Firebird 4.0 Release Notes

Helen Borrie (Collator/Editor)

21 August 2017 - Document v.0400-07 - for Firebird 4.0 Alpha 1 Release

Firebird 4.0 Release Notes

21 August 2017 - Document v.0400-07 - for Firebird 4.0 Alpha 1 Release

Helen Borrie (Collator/Editor)

Table of Contents

List of Tables

Chapter 1

General Notes

Thank you for reviewing this Firebird 4.0 Alpha release. We cordially invite you to test it hard against your expectations and engage with us in identifying and fixing any bugs you might encounter.

ODS13 is introduced and it's a major ODS upgrade, so older databases cannot be opened with a Firebird 4 server. At this point in development, nothing more than a backup/restore is needed if you want to upgrade an existing database for your alpha testing. The engine library is named `engine13.dll` (Windows) and `engine13.so` (POSIX). The security database is named `security4.fdb`. Binaries layout and configuration are unchanged from Firebird 3.

The “known incompatibilities” chapter and other migration issues will be documented for Beta.

Bug Reporting

Bugs fixed since the most recent sub-release of Firebird 3 are listed and described in the chapter entitled Bugs Fixed.

- 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 4.0 installation.

--*The Firebird Project*

Chapter 2

New In Firebird 4.0

Summary of New Features

The following lists summarise the planned features and changes, with links to the topics covering items available to test in this Alpha release.

Complete in Alpha

Physical standby solution

Physical standby solution (incremental restore via nbackup). CORE-2216, CORE-2990 Vlad Khorsun / Roman Simakov

The changes are described in more detail in the Utilities chapter in the topic *nBackup*: GUID-based Backup and In-Place Merge.

Extend length of metadata identifiers

Metadata names longer than 31 characters (new maximum 63 characters). CORE-749 Adriano dos Santos Fernandes

The changes are described in more detail in the chapter Data Definition Language, in the topic Extended Length for Object Names.

Configurable time-outs

Timeouts for statements / transactions / connections CORE-658, CORE-985 Vlad Khorsun

The changes for statements and connections are described in more detail in the chapter Changes in the Firebird Engine in the topic Timeouts at Two levels.

Extended precision for numerics

Numerics with precision longer than 18 digits, improved intermediate calculations for shorter numerics CORE-4409 Alex Peshkov / Dmitry Yemanov

The high-precision numeric type is described in more detail in the Data Definition Language chapter in the topic Data type DECFLOAT.

Enhanced system privileges

Predefined system roles, administrative permissions. CORE-2557 Alex Peshkov

The changes are described in more detail in the Security chapter in the topic Enhanced System Privileges.

Extended window functions

Extended window functions CORE-1688 Adriano dos Santos Fernandes

The changes are described in more detail in the Data Manipulation Language chapter in the topics Frames for Window Functions, Named Windows and More Window Functions.

GRANT ROLE TO ROLE

Granting roles to other roles. CORE-1815 Roman Simakov / Alex Peshkov

The changes are described in more detail in the Security chapter in the topic Granting a Role to Another Role.

User groups
User groups / cumulative permissions. CORE-751 Roman Simakov / Alex Peshkov
The changes are described in more detail in the Security chapter in the topic Granting a Role to Another Role.

Pending for Beta

Replication
Built-in logical (row level) replication, both synchronous and asynchronous. (Dmitry Yemanov &Roman Simakov)

Tracker ticket CORE-2021

Batch operations in the API
Batch API operations, bulk load optimizations (Alex Peshkov)

Tracker ticket CORE-820

Enhanced optimizer statistics
Optimizer statistics: more data (including histograms), auto-update (Dmitry Yemanov & Vlad Khorsun)

Tracker tickets CORE-1082 & CORE-1686

Improve performance of gbak restore
Improve performance of *gbak restore*, including parallel operations. (Vlad Khorsun)

Tracker ticket CORE-2992

Support for time zones
Support for time zones (Adriano dos Santos Fernandes)

Tracker tickets CORE-694 & CORE-909

New data access paths for queries
New data access paths, subquery transformations (Dmitry Yemanov)

Tracker ticket CORE-4823

TRUNCATE TABLE
TRUNCATE TABLE command (Dmitry Yemanov)

Tracker ticket CORE-2479

Chapter 3

Changes in the Firebird Engine

The Firebird engine, V.4, presents no radical changes in architecture or operation. Improvements and enhancements continue, including an doubling of the maximum database page size and the long-awaited ability to impose timeouts on connections and statements that could be troublesome.

Firebird 4 creates databases with the on-disk structure numbered 13—"ODS 13". The remote interface protocol number is 16.

Extended Maximum Page Size

Dmitry Yemanov

Tracker ticket CORE-2192

The maximum page size for databases created under ODS 13 has been extended from 16 Kb to 32 Kb.

xinetd Support on Linux Replaced

Alex Peshkov

Tracker ticket CORE-5238

On Linux, Firebird 4 uses the same network listener process (Firebird) for all architectures. For Classic, the main (listener) process starts up via *init/systemd*, binds to the 3050 port and spawns a worker firebird process for every connection—similarly to what happens on Windows.

Timeouts at Two levels

Vladyslav Khorsun

Tracker ticket CORE-5488

Firebird 4 introduces configurable timeouts for running SQL statements and for idle connections (sessions).

Idle Session Timeouts

An idle session timeout allows a user connection to close automatically after a specified period of inactivity. The database admin could use it to enforce closure of old connections that have become inactive, to reduce unnecessary consumption of resources. It could also be used by application and tools developers as an alternative to writing their own modules for controlling connection lifetime.

By default, the idle timeout is not enabled. No minimum or maximum limit is imposed but a reasonably large period, such as a few hours, is recommended.

How the Idle Session Timeout Works

- When the user API call leaves the engine (returns to the calling connection) a special idle timer associated with the current connection is started
- When another user API call from that connection enters the engine, the idle timer is stopped
- If the idle time is attained, the engine immediately closes the connection in the same way as with asyncronous connection cancellation:
  - all active statements and cursors are closed
  - all active transactions are rolled back
  - The network connection remains open at this point, allowing the client application to get the exact error code on the next API call. The network connection will be closed on the server side, after an error is reported or in due course as a result of a network timeout from a client-side disconnection.

> **Note**
> Whenever a connection is cancelled, the next user API call returns the error **isc_att_shutdown** with a secondary error specifying the exact reason. Now, we have
>
> | | |
> |---|---|
> | isc_att_shut_idle: | Idle timeout expired |
>
> in addition to
>
> | | |
> |---|---|
> | isc_att_shut_killed: | Killed by database administrator |
> | isc_att_shut_db_down: | Database is shut down |
> | isc_att_shut_engine: | Engine is shut down |

Setting the Idle Session Timeout

> **Note**
> The idle timer will not start if the timeout period is set to zero.

An idle session timeout can be set:

- At database level the database administrator can set the configuration parameter **ConnectionIdleTimeout**, an integer value **in minutes**. The default value of zero means no timeout is set. It is configurable per-database, so it may be set globally in `firebird.conf` and overridden for individual databases in `databases.conf` as required.

  The scope of this method is all user connections, except system connections (garbage collector, cache writer, etc.).

- at connection level, the idle session timeout is supported by both the API and a new SQL statement syntax. The scope of this method is specific to the supplied connection (attachment). Its value in the API is **in seconds**. In the SQL syntax it can be hours, minutes or seconds. Scope for this method is the connection to which it is applied.

Determining the Timeout that is In Effect

The effective idle timeout value is determined whenever a user API call leaves the engine, checking first at connection level and then at database level. A connection-level timeout can override the value of a database-level setting, as long as the period of time for the connection-level setting is no longer than any non-zero timeout that is applicable at database level.

> **Important**
>
> Take note of the difference between the time units at each level. At database and connection levels, timeout is in seconds; in the SQL syntax it can be hours, minutes or seconds, defaulting to minutes where units are not stated.
>
> Absolute precision is not guaranteed in any case, especially when the system load is high, but timeouts are guaranteed not to expire earlier than the moment specified.

SQL Syntax for Setting an Idle Session Timeout

The statement for setting an idle timeout at connection level can run outside transaction control and takes effect immediately. The syntax pattern is as follows:

```
SET SESSION IDLE TIMEOUT <value> [HOUR | MINUTE | SECOND]
```

If the time unit is not set, it defaults to MINUTE.

Support at API Level

Get\set idle connection timeout, seconds

```
interface Attachment
 uint getIdleTimeout(Status status);
 void setIdleTimeout(Status status, uint timeOut);
```

The values of the idle connection timeout at both configuration and connection levels, along with the current actual timeout, can be obtained using the **isc_database_info()** API with some new info tags:

fb_info_ses_idle_timeout_db	Value set at config level
fb_info_ses_idle_timeout_att	Value set at given connection level
fb_info_ses_idle_timeout_run	Actual timeout value for the given connection, evaluated considering the values set at config and connection levels, see Determining the Timeout that is In Effect above.

Notes regarding remote client implementation

1. Attachment::setIdleTimeout() issues a “SET SESSION IDLE TIMEOUT” SQL statement
2. Attachment::getIdleTimeout() calls isc_database_info() with the fb_info_ses_idle_timeout_att tag
3. If the protocol of the remote Firebird server is less than 16, it does not support idle connection timeouts. If that is the case,
   - Attachment::setIdleTimeout() will return the error isc_wish_list
   - Attachment::getIdleTimeout() will return zero and set the isc_wish_list error
   - isc_database_info() will return the usual isc_info_error tag in the info buffer

Context Variable Relating to Idle Session Timeouts

The 'SYSTEM' context has a new variable: `SESSION_IDLE_TIMEOUT`. It contains the current value of idle connection timeout that was set at connection level, or zero, if no timeout was set.

Idle Session Timeouts in the Monitoring Tables

In MON$ATTACHMENTS:

MON$IDLE_TIMEOUT	Connection level idle timeout
MON$IDLE_TIMER	Idle timer expiration time

MON$IDLE_TIMEOUT contains timeout value set at connection level, in seconds. Zero, if timeout is not set.

MON$IDLE_TIMER contains NULL if an idle timeout was not set or if a timer is not running.

Statement Timeouts

The statement timeout feature enables the ability to set a timeout for an SQL statement, allowing execution of a statement to be stopped authomatically when it has been running longer than the given timeout period. It gives the database administrator an instrument for limiting excessive resource consumption from heavy queries.

Statement timeouts could be useful to application developers when creating and debugging complex queries without advance knowledge of execution time. Testers and others could find them handy for detecting long running queries and establishing finite run times for test suites.

How the Statement Timeout Works

When the statement starts execution or a cursor is opened, the engine starts a special timer. It is stopped when the statement completes execution or the last record has been fetched by the cursor.

Note
FETCH does not reset this timer.

When the timeout point is reached:

- if statement execution is active, it stops at closest possible moment
- if statement is not active currently (between fetches, for example), it is marked as cancelled and the next fetch will actually break execution and return an error

> **Statement types excluded from timeouts**
>
> Statement timeouts are not applicable to some types of statement and will simply be ignored:
>
> - All DDL statements
> - All internal queries issued by the engine itself

Setting a Statement Timeout

> **Note**
> The timer will not start if the timeout period is set to zero.

A statement timeout can be set:

- at database level, by the database administrator, by setting the configuration parameter **StatementTimeout** in firebird.conf or databases.conf, an integer representing the number of seconds after which statement execution will be cancelled automatically by the engine. Zero means no timeout is set. A non-zero setting will affect all statements in all connections.
- at connection level, using the API and\or the new SQL statement syntax for setting a statement timeout. A connection-level setting (via SQL or the API) affects all statements for the given connection; Units for the timeout period at this level can be specified to any granularity from hours to milliseconds.
- at statement level, using the API, in milliseconds

Determining the Statement Timeout that is In Effect

The statement timeout value that is in effect is determined whenever a statement starts executing or a cursor is opened. In searching out the timeout in effect, the engine goes up through the levels, from statement through to database and/or global levels until it finds a non-zero value. If the value in effect turns out to be zero then no statement timer is running and no timeout applies.

A statement-level or connection-level timeout can override the value of a database-level setting, as long as the period of time for the lower-level setting is no longer than any non-zero timeout that is applicable at database level.

> **Important**
>
> Take note of the difference between the time units at each level. At database level, timeout is in minutes; in the API, it is in seconds; in the SQL syntax it can be hours, minutes, seconds or milliseconds, defaulting to seconds where units are not stated.
>
> Absolute precision is not guaranteed in any case, especially when the system load is high, but timeouts are guaranteed not to expire earlier than the moment specified.

Whenever a statement times out and is cancelled, the next user API call returns the error **isc_cancelled** with a secondary error specifying the exact reason, viz.,

isc_cfg_stmt_timeout:	Config level timeout expired
isc_att_stmt_timeout:	Attachment level timeout expired
isc_req_stmt_timeout:	Statement level timeout expired

Notes about Statement Timeouts

1. A client application could wait longer than the time than set by the timeout value if the engine needs to undo a large number of actions as a result of the statement cancellation
2. When the engine runs an EXECUTE STATEMENT statement, it passes the remainder of the currently active timeout to the new statement. If the external (remote) engine does not support statement timeouts, the local engine silently ignores any corresponding error.
3. When engine acquires some lock from the lock manager, it tries to lower the value of the lock timeout using the remainder of the currently active statement timeout, if possible. Due to lock manager internals, any statement timeout remainder will be rounded up to whole seconds.

SQL Syntax for Setting a Statement Timeout

The statement for setting a statement execution timeout at connection level can run outside transaction control and takes effect immediately. The statement syntax pattern is:

```
SET STATEMENT TIMEOUT <value> [HOUR | MINUTE | SECOND | MILLISECOND]
```

If the time part unit is not set, it defaults to SECOND.

Support for Statement Timeouts at API Level

statement execution timeout at connection level, milliseconds:

```
interface Attachment
 uint getStatementTimeout(Status status);
 void setStatementTimeout(Status status, uint timeOut);
```

Get\set statement execution timeout at statement level, milliseconds:

```
interface Statement
 uint getTimeout(Status status);
 void setTimeout(Status status, uint timeOut);
```

Set statement execution timeout at statement level using ISC API, milliseconds:

```
ISC_STATUS ISC_EXPORT fb_dsql_set_timeout(ISC_STATUS*, isc_stmt_handle*, ISC_ULONG);
```

Getting the statement execution timeout at config and\or connection levels can be done using the `isc_database_info()` API with some new info tags:

fb_info_statement_timeout_db
fb_info_statement_timeout_att

Getting the statement execution timeout at statement level can be done using the `isc_dsql_info()` API with some new info tags:

isc_info_sql_stmt_timeout_user	Timeout value of given statement
isc_info_sql_stmt_timeout_run	Actual timeout value of given statement. Valid only for statements currently executing, i.e., when a timeout timer is actually running. Evaluated considering the values set at config, connection and statement levels, see Determining the Statement Timeout that is In Effect above.

Notes regarding remote client implementation

1. Attachment::setStatementTimeout() issues a “SET STATEMENT TIMEOUT” SQL statement
2. Attachment::getStatementTimeout() calls isc_database_info() with the fb_info_statement_timeout_att tag
3. Statement::setTimeout() saves the given timeout value and passes it with op_execute and op_execute2 packets
4. Statement::getTimeout() returns the saved timeout value
5. fb_dsql_set_timeout() is a wrapper over Statement::setTimeout()
6. If the protocol of the remote Firebird server is less than 16, it does not support statement timeouts. If that is the case,
   - “set” and “get” functions will return an isc_wish_list error
   - “info” will return the usual isc_info_error tag in the info buffer

Context Variable relating to Statement Timeouts

The 'SYSTEM' context has a new variable: `STATEMENT_TIMEOUT`. It contains the current value of the statement execution timeout that was set at connection level, or zero, if no timeout was set.

Statement Timeouts in the Monitoring Tables

In MON$ATTACHMENTS:

MON$STATEMENT_TIMEOUT	Connection level statement timeout

In MON$STATEMENTS:

MON$STATEMENT_TIMEOUT	Connection level statement timeout
MON$STATEMENT_TIMER	Timeout timer expiration time

MON$STATEMENT_TIMEOUT contains timeout value set at connection or statement level, in milliseconds. Zero, if timeout is not set.

MON$STATEMENT_TIMER contains NULL if no timeout was set or if a timer is not running.

Support for Statement Timeouts in isql

A new command has been introduced in *isql* to enable an execution timeout in milliseconds to be set for the next statement. The syntax is:

```
SET LOCAL_TIMEOUT <int>
```

After statement execution, the timer is automatically reset to zero.

Chapter 4

Changes to the Firebird API and ODS

ODS (On-Disk Structure) Changes

New ODS Number

Firebird 4.0 creates databases with an ODS (On-Disk Structure) version of 13.

Application Programming Interfaces

The wire protocol version for the Firebird 4.0 API is 16. Additions include

Session Timeouts

See Support for Session Timeouts at API Level in the chapter “Changes in the Firebird Engine”.

Statement Timeouts

See Support for Statement Timeouts at API Level in the chapter “Changes in the Firebird Engine”.

Chapter 5

Configuration Additions and Changes

Parameters for Timeouts

Two new parameters are available for global and per-database configuration, respectively, of server-wide and database-wide idle session and statement timeouts. They are discussed in detail elsewhere (see links).

ConnectionIdleTimeout

The value is integer, expressing minutes. Study the notes on idle session timeouts carefully to understand how this configuration fits in with related settings via SQL and the API.

See Setting the Session Timeout in the chapter “Changes to the Firebird Engine”.

StatementTimeout

The value is integer, expressing seconds. Study the notes on statement timeouts carefully to understand how this configuration fits in with related settings via SQL and the API.

See Setting a Statement Timeout in the chapter “Changes to the Firebird Engine”.

Parameters to Restrict Length of Object Identifiers

Object identifiers in an ODS 13 database can be up to 63 characters in length and the engine stores them in UTF-8, not UNICODE_FSS as previously. Two new global or per-database parameters are available if you need to restrict either the byte-length or the character-length of object names in ODS 13 databases for some reason.

Longer object names are optional, of course. Reasons you might need to restrict their length could include:

- Constraints imposed by the client language interface of existing applications, such as *gpre* or Delphi
- In-house coding standards
- Interoperability for cross-database applications such as a third-party replication system or an in-house system that uses multiple versions of Firebird

This is not an exhaustive list. It is the responsibility of the developer to test usage of longer object names and establish whether length restriction is necessary.

Whether setting one or both parameters has exactly the same effect will depend on the characters you use. Any non-ASCII character requires 2 bytes or more in UTF-8, so one cannot assume that byte-length and character-length have a direct relationship in all situations.

The two settings are verified independently and if either constrains the length limit imposed by the other, use of the longer identifier will be disallowed.

> **Warning**
> If you set either parameter globally, i.e., in firebird.conf, it will affect all databases, including the security database. That has the potential to cause problems!

MaxIdentifierByteLength

Sets a limit for the number of bytes allowed in an object identifier. It is an integer, defaulting to 252 bytes, i.e., 63 characters * 4, 4 being the maximum number of bytes for each character.

To set it to the limit in previous Firebird versions, use 31.

MaxIdentifierCharLength

Sets a limit for the number of characters allowed in an object identifier. It is an integer, defaulting to 63, the new limit implemented in Firebird 4.

Chapter 6

Security

Security improvements in Firebird 4 include:

Enhanced System Privileges

Alex Peshkov

Tracker ticket CORE-5343

This feature enables granting and revoking some special privileges for regular users to perform tasks that have been historically limited to SYSDBA only, for example:

- Run utilities such as gbak, gfix, nbackup and so on
- Shut down a database and bring it online
- Trace other users' attachments
- Access the monitoring tables

The implementation involved creating a set of SYSTEM PRIVILEGES, analogous to object privileges, from which lists of privileged tasks could be assigned to roles.

List of Valid System Privileges

The following table lists the names of the valid system privileges that can be granted and revoked to and from roles.

USER_MANAGEMENT	Manage users
READ_RAW_PAGES	Read pages in raw format using Attachment::getInfo()
CREATE_USER_TYPES	Add/change/delete non-system records in RDB$TYPES
USE_NBACKUP_UTILITY	Use nbackup to create database copies
CHANGE_SHUTDOWN_MODE	Shut down database and bring online
TRACE_ANY_ATTACHMENT	Trace other users' attachments
MONITOR_ANY_ATTACHMENT	Monitor (tables MON$) other users' attachments
ACCESS_SHUTDOWN_DATABASE	Access database when it is shut down
CREATE_DATABASE	Create new databases (given in security.db)
DROP_DATABASE	Drop this database
USE_GBAK_UTILITY	Use appropriate utility

USE_GSTAT_UTILITY	...
USE_GFIX_UTILITY>	...
IGNORE_DB_TRIGGERS	Insruct engine not to run DB-level triggers
CHANGE_HEADER_SETTINGS	Modify parameters in DB header page
SELECT_ANY_OBJECT_IN_DATABASE	Use SELECT for any selectable object
ACCESS_ANY_OBJECT_IN_DATABASE	Access (in any possible way) any object
MODIFY_ANY_OBJECT_IN_DATABASE	Modify (up to drop) any object
CHANGE_MAPPING_RULES	Change authentication mappings
USE_GRANTED_BY_CLAUSE	Use GRANTED BY in GRANT and REVOKE operators
GRANT_REVOKE_ON_ANY_OBJECT	GRANT and REVOKE rights on any object in database
GRANT_REVOKE_ANY_DDL_RIGHT	GRANT and REVOKE any DDL rights
CREATE_PRIVILEGED_ROLES	Use SET SYSTEM PRIVILEGES in roles

New Grantee Type SYSTEM PRIVILEGE

At a lower level, a new grantee type SYSTEM PRIVILEGE enables the SYSDBA to grant and revoke specific access privileges on database objects to a named system privilege. For example,

```
GRANT ALL ON PLG$SRP_VIEW TO SYSTEM PRIVILEGE USER_MANAGEMENT
```

grants to users having USER_MANAGEMENT privilege all rights to the view that is used in the SRP user management plug-in.

Assigning System Privileges to a Role

To put all this to use, we have some new clauses in the syntax of the CREATE ROLE and ALTER ROLE statements for attaching a list of the desired system privileges to a new or existing role.

The SET SYSTEM PRIVILEGES Clause

The syntax pattern for setting up or changing these special roles is as follows:

```
CREATE ROLE <name> SET SYSTEM PRIVILEGES TO <privilege1> {, <privilege2> {, ...  <privilegeN> }}
ALTER ROLE <name> SET SYSTEM PRIVILEGES TO <privilege1> {, <privilege2> {, ...  <privilegeN> }}
```

Both statements assign a non-empty list of system privileges to role <name>. The ALTER ROLE statement clears privileges previously assigned to the named role, before constructing the new list.

> **Important**
> Be aware that each system privilege provides a very thin level of control. For some tasks it may be necessary to give the user more than one privilege to perform some task. For example, add IGNORE_DB_TRIGGERS to USE_GSTAT_UTILITY because gstat needs to ignore database triggers.

Note that this facility provides a solution to an old Tracker request (CORE-2557) to implement permissions on the monitoring tables:

```
CREATE ROLE MONITOR SET SYSTEM PRIVILEGES TO MONITOR_ANY_ATTACHMENT;
GRANT MONITOR TO ROLE MYROLE;
```

Dropping System Privileges from a Role

This statement id used to clear the list of system privileges from the named role:

```
ALTER ROLE <name> DROP SYSTEM PRIVILEGES
```

The role <name> is not dropped, just the list attached to it.

Function RDB$SYSTEM_PRIVILEGE

To accompany all this delegation of power is a new built-in function, RDB$SYSTEM_PRIVILEGE(). It takes a valid system privilege as an argument and returns True if the current attachment has the given system privilege.

Format:

```
    RDB$SYSTEM_PRIVILEGE( <privilege> )
```

Example

```
select rdb$system_privilege(user_management) from rdb$database;
```

Granting a Role to Another Role

Roman Simakov

Tracker ticket CORE-1815

Firebird 4 allows a role to be granted to another role—a phenomenon that has been nicknamed “cumulative roles”. If you hear that term, it is referring to roles that are embedded within other roles by way of `GRANT ROLE a TO ROLE b`, something Firebird would not allow before.

> **Important**
>
> Take careful note that the GRANT ROLE syntax has been extended, along with its effects.

Syntax Pattern

```
GRANT [DEFAULT] <role name> TO [USER | ROLE] <user/role name> [WITH ADMIN OPTION];
REVOKE [DEFAULT] <role name> FROM [USER | ROLE] <user/role name> [WITH ADMIN OPTION];
```

The DEFAULT Keyword

If the optional DEFAULT keyword is included, the role will be used every time the user logs in, even if the role is not specified explicitly in the login credentials. During attachment, the user will get the privileges of all roles that have been granted to him/her with the DEFAULT property. This set will include all the privileges of all the embedded roles that have been granted to the <role name> role with the DEFAULT property.

Setting (or not setting) a role in the login does not affect the default role. The set of rights, given (by roles) to the user after login is the union of the login role (when set), all default roles granted to the user and all roles granted to this set of roles.

> **Note**
> A user still cannot acquire any privileges associated with a base role that has not been granted to his account or has been revoked.

WITH ADMIN OPTION Clause

If a user is to be allowed to grant a role to another user or to another role, the WITH ADMIN OPTION should be included. Subsequently the user will be able to grant any role in the sequence of roles granted to him, provided every role in the sequence has WITH ADMIN OPTION.

Example Using a Cumulative Role

```
CREATE DATABASE 'LOCALHOST:/TMP/CUMROLES.FDB';
CREATE TABLE T(I INTEGER);
CREATE ROLE TINS;
CREATE ROLE CUMR;
GRANT INSERT ON T TO TINS;
GRANT DEFAULT TINS TO CUMR WITH ADMIN OPTION;
GRANT CUMR TO USER US WITH ADMIN OPTION;
CONNECT 'LOCALHOST:/TMP/CUMROLES.FDB' USER 'US' PASSWORD 'PAS';
INSERT INTO T VALUES (1);
GRANT TINS TO US2;
```

Revoking the DEFAULT Property of a Role Assignment

To remove the DEFAULT proprty of a role assignment without revoking the role itself, include the DEFAULT keyword in the REVOKE statement:

```
REVOKE DEFAULT ghost FROM USER henry
REVOKE DEFAULT ghost FROM ROLE poltergeist
```

Otherwise, revoking a role altogether from a user is unchanged. However, now a role can be revoked from a role. For example,

```
REVOKE ghost FROM USER henry
REVOKE ghost FROM ROLE poltergeist
```

Function RDB$ROLE_IN_USE

Roman Simakov

Tracker ticket CORE-2762

A new built-in function lets the current user check whether a specific role is available under his/her current credentials. It takes a single-quoted role name as a string argument and returns a Boolean result.

> **Warning**
> The function should take a string of arbitrary length. However, at this point, it is limited to 32 characters because it appears not to recognise longer role names. A bug report was submitted: you may check the status of the ticket to determine whether the fix was completed for Alpha 1.

Format:

```
RDB$ROLE_IN_USE(<role_name>)
```

List Currently Active Roles

Tracker ticket CORE-751

To get a list of currently active roles you can run:

```
SELECT * FROM RDB$ROLES WHERE RDB$ROLE_IN_USE(RDB$ROLE_NAME)
```

SQL SECURITY Feature

Roman Simakov

Tracker ticket CORE-5568

This new feature in Firebird 4 enables executable objects (triggers, stored procedures, stored functions) to be defined to run in the context of an SQL SECURITY clause, as defined in the SQL standards (2003, 2011).

The SQL SECURITY scenario has two contexts: INVOKER and DEFINER. The INVOKER context corresponds to the privileges currently available to the CURRENT_USER or the non-human caller, while DEFINER corresponds to those available to the owner of the object.

The SQL SECURITY property is an optional part of an object's definition that can be applied to the object with DDL statements. The property cannot be dropped but it can be changed from INVOKER to DEFINER and vice versa.

It is not the same thing as SQL privileges, which are applied to **users** (human and some less animate types) to give them various types of access to database objects. When an executable object in Firebird needs access to a table, a view or another executable object, the target object is not accessible if the invoker does not have the necessary privileges on it. That has been the situation in previous Firebird versions and remains so in Firebird 4. That is, by default, all executable objects have the SQL SECURITY INVOKER property in Firebird 4. Any caller lacking the necessary privileges will be rejected.

If a routine has the SQL SECURITY DEFINER property applied to it, the invoking user or routine will be able to execute it if the required privileges have been granted to its owner, without the need for the caller to be granted those privileges specifically.

In summary:

- If INVOKER is set, the access rights for executing the call to an executable object are determined by checking the current user's active set of privileges
- If DEFINER is set, the access rights of the object owner will be applied instead, regardless of the current user's active privilege set

Syntax Patterns

```
CREATE TABLE <table-name> (...) [SQL SECURITY {DEFINER | INVOKER}]
ALTER TABLE <table-name> ... [{ALTER SQL SECURITY {DEFINER | INVOKER} | DROP SQL SECURITY}]
CREATE [OR ALTER] FUNCTION <function-name> ... [SQL SECURITY {DEFINER | INVOKER}] AS ...
CREATE [OR ALTER] PROCEDURE <procedure-name> ... [SQL SECURITY {DEFINER | INVOKER}] AS ...
CREATE [OR ALTER] TRIGGER <trigger-name> ... [SQL SECURITY {DEFINER | INVOKER} | DROP SQL SECURI
CREATE [OR ALTER] PACKAGE <package-name> [SQL SECURITY {DEFINER | INVOKER}] AS ...

ALTER DATABASE SET DEFAULT SQL SECURITY {DEFINER | INVOKER}
```

Packaged Routines

An explicit SQL SECURITY clause is not valid for procedures and functions defined in a package and will cause an error.

Triggers

Triggers inherit the setting of the SQL SECURITY property from the table, but it can be overriden explicitly. If the property is changed for a table, triggers that do not carry the overridden property will not see the effect of the change until next time the trigger is loaded into the metadata cache.

To remove an explicit SQL SECURITY option from a trigger, e.g. one named tr_ins, you can run

```
alter trigger tr_ins DROP SQL SECURITY;
```

To set it again to SQL SECURITY INVOKER, run

```
alter trigger tr_ins sql security invoker;
```

Examples Using the SQL SECURITY Property

1. With DEFINER set for table t, user US needs only the SELECT privilege on it. If it were set for INVOKER, the user would need also the EXECUTE privilege on function f.

```
set term ^;
create function f() returns int
as
begin
    return 3;
end^
set term ;^
create table t (i integer, c computed by (i + f())) SQL SECURITY DEFINER;
insert into t values (2);
grant select on table t to user us;

commit;

connect 'localhost:/tmp/7.fdb' user us password 'pas';
select * from t;
```

2. With DEFINER set for function f, user US needs only the EXECUTE privilege on it. If it were set for INVOKER, the user would need also the INSERT privilege on table t.

```
set term ^;
create function f (i integer) returns int SQL SECURITY DEFINER
as
begin
  insert into t values (:i);
  return i + 1;
end^
set term ;^
grant execute on function f to user us;

commit;

connect 'localhost:/tmp/59.fdb' user us password 'pas';
select f(3) from rdb$database;
```

3. With DEFINER set for procedure p, user US needs only the EXECUTE privilege on it. If it were set for INVOKER, either the user or the procedure would need also the INSERT privilege on table t.

```
set term ^;
create procedure p (i integer) SQL SECURITY DEFINER
as
begin
  insert into t values (:i);
end^
set term ;^

grant execute on procedure p to user us;
commit;
```

```
connect 'localhost:/tmp/17.fdb' user us password 'pas';
execute procedure p(1);
```

4. With DEFINER set for trigger tr, user US needs only the INSERT privilege on it. If it were set for INVOKER, either the user would need also the INSERT privilege on table t.

```
create table tr (i integer);
create table t (i integer);
set term ^;
create trigger tr_ins for tr after insert SQL SECURITY DEFINER
as
begin
  insert into t values (NEW.i);
end^
set term ;^
grant insert on table tr to user us;

commit;

connect 'localhost:/tmp/29.fdb' user us password 'pas';
insert into tr values(2);
```

The result would be the same if SQL SECURITY DEFINER were specified for table TR:

```
create table tr (i integer) SQL SECURITY DEFINER;
create table t (i integer);
set term ^;
create trigger tr_ins for tr after insert
as
begin
  insert into t values (NEW.i);
end^
set term ;^
grant insert on table tr to user us;

commit;

connect 'localhost:/tmp/29.fdb' user us password 'pas';
insert into tr values(2);
```

5. With DEFINER set for package pk, user US needs only the EXECUTE privilege on it. If it were set for INVOKER, either the user would need also the INSERT privilege on table t.

```
create table t (i integer);
set term ^;
create package pk SQL SECURITY DEFINER
as
begin
    function f(i integer) returns int;
end^

create package body pk
as
begin
    function f(i integer) returns int
```

```
    as
    begin
      insert into t values (:i);
      return i + 1;
    end
end^
set term ;^
grant execute on package pk to user us;

commit;

connect 'localhost:/tmp/69.fdb' user us password 'pas';
select pk.f(3) from rdb$database;
```

Chapter 7

Data Definition Language (DDL)

Quick Links

- Extended Length for Object Names
- Data type DECFLOAT
- Aliases for Binary String Types
- Extensions to the IDENTITY Type

DDL Enhancements

Enhancements have been added to the SQL data definition language lexicon in Firebird 4 include a new, high-precision floating-point data type and more extensions for the IDENTITY type.

New and extended DDL statements supporting the new security features are described in the Security chapter.

Extended Length for Object Names

Adriano dos Santos Fernandes

Tracker ticket CORE-749

The maximum length of objects names from this version forward is 63 characters, up from the previous maximum of 31 bytes.

Multi-byte identifiers can also be long now. For example, the previous limit allowed only 15 Cyrillic characters; now, they could be up to 63.

> **Note**
> Double quotes are not counted.

Restricting the Length

If, for some reason, you need to restrict the maximum size of object names, either globally or for individual databases, two new configuration parameters are available in `firebird.conf` and/or `databases.conf`: see Parameters to Restrict Length of Object Identifiers in the Configuration chapter for further details.

Data type DECFLOAT

Alex Peshkov

Tracker ticket CORE-5525

DECFLOAT is a DB2-compatible numeric type that stores floating-point numbers precisely, unlike FLOAT or DOUBLE PRECISION that provide a binary approximation of the purported precision. Firebird 4 accords with the IEEE standard by providing both 16-bit and 34-bit precision for this type. All intermediate calculations are performed with 34-bit values.

> **16-bit and 34-bit**
>
> The “16” and “34” refer to the mazimum precision in Base-10 digits. See https://en/wikipedia.org/wiki/iEEE_754#Basic_and_interchange_formats for a comprehensive table.

Syntax Rules

```
DECFLOAT(16)
DECFLOAT(34)
```

Storage complies with IEEE 754, storing data as 64 and 128 bits, respectively.

Examples

```
DECLARE VARIABLE VAR1 DECFLOAT(34);
--
CREATE TABLE TABLE1 (FIELD1 DECFLOAT(16));
```

Aspects of DECFLOAT Usage

Length of Literals

The length of DECFLOAT literals cannot exceed 1024 characters. Scientific notation is required for longer values. For example, **0.0<1020 zeroes>11** cannot be used as a literal, the equivalent in scientific notation, **1.1E-1022** is valid. Similarly, **10<1022 zeroes>0** can be presented as **1.0E1024**.

Use with Standard Functions

A number of standard scalar functions can be used with expressions and values of the DECFLOAT type. They are:

ABS	EXP	LN	LOG10	SIGN
CEILING	FLOOR	LOG	POWER	SQRT

The aggregate functions SUM, AVG, MAX and MIN work with DECFLOAT data, as do all of the statistics aggregates (like but not limited to STDDEV or CORR).

Special Functions for DECFLOAT

Firebird supports four functions, designed to support DECFLOAT data specifically:

- COMPARE_DECFLOAT—compares two DECFLOAT values to be equal, different or unordered. Returns a SMALLINT value, one of:

0	Values are equal
1	First value is less than second
2	First value is greater than second
3	Values are unordered, i.e., one or both is NAN / SNAN

  Unlike the comparison operators ('<', '=', '>', etc.) comparison is exact: `COMPARE_DECFLOAT(2.17, 2.170)` returns 2, not 0.

- NORMALIZE_DECFLOAT—takes a single DECFLOAT argument and returns it in its simplest form. That means that for any non-zero value, trailing zeros are removed with appropriate correction of the exponent.

  For example, `NORMALIZE_DECFLOAT(12.00)` returns 12 and `NORMALIZE_DECFLOAT(120)` returns 1.2E+2.

- QUANTIZE— takes two DECFLOAT arguments. The returned value is the first argument scaled using the second value as a pattern.

  For example, `QUANTIZE(1234, 9.999)` returns 1234.000.

- TOTALORDER—compares two DECFLOAT values including any special value. The comparison is exact. Returns a SMALLINT value, one of:

-1	First value is less than second
0	Values are equal
1	First value is greater than second

  For TOTALORDER comparisons, DECFLOAT values are ordered as follows:

```
-nan < -snan < -inf < -0.1 < -0.10 < -0 < 0 < 0.10 < 0.1 < inf < snan < nan
```

Session Control Operator SET DECFLOAT

Firebird supports the session control operator SET DECFLOAT which has three forms, as follows:

- SET DECFLOAT ROUND <mode> controls the rounding mode used in operations with DECFLOAT values. Valid modes are:

CEILING	towards +infinity
UP	away from 0
HALF_UP	to nearest, if equidistant, then up
HALF_EVEN	to nearest, if equidistant, ensure last digit in the result will be even
HALF_DOWN	to nearest, if equidistant, then down
DOWN	towards 0
FLOOR	towards -infinity
REROUND	up if digit to be rounded is 0 or 5, down in other cases

- SET DECFLOAT TRAPS TO <comma-separated traps list which may be empty> controls which exceptional conditions cause a trap. Valid traps are:

Division_by_zero	(set by default)
Inexact	
Invalid_operation	(set by default)
Overflow	(set by default)
Underflow	(set by default)

- SET DECFLOAT BIND <bind-type> controls how DECFLOAT values are represented externally, i.e. in messages or in the XSQLDA. The range of bindings is useful if one plans to use DECFLOAT values with some old client that does not support the native format. One can choose between strings (ideal precision, but poor support for further processing), floating point values (ideal support for further processing but poor precision) or scaled integers (good support for further processing and the required precision but having a very limited range of values). CHAR binding is a satisfactory choice for most general purpose GUI client tools.

  Valid binding types are:

NATIVE	Use IEEE754 binary representation
CHAR/CHARACTER	Use ASCII string
DOUBLE PRECISION	Use the same 8-byte floating-point representation as is used for DOUBLE PRECISION fields
BIGINT	As BIGINT, with optional comma-separated SCALE clause, e.g., `BIGINT,3`

Aliases for Binary String Types

Dimitry Sibiryakov

Tracker ticket CORE-5064

Data types named BINARY(n), VARBINARY(n) and BINARY VARYING(n) have been added to the lexicon as optional aliases for defining string columns in CHARACTER SET OCTETS.

BINARY(n) is an alias for CHAR(n) CHARACTER SET OCTETS, while VARBINARY(n) and BINARY VARYING(n) are aliases for VARCHAR(n) CHARACTER SET OCTETS and for each other.

Extensions to the IDENTITY Type

Adriano dos Santos Fernandes

An IDENTITY column is one that is formally associated with an internal sequence generator and has its value set automatically when omitted from an INSERT statement.

The IDENTITY sub-type appeared in Firebird 3 and has undergone a number of extensions in V.4, including implementation of DROP IDENTITY, the GENERATED ALWAYS and OVERRIDE directives and the INCREMENT BY option.

Extended Syntax for Managing IDENTITY Columns

```
<column definition> ::=
  <name> <type> GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY [ ( <identity column option>... ) ]

<identity column option> ::=
  START WITH <value> |  INCREMENT [ BY ] <value>

<alter column definition> ::=
  <name> <set identity column generation clause> [ <alter identity column option>... ] |
  <name> <alter identity column option>... |
  <name> DROP IDENTITY

 <set identity column generation clause> ::=
  SET GENERATED { ALWAYS | BY DEFAULT }

<alter identity column option> ::=
  RESTART [ WITH <value> ] | SET INCREMENT [ BY ] <value>
```

Rules and Characteristics

- The type of an identity column must be an exact number type with zero scale, comprising SMALLINT, INTEGER, BIGINT, NUMERIC(s,0) and DECIMAL(s,0).
- Identity columns cannot have a DEFAULT value or be defined as COMPUTED BY <expr>
- A regular column cannot be altered to be an identity column
- Identity columns cannot be defined or made non-nullable
- The engine does not enforce uniqueness automatically. A unique constraint or index of the required kind must be defined explicitly.
- An INCREMENT value cannot be zero

The Firebird 4 Extensions to IDENTITY

The Firebird 3 implementation was minimal, effectively formalizing the traditional way of implementing generated keys in Firebird, without many options. Firebird 4 puts some meat on those bones.

The GENERATED ALWAYS and BY DEFAULT Directives

Tracker ticket CORE-5463

The earlier implementation behaved like the traditional Firebird setup for generating integer keys automatically when the column was omitted from the insert operation's column list. If the column was not listed, the IDENTITY generator would supply the value.

A GENERATED BY clause is mandatory. The GENERATED BY DEFAULT directive, present in the Firebird 3 syntax, implemented this behaviour formally without the alternative GENERATED ALWAYS option, :

```
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 order by id;

commit;
```

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

The GENERATED ALWAYS directive introduces alternative behaviour that enforces the use of the identity generator, whether or not the user supplies a value.

> **Overriding the defined behaviour**
>
> For one-off cases this enforcement can be overridden in DML by including an OVERRIDING SYSTEM VALUE clause.
>
> On the other hand, for one-off cases where you want to override the defined action for a column defined with the GENERATED BY DEFAULT directive to behave as though it were defined as GENERATED ALWAYS and ignore any DML-supplied value, the clause OVERRIDING USER VALUE is available.
>
> For more details, see OVERRIDING Clause for IDENTITY Columns in the DML chapter.

Changing the Defined Behaviour

The ALTER COLUMN clause of ALTER TABLE now has syntax for changing the default GENERATED behaviour from BY DEFAULT to ALWAYS, or vice versa:

```
alter table objects
  alter id
  SET GENERATED ALWAYS;
```

DROP IDENTITY Clause

Tracker ticket CORE-5431

For a situation where you want to drop the IDENTITY property from a column but retain the data, the DROP IDENTITY clause is available to the ALTER TABLE statement:

```
alter table objects
  alter id
  DROP IDENTITY;
```

INCREMENT BY Option for IDENTITY Columns

Tracker ticket CORE-5430

By default, identity columns start at 1 and increment by 1. The INCREMENT BY option can now be used to set the increment for some positive step of 2 or more:

```
create table objects (
  id integer generated BY DEFAULT as
     identity START WITH 10000 INCREMENT BY 10
     primary key,
  name varchar(15)
);
```

Changing the Increment (Step) Value

For changing the step value of the sequence produced by an IDENTITY generator, the SET INCREMENT clause is available in the latest ALTER TABLE statement syntax:

```
alter table objects
  alter id SET INCREMENT BY 5;
```

> **Note**
> Changing the step value does not affect existing data.

Implementation

Two columns have been inserted in RDB$RELATION_FIELDS: RDB$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. RDB $IDENTITY_TYPE stores the value 0 for GENERATED ALWAYS, 1 for GENERATED BY DEFAULT, and NULL for non-identity columns.

Chapter 8

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

DEFAULT Context Value for Inserting and Updating

Adriano dos Santos Fernandes

Tracker ticket CORE-5449

Support has been implemented to enable the declared default value for a column or domain to be included directly in INSERT, UPDATE, MERGE and UPDATE OR INSERT statements by use of the keyword DEFAULT in the column's position. If DEFAULT appears in the position of a column that has no default value defined, the engine will attempt to write NULL to that column.

The feature is defined in (SQL:2011): 6.5 <contextually typed value specification>.

Simple Examples

```
insert into sometable (id, column1)
values (DEFAULT, 'name')
--
update sometable
  set column1 = 'a', column2 = default
```

Notes

If id is an identity column, the identity value will be generated, even if there is an UPDATE ... SET command associated with the column.

If DEFAULT is specified on a computed column, the parser will allow it but it will have no effect.

In columns populated by triggers in the traditional way, the value from DEFAULT enters the NEW context variable of any BEFORE INSERT or BEFORE UPDATE trigger.

DEFAULT vs DEFAULT VALUES

Since v.2.1, Firebird has supported the DEFAULT VALUES clause. The two clauses are not the same. The DEFAULT VALUES clause is an alternative to the VALUES clause and can be used only when all of the colums specified in the column list have been defined with default values.

OVERRIDING Clause for IDENTITY Columns

Adriano dos Santos Fernandes

Tracker ticket CORE-5463

Identity columns defined with the BY DEFAULT attibute can be overriden in statements that insert rows (INSERT, UPDATE OR INSERT, MERGE ... WHEN NOT MATCHED) just by specifying the value in the values list. For identity columns defined with the GENERATE ALWAYS attribute, that kind of override is not allowed.

Making the value passed in the INSERT statement for an ALWAYS column acceptable to the engine requires use of the OVERRIDING clause with the SYSTEM VALUE sub-clause, as illustrated below:

```
insert into objects (id, name)
  OVERRIDING SYSTEM VALUE values (11, 'Laptop');
```

OVERRIDING supports another sub-clause, USER VALUE, for use with BY DEFAULT columns to direct the engine to ignore the value passed in INSERT and use the sequence defined for the identity column:

```
insert into objects (id, name)
  OVERRIDING USER VALUE values (12, 'Laptop');  -- 12 is not used
```

Extension of SQL Windowing Features

Adriano dos Santos Fernandes

In addition to the OVER clause, Firebird window functions can now use partitions, order and frames.

Syntax Pattern

The pattern for Firebird 4 windowing syntax is as follows:

```
<window function> ::=
<window function name>([<expr> [, <expr> ...]])
    OVER {<window specification> | <existing window name>}

<window specification> ::=
  ([<existing window name>] [<window partition>] [<window order>] [<window frame>])

<window partition> ::=
  PARTITION BY <expr> [, <expr> ...]

<window order> ::=
  ORDER BY <expr> [<direction>] [<nulls placement>] [, <expr> [<direction>] [<nulls placement>]]

<window frame> ::=
  {RANGE | ROWS} <window frame extent>

<window frame extent> ::=
  {<window frame start> | <window frame between>}

<window frame start> ::=
  {UNBOUNDED PRECEDING | <expr> PRECEDING | CURRENT ROW}

<window frame between> ::=
  BETWEEN {UNBOUNDED PRECEDING | <expr> PRECEDING | <expr> FOLLOWING | CURRENT ROW} AND
          {UNBOUNDED FOLLOWING | <expr> PRECEDING | <expr> FOLLOWING | CURRENT ROW}

<direction> ::=
  {ASC | DESC}

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

<query spec> ::=
  SELECT
    [<limit clause>]
    [<distinct clause>]
    <select list>
    [<where clause>]
    [<group clause>]
    [<having clause>]
    [<named windows clause>]
    [<plan clause>]

<named windows clause> ::=
  WINDOW <window definition> [, <window definition>] ...

<window definition> ::=
  <new window name> AS <window specification>
```

Frames for Window Functions

Tracker ticket CORE-3647

A *frame* can be specified, within which certain window functions are to work.

Syntax Elements for Frames

The following extract from the syntax pattern above explains the elements that affect frames:

```
<window frame> ::=
  {RANGE | ROWS} <window frame extent>

<window frame extent> ::=
  {<window frame start> | <window frame between>}

<window frame start> ::=
  {UNBOUNDED PRECEDING | <expr> PRECEDING | CURRENT ROW}

<window frame between> ::=
  BETWEEN {UNBOUNDED PRECEDING | <expr> PRECEDING | <expr> FOLLOWING | CURRENT ROW} AND
          {UNBOUNDED FOLLOWING | <expr> PRECEDING | <expr> FOLLOWING | CURRENT ROW}
```

The frame comprises three pieces: unit, start bound and end bound. The unit can be RANGE or ROWS and defines how the bounds will work. The bounds are:

<expr> PRECEDING
<expr> FOLLOWING
CURRENT ROW

- With RANGE, the ORDER BY should specify only one expression, and that expression should be of a numeric, date, time or timestamp type. For <expr> PRECEDING and <expr> FOLLOWING bounds, <expr> is subtracted from the order expression in the case of PRECEDING and added to it in the case of FOLLOWING. For CURRENT ROW, the order expression is used as-is.

  All rows inside the partition that are between the bounds are considered part of the resulting window frame.

- With ROWS, order expressions are not limited by number or type. For this unit, <expr> PRECEDING, <expr> FOLLOWING and CURRENT ROW relate to the row position under the partition, and not to the values of the ordering keys.

UNBOUNDED PRECEDING and UNBOUNDED FOLLOWING work identically with RANGE and ROWS. UNBOUNDED PRECEDING looks for the first row and UNBOUNDED FOLLOWING the last one, always inside the partition.

The frame syntax with <window frame start> specifies the start frame, with the end frame being CURRENT ROW.

Some window functions discard frames:

- ROW_NUMBER, LAG and LEAD always work as ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
- DENSE_RANK, RANK, PERCENT_RANK and CUME_DIST always work as RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW.
- FIRST_VALUE, LAST_VALUE and NTH_VALUE respect frames, but the RANGE unit behaviour is identical to ROWS.

Navigational Functions with Frames

Navigational functions, implemented in Firebird 3, get the simple (non-aggregated) value of an expression from another row that is within the same partition. They can operate on frames. These are the syntax patterns:

```
  <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> ] ] )
```

When FIRST_VALUE, LAST_VALUE and NTH_VALUE operate on a window frame, the default frame is RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW. This is likely to produce strange results for LAST_VALUE in particular, and also for NTH_VALUE.

Example Using Frames

When the ORDER BY window clause is used but a frame clause is omitted, the default frame just described causes the query below to produce weird behaviour for the `sum_salary` column. It sums from the partition start to the current key, instead of summing the whole partition.

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

Result:

```
 | id | salary | sum_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 |
```

A frame can be set explicitly to sum the whole partition, as follows:

```
 select
     id,
     salary,
     sum(salary) over (
       order by salary
       ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
                     ) sum_salary
   from employee
   order by salary;
```

Result:

```
 | id | salary | sum_salary |
 |---:|-------:|-----------:|
 |  3 |   8.00 |      49.00 |
 |  4 |   9.00 |      49.00 |
 |  1 |  10.00 |      49.00 |
 |  5 |  10.00 |      49.00 |
 |  2 |  12.00 |      49.00 |
```

This query “fixes” the weird nature of the default frame clause, producing a result similar to a simple `OVER ()` clause without `ORDER BY`.

We can use a range frame to compute the count of employees with salaries between (an employee's salary - 1) and (his salary + 1) with this query:

```
select
    id,
    salary,
    count(*) over (
      order by salary
      RANGE BETWEEN 1 PRECEDING AND 1 FOLLOWING
    ) range_count
  from employee
  order by salary;
```

Result:

```
| id | salary | range_count |
|---:|-------:|------------:|
|  3 |   8.00 |           2 |
|  4 |   9.00 |           4 |
|  1 |  10.00 |           3 |
|  5 |  10.00 |           3 |
|  2 |  12.00 |           1 |
```

Named Windows

Tracker ticket CORE-5346

In a query with the WINDOW clause, a window can be explicitly named to avoid repetitive or confusing expressions.

A named window can be used

1. in the OVER element to reference a window definition, e.g. OVER <window-name>
2. as a base window of another named or inline (OVER) window, if it is not a window with a frame (ROWS or RANGE clauses).

> **Note**
> a window with a base window cannot have PARTITION BY, nor override the ordering (ORDER BY sequence) of a base window.

Example Using Named Windows

```
select
    id,
    department,
    salary,
    count(*) over w1,
    first_value(salary) over w2,
    last_value(salary) over w2
  from employee
  window w1 as (partition by department),
         w2 as (w1 order by salary)
  order by department, salary;
```

More Window Functions

Adriano dos Santos Fernandes
Hajime Nakagami

Tracker ticket CORE-1688

More ANSI SQL:2003 window functions—the ranking functions PERCENT_RANK, CUME_DIST and NTILE.

Ranking Functions

```
<ranking window function> ::=
    DENSE_RANK() |
    RANK() |
    PERCENT_RANK() |
    CUME_DIST() |
    NTILE(<expr>) |
    ROW_NUMBER()
```

Ranking functions compute the ordinal rank of a row within the window partition. The basic functions in this category, present since Firebird 3, are DENSE_RANK, RANK and ROW_NUMBER. These function enable creation of various types of incremental counters to generate sets in ways that are analogous with operations such as SUM(1) OVER (ORDER BY SALARY).

The new functions implemented in Firebird 4 are:

- PERCENT_RANK is a ratio of RANK to group count.
- CUME_DIST is the cumulative distribution of a value in a group.
- NTILE takes an argument and distributes the rows into the specified number of groups. The argument is restricted to integral positive literal, variable (:var) and DSQL parameter (?).

Simple Example

The following example illustrates the behaviour of ranking functions. SUM is included for comparison.

```
select
    id,
    salary,
    dense_rank() over (order by salary),
    rank() over (order by salary),
    percent_rank() over (order by salary),
    cume_dist() over (order by salary),
    ntile(3) over (order by salary),
    row_number() over (order by salary),
    sum(1) over (order by salary)
  from employee
  order by salary;
```

The result set looks something like the following, although trailing zeroes have been truncated here in order to fit the lines to the document page:

```
id  salary   dense_rank   rank   percent_rank   cume_dist   ntile   row_number   sum
3     8.00            1      1      0.0000000  0.20000000       1            1     1
4     9.00            2      2      0.2500000  0.40000000       1            2     2
1    10.00            3      3      0.5000000  0.80000000       2            3     4
5    10.00            3      3      0.5000000  0.80000000       2            4     4
2    12.00            4      5      1.0000000  1.00000000       3            5     5
```

Optional AUTOCOMMIT for SET TRANSACTION

Dmitry Yemanov

Tracker ticket CORE-5119

Autocommit mode is now supported in the SET TRANSACTION statement syntax.

Example

```
SET TRANSACTION SNAPSHOT NO WAIT AUTO COMMIT;
```

Built-in Functions

Additions and changes to the set of built-in functions in Firebird 4.

New Built-in Functions

Two new built-in functions were added to support the new security features. They are not described here—the descriptions are located in the Security chapter. They are:

- RDB$SYSTEM_PRIVILEGE
- RDB$ROLE_IN_USE

Some special functions were added for operations on DECFLOAT data: :

- COMPARE_DECFLOAT—compares two DECFLOAT values to be equal, different or unordered
- NORMALIZE_DECFLOAT—takes a single DECFLOAT argument and returns it in its simplest form
- QUANTIZE— takes two DECFLOAT arguments and returns the first argument scaled using the second value as a pattern

Detailed descriptions are in the DDL chapter, in the topic Special Functions for DECFLOAT.

Changes to Built-in Functions

Functions improved in this release:

HASH()

Adriano dos Santos Fernandes

Tracker ticket CORE-4436

Returns a hash for a string using a specified algorithm. Format is:

```
HASH( <string> [ USING <algorithm> ] )

algorithm ::= { MD5 | SHA1 | SHA256 | SHA512 }
```

The syntax with the optional USING clause is introduced in FB 4.0 and returns VARCHAR strings in character set OCTETS.

> **Important**
>
> The syntax without the USING clause is still supported. It uses the 64-bit variation of the non-cryptographic PJW hash function (also known as ELF64):
>
> https://en.wikipedia.org/wiki/PJW_hash_function
>
> which is very fast and can be used for general purposes (hash tables, etc), but its collision quality is sub-optimal. Other hash functions (specified explicitly in the USING clause) should be used for more reliable hashing.

Examples

```
select hash(x using sha256) from y;
--
select hash(x) from y; -- not recommended
```

Chapter 9

Procedural SQL (PSQL)

Recursion is now supported in sub-routines. A few improvements have been implemented to help in logging exceptions from the various error contexts supported in PSQL.

Recursion for subroutines

Adriano dos Santos Fernandes

Tracker ticket CORE-5380

Starting in FB 4, subroutines may be recursive or call other subroutines.

Examples

A couple of recursive sub-functions in EXECUTE BLOCK:

```
execute block returns (i integer, o integer)
as
    -- Recursive function without forward declaration.
    declare function fibonacci(n integer) returns integer
    as
    begin
      if (n = 0 or n = 1) then
       return n;
     else
       return fibonacci(n - 1) + fibonacci(n - 2);
    end
    begin
      i = 0;

      while (i < 10)
      do
      begin
        o = fibonacci(i);
        suspend;
        i = i + 1;
      end
   end

-- With forward declaration and parameter with default values.

execute block returns (o integer)
as
    -- Forward declaration of P1.
    declare procedure p1(i integer = 1) returns (o integer);

    -- Forward declaration of P2.
    declare procedure p2(i integer) returns (o integer);
```

```
    -- Implementation of P1 should not re-declare parameter default value.
    declare procedure p1(i integer) returns (o integer)
    as
    begin
        execute procedure p2(i) returning_values o;
    end

    declare procedure p2(i integer) returns (o integer)
    as
    begin
        o = i;
    end
begin
    execute procedure p1 returning_values o;
    suspend;
end
```

A Helper for Logging Context Errors

A new system function enables the module to pass explicit context information from the error block to a logging routine.

System Function RDB$ERROR()

Dmitry Yemanov

Tracker tickets CORE-2040 and CORE-1132

The function RDB$ERROR() takes a PSQL error context as input and returns the specific context of the active exception. Its scope is confined to the context of the exception-handling block in PSQL. Outside the exception handling block, RDB$ERROR always contains NULL.

The type of the return value depends on the context.

Syntax Rules

```
RDB$ERROR ( context )
context ::= { GDSCODE | SQLCODE | SQLSTATE | EXCEPTION | MESSAGE }
```

Contexts

GDSCODE	INTEGER	Context variable: refer to documentation
SQLCODE	INTEGER	Context variable: refer to documentation
SQLSTATE	CHAR(5) CHARACTER SET ASCII	Context variable: refer to documentation

EXCEPTION	VARCHAR(63) CHARACTER SET UTF8	Returns name of the active user-defined exception or NULL if the active exception is a system one
MESSAGE	VARCHAR(1024) CHARACTER SET UTF8	Returns interpreted text for the active exception

Example

```
BEGIN
  ...
WHEN ANY DO
  EXECUTE PROCEDURE P_LOG_EXCEPTION(RDB$ERROR(MESSAGE));
END
```

Chapter 10

Monitoring & Command-line Utilities

Improvements and additions to the Firebird utilities continue.

Monitoring

Additions to MON$ATTACHMENTS and MON$STATEMENTS to report on session and statement timeouts. Refer to Timeouts at Two levels in the chapter “Changes in the Firebird Engine” for details.

New columns in the tables:

- In MON$ATTACHMENTS:

MON$IDLE_TIMEOUT	Connection level idle timeout
MON$IDLE_TIMER	Idle timer expiration time
MON$STATEMENT_TIMEOUT	Connection level statement timeout

- In MON$STATEMENTS:

MON$STATEMENT_TIMEOUT	Connection level statement timeout
MON$STATEMENT_TIMER	Timeout timer expiration time

nBackup: UUID-based Backup and In-Place Merge

Roman Simakov
Vlad Khorsun

Tracker ticket CORE-2216

The *nBackup* utility in Firebird 4 can perform a physical backup that uses the GUID (UUID) of the most recent backup of a read-only standby database to establish the backup target file. Increments from the <source database> can be applied continuously to the standby database, eliminating the need to keep and apply all increments since the last full backup.

The new style of “warm” backup and merge to a standby database can be run without affecting an existing multilevel backup scheme on the live database.

Making Backups

The syntax pattern for this form of backup with *nBackup* is as follows:

```
nbackup -B[ACKUP] <level> | <GUID> <source database> [<backup file>]
```

Merging-in-Place from the Backup

The syntax pattern for an in-place “restore” to merge the incremental backup file with the standby database is:

```
nbackup -I[NPLACE] -R[ESTORE] <standby database> <backup file>
```

> **Note**
>
> “Restore” here means merging the increment from the backup file with the standby database.
>
> *Switch names may change before the final release.*

Example of an On-line Backup and Restore

1. Use *gstat* to get the UUID of the standby database:

```
  gstat -h <standby database>
...
    Variable header data:
        Database backup GUID: {8C519E3A-FC64-4414-72A8-1B456C91D82C}
```

2. Use the backup UUID to produce an incremental backup:

```
  nbackup -B {8C519E3A-FC64-4414-72A8-1B456C91D82C} <source database> <backup file>
```

3. Apply increment to the standby database:

```
  nbackup -I -R <standby database> <backup file>
```

isql: Support for Statement Timeouts

A new command has been introduced in *isql* to enable an execution timeout in milliseconds to be set for the next statement. The syntax is:

```
SET LOCAL_TIMEOUT <int>
```

After statement execution, the timer is automatically reset to zero.

Chapter 11

Bugs Fixed

Firebird 4.0 Alpha 1 Release: Bug Fixes

The following fixes to pre-existent bugs are noted:

(CORE-5545) Using the POSITION parameter with the [RE]CREATE TRIGGER syntax would cause an "unknown token" error if POSITION was written in the logically correct place, i.e., after the main clauses of the statement. For example, the following should work because POSITION comes after the other specifications:

```
RECREATE TRIGGER T1
BEFORE INSERT
ON tbl
POSITION 1 AS
BEGIN
  --
END
```

However, it would exhibit the error, while the following would succeed:

```
RECREATE TRIGGER T1
BEFORE INSERT
POSITION 1
ON tbl
AS
BEGIN
  --
END
```

The fix makes the first example correct and the second should throw the error.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-5454) Inserting into an updatable view without an explicit column list would fail.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-5408) The result of a Boolean expression could not be concatenated with a string literal.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-5404) Inconsistent column and line references were being returned in error messages for faulty PSQL definitions.

fixed by A. dos Santos Fernandes

~ ~ ~

(CORE-5237) Processing of the *include* clause in configuration files was mishandling dot (.) and asterisk (*) characters in the file name and path of the included file.

fixed by D. Sibiryakov

~ ~ ~

(CORE-5223) Double dots in file names for databases were prohibited if the *DatabaseAccess* configuration parameter was set to restrict access to a list of directories.

fixed by D. Sibiryakov

~ ~ ~

(CORE-5141) Field definition would allow multiple NOT NULL clauses. For example,

```
create table t (a integer not null not null not null)
```

The fix makes the behaviour consistent with CREATE DOMAIN behaviour and the example will return the error “Duplicate specification of NOT NULL - not supported”.

fixed by D. Sibiryakov

~ ~ ~

(CORE-4985) A non-privileged user could implicitly count records in a restricted table.

fixed by D. Yemanov

~ ~ ~

(CORE-4701) Garbage collection for indexes and BLOBs was not taking data in the Undo log into account.

fixed by D. Sibiryakov

~ ~ ~

(CORE-4483) In PSQL, data changed by executing a procedure was not visible to the WHEN handler if the exception occurred in the called procedure.

fixed by D. Sibiryakov

~ ~ ~

(CORE-4424) In PSQL, execution flow would roll back to the wrong savepoint if multiple exception handlers were executed at the same level.

fixed by D. Sibiryakov

~ ~ ~

Chapter 12

Firebird 4.0 Project Teams

Table 12.1. 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
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
Pavel Zotov	Russian Federation	QA tester and tools developer
Philippe Makowski	France	QA tester and maintainer of EPEL kits
Paul Reeves	France	Windows 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
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
Pavel Menshchikov	Russian Federation	Documentation translator

Developer	Country	Major Tasks
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.