data type (8 bits) comprises the distinct truth values TRUE and FALSE. Unless prohibited by a NOT NULL constraint, the BOOLEAN data type also supports the truth value UNKNOWN as the null value. The specification does not make a distinction between the NULL value of this data type and the truth value UNKNOWN that is the result of an SQL predicate, search condition, or boolean value expression: they may be used interchangeably to mean exactly the same thing.

As with many programming languages, the SQL BOOLEAN values can be tested with implicit truth values. For example, **field1 OR field2** and **NOT field1** are valid expressions.

The IS Operator

Predications use the operator IS [NOT] for matching. For example, **field1 IS FALSE**, or **field1 IS NOT TRUE**.

> **Note**
>
> Equivalence operators (“=”, “!=”, “<>” and so on) are valid in all comparisons.

Examples

```
CREATE TABLE TBOOL (ID INT, BVAL BOOLEAN);
COMMIT;

INSERT INTO TBOOL VALUES (1, TRUE);
INSERT INTO TBOOL VALUES (2, 2 = 4);
INSERT INTO TBOOL VALUES (3, NULL = 1);
COMMIT;

SELECT * FROM TBOOL
          ID    BVAL
============ =======
           1 <true>
           2 <false>
           3 <null>

-- Test for TRUE value
SELECT * FROM TBOOL WHERE BVAL
          ID    BVAL
============ =======
           1 <true>

-- Test for FALSE value
SELECT * FROM TBOOL WHERE BVAL IS FALSE
          ID    BVAL
============ =======
           2 <false>

-- Test for UNKNOWN value
SELECT * FROM TBOOL WHERE BVAL IS UNKNOWN
          ID    BVAL
============ =======
           3 <null<

-- Boolean values in SELECT list
SELECT ID, BVAL, BVAL AND ID < 2
  FROM TBOOL
          ID    BVAL
============ ======= =======
           1 <true>  <true>
           2 <false> <false>
           3 <null>  <false>

-- PSQL Declaration with start value
DECLARE VARIABLE VAR1 BOOLEAN = TRUE;

-- Valid syntax, but as with a comparison
-- with NULL, will never return any record
SELECT * FROM TBOOL WHERE BVAL = UNKNOWN
SELECT * FROM TBOOL WHERE BVAL <> UNKNOWN
```

Notes

- Represented in the API with the FB_BOOLEAN type and FB_TRUE and FB_FALSE constants.
- The value TRUE is greater than the value FALSE.
- Although BOOLEAN is not implicitly convertible to any other datatype, it can be explicitly converted to and from string with CAST.
- For compatibility reasons, the non-reserved keywords INSERTING, UPDATING and DELETING continue to behave as Boolean expressions when used in context in PSQL, while behaving as values if they are column or variable names in non-Boolean expressions.

  The following example uses the word INSERTING in all three ways:

```
SELECT
  INSERTING,          -- value
  NOT INSERTING       -- keyword
FROM TEST
WHERE
  INSERTING           -- keyword
  AND INSERTING IS TRUE -- value
```

Identity Column Type

Adriano dos Santos Fernandes

An identity column is a column associated with an internal sequence generator. Its value is set automatically set when the column is omitted in an INSERT statement.

Syntax Pattern

```
<column definition> ::=
    <name> <type> GENERATED BY DEFAULT AS IDENTITY <constraints>
```

Rules

- The data type of an identity column must be an exact number type with zero scale. Allowed types are thus SMALLINT, INTEGER, BIGINT, NUMERIC(x,0) and DECIMAL(x,0).
- An identity column cannot have DEFAULT or COMPUTED value.

Notes

- An identity column cannot be altered to become a regular column. The reverse is also true.
- Identity columns are implicitly NOT NULL (non-nullable).
- Uniqueness is not enforced automatically. A UNIQUE or PRIMARY KEY constraint is required to guarantee uniqueness.

Example

```
create table objects (
  id integer generated by default as identity primary key,
  name varchar(15)
);

insert into objects (name) values ('Table');
insert into objects (name) values ('Book');
insert into objects (id, name) values (10, 'Computer');

select * from objects;

          ID NAME
============ ===============
           1 Table
           2 Book
          10 Computer
```

Implementation Details

Two new columns have been inserted in RDB$RELATION_FIELDS to support identity columns: RDB $GENERATOR_NAME and RDB$IDENTITY_TYPE.

- RDB$GENERATOR_NAME stores the automatically created generator for the column. In RDB$GENERATORS, the value of RDB$SYSTEM_FLAG of that generator will be 6.
- Currently, RDB$IDENTITY_TYPE always stores the value 0 (GENERATED BY DEFAULT) for identity columns and NULL for non-identity columns. In the future this column will be able to store the value 1 (GENERATED ALWAYS) when that type of identity column is supported by Firebird.

Manage Nullability in Domains and Columns

A. dos Santos Fernandes

ALTER syntax is now available to change the nullability of a table column or a domain

Syntax Pattern

```
ALTER TABLE <table name> ALTER <field name> [NOT] NULL

ALTER DOMAIN <domain name> [NOT] NULL
```

> **Notes**
>
> The success of a change in a table column from NULL to NOT NULL is subject to a full data validation on the table, so ensure that the column has no nulls before attempting the change.
>
> A change in a domain subjects all the tables using the domain to validation.
>
> An explicit NOT NULL on a column that depends on a domain prevails over the domain. In this situation, the changing of the domain to make it nullable does not propagate to the column.

Modify Generators (Sequences)

More statement options have been added for modifying generators (sequences). Where previously in SQL the only option was **ALTER SEQUENCE <sequence name> RESTART WITH <value>**, now a full lexicon is provided and GENERATOR and SEQUENCE are synonyms for the full range of commands.

RESTART can now be used on its own to restart the sequence at its previous start or restart value. A new column `RDB$INITIAL_VALUE` is added to the system table RDB$GENERATORS to store that value.

Syntax Forms

```
{ CREATE | RECREATE } { SEQUENCE | GENERATOR } <sequence name> [ START WITH <value> ]

CREATE OR ALTER { SEQUENCE | GENERATOR } <sequence name> { RESTART | START WITH <value> }

ALTER { SEQUENCE | GENERATOR } <sequence name> RESTART [ WITH <value> ]
```

Alter the Default Character Set

A. dos Santos Fernandes

```
ALTER DATABASE
  ...
  ALTER DEFAULT CHARACTER SET <new_charset>
```

The alteration does not change any existing data. The new default character set is used only in subsequent DDL commands and will assume the default collation of the new character set.

BLOB in COMPUTED BY Expressions

Adriano dos Santos Fernandes

For Example

```
ALTER TABLE ATABLE
  ADD ABLOB
  COMPUTED BY (SUBSTRING(BLOB_FIELD FROM 1 FOR 20))
```

Improved Management of SQL Privileges

A. Peshkov

Some improvements will make it simpler to implement detailed management of SQL privileges.

GRANT/REVOKE Rights GRANTED BY Specified User

Previously, the grantor or revoker of SQL privileges was always the current user. This change makes it so that a different grantor or revoker can be specified in GRANT and REVOKE commands.

Syntax Pattern

```
grant <right> to <object> [ { granted by | as } [ user ] <username> ]
revoke <right> from <object> [ { granted by | as } [ user ] <username> ]
```

The **GRANTED BY** clause form is recommended by the SQL standard. The alternative form using **AS** is supported by Informix and possibly some other servers and is included for better compatibility.

Example (working as SYSDBA)

```
create role r1;
grant r1 to user1 with admin option;
grant r1 to public granted by user1;

-- (in isql)
show grant;
/* Grant permissions for this database */
GRANT R1 TO PUBLIC GRANTED BY USER1
GRANT R1 TO USER1 WITH ADMIN OPTION
```

REVOKE ALL ON ALL

When a user is removed from the security database or another authentication source, this new command is useful for revoking its access to all objects in the database.

Syntax Pattern

```
REVOKE ALL ON ALL FROM [USER] username
REVOKE ALL ON ALL FROM [ROLE] rolename
```

Example

```
# gsec -del guest
# isql employee
fbs bin # ./isql employee
Database:  employee
SQL> REVOKE ALL ON ALL FROM USER guest;
SQL>
```

Chapter 9

Data Manipulation Language (DML)

In this chapter are the additions and improvements that have been added to the SQL data manipulation language subset in Firebird 3.0.

Quick Links

Supplemental SQL 2008 Features for MERGE

Adriano dos Santos Fernandes

In summary, support for MERGE was supplemented with the introduction of these features:

- Addition of the DELETE extension (CORE-2005)
- Enabling the use of multiple WHEN MATCHED | NOT MATCHED clauses (CORE-3639) and ability to apply conditions to WHEN MATCHED | NOT MATCHED
- Addition of the RETURNING ... INTO ... clause (CORE-3020)

The purpose of MERGE is to read data from the source and INSERT or UPDATE in the target table according to a condition. It is available in DSQL and PSQL.

Syntax Pattern

```
 <merge statement> ::=
MERGE
 INTO <table or view> [ [AS] <correlation name> ]
 USING <table or view or derived table> [ [AS] <correlation name> ]
 ON <condition>
 <merge when>...
 <returning clause>

 <merge when> ::=
<merge when matched> |
<merge when not matched>

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

 <merge when not matched> ::=
WHEN NOT MATCHED [ AND <condition> ] THEN
 INSERT [ <left paren> <column list> <right paren> ]
  VALUES <left paren> <value list> <right paren>
```

Rules

At least one of <merge when matched> or <merge when not matched> should be specified.

Example

```
MERGE INTO customers c
  USING
    (SELECT * FROM customers_delta WHERE id > 10) cd
     ON (c.id = cd.id)
  WHEN MATCHED THEN
    UPDATE SET name = cd.name
  WHEN NOT MATCHED THEN
    INSERT (id, name)
    VALUES (cd.id, cd.name)
```

> **Notes**
>
> A right join is made between the INTO (left-side) and USING tables using the condition. UPDATE is called when a record exists in the left table (INTO), otherwise INSERT is called.
>
> As soon as it is determined whether or not the source matches a record in the target, the set formed from the corresponding (WHEN MATCHED / WHEN NOT MATCHED) clauses is evaluated in the order specified, to check their optional conditions. The first clause whose condition evaluates to true is the one which will be executed, and the subsequent ones will be ignored.
>
> If no record is returned in the join, INSERT is not called.

Window (Analytical) Functions

Adriano dos Santos Fernandes

According to the SQL specification, window functions (also know as analytical functions) are a kind of aggregation, but one that does not “filter” the result set of a query. The rows of aggregated data are mixed with the query result set.

The window functions are used with the OVER clause. They may appear only in the SELECT list or the ORDER BY clause of a query.

Besides the OVER clause, Firebird window functions may be *partitioned* and *ordered*.

Syntax Pattern

```
<window function> ::= <window function name>([<expr> [, <expr> ...]]) OVER (
  [PARTITION BY <expr> [, <expr> ...]]
  [ORDER BY <expr>
    [<direction>]
    [<nulls placement>]
    [, <expr> [<direction>] [<nulls placement>] ...]
)

<direction> ::= {ASC | DESC}

<nulls placement> ::= NULLS {FIRST | LAST}
```

Aggregate Functions Used as Window Functions

All aggregate functions may be used as window functions, adding the OVER clause.

Imagine a table EMPLOYEE with columns ID, NAME and SALARY, and the need to show each employee with his respective salary and the percentage of his salary over the payroll.

A normal query could achieve this, as follows:

```
select
    id,
    department,
    salary,
    salary / (select sum(salary) from employee) percentage
  from employee
  order by id;
```

Results

```
  id  department  salary  percentage
  --  ----------  ------  ----------
  1   R & D        10.00      0.2040
  2   SALES        12.00      0.2448
  3   SALES         8.00      0.1632
  4   R & D         9.00      0.1836
  5   R & D        10.00      0.2040
```

The query is repetitive and lengthy to run, especially if EMPLOYEE happened to be a complex view.

The same query could be specified in a much faster and more elegant way using a window function:

```
select
    id,
    department,
    salary,
    salary / sum(salary) OVER () percentage
  from employee
  order by id;
```

Here, **sum(salary) over ()** is computed with the sum of all SALARY from the query (the employee table).

Partitioning

Like aggregate functions, that may operate alone or in relation to a group, window functions may also operate on a group, which is called a “partition”.

Syntax Pattern

```
    <window function>(...) OVER (PARTITION BY <expr> [, <expr> ...])
```

Aggregation over a group could produce more than one row, so the result set generated by a partition is joined with the main query using the same expression list as the partition.

Continuing the employee example, instead of getting the percentage of each employee's salary over the all-employees total, we would like to get the percentage based on just the employees in the same department:

```
select
    id,
    department,
    salary,
    salary / sum(salary) OVER (PARTITION BY department) percentage
  from employee
  order by id;
```

Results

```
  id  department  salary  percentage
  --  ----------  ------  ----------
  1   R & D        10.00      0.3448
  2   SALES        12.00      0.6000
  3   SALES         8.00      0.4000
  4   R & D         9.00      0.3103
  5   R & D        10.00      0.3448
```

Ordering

The ORDER BY sub-clause can be used with or without partitions and, with the standard aggregate functions, make them return the partial aggregations as the records are being processed.

Example

```
select
    id,
    salary,
    sum(salary) over (order by salary) cumul_salary
  from employee
  order by salary;
```

The result set produced:

```
  id  salary  cumul_salary
  --  ------  ------------
  3     8.00          8.00
  4     9.00         17.00
  1    10.00         37.00
  5    10.00         37.00
  2    12.00         49.00
```

Then cumul_salary returns the partial/accumulated (or running) aggregation (of the SUM function). It may appear strange that 37.00 is repeated for the ids 1 and 5, but that is how it should work. The ORDER BY keys are grouped together and the aggregation is computed once (but summing the two 10.00). To avoid this, you can add the ID field to the end of the ORDER BY clause.

It's possible to use multiple windows with different orders, and ORDER BY parts like ASC/DESC and NULLS FIRST/LAST.

With a partition, ORDER BY works the same way, but at each partition boundary the aggregation is reset.

All aggregation functions, other than LIST(), are usable with ORDER BY.

Exclusive window functions

Beyond aggregate functions are the exclusive window functions, currently divided into *ranking* and *navigational* categories. Both sets can be used with or without partition and ordering, although the usage does not make much sense without ordering.

Ranking Functions

The rank functions compute the ordinal rank of a row within the window partition. In this category are the functions DENSE_RANK, RANK and ROW_NUMBER.

Syntax

```
    <ranking window function> ::=
        DENSE_RANK() |
        RANK() |
        ROW_NUMBER()
```

The ranking functions can be used to create different type of incremental counters. Consider **SUM(1) OVER (ORDER BY SALARY)** as an example of what they can do, each of them in a different way. Following is an example query, also comparing with the SUM behavior.

```
select
    id,
    salary,
    dense_rank() over (order by salary),
    rank() over (order by salary),
    row_number() over (order by salary),
    sum(1) over (order by salary)
  from employee
  order by salary;
```

The result set:

```
  id  salary  dense_rank  rank  row_number  sum
  --  ------  ----------  ----  ----------  ---
  3     8.00           1     1           1    1
  4     9.00           2     2           2    2
  1    10.00           3     3           3    4
  5    10.00           3     3           4    4
  2    12.00           4     5           5    5
```

The difference between DENSE_RANK and RANK is that there is a gap related to duplicate rows (relative to the window ordering) only in RANK. DENSE_RANK continues assigning sequential numbers after the duplicate salary. On the other hand, ROW_NUMBER always assigns sequential numbers, even when there are duplicate values.

Navigational Functions

The navigational functions get the simple (non-aggregated) value of an expression from another row of the query, within the same partition.

Syntax

```
    <navigational window function> ::=
        FIRST_VALUE(<expr>) |
        LAST_VALUE(<expr>) |
        NTH_VALUE(<expr>, <offset>) [FROM FIRST | FROM LAST] |
        LAG(<expr> [ [, <offset> [, <default> ] ] ) |
        LEAD(<expr> [ [, <offset> [, <default> ] ] )
```

> **Important to Note**
>
> FIRST_VALUE, LAST_VALUE and NTH_VALUE also operate on a window frame. Currently, Firebird always frames from the first to the current row of the partition, not to the last. This is likely to produce strange results for NTH_VALUE and especially LAST_VALUE.

Example

```
select
    id,
    salary,
```

```
    first_value(salary) over (order by salary),
    last_value(salary) over (order by salary),
    nth_value(salary, 2) over (order by salary),
    lag(salary) over (order by salary),
    lead(salary) over (order by salary)
  from employee
  order by salary;
```

The result set:

```
  id  salary  first_value  last_value  nth_value     lag    lead
  --  ------  -----------  ----------  ---------  ------  ------
  3     8.00         8.00        8.00     <null>  <null>    9.00
  4     9.00         8.00        9.00       9.00    8.00   10.00
  1    10.00         8.00       10.00       9.00    9.00   10.00
  5    10.00         8.00       10.00       9.00   10.00   12.00
  2    12.00         8.00       12.00       9.00   10.00  <null>
```

FIRST_VALUE and LAST_VALUE get, respectively, the first and last value of the ordered partition.

NTH_VALUE gets the n-th value, starting from the first (default) or the last record, from the ordered partition. An offset of 1 from first would be equivalent to FIRST_VALUE; an offset of 1 from last is equivalent to LAST_VALUE.

LAG looks for a preceding row, and LEAD for a following row. LAG and LEAD get their values within a distance respective to the current row and the offset (which defaults to 1) passed.

In a case where the offset points outside the partition, the default parameter (which defaults to NULL) is returned.

Advanced Plan Output

Dmitry Yemanov

PLAN output can now be output in a more structured and comprehensible form, e.g.

```
SELECT statement
   -> First [10]
     -> Sort [SUM, O_ORDERDATE]
       -> Aggregate
         -> Sort [L_ORDERKEY, O_ORDERDATE, O_SHIPPRIORITY]
           -> Inner Loop Join
             -> Filter
               -> Table #ORDERS# Access By ID
                 -> Bitmap
                   -> Index #ORDERS_ORDERDATE# Range Scan
             -> Filter
               -> Table #CUSTOMER# Access By ID
                 -> Bitmap
                   -> Index #CUSTOMER_PK# Unique Scan
             -> Filter
               -> Table #LINEITEM# Access By ID
                 -> Bitmap
                   -> Index #LINEITEM_PK# Unique Scan
```

Advanced PLAN Output in isql

New syntax **SET EXPLAIN [ON | OFF]** has been added to the *isql* utility to surface this option. For details, refer to SET EXPLAIN Extensions for Viewing Detailed Plans in the **Utilities** chapter.

Internal Functions

Additions and enhancements to the internal functions set are:

SUBSTRING with Regular Expressions

Adriano dos Santos Fernandes

A substring search can now use a regular expression.

Search Pattern

```
SUBSTRING(<string> [NOT] SIMILAR TO <pattern> ESCAPE <char>)
```

Discussion: TrackerCORE-2006

For more information about the use of SIMILAR TO expressions, refer to `README.similar_to.txt` in the /doc/ subdirectory of your Firebird installation.

New Inverse Hyperbolic Trigonometric Functions

Claudio Valderrama C.

The six inverse hyperbolic trigonometric functions have been implemented internally. They are:

ACOSH
: Returns the hyperbolic arc cosine of a number (expressed in radians). Format: **ACOSH(<number>)**

ASINH
: Returns the hyperbolic arc sine of a number (expressed in radians). Format: **ASINH(<number>)**

ATANH
: Returns the hyperbolic arc tangent of a number (expressed in radians). Format: **ATANH(<number>)**

COSH
: Returns the hyperbolic cosine of an angle (expressed in radians). Format: **COSH(<number>)**

SINH
: Returns the hyperbolic sine of an angle (expressed in radians). Format: **SINH(<number>)**

TANH
: Returns the hyperbolic tangent of an angle (expressed in radians). Format: **TANH(<number>)**

TRIM() BLOB Arguments Lose 32 KB limit

Adriano dos Santos Fernandes

In prior versions, TRIM(substring from string) allowed BLOBs for both arguments, but the first argument had to be smaller than 32 KB. Now both arguments can take BLOBs of >= 32 KB.

Some Useful DML Improvements

A collection of useful DML improvements is released with Firebird 3.

Alternatives for Embedding Quotes in String Literals

Adriano dos Santos Fernandes

It is now possible to use a character, or character pair, other than the doubled (escaped) apostrophe, to embed a quoted string inside another string. The keyword **q** or **Q** preceding a quoted string informs the parser that certain left-right pairs or pairs of identical characters within the string are the delimiters of the embedded string literal.

Syntax

```
<alternate string literal> ::=
    { q | Q } <quote> <alternate start char> [ { <char> }... ] <alternate end char> <quote>
```

> **Rules**
>
> When <alternate start char> is '(', '{', '[' or '<', <alternate end char> is paired up with its respective “partner”, viz. ')', '}', ']' and '>'. In other cases, <alternate end char> is the same as <alternate start char>.
>
> Inside the string, i.e., <char> items, single (not escaped) quotes could be used. Each quote will be part of the result string.

Examples

```
select q'{abc{def}ghi}' from rdb$database;        -- result: abc{def}ghi
select q'!That's a string!' from rdb$database;    -- result: That's a string
```

Prohibit Edgy Mixing of Implicit/Explicit Joins

Dmitry Yemanov

While mixing of implicit and explict join syntaxes is not recommended at all, the parser still allows them. Certain “mixes” actually cause the optimizer to produce unexpected results, including “No record for fetch” errors. The same edgy styles are prohibited by other SQL engines and now they are prohibited in Firebird.

To visit some discussion on the subject, see the Tracker ticket CORE-2812.

Left-side Parameters Supported

Adriano dos Santos Fernandes

The following style of subquery, with the parameter in the left side of a WHERE...IN (SELECT...) condition, would fail with the error “The data type of the parameter is unknown”.

```
SELECT <columns> FROM table_1 t1
  WHERE <conditions on table_1>
  AND (? IN (SELECT some_col FROM table_2 t2 WHERE t1.id = t2.ref_id))
```

> **Note**
>
> Better SQL coding practice would be to use EXISTS in these cases; however, developers were stumbling over this problem when using generated SQL from Hibernate, which used the undesirable style.

Enhancements to the RETURNING Clause

Adriano dos Santos Fernandes

Two usage enhancements were added to the RETURNING clause:

RETURNING Clause Value Can be Aliased

When using the RETURNING clause to return a value to the client, the value can now be passed under an alias.

Example Without and With Aliases

```
UPDATE T1 SET F2 = F2 * 10
  RETURNING OLD.F2, NEW.F2; -- without aliases

UPDATE T1 SET F2 = F2 * 10
  RETURNING OLD.F2 OLD_F2, NEW.F2 AS NEW_F2; -- with aliases
```

> **Note**
>
> The keyword AS is optional.

RETURNING Clause from Positioned Updates and Deletes

Support has been added for a RETURNING clause in positioned (WHERE CURRENT OF) UPDATE and DELETE statements.

Example

```
UPDATE T1 SET F2 = F2 * 10 WHERE CURRENT OF C
```

```
    RETURNING NEW.F2;
```

Cursor Stability

Vlad Khorsun

Until this release, Firebird suffered from an infamous bug whereby a data modification operation could loop infinitely and, depending on the operation, delete all the rows in a table, continue updating the same rows ad infinitum or insert rows until the host machine ran out of resources. All DML statements were affected (INSERT, UPDATE, DELETE, MERGE). It occurred because the engine used an implicit cursor for the operations.

To ensure stability, rows to be inserted, updated or deleted had to be marked in some way in order to avoid multiple visits. Another workaround was to force the query to have a SORT in its plan, in order to materialize the cursor.

From Firebird 3, engine uses the Undo log to check whether a row was already inserted or modified by the current cursor.

> **Important**
>
> This stabilisation does NOT work with SUSPEND loops in PSQL.

An Improvement for GTTs

Vlad Khorsun

Global temporary tables (GTTs) are now writable even in read-only transactions. The effect is as follows.-

Read-only transaction in read-write database
: Writable in both ON COMMIT PRESERVE ROWS and ON COMMIT DELETE ROWS

Read-only transaction in read-only database
: Writable in ON COMMIT DELETE ROWS only

Also

- Rollback for GTT ON COMMIT DELETE ROWS is faster
- Rows do not need to be backed out on rollback
- Garbage collection in GTT is not delayed by active transactions of other connections
-

> **Note**
>
> The same refinements were also backported to Firebird 2.5.1.

An Improvement for DML Strings

Adriano dos Santos Fernandes

Strings in DML queries are now transformed or validated to avoid passing malformed strings to the engine internals, for example, to the MON$STATEMENTS.MON$SQL_TEXT column.

The solution adopted depends on the character set of the attachment.-

- NONE—non-ASCII characters are transformed to question marks
- Others—the string is checked for malformed characters

Optimizations

Optimizations made for this release included.-

SIMILAR TO

Adriano dos Santos Fernandes

The performance of SIMILAR TO was improved.

OR'ed Parameter in WHERE Clause

Dmitry Yemanov

Performance for **(table.field = :param or :param = -1)** in the WHERE clause was enhanced.

Dialect 1 Interface

Adriano dos Santos Fernandes

Selection of SQL_INT64, SQL_DATE and SQL_TIME in dialect 1 was enabled.

See Tracker CORE-3972

Chapter 10

Procedural SQL (PSQL)

Advancements in procedural SQL (PSQL) include:

Quick Links

PSQL Stored Functions

Dmitry Yemanov

It is now possible to write a scalar function in PSQL and call it just like an internal function.

Syntax for the DDL

```
{CREATE [OR ALTER] | ALTER | RECREATE} FUNCTION <name>
[(param1 [, ...])]
RETURNS lt;type>
AS
BEGIN
  ...
END
```

> **Tip**
>
> The CREATE statement is the *declaration syntax* for PSQL functions, parallel to DECLARE for legacy UDFs.

Example

```
CREATE FUNCTION F(X INT) RETURNS INT
AS
```

```
BEGIN
  RETURN X+1;
END;
SELECT F(5) FROM RDB$DATABASE;
```

PSQL Sub-routines

Adriano dos Santos Fernandes

The header of a PSQL module (stored procedure, stored function, trigger, executable block) can now accept sub-procedure and sub-function blocks in the header declarations for use within the body of the module.

Syntax for Declaring a Sub-procedure

```
DECLARE PROCEDURE <name> [(param1 [, ...])]
[RETURNS (param1 [, ...])]
AS
...
```

Syntax for declaring a Sub-function

```
DECLARE FUNCTION <name> [(param1 [, ...])]
RETURNS <type>
AS
...
```

Examples

```
SET TERM ^;
--
-- Sub-function in EXECUTE BLOCK
--
EXECUTE BLOCK RETURNS (N INT)
AS
  DECLARE FUNCTION F(X INT) RETURNS INT
  AS
  BEGIN
    RETURN X+1;
  END
BEGIN
  N = F(5);
  SUSPEND;
END ^
--
-- Sub-function inside a stored function
--
CREATE OR ALTER FUNCTION FUNC1 (n1 INTEGER, n2 INTEGER)
  RETURNS INTEGER
  AS
  DECLARE FUNCTION SUBFUNC (n1 INTEGER, n2 INTEGER)
```

```
    RETURNS INTEGER
    AS
    BEGIN
      RETURN n1 + n2;
    END
    BEGIN
      RETURN SUBFUNC(n1, n2);
    END ^
--
select func1(5, 6) from rdb$database ^
```

Packages

A. dos Santos Fernandes

> **Note**
>
> This feature was sponsored with donations gathered at the Fifth Brazilian Firebird Developers' Day

A package is a group of procedures and functions managed as one entity. The notion of “packaging” the code components of a database operation addresses several objectives:

Modularisation
: The idea is to separate blocks of interdependent code into logical modules, as programming languages do.

  In programming it is well recognised that grouping code in various ways, in namespaces, units or classes, for example, is a good thing. With standard procedures and functions in the database this is not possible. Although they can be grouped in different script files, two problems remain:

  1. The grouping is not represented in the database metadata.

  2. Scripted routines all participate in a flat namespace and are callable by everyone (we are not referring to security permissions here).

To facilitate dependency tracking
: We want a mechanism to facilitate dependency tracking between a collection of related internal routines, as well as between this collection and other routines, both packaged and unpackaged.

  Firebird packages come in two parts: a *header* (keyword PACKAGE) and a *body* (keyword PACKAGE BODY). This division is very similar to a Delphi unit, the header corresponding to the interface part and the body corresponding to the implementation part.

  The header is created first (CREATE PACKAGE) and the body (CREATE PACKAGE BODY) follows.

  Whenever a packaged routine determines that it uses a certain database object, a dependency on that object is registered in Firebird system tables. Thereafter, to drop, or maybe alter that object, you first need to remove what depends on it. As it is a package body that depends on it, that package body can just be dropped, even if some other database object depends on this package. When the body is dropped, the header remains, allowing you to recreate its body once the changes related to the removed object are done.

To facilitate permission management
: It is good practice in general to create routines to require privileged use and to use roles or users to enable the privileged use. As Firebird runs routines with the caller privileges, it is necessary to grant also resource

usage to each routine when these resources would not be directly accessible to the caller. Usage of each routine to needs to be granted to users and/or roles.

Packaged routines do not have individual privileges. The privileges act on the package. Privileges granted to packages are valid for all package body routines, including private ones, but are stored for the package header.

For example:

```
GRANT SELECT ON TABLE secret TO PACKAGE pk_secret;
GRANT EXECUTE ON PACKAGE pk_secret TO ROLE role_secret;
```

To enable "private scope"

This objective was to introduce private scope to routines, viz., to make them available only for internal usage within the defining package.

All programming languages have the notion of routine scope, which is not possible without some form of grouping. Firebird packages also work like Delphi units in this regard. If a routine is not declared in the package header (interface) and is implemented in the body (implementation), it becomes a private routine. A private routine can only be called from inside its package.

Packaging Syntax

```
<package_header> ::=
    { CREATE [OR ALTER] | ALTER | RECREATE } PACKAGE <name>
    AS
    BEGIN
        [ <package_item> ... ]
    END

<package_item> ::=
    <function_decl> ; |
    <procedure_decl> ;

<function_decl> ::=
    FUNCTION <name> [( <parameters> )] RETURNS <type>

<procedure_decl> ::=
    PROCEDURE <name> [( <parameters> ) [RETURNS ( <parameters> )]]

<package_body> ::=
    { CREATE | RECREATE } PACKAGE BODY <name>
    AS
    BEGIN
        [ <package_item> ... ]
        [ <package_body_item> ... ]
    END

<package_body_item> ::=
    <function_impl> |
    <procedure_impl>

<function_impl> ::=
    FUNCTION <name> [( <parameters> )] RETURNS <type>
    AS
```

```
        BEGIN
           ...
        END
        |
        FUNCTION <name>  [( <parameters> )] RETURNS <type>
            EXTERNAL NAME '<name>' ENGINE <engine>

    <procedure_impl> ::=
        PROCEDURE <name> [( <parameters> ) [RETURNS ( <parameters> )]]
        AS
        BEGIN
           ...
        END
        |
        PROCEDURE <name> [( <parameters> ) [RETURNS ( <parameters> )]]
            EXTERNAL NAME '<name>' ENGINE <engine>

    <drop_package_header> ::=
        DROP PACKAGE <name>

    <drop_package_body> ::=
        DROP PACKAGE BODY <name>
```

Syntax rules

- A package body should implement all routines declared in the header and in the body start, with the same signature. {ADRIANO, this terminology needs explanation.-Ed.]
- Default value for procedure parameters could not be redefined (be informed in <package_item> and <package_body_item>). That means, they can be in <package_body_item> only for private procedures not declared. [ADRIANO, sorry, I cannot make any sense of this.-Ed.]

> **Notes**
>
> DROP PACKAGE drops the package body before dropping its header.
>
> UDF declarations (DECLARE EXTERNAL FUNCTION) are currently not supported inside packages.

Simple Packaging Example

```
SET TERM ^;
-- package header, declarations only
CREATE OR ALTER PACKAGE TEST
AS
BEGIN
  PROCEDURE P1(I INT) RETURNS (O INT); -- public procedure
END

-- package body, implementation
RECREATE PACKAGE BODY TEST
AS
BEGIN
  FUNCTION F1(I INT) RETURNS INT; -- private function
  PROCEDURE P1(I INT) RETURNS (O INT)
  AS
```

```
  BEGIN
  END
  FUNCTION F1(I INT) RETURNS INT
  AS
  BEGIN
    RETURN 0;
  END
END ^
```

> **Note**
>
> More examples can be found in the Firebird installation, in ../examples/package/.

DDL triggers

A. dos Santos Fernandes

> **Note**
>
> This feature was sponsored with donations gathered at the Fifth Brazilian Firebird Developers' Day

The purpose of a “DDL trigger” is to enable restrictions to be placed on users who attempt to create, alter or drop a DDL object.

Syntax Pattern

```
<database-trigger> ::=
 {CREATE | RECREATE | CREATE OR ALTER}
  TRIGGER <name>
  [ACTIVE | INACTIVE]
  {BEFORE | AFTER} <ddl event>
  [POSITION <n>]
 AS
 BEGIN
  ...
 END

<ddl event> ::=
 ANY DDL STATEMENT
  | <ddl event item> [{OR <ddl event item>}...]

<ddl event item> ::=
       CREATE TABLE
  | ALTER TABLE
  | DROP TABLE
  | CREATE PROCEDURE
  | ALTER PROCEDURE
  | DROP PROCEDURE
  | CREATE FUNCTION
  | ALTER FUNCTION
```

| DROP FUNCTION
| CREATE TRIGGER
| ALTER TRIGGER
| DROP TRIGGER
| CREATE EXCEPTION
| ALTER EXCEPTION
| DROP EXCEPTION
| CREATE VIEW
| ALTER VIEW
| DROP VIEW
| CREATE DOMAIN
| ALTER DOMAIN
| DROP DOMAIN
| CREATE ROLE
| ALTER ROLE
| DROP ROLE
| CREATE SEQUENCE
| ALTER SEQUENCE
| DROP SEQUENCE
| CREATE USER
| ALTER USER
| DROP USER
| CREATE INDEX
| ALTER INDEX
| DROP INDEX
| CREATE COLLATION
| DROP COLLATION
| ALTER CHARACTER SET
| CREATE PACKAGE
| ALTER PACKAGE
| DROP PACKAGE
| CREATE PACKAGE BODY
| DROP PACKAGE BODY

Important Rule

The event type [BEFORE | AFTER] of a DDL trigger cannot be changed.

Semantics

1. BEFORE triggers are fired before changes to the system tables. AFTER triggers are fired after system table changes.

2. When a DDL statement fires a trigger that raises an exception (BEFORE or AFTER, intentionally or unintentionally) the statement will not be committed. That is, exceptions can be used to ensure that a DDL operation will fail if the conditions are not precisely as intended.

3. DDL trigger actions are executed only when *committing* the transaction in which the affected DDL command runs. Never overlook the fact that what is possible to do in an AFTER trigger is exactly what is possible to do after a DDL command without autocommit. You cannot, for example, create a table in the trigger and use it there.

4. With “CREATE OR ALTER” statements, a trigger is fired one time at the CREATE event or the ALTER event, according to the previous existence of the object. With RECREATE statements, a trigger is fired for the DROP event if the object exists, and for the CREATE event.

5. ALTER and DROP events are generally not fired when the object name does not exist. For the exception, see point 6.

6. The exception to rule 5 is that BEFORE ALTER/DROP USER triggers fire even when the user name does not exist. This is because, underneath, these commands perform DML on the security database and the verification is not done before the command on it is run. This is likely to be different with embedded users, so do not write code that depends on this.

7. If some exception is raised after the DDL command starts its execution and before AFTER triggers are fired, AFTER triggers will not be fired.

8. Packaged procedures and triggers do not fire individual **{CREATE | ALTER | DROP} {PROCEDURE | FUNCTION}** triggers.

Support in Utilities

A DDL trigger is a type of database trigger, so the parameters **-nodbtriggers** (GBAK and ISQL) and **-T** (NBACKUP) apply to them. Remember that only the database owner and SYSDBA can use these switches.

Permissions

Only the database owner and SYSDBA can create, alter or drop DDL triggers.

DDL_TRIGGER Context Namespace

The introduction of DDL triggers brings with it the new **DDL_TRIGGER** namespace for use with RDB $GET_CONTEXT. Its usage is valid only when a DDL trigger is running. Its use is valid in stored procedures and functions called by DDL triggers.

The DDL_TRIGGER context works like a stack. Before a DDL trigger is fired, the values relative to the executed command are pushed onto this stack. After the trigger finishes, the values are popped. So in the case of cascade DDL statements, when an user DDL command fires a DDL trigger and this trigger executes another DDL command with EXECUTE STATEMENT, the values of the DDL_TRIGGER namespace are the ones relative to the command that fired the last DDL trigger on the call stack.

Elements of DDL_TRIGGER Context

- EVENT_TYPE: event type (CREATE, ALTER, DROP)
- OBJECT_TYPE: object type (TABLE, VIEW, etc)
- DDL_EVENT: event name (<ddl event item>), where <ddl_event_item> is EVENT_TYPE || ' ' || OBJECT_TYPE

- OBJECT_NAME: metadata object name
- SQL_TEXT: sql statement text

Examples Using DDL Triggers

Here is how you might use a DDL trigger to enforce a consistent naming scheme, in this case, stored procedure names should begin with the prefix “SP_”:

```
create exception e_invalid_sp_name 'Invalid SP name (should start with SP_)';

set term !;

create trigger trig_ddl_sp before CREATE PROCEDURE
as
begin
    if (rdb$get_context('DDL_TRIGGER', 'OBJECT_NAME') not starting 'SP_') then
        exception e_invalid_sp_name;
end!

-- Test

create procedure sp_test
as
begin
end!

create procedure test
as
begin
end!

-- The last command raises this exception and procedure TEST is not created
-- Statement failed, SQLSTATE = 42000
-- exception 1
-- -E_INVALID_SP_NAME
-- -Invalid SP name (should start with SP_)
-- -At trigger 'TRIG_DDL_SP' line: 4, col: 5

set term ;!
```

Implement custom DDL security, in this case restricting the running of DDL commands to certain users:

```
create exception e_access_denied 'Access denied';

set term !;

create trigger trig_ddl before any ddl statement
as
begin
    if (current_user <> 'SUPER_USER') then
        exception e_access_denied;
end!

-- Test

create procedure sp_test
as
```

```
begin
end!

-- The last command raises this exception and procedure SP_TEST is not created
-- Statement failed, SQLSTATE = 42000
-- exception 1
-- -E_ACCESS_DENIED
-- -Access denied
-- -At trigger 'TRIG_DDL' line: 4, col: 5

set term ;!
```

Use a trigger to log DDL actions and attempts:

```
create sequence ddl_seq;

create table ddl_log (
    id bigint not null primary key,
    moment timestamp not null,
    user_name varchar(31) not null,
    event_type varchar(25) not null,
    object_type varchar(25) not null,
    ddl_event varchar(25) not null,
    object_name varchar(31) not null,
    sql_text blob sub_type text not null,
    ok char(1) not null
);

set term !;

create trigger trig_ddl_log_before before any ddl statement
as
    declare id type of column ddl_log.id;
begin
    -- We do the changes in an AUTONOMOUS TRANSACTION, so if an exception happens
    -- and the command didn't run, the log will survive.
    in autonomous transaction do
    begin
        insert into ddl_log (id, moment, user_name, event_type, object_type,
                             ddl_event, object_name, sql_text, ok)
            values (next value for ddl_seq, current_timestamp, current_user,
                    rdb$get_context('DDL_TRIGGER', 'EVENT_TYPE'),
                    rdb$get_context('DDL_TRIGGER', 'OBJECT_TYPE'),
                    rdb$get_context('DDL_TRIGGER', 'DDL_EVENT'),
                    rdb$get_context('DDL_TRIGGER', 'OBJECT_NAME'),
                    rdb$get_context('DDL_TRIGGER', 'SQL_TEXT'),
                    'N')
            returning id into id;
        rdb$set_context('USER_SESSION', 'trig_ddl_log_id', id);
    end
end!

-- Note: the above trigger will fire for this DDL command. It's good idea to
-- use -nodbtriggers when working with them!
create trigger trig_ddl_log_after after any ddl statement
as
begin
    -- Here we need an AUTONOMOUS TRANSACTION because the original transaction
    -- will not see the record inserted on the BEFORE trigger autonomous
    -- transaction if user transaction is not READ COMMITTED.
```

```
    in autonomous transaction do
       update ddl_log set ok = 'Y'
       where id = rdb$get_context('USER_SESSION', 'trig_ddl_log_id');
end!

commit!

set term ;!

-- Delete the record about trig_ddl_log_after creation.
delete from ddl_log;
commit;

-- Test

-- This will be logged one time
-- (as T1 did not exist, RECREATE acts as CREATE) with OK = Y.
recreate table t1 (
    n1 integer,
    n2 integer
);

-- This will fail as T1 already exists, so OK will be N.
create table t1 (
    n1 integer,
    n2 integer
);

-- T2 does not exist. There will be no log.
drop table t2;

-- This will be logged twice
-- (as T1 exists, RECREATE acts as DROP and CREATE) with OK = Y.
recreate table t1 (
    n integer
);

commit;



select id, ddl_event, object_name, sql_text, ok
          from ddl_log order by id;

                    ID DDL_EVENT                OBJECT_NAME                     SQL_TEXT OK
====================== ========================= ======================= ================ ======
                     2 CREATE TABLE              T1                                80:3 Y
==============================================================================
SQL_TEXT:
recreate table t1 (
    n1 integer,
    n2 integer
)
==============================================================================
                     3 CREATE TABLE              T1                                80:2 N
==============================================================================
SQL_TEXT:
create table t1 (
    n1 integer,
    n2 integer
)
==============================================================================
```

```
                    4 DROP TABLE                T1                                        80:6 Y
================================================================================
SQL_TEXT:
recreate table t1 (
    n integer
)
================================================================================
                    5 CREATE TABLE              T1                                        80:9 Y
================================================================================
SQL_TEXT:
recreate table t1 (
    n integer
)
================================================================================
```

Exceptions with parameters

Adriano dos Santos Fernandes

An exception can now be defined with a message containing slots for parameters which are filled and passed when raising the exception, using the syntax pattern

EXCEPTION <name> USING (<value list>)

Examples

create exception e_invalid_val 'Invalid value @1 for the field @2';

```
...
if (val < 1000) then
   thing = val;
else
   exception e_invalid_val using (val, 'thing');
end
```

CREATE EXCEPTION EX_BAD_SP_NAME
'Name of procedures must start with "@1" : "@2"';

```
CREATE TRIGGER TRG_SP_CREATE BEFORE CREATE PROCEDURE
AS
DECLARE SP_NAME VARCHAR(255);
BEGIN
  SP_NAME = RDB$GET_CONTEXT('DDL_TRIGGER', 'OBJECT_NAME');

  IF (SP_NAME NOT STARTING 'SP_')
  THEN EXCEPTION EX_BAD_SP_NAME USING ('SP_', SP_NAME);
END;
```

Notes

The status vector is generated using this code combination: isc_except, <exception number>, isc_formatted_exception, <formatted exception message>, <exception parameters>

Since a new error code (isc_formatted_exception) is used, the client must be v.3.0, or at least use the `firebird.msg` file from v.3.0, in order to translate the status vector to a string.

Considering, in left-to-right order, each parameter passed in the exception-raising statement as “the Nth”, with N starting at 1:

- If an Nth parameter is not passed, the text is not substituted.
- If NULL is passed, it is replaced by the string '*** null ***'.
- If more parameters are passed than are defined in the exception message, the surplus ones are ignored.
- The total length of the message, including the values of the parameters, is still limited to 1053 bytes.

CONTINUE in Looping Logic

Adriano dos Santos Fernandes

CONTINUE is a complementary command to BREAK/LEAVE, allowing flow of control to break (leave) and start of the next iteration of a FOR/WHILE loop.

Syntax

```
  CONTINUE [<label>];
```

Example

```
FOR SELECT A, D FROM ATABLE INTO :achar, :ddate
  DO BEGIN
    IF (ddate < current_data - 30) THEN
      CONTINUE;
    ELSE
      /* do stuff */
    ...
  END
```

PSQL Cursor Stabilization

Vlad Khorsun

PSQL cursors without SUSPEND inside are now stable:

```
FOR SELECT ID FROM T WHERE VAL IS NULL INTO :ID
  DO BEGIN
    UPDATE T SET VAL = 1
     WHERE ID = :ID;
```

```
  END
```

Previously, this block would loop interminably. Now, the loop will not select the value if it was set within the loop.

> **Note**
>
> This could change the behaviour of legacy code.

If there is a SUSPEND inside the block, the old instability remains#this query still produces the infinite loop:

```
FOR SELECT ID FROM T INTO :ID
  DO BEGIN
    INSERT INTO T (ID) VALUES (:ID);
    SUSPEND;
  END
```

Some Size Limits Removed Using New API

Dmitry Yemanov

If and only if the new API is in use:

- The size of the body of a stored procedure or a trigger can exceed the traditional limit of 32 KB. The theoretical limit provided by the new API is 4GB. At the moment, as a security measure, the hardcoded limit is 10MB. It may change before the final release.
- The total size of all input or output parameters for a stored procedure or a user-defined DSQL query is no longer limited to the traditional size of (64KB - overhead).

SQLSTATE in Exception Handler

Dmitry Yemanov

Before the final release of Firebird 3 an SQLSTATE code will become a valid condition for trapping an exception with a WHEN statement. It is not implemented in the Alpha 1 release.

Chapter 11

Command-line Utilities

No new utilities are released with Firebird 3.0. Existing utilities have undergone a few improvements.

Monitoring

Dmitry Yemanov

New information is now available in MON$ATTACHMENTS, viz.

- Operating system user name. See Tracker CORE-3779.
- Protocol and client library version. See Tracker CORE-2780.
- Client host name. See Tracker CORE-2187.

isql

SET EXPLAIN Extensions for Viewing Detailed Plans

Dmitry Yemanov

A new SET option is added: SET EXPLAIN [ON | OFF]. It extends the SET PLAN option to report the explained plan instead of the standard one.

If SET PLAN is omitted, then SET EXPLAIN turns the plan output on. SET PLANONLY works as in previous versions.

Usage options

```
SET PLAN = simple plan + query execution
SET PLANONLY = simple plan, no query execution
SET PLAN + SET EXPLAIN = explained plan + query execution
SET PLAN + SET EXPLAIN + SET PLANONLY = explained plan, no query execution
SET EXPLAIN = explained plan + query execution
SET EXPLAIN + SET PLANONLY = explained plan, no query execution
```

Metadata Extract

Claudio Valderrama C.

The metadata extract tool (*-[e]x[tract]* switch) was improved to create a script that takes the dependency order of objects properly into account.

Path to INPUT Files

Adriano dos Santos Fernandes

The INPUT command will now use a relative path based on the directory of the last-opened, unclosed file in the chain to locate the next file.

fb_lock_print

Input Arguments

Dmitry Yemanov

fb_lock_print now accepts 32-bit integers as the input arguments for seconds and intervals. Previously they were limited to SMALLINT.

Useability Improvements

Vlad Khorsun

A few other small improvements:

1. More detailed usage help is available from the command line (-help).
2. Events history and list of owners are no longer output by default: they may be requested explicitly if required. Header-only is the new default.
3. New -o[wners] switch to print only owners (locks) with pending requests

Other Tweaks

Some implementation annoyances were cleared up in several utilities.

All Command-line Utilities

Resolution of Database Path

Alex Peshkov

All utilities resolve database paths in `databases.conf` when they need to access a database file directly. But not all of them would follow the same rules when expanding a database name. Now, they do.

Help and Version Information

Claudio Valderrama C.

All command-line utilities except *gpre* and *qli* now present help and version information in a unified and coherent way.

No info yet at CORE-2540.

War on Hard-coded Messages

Claudio Valderrama C.

Hard-coded messages were replaced with the regular parameterised-style ones in *tracemanager* and *nbackup*.

War on Arbitrary Switch Syntax

Claudio Valderrama C.

Switch options in *qli* and *nbackup* were made to check the correctness (or not) of the abbreviated switch options presented.

Chapter 12

Bugs Fixed

Firebird 3.0 Release

The following improvements and bug fixes were reported as fixed prior to the v.3.0.0 release:

Core Engine

(CORE-4135) Sweep was blocking the establishment of concurrent attachments in Superserver.

fixed by V. Khorsun

~ ~ ~

(CORE-4134) A race condition could occur when auto-sweep was started.

fixed by V. Khorsun

~ ~ ~

(CORE-4074) COMPUTED BY columns and POSITION function could produce garbled results.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-4027) Creating a table with computed fields containing SELECT FIRST could produce a corrupted result.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-3973) The SQLDA for an aliased column in a grouped query was missing the original table name, column name and owner.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-3947) Wrong results were produced when a column in the WHERE clause used the collation option (NUMERIC-SORT=1).

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-3941) A unique expression index would exhibit a memory alignment problem.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-3929) The invalid error “attempted update of read-only column” would appear when selecting MINVALUE from a list of more than 255 elements.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-3894) When an attempt was made to reduce the size of a CHAR or VARCHAR column, the numbers delivered in the error message were incorrect.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-3874) A computed column would appear in non-existent rows output from a left join.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-3820) Some character sets were duplicated in the system table RDB$TYPES.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-3754) SIMILAR TO was not working correctly.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-3735) An unprivileged user could delete from the system tables RDB$DATABASE, RDB$COLLATIONS and RDB$CHARACTER_SETS.

fixed by D. Yemanov

~ ~ ~

(CORE-3694) “Internal consistency check” would occur in a query with grouping by subquery+stored procedure+aggregate.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-3672) It was not possible to use the SUBSTRING function to create a computed index for large columns.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-3638) Some collation tweaking: FR_CA_CI_AI collation was introduced; FR_FR was changed to be identical to FR_CA and FR_FR_CI_AI was changed to be identical to the new FR_CA_CI_AI.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-3476) The LIST function was concatenating binary blobs as though they were text.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-3401) Collation errors could occur with the use of **[type of] <domain>** and **type of <column>**.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-3373) It was possible to store a string of length 31 characters into a VARCHAR(25) column.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-3338) *Regression:* Code changes had disabled support for expression indexes with COALESCE, CASE and DECODE.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-3317) Success of row deletion depended on the order of insertion of the rows.

fixed by V. Khorsun

~ ~ ~

(CORE-3310) A complex expression involving RDB$GET_CONTEXT and BETWEEN worked in DSQL but failed with a conversion error when selected in a view definition.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-3260) Interlock.h was not portable.

fixed by A. Peshkov

~ ~ ~

(CORE-3250) The Firebird server could not be started under any user name other than “root”, “firebird”, “interbas” or “interbase”.

fixed by A. Peshkov

~ ~ ~

(CORE-3239) The collation UTF8 UNICODE_CI could not be used in a compound index.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-3204) A constraint violation error involving CAST was not being raised inside views.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-3052) Comparisons involving multiple index segments could produce wrong result sets.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-2988) The concurrent transaction number was not being reported when a lock timeout occurred.

fixed by N. Samofatov

~ ~ ~

(CORE-2957) COUNT(*) from a big table could return a negative result.

fixed by D. Yemanov

~ ~ ~

(CORE-2952) Character class names in SIMILAR TO expressions could be case-sensitive or case-insensitive, depending on the collation of the left part, whereas they should be unequivocally case-insensitive.

fixed by D. Sibiryakov

~ ~ ~

(CORE-2932) An ALTER TABLE..ALTER COLUMN..ALTER POSITION operation could result in wrong column positions.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-2922) The character set used in a constant was not being registered as a dependency.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-2913) COLLATE expressions were being applied incorrectly.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-2798) Plan output lacked the names of views when selecting from views that contained procedure calls.

fixed by D. Yemanov

~ ~ ~

(CORE-2796) DB_KEY was always zero for rows in external tables.

fixed by D. Yemanov

~ ~ ~

(CORE-2678) A full outer join could not use available indices, resulting in very slow execution sometimes.

fixed by D. Yemanov

~ ~ ~

(CORE-2508) Use of certain choices of character in double-quoted index names, for example a bracket character, could defeat the parsing logic when generating a human-readable plan.

fixed by D. Yemanov

~ ~ ~

(CORE-2155) A join of a stored procedure with a view or a table could fail with the error “No current record for fetch operation”.

fixed by D. Yemanov

~ ~ ~

(CORE-1712) A buffer overrun error was being caught erroneously in a DOUBLE PRECISION to VARCHAR conversion in a Dialect 1 database.

fixed by C. Valderrama C.

~ ~ ~

(CORE-1605) An aggregated query was causing “Bugcheck 232 (invalid operation)”.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-1550) An unnecessary index scan was executed when the same index is mapped to both WHERE and ORDER BY clauses.

fixed by D. Yemanov

~ ~ ~

API/Remote Interface

(CORE-3718) The cient library could hang after an unsuccessful attempt to connect to the remote auxiliary (events) port.

fixed by A. Peshkov

~ ~ ~

(CORE-3475) Parameters inside the CAST function were being wrongly described in the SQLDA as non-nullable.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-3269) The client would perform *detach* incorrectly when the server became unavailable.

fixed by A. Peshkov

~ ~ ~

(CORE-2484) An erroneous “Success” message would be returned in the error status vector when failing to connect to a trash database file.

fixed by C. Valderrama C.

~ ~ ~

(CORE-2431) String values in error messages were not converted to the connection character set.

fixed by A. dos Santos Fernandes

~ ~ ~

Procedural Language

(CORE-4018) Use of a system domain in declarations of arguments or return values in a stored procedure could prevent the procedure from being modifiable.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-3737) EXECUTE BLOCK parameter definitions were not being respected and could cause wrong behavior with respect to character sets.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-3545) Validation of domain CHECK constraints when used in PSQL declarations was inconsistent: it was using the type of the expression, instead of the type of the variable.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-3055) The names of variables or arguments could be wrong or absent in error messages when more than 256 variables were used.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-3047) Resolution of EXECUTE BLOCK parameter collations was using wrong logic.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-2204) Constraints on stored procedure output parameters were checked even when the procedure returned no rows.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-1620) Incorrect error message (an absurd column number) was returned if an empty SQL string was prepared for EXECUTE STATEMENT.

fixed by D. Yemanov

~ ~ ~

Data Definition Language

(CORE-3114) Attempting to drop a non-existent generator (sequence) would result in a serious exception.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-3056) Problems could occur if further DDL commands were issued in the same transaction following a CREATE COLLATION command.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-2696) The ALTER TABLE command allowed the addition of a column with a NOT NULL definition, allowing a non-savvy DBAdmin to wreck the table.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-1748) Unrestorable backup: a problem which would occur if ALTER TABLE...ADD COLUMN added a column with a NOT NULL constraint. The fix for CORE-2696 has now made it impossible to do this.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-1518) Adding a non-nullable column to a populated table would render the table inconsistent. The fix for CORE-2696 has now made it impossible to do this.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-1355) Client tools tended to be confused about how to interpret a NULL that is returned from a non-nullable column. The fix for CORE-2696 has now made it impossible to add a non-nullable column to a populated table.

It is not clear, though, whether this part of the fix makes it mandatory to specify a default value for a non-nullable column.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-634) Bad behaviour of DELETE when the WHERE clause was a subquery involving FIRST/SKIP: the operation would zap every row in the table.

fixed by V. Khorsun

~ ~ ~

(CORE-304) Any user could alter or drop generators and exceptions#a metadata security hole.

fixed by D. Yemanov

~ ~ ~

Data Manipulation Language & DSQL

(CORE-4144) When when preparing a query with UNION, the error “context already in use (BLR error)” was wrongly being thrown.

fixed by V. Khorsun

~ ~ ~

(CORE-4005) Recursive CTEs were returning a wrong error message.

fixed by V. Khorsun

~ ~ ~

(CORE-3416) Inserting a word containing the 8-bit character 'ä' into a CHARACTER SET ASCII column would succeed instead of throwing a transliteration error.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-3201) The internal function ATAN2 was returning an incorrect value with arguments (0, 0).

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-3174) An expression index involving TRIM could lead to an incorrect indexed lookup.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-2699) A common table expression context could be used with parameters.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-2606) A multi-byte CHAR value requested as VARCHAR was returned with padded spaces.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-2238) With UTF8 and large varchar fields, IS DISTINCT FROM would cause the error “Implementation limit exceeded”.

fixed by D. Yemanov

~ ~ ~

(CORE-1188) STARTING WITH ? (where the parameter value supplied is an empty string) would fail if the plan used a compound index.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-92) Infinite insertion cycle: INSERT INTO THIS_TABLE SELECT ... FROM THIS_TABLE would loop forever until resources were exhausted.

fixed by V. Khorsun

~ ~ ~

Command-line Utilities

(CORE-2547) Utilities did not always honour the minimum number of characters required to recognise an option.

fixed by C. Valderrama C.

~ ~ ~

Other old bugs that were fixed in utilities:

FbGuard

(CORE-2784) Guardian would keep creating more and more threads each time FBServer died.

fixed by C. Valderrama C.

~ ~ ~

(CORE-1595) Firebird Guardian's tray icon would disappear after a Windows Explorer crash.

fixed by C. Valderrama C.

~ ~ ~

isql

(CORE-4137) *isql* was generating metadata script output with syntax errors in the CHARACTER SET clause, e.g., `CHARACTER SETISO8859_1`.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-3431) *isql* was padding UTF-8 data incorrectly.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-2788) *isql* would extract the array dimensions after the character set name.

fixed by C. Valderrama C.

~ ~ ~

gbak

(CORE-3575) *gbak* did not support backup volumes of size greater than 4GB.

fixed by A. Peshkov

~ ~ ~

(CORE-2740) *gbak* would restore invalid views without any warning to the user.

fixed by C. Valderrama C.

~ ~ ~

(CORE-2545) Several validations were lacking in *gbak*.

fixed by C. Valderrama C.

~ ~ ~

nbackup

(CORE-2543) *nbackup* could hide the real cause of a failure.

fixed by C. Valderrama C.

~ ~ ~

International Language Support

(CORE-4136) The “Sharp-S” character was being treated incorrectly in the UNICODE_CI_AI collation.

fixed by A. dos Santos Fernandes

~ ~ ~

Chapter 13

Firebird 3.0 Project Teams

Table 13.1. Firebird Development Teams

Developer	Country	Major Tasks
Dmitry Yemanov	Russian Federation	Full-time database engineer/implementor, core team leader
Alex Peshkov	Russian Federation	Full-time security features coordinator; buildmaster; porting authority
Claudio Valderrama	Chile	Code scrutineer; bug-finder and fixer; ISQL enhancements; UDF fixer, designer and implementor
Vladyslav Khorsun	Ukraine	Full-time DB engineer, SQL feature designer/implementor
Adriano dos Santos Fernandes	Brazil	International character-set handling; text and text BLOB enhancements; new DSQL features; code scrutineering
Roman Simakov	Russian Federation	Engine contributions
Paul Beach	France	Release Manager; HP-UX builds; MacOS Builds; Solaris Builds
Pavel Cisar	Czech Republic	QA tools designer/coordinator
Philippe Makowski	France	QA tester
Paul Reeves	France	Win32 installers and builds
Mark Rotteveel	The Netherlands	Jaybird implementor and co-coordinator
Jiri Cincura	Czech Republic	Developer and coordinator of .NET providers
Alexander Potapchenko	Russian Federation	Developer and coordinator of ODBC/JDBC driver for Firebird
Stephen Boyd	Canada	GPRE contributions
Alexey Kovyazin	Russian Federation	Website coordinator
Paul Vinkenoog	The Netherlands	Coordinator, Firebird documentation project; documentation writer and tools developer/implementor
Norman Dunbar	U.K.	Documentation writer

Developer	Country	Major Tasks
Pavel Menshchikov	Russian Federation	Documentation translator
Tomneko Hayashi	Japan	Documentation translator
Umberto (Mimmo) Masotti	Italy	Documentation translator
Helen Borrie	Australia	Release notes editor; Chief of Thought Police

Appendix A: Licence Notice

The contents of this Documentation are subject to the Public Documentation License Version 1.0 (the “License”); you may only use this Documentation if you comply with the terms of this Licence. Copies of the Licence are available at http://www.firebirdsql.org/pdfmanual/pdl.pdf (PDF) and http://www.firebirdsql.org/manual/pdl.html (HTML).

The Original Documentation is entitled *Firebird 3.0 Release Notes*.

The Initial Writer of the Original Documentation is: Helen Borrie. Persons named in attributions are Contributors.