



Firebird 4.0 Release Notes

Firebird Project: Core Developers, Helen Borrie, Dmitry Yemanov

Version 0400-37, 26 May 2021

Table of Contents

1. General Notes	7
Bug Reporting	7
Documentation	7
2. New In Firebird 4.0	8
Summary of New Features	8
Complete In Firebird 4.0 Final Release	8
Complete In Release Candidate 1	9
Complete In Beta 2	11
Complete In Beta 1	15
Compatibility with Older Versions	17
3. Changes in the Firebird Engine	18
Maximum Page Size Increased To 32KB	18
External Functions (UDFs) Feature Deprecated	18
Support for International Time Zones	18
Session Time Zone	19
Time Zone Format	19
Data Types for Time Zone Support	19
API Support for Time Zones	20
Time Zone Statements and Expressions	20
Virtual table RDB\$TIME_ZONES	20
Package RDB\$TIME_ZONE_UTIL	20
Updating the Time Zone Database	22
Firebird Replication	22
Replication Modes	23
Access Modes	23
Journaling	23
Error Reporting	24
Setting Up Replication	25
Pooling of External Connections	29
Key Characteristics of Connection Pooling	29
How the Connection Pool Works	30
Managing the Connection Pool	31
Querying the Connection Pool	31
Parameters for Configuring the Connection Pool	31
Timeouts at Two levels	32
Idle Session Timeouts	32
Statement Timeouts	35
Commit Order for Capturing the Database Snapshot	40

The 'Commit Order' Approach	40
Read Consistency for Statements in Read-Committed Transactions.....	42
Garbage Collection	45
Precision Improvement for Calculations Involving NUMERIC and DECIMAL	46
Increased Number of Formats for Views.....	46
Optimizer Improvement for GROUP BY.....	47
<i>xinetd</i> Support on Linux Replaced.....	47
Support for RISC v.64 Platform.....	47
Virtual table RDB\$CONFIG.....	47
4. Changes to the Firebird API and ODS	49
ODS (On-Disk Structure) Changes	49
New ODS Number	49
New System Tables	49
New Columns in System Tables	49
Application Programming Interfaces	50
Services Cleanup	50
Services API Extensions	50
Timeouts for Sessions & Statements	50
New Isolation Sub-level for READ COMMITTED Transactions.....	50
Support for Batch Insert and Update Operations in the API.....	51
API Support for Time Zones	57
API Support for DECFLOAT and Long Numerics.....	60
Additions to Other Interfaces	62
Extensions to various getInfo() Methods	63
Additions to the Legacy (ISC) API.....	65
5. Reserved Words and Changes.....	66
New Keywords in Firebird 4.0	66
Reserved	66
Non-reserved	66
6. Configuration Additions and Changes.....	67
Parameters for Timeouts	67
ConnectionIdleTimeout	67
StatementTimeout	67
Parameters for External Connection Pooling	67
ExtConnPoolSize	67
ExtConnPoolLifetime	67
Parameters to Restrict Length of Object Identifiers	67
MaxIdentifierByteLength	68
MaxIdentifierCharLength	68
Parameters Supporting Read Consistency in Transactions	68
ReadConsistency	68

TipCacheBlockSize	69
SnapshotsMemSize	69
Other Parameters	69
ClientBatchBuffer	69
DataTypeCompatibility	69
DefaultTimeZone	70
OutputRedirectionFile	70
Srp256 becomes the default authentication method	70
ChaCha is added as a default wire encryption method	70
TempCacheLimit at database level	70
UseFileSystemCache is added as a replacement for FileSystemCacheThreshold	70
InlineSortThreshold	71
7. Security	72
Enhanced System Privileges	72
List of Valid System Privileges	72
New Grantee Type SYSTEM PRIVILEGE	73
Assigning System Privileges to a Role	73
Function RDB\$SYSTEM_PRIVILEGE	74
Granting a Role to Another Role	74
The DEFAULT Keyword	75
WITH ADMIN OPTION Clause	75
Example Using a Cumulative Role	75
Revoking the DEFAULT Property of a Role Assignment	75
Function RDB\$ROLE_IN_USE	76
SQL SECURITY Feature	76
Triggers	77
Examples Using the SQL SECURITY Property	78
Built-in Cryptographic Functions	81
ENCRYPT() and DECRYPT()	81
RSA_PRIVATE()	82
RSA_PUBLIC()	82
RSA_ENCRYPT()	82
RSA_DECRYPT()	83
RSA_SIGN_HASH()	83
RSA_VERIFY_HASH()	84
Improvements to Security Features	85
User Managing Other Users	85
8. Management Statements	86
Connections Pooling Management	86
ALTER EXTERNAL CONNECTIONS POOL	86
ALTER SESSION RESET	87

Errors handling	88
Time Zone Management	89
SET TIME ZONE	89
Timeout Management	89
Setting DECFLOAT Properties	89
Setting Data Type Coercion Rules	90
9. Data Definition Language (DDL)	93
Quick Links	93
Extended Length for Object Names	93
Restricting the Length	93
New Data Types	93
Data Type INT128	93
Data Types TIME WITH TIME ZONE and TIMESTAMP WITH TIME ZONE	94
Data Type DECFLOAT	94
DDL Enhancements	95
Increased Precision for Exact Numeric Types	96
Standard Compliance for Data Type FLOAT	97
Data Type Extensions for Time Zone Support	97
Aliases for Binary String Types	98
Extensions to the IDENTITY Type	98
Excess parameters in EXECUTE STATEMENT	102
Replication Management	102
10. Data Manipulation Language (DML)	104
Quick Links	104
Lateral Derived Tables	104
DEFAULT Context Value for Inserting and Updating	105
DEFAULT vs DEFAULT VALUES	106
OVERRIDING Clause for IDENTITY Columns	106
Extension of SQL Windowing Features	106
Frames for Window Functions	108
Named Windows	111
More Window Functions	112
FILTER Clause for Aggregate Functions	113
Syntax for FILTER Clauses	114
Optional AUTOCOMMIT for SET TRANSACTION	114
Sharing Transaction Snapshots	114
Expressions and Built-in Functions	115
New Functions and Expressions	115
Changes to Built-in Functions and Expressions	122
SUBSTRING()	123
UDF Changes	123

Miscellaneous DML Improvements	124
Improve Error Message for an Invalid Write Operation	124
Improved Failure Messages for Expression Indexes	124
RETURNING * Now Supported	124
11. Procedural SQL (PSQL)	126
Recursion for subroutines	126
A Helper for Logging Context Errors	127
System Function RDB\$ERROR()	127
Allow Management Statements in PSQL Blocks	128
12. Monitoring & Command-line Utilities	130
Monitoring	130
<i>nbackup</i>	131
UUID-based Backup and In-Place Merge	131
Restore and Fixup for Replica Database	132
<i>isql</i>	132
Support for Statement Timeouts	132
Better transaction control	132
<i>gbak</i>	134
Backup and Restore with Encryption	134
Enhanced Restore Performance	136
Friendlier “-fix_fss_*” Messages	136
Ability to Backup/Restore Only Specified Tables	137
<i>gfix</i>	137
Configuring and managing replication	137
13. Compatibility Issues	138
SQL	138
Deprecation of Legacy SQL Dialect 1	138
Read Consistency for READ COMMITTED transactions Used By Default	138
Deprecation of External Functions (UDFs)	138
Changes in DDL and DML Due to Timezone Support	140
Prefixed Implicit Date/Time Literals Now Rejected	140
Starting Value of Sequences	141
INSERT ... RETURNING Now Requires a SELECT privilege	142
Utilities	142
Deprecation of QLI	142
14. Bugs Fixed	143
Firebird 4.0 Release: Bug Fixes	143
Core Engine	143
Server Crashes/Hang-ups	147
API/Remote Interface	148
Build Issues	148

Utilities	149
Firebird 4.0 Release Candidate 1: Bug Fixes	149
Core Engine	149
Server Crashes/Hang-ups	154
API/Remote Interface	154
Build Issues	155
Utilities	155
Firebird 4.0 Beta 2 Release: Bug Fixes	155
Core Engine	155
Server Crashes/Hang-ups	160
API/Remote Interface	161
Build Issues	161
Utilities	162
Firebird 4.0 Beta 1 Release: Bug Fixes	164
Core Engine	164
Server Crashes/Hang-ups	168
Security	168
Utilities	169
Build Issues	170
Firebird 4.0 Alpha 1 Release: Bug Fixes	170
15. Firebird 4.0 Project Teams	173
Appendix A: Licence Notice	174

Chapter 1. General Notes

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

ODS 13 is introduced and it's a major ODS upgrade, so older databases cannot be opened with a Firebird 4 server. The engine library is named `engine13.dll` (Windows) and `libEngine13.so` (POSIX). The security database is named `security4.fdb`. Binaries layout and configuration are unchanged from Firebird 3.



That said, you can copy the Firebird engine library from the Firebird 3.0 distribution package (named `engine12.dll` (Windows) and `libEngine12.so` (POSIX), and located inside the `/plugins` sub-directory) to continue working with databases in ODS12 without needing a backup/restore. However, new features introduced with Firebird 4.0 will not be accessible.

Known incompatibilities are detailed in the [Compatibility Issues](#) chapter.

Bug Reporting

Bugs fixed since Firebird 3.0.7 and Firebird 4.0 Release Candidate 1 are listed and described in the [Bugs Fixed](#) chapter.

- 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 has not 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 beta 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](#).

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

Complete In Firebird 4.0 Final Release

Renamed functions `RSA_SIGN` and `RSA_VERIFY`

Functions priorly introduced as `RSA_SIGN` and `RSA_VERIFY` have been renamed to `RSA_SIGN_HASH` and `RSA_VERIFY_HASH` respectively.

Tracker ticket [#6806](#)

Session time zone is available via the system context

The `SYSTEM` namespace of the built-in function `RDB$GET_CONTEXT` has been extended with the `SESSION_TIMEZONE` parameter that returns the time zone of the current session.

Tracker ticket [#6786](#)

More efficient implementation of `SUBSTRING` for UTF-8 character set

Built-in function `SUBSTRING` has been improved to provide better performance when called with *UTF-8* input strings.

Tracker ticket [#6769](#)

Allow replication plugins to refuse the current attachment

The replication plugin API has been extended to allow the plugin to validate the attachment and return an error if it refuses to replicate its changes.

Tracker ticket [#6767](#)

Immediate execution of `DSQL_drop` and `DSQL_unprepare` commands

Delayed execution of `DSQL_drop` and `DSQL_unprepare` options of the `dsql_free_statement()` operation has been replaced with an immediate execution in order to release resource locks faster.

Tracker ticket [#6748](#)

Provide time zone ID constants in public headers

Time zone IDs used by the Firebird engine have been made available in the header files of the public API.

Tracker ticket [#6715](#)

Make `RDB$GET_TRANSACTION_CN` working the same way in Super and Classic

Built-in function `GET_TRANSACTION_CN` has been fixed to return the same result (proper commit

number) for the given transaction number in all architectures. Priorly it might return *NULL* in Classic Server.

Tracker ticket [#6253](#)

Complete In Release Candidate 1

ALTER SESSION RESET statement

New command to reset user session environment to its initial (default) state has been added.

For full details, see [ALTER SESSION RESET Statement](#) in the [Management Statements](#) chapter.

Tracker ticket [CORE-5832](#)

New virtual table RDB\$CONFIG

This table exposes configuration settings actual for the current database.

For full details, see [Virtual table RDB\\$CONFIG](#) in the [Engine](#) chapter.

Tracker ticket [CORE-3708](#)

Report replica mode through `isc_database_info`, `MON$DATABASE` and `SYSTEM` context

The replica state of the database (*none / read-only / read-write*) is now surfaced via the `MON$DATABASE` table and `Attachment::getInfo()` API call. It can also be read using the context variable `REPLICA_MODE` of the `SYSTEM` namespace.

For full details, see [Monitoring](#) in the [Monitoring & Command-line Utilities](#) chapter and [Extensions to various `getInfo\(\)` Methods](#) in the [Changes to the Firebird API and ODS](#) chapter.

Tracker ticket [CORE-6474](#)

Tracing of session management statements

The trace manager has been extended to report also the new category of session management statements, e.g. `ALTER SESSION RESET`.



Trace plugin developers should be prepared to accept the `NULL` transaction inside the `ITracePlugin::trace_dsql_execute()` method, similarly to how it should have been handled for the `trace_dsql_prepare()` method of the same interface.

Tracker ticket [CORE-6469](#)

Ability to retrieve next attachment ID and next statement ID

Counters representing next attachment ID and next statement ID are now surfaced via the `MON$DATABASE` table and `Attachment::getInfo()` API call.

For full details, see [Monitoring](#) in the [Monitoring & Command-line Utilities](#) chapter and [Extensions to various `getInfo\(\)` Methods](#) in the [Changes to the Firebird API and ODS](#) chapter.

Tracker ticket [CORE-6300](#)

SQL standard syntax for timezone offsets

Timezone offset in timestamp/time literal, CAST, SET TIME ZONE and AT TIME ZONE now follows SQL standard syntax only.

Tracker ticket [CORE-6429](#)

No -pidfile option anymore

PIDFile/-pidfile directive/option has been removed from Firebird Super(Server/Classic) systemd unit.

Tracker ticket [CORE-6413](#)

Time zone displacement in configuration

Usage of time zone displacement is now allowed in configuration setting DefaultTimeZone.

Tracker ticket [CORE-6395](#)

Better dependency tracking when installing Firebird on Linux

Presence of tomcrypt & curses libraries is now checked before installing Firebird.

Tracker ticket [CORE-6366](#)

INT128 as a dedicated data type

INT128 data type has been added as explicit basic type for high precision numerics.

Tracker ticket [CORE-6342](#)

API cleanup

Util methods that return interface pointers by legacy handle are replaced with plain C functions.

Tracker ticket [CORE-6320](#)

Ability to update the supported time zones

Now it's possible to update list of time zones (names and ids) without source code recompilation.

Tracker ticket [CORE-6308](#)

Support for nbackup -fixup via Services API

Allow to fixup (nbackup) a database using Services API

Tracker ticket [CORE-5085](#)

Better error reporting for user management commands

Explicit message about missing password is now raised for CREATE [OR ALTER] USER statements.

Tracker ticket [CORE-4841](#)

Improved sorting performance

Sorting performance has been improved for cases when long VARCHARs are involved.

Tracker ticket [CORE-2650](#)

Complete In Beta 2

SET BIND OF "type1" TO "type2" statement

New session-management statement SET BIND defines data type coercion rules between server-side and client-side data types.

For full details, see [SET BIND Statement](#) in the [Management Statements](#) chapter.

Tracker ticket [CORE-6287](#).

SQL-level replication management

ALTER DATABASE and CREATE/ALTER TABLE statements are extended to allow SQL-level management for the replicated table set and current replication state. For details, see [Replication Management](#) in the [Data Definition Language](#) chapter.

Tracker ticket [CORE-6285](#).

FLOAT datatype is now SQL standard compliant

FLOAT(p) definition is changed to represent precision in binary digits (as defined by the SQL specification) rather than in decimal digits as before. For details, see [Standard Compliance for Data Type FLOAT](#) in the [Data Definition Language](#) chapter.

Tracker ticket [CORE-6109](#).

Starting multiple transactions using the same initial transaction snapshot

SET TRANSACTION statement makes it possible to share the same transaction snapshot among multiple transactions (possibly started by different attachments). For details, see [Sharing Transaction Snapshots](#) in the [Data Manipulation Language](#) chapter.

Tracker ticket [CORE-6018](#).

Better transaction control in ISQL

ISQL can now (optionally) remember the transaction parameters of the last started transaction and reuse them for subsequent transactions. For details, see [Keeping Transaction Parameters](#) in the [Utilities](#) chapter.

Tracker ticket [CORE-4933](#).

Lateral derived tables

Support for SQL:2011 compliant lateral derived tables. For details, see [Lateral Derived Tables](#) in the [Data Manipulation Language](#) chapter.

Tracker ticket [CORE-3435](#).

Convenient usage of TIMESTAMP/TIME WITH TIME ZONE when appropriate ICU library is not installed on the client side

In order to work with time zone names introduced with the new data types TIME WITH TIME ZONE

and `TIMESTAMP WITH TIME ZONE`, the Firebird client library provides API extensions that internally use the ICU library. If the ICU library is missing (or has an incorrect version), the time value would be represented in GMT which may be inconvenient.

To provide a better workaround to this issue, the so called *EXTENDED* format of the time zone information has been introduced. It includes both time zone name and its corresponding GMT offset. The GMT offset will be used as a fallback in the case of missing or mismatched ICU library. For details see [SET BIND Statement](#) in the [Management Statements](#) chapter.

Tracker ticket [CORE-6286](#).

Options in user management statements can be specified in arbitrary order

DDL statements `CREATE USER`, `RECREATE USER`, `ALTER USER`, `ALTER CURRENT USER` and `CREATE OR ALTER USER` now allow their options (`PASSWORD`, `FIRSTNAME`, `TAGS`, etc) to be specified in arbitrary order.

Tracker ticket [CORE-6279](#).

Efficient table scans for DBKEY-based range conditions

Range conditions (less-than and more-than) applied to a `RDB$DB_KEY` pseudo-column are now executed using a range table scan instead of a full table scan, thus providing better performance of such queries.

Tracker ticket [CORE-6278](#).

Increased parsing speed of long queries

Stack growth increment inside the SQL parser has been increased to allow less memory reallocations/copies and thus improve the parsing performance for long queries.

Tracker ticket [CORE-6274](#).

API methods to set various names (field, relation, etc.) in the metadata builder

Methods `setField()`, `setRelation()`, `setOwner()`, `setAlias()` have been added to the `IMetadataBuilder` interface of the Firebird API to set up the corresponding values for the given API message.

Tracker ticket [CORE-6268](#).

SUSPEND is prohibited in procedures and EXECUTE BLOCK without RETURNS

If a stored procedure or an `EXECUTE BLOCK` statement misses the `RETURNS` declaration (i.e. it has no output parameters), then the `SUSPEND` statement inside its body is prohibited and error `isc_suspend_without_returns` is raised.

Tracker ticket [CORE-6239](#).

Improve performance when using SRP plugin for authentication

Connections cache has been implemented inside the SRP authentication plugin to improve the performance.

Tracker ticket [CORE-6237](#).

Delivery of key known to the client to any database connection

It makes it possible to run standard utilities (like *gfx*) or service tasks against an encrypted database on remote server in the cases when the database key is known to the client.

Tracker ticket [CORE-6220](#).

Support for specials (inf/nan) when sorting DECFLOAT values

Special values (like *INF/NaN*) have been taken into account when sorting DECFLOAT values, the output order is now consistent with their comparison rules.

Tracker ticket [CORE-6219](#).

Extend trace record for COMMIT/ROLLBACK RETAINING to show old/new transaction IDs

COMMIT/ROLLBACK RETAINING statement preserves the current transaction context but generates a new transaction ID. The trace output has been extended to show this new transaction ID in the COMMIT_RETAINING and ROLLBACK_RETAINING trace events and also show both initial and new transaction IDs in every transaction identifier in the trace records.

Tracker ticket [CORE-6095](#).

Show OS-specific error when entrypoint is not found in dynamic library

When the dynamic library loaded by the Firebird engine misses the required entrypoint, the reported error now includes the OS-specific information.

Tracker ticket [CORE-6069](#).

Change behavior of skipped and repeated wall times within time zones

Within time zones, some wall times do not exist (DST starting) or repeat twice (DST ending). Firebird has been modified to handle these situations accordingly to the ECMAScript standard. For example:

- 1:30 AM on November 5, 2017 in America/New_York is repeated twice (fall backward), but it must be interpreted as 1:30 AM UTC-04 instead of 1:30 AM UTC-05.
- 2:30 AM on March 12, 2017 in America/New_York does not exist, but it must be interpreted as 2:30 AM UTC-05 (equivalent to 3:30 AM UTC-04).

Tracker ticket [CORE-6058](#).

Built-in functions converting binary string to hexadecimal representation and vice versa

Functions HEX_ENCODE and HEX_DECODE have been added to convert between binary strings and their hexadecimal representations. See [HEX_ENCODE\(\)](#) and [HEX_DECODE\(\)](#) for their description.

Tracker ticket [CORE-6049](#).

Ability to see the current state of database encryption

Column MON\$CRYPT_STATE has been added to the table MON\$DATABASE. It has four possible states:

- 0 - not encrypted
- 1 - encrypted

- 2 - decryption is in progress
- 3 - encryption is in progress

Tracker ticket [CORE-6048](#).

DPB properties for DECFLOAT configuration

New DPB items have been added to the API that can be used to set up the DECFLOAT properties for the current attachment. See also [Setting DECFLOAT Properties](#) in the [Management Statements](#) chapter.

Tracker ticket [CORE-6032](#).

Transaction info item `fb_info_tra_snapshot_number` in the API

New TPB item `fb_info_tra_snapshot_number` has been added to the API that returns the snapshot number of the current transaction.

Tracker ticket [CORE-6017](#).

EXECUTE STATEMENT with excess parameters

Input parameters of EXECUTE STATEMENT command may be declared using the EXCESS prefix to indicate that they can be missing in the query text. See [Excess parameters in EXECUTE STATEMENT](#) in the [Data Definition Language](#) chapter for details.

Tracker ticket [CORE-5658](#).

Ability to backup/restore only tables defined via a command line argument (pattern)

New command-line switch `-INCLUDE_DATA` has been added to *gbak*, see [Ability to Backup/Restore Only Specified Tables](#) in the [Utilities](#) chapter.

Tracker ticket [CORE-5538](#).

RECREATE USER statement

New DDL statement RECREATE USER has been added to drop and re-create the specified user in a single step.

Tracker ticket [CORE-4726](#).

Authenticate user in "EXECUTE STATEMENT ON EXTERNAL DATA SOURCE" by hash of the current password

New sample plugin named ExtAuth has been added to the Firebird distribution package. It allows to omit user name and password when calling EXECUTE STATEMENT against a trusted group of servers sharing the same ExtAuth plugin and the key specific for that group. See `/firebird/examples/extauth/INSTALL` for more details.

Tracker ticket [CORE-3983](#).

Extended precision for numerics

Fixed point numerics with precision up to 38 digits are now supported, along with improved intermediate calculations for shorter numerics. For details, see [Increased Precision for NUMERIC](#)

and [DECIMAL Types](#) in the [Data Definition Language](#) chapter.

Complete In Beta 1

Support for international time zones

International time zone support from Firebird 4.0 onward comprises data types, functions and internal algorithms to manage date/time detection, storage and calculations involving international time zones based on UTC (Adriano dos Santos Fernandes).

For full details, see [Support for International Time Zones](#) in the [Engine](#) chapter.

Tracker tickets [CORE-694](#) and [CORE-909](#)

Built-in replication

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

For details, see [Firebird Replication](#) in the [Engine](#) chapter.

Tracker ticket [CORE-2021](#)

New way to capture the database snapshot

Introducing a new methodology for the Firebird engine to capture the snapshots for retaining the consistency of a transaction's view of database state. The new approach enables read consistency to be maintained for the life of a statement in READ COMMITTED transactions and also allows more optimal garbage collection.

The changes are described in more detail in the topic [Commit Order for Capturing the Database Snapshot](#) in the chapter [Changes in the Firebird Engine](#).

Pooling of external connections

The external data source (EDS) subsystem has been augmented by a pool of external connections. The pool retains unused external connections for a period to reduce unnecessary overhead from frequent connections and disconnections by clients using the same connection strings (Vlad Khorsun).

For details, see [Pooling of External Connections](#) in the [Engine](#) chapter.

Tracker ticket [CORE-5990](#)

Physical standby solution

Physical standby solution (incremental restore via nbackup).

The changes are described in more detail in the Utilities chapter in the topic [nBackup: GUID-based Backup and In-Place Merge](#).

Extended length of metadata identifiers

Metadata names longer than 31 bytes: new maximum length of object names is 63 characters.

The changes are described in more detail in the topic [Extended Length for Object Names](#) in the

chapter [Data Definition Language](#).

Configurable time-outs

Timeout periods configurable for statements, transactions and connections.

The changes for statements and connections are described in more detail in the topic [Timeouts at Two levels](#) in the chapter [Changes in the Firebird Engine](#) (Vlad Khorsun).

Tracker tickets [CORE-658](#) and [CORE-985](#)

New DECFLOAT data type

The SQL:2016 standard-compliant high-precision numeric type DECFLOAT is introduced, along with related operational functions. It is described in detail in the topic [Data type DECFLOAT](#) in the chapter [Data Definition Language](#).

Enhanced system privileges

Predefined system roles, administrative permissions.

The changes are described in more detail in the topic [Enhanced System Privileges](#) in the [Security](#) chapter.

See also the [Management Statements](#) chapter for some background about what the new system privileges are intended for.

GRANT ROLE TO ROLE

Granting roles to other roles, described in detail in the topic [Granting a Role to Another Role](#) in the [Security](#) chapter.

User groups

User groups and cumulative permissions are described in detail in the topic [Granting a Role to Another Role](#) in the [Security](#) chapter.

Batch operations in the API

Batch API operations, bulk load optimizations, support for passing BLOBs in-line.

Tracker ticket [CORE-820](#)

For details, see [Support for Batch Insert and Update Operations in the API](#).

Window functions extensions

Extensions to window functions are described in detail in the [Data Manipulation Language](#) chapter in the topics [Frames for Window Functions](#), [Named Windows](#) and [More Window Functions](#).

FILTER Clause for Aggregate Functions

FILTER clause implemented for aggregate functions, see [FILTER Clause for Aggregate Functions](#) in the [Data Manipulation Language](#) chapter.

Tracker ticket [CORE-5768](#)

Enhanced RETURNING clause in DML to enable returning all current field values

Introduces the RETURNING * syntax, and variants, to return a complete set of field values after committing a row that has been inserted, updated or deleted (Adriano dos Santos Fernandes). For details, see [RETURNING * Now Supported](#) in the [Data Manipulation Language](#) chapter.

Tracker ticket [CORE-3808](#)

Built-in functions FIRST_DAY and LAST_DAY

New date/time functions FIRST_DAY and LAST_DAY, see [Two New Date/Time Functions](#) in the [Data Manipulation Language](#) chapter.

Tracker ticket [CORE-5620](#)

Built-in Cryptographic functions

New security-related functions, including eight cryptographic ones, see [Built-in Cryptographic Functions](#) in the [Security](#) chapter.

Tracker ticket [CORE-5970](#)

Monitoring Compression and Encryption Status of Attachments

Compression and encryption status of a connection are now available in the monitoring table MON\$ATTACHMENTS:

- MON\$WIRE_COMPRESSED (wire compression enabled = 1, disabled = 0)
- MON\$WIRE_ENCRYPTED (wire encryption enabled = 1, disabled = 0)

Tracker ticket [CORE-5536](#)

Improve performance of *gbak restore*

The new Batch API was used to improve the performance of *gbak restore*, including parallel operations.

Tracker tickets [CORE-2992](#) and [CORE-5952](#)

Backup and Restore with Encryption

Support for backing up and restoring encrypted databases using the crypt and keyholder plugins — see [Backup and Restore with Encryption](#) in the [Utilities](#) chapter.

Also available is compression and decompression of both encrypted and non-encrypted backups.

Compatibility with Older Versions

Notes about compatibility with older Firebird versions are collated in the “[Compatibility Issues](#)” chapter.

Chapter 3. Changes in the Firebird Engine

The Firebird engine, version 4, presents no radical changes in architecture or operation. Improvements and enhancements continue, including a doubling of the maximum database page size and the long-awaited ability to impose timeouts on connections and statements that could be troublesome, primary-replica replication and international time zone support.

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

Maximum Page Size Increased To 32KB

Dmitry Yemanov

Tracker ticket [CORE-2192](#)

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

External Functions (UDFs) Feature Deprecated

The original design of external functions (UDF) support has always been a source of security problems. The most dangerous security holes, that occurred when UDFs and external tables were used simultaneously, were fixed as far back as Firebird 1.5. Nevertheless, UDFs have continued to present vulnerability issues like server crashes and the potential to execute arbitrary code.

The use of UDFs has been aggressively deprecated in Firebird 4:

- The default setting for the configuration parameter `UdfAccess` is `NONE`. In order to run UDFs at all will now require an explicit configuration of `Restrict UDF`
- The UDF libraries (`ib_udf`, `fbudf`) are no longer distributed in the installation kits
- Most of the functions in the libraries previously distributed in the shared (dynamic) libraries `ib_udf` and `fbudf` had already been replaced with built-in functional analogs. A few remaining UDFs have been replaced with either analog routines in a new library of UDRs named `udf_compat` or converted to stored functions.

Refer to [Deprecation of External Functions \(UDFs\)](#) in the [Compatibility](#) chapter for details and instructions about upgrading to use the safe functions.

- Replacement of UDFs with UDRs or stored functions is strongly recommended

Support for International Time Zones

Adriano dos Santos Fernandes

Tracker tickets [CORE-909](#) and [CORE-694](#)

Time zone support from Firebird 4.0 onward consists of

- data types `TIME WITH TIME ZONE` and `TIMESTAMP WITH TIME ZONE`; implicitly also `TIME WITHOUT TIME ZONE` and `TIMESTAMP WITHOUT TIME ZONE` as aliases for the existing types `TIME` and `TIMESTAMP`
- expressions and statements to work with time zones
- conversion between data types without/with time zones



The data types `TIME WITHOUT TIME ZONE`, `TIMESTAMP WITHOUT TIME ZONE` and `DATE` are defined to use the *session time zone* when converting from or to a `TIME WITH TIME ZONE` or `TIMESTAMP WITH TIME ZONE`. `TIME` and `TIMESTAMP` are synonymous to their respective `WITHOUT TIME ZONE` data types.

Session Time Zone

As the name implies, the session time zone, can be different for each database attachment. It can be set explicitly in the DPB or SPB with the item `isc_dpb_session_time_zone`; otherwise, by default, it uses the same time zone as the operating system of the Firebird server process. This default can be overridden in `firebird.conf`, see [DefaultTimeZone setting](#) in the [Configuration Additions and Changes](#) chapter.

Subsequently, the time zone can be changed to a given time zone using a `SET TIME ZONE` statement or reset to its original value with `SET TIME ZONE LOCAL`.

Time Zone Format

A time zone is a string, either a time zone region (for example, `'America/Sao_Paulo'`) or a displacement from GMT in hours:minutes (for example, `'-03:00'`).

A time/timestamp with time zone is considered equal to another time/timestamp with time zone if their conversions to UTC are equivalent. For example, time `'10:00 -02:00'` and time `'09:00 -03:00'` are equivalent, since both are the same as time `'12:00 GMT'`.



The same equivalence applies in `UNIQUE` constraints and for sorting purposes.

Data Types for Time Zone Support

The syntax for declaring the data types `TIMESTAMP` and `TIME` has been extended to include arguments defining whether the field should be defined with or without time zone adjustments, i.e.,

```
TIME [ { WITHOUT | WITH } TIME ZONE ]
```

```
TIMESTAMP [ { WITHOUT | WITH } TIME ZONE ]
```

The default for both `TIME` and `TIMESTAMP` is `WITHOUT TIME ZONE`. For more details, see [Data Type Extensions for Time Zone Support](#) in the [Data Definition Language](#) chapter.

API Support for Time Zones

- Structures (structs)
- Functions

Time Zone Statements and Expressions

Additions and enhancements to syntax in DDL and DML are listed in this section. Follow the links indicated to the details in the DDL and DML chapters.

Statement **SET TIME ZONE**

Changes the session time zone

Expression **AT**

Translates a time/timestamp value to its corresponding value in another time zone

Expression **EXTRACT**

Two new arguments have been added to the **EXTRACT** expression: **TIMEZONE_HOUR** and **TIMEZONE_MINUTE** to extract the time zone hours displacement and time zone minutes displacement, respectively.

Expression **LOCALTIME**

Returns the current time as a **TIME WITHOUT TIME ZONE**, i.e., in the session time zone

Expression **LOCALTIMESTAMP**

Returns the current timestamp as a **TIMESTAMP WITHOUT TIME ZONE**, i.e., in the session time zone

Expressions **CURRENT_TIME** and **CURRENT_TIMESTAMP**

In version 4.0, **CURRENT_TIME** and **CURRENT_TIMESTAMP** now return **TIME WITH TIME ZONE** and **TIMESTAMP WITH TIME ZONE**, with the time zone set by the session time zone

Virtual table **RDB\$TIME_ZONES**

A virtual table listing time zones supported in the engine. Columns:

- **RDB\$TIME_ZONE_ID** type **INTEGER**
- **RDB\$TIME_ZONE_NAME** type **CHAR(63)**

Package **RDB\$TIME_ZONE_UTIL**

A package of time zone utility functions and procedures:

Function **DATABASE_VERSION**

RDB\$TIME_ZONE_UTIL.DATABASE_VERSION returns the version of the time zone database as a **VARCHAR(10) CHARACTER SET ASCII**.

Example

```
select rdb$time_zone_util.database_version() from rdb$database;
```

Returns:

```
DATABASE_VERSION
=====
2020d
```

Procedure TRANSITIONS

RDB\$TIME_ZONE_UTIL.TRANSITIONS returns the set of rules between the start and end timestamps.

The input parameters are:

- RDB\$TIME_ZONE_NAME type CHAR(63)
- RDB\$FROM_TIMESTAMP type TIMESTAMP WITH TIME ZONE
- RDB\$TO_TIMESTAMP type TIMESTAMP WITH TIME ZONE

Output parameters:

RDB\$START_TIMESTAMP

type TIMESTAMP WITH TIME ZONE — The start timestamp of the transition

RDB\$END_TIMESTAMP

type TIMESTAMP WITH TIME ZONE — The end timestamp of the transition

RDB\$ZONE_OFFSET

type SMALLINT — The zone's offset, in minutes

RDB\$DST_OFFSET

type SMALLINT — The zone's DST offset, in minutes

RDB\$EFFECTIVE_OFFSET

type SMALLINT — Effective offset (ZONE_OFFSET + DST_OFFSET)

Example

```
select *
  from rdb$time_zone_util.transitions(
    'America/Sao_Paulo',
    timestamp '2017-01-01',
    timestamp '2019-01-01');
```

Returns (RDB\$ prefix left off for brevity):

START_TIMESTAMP	END_TIMESTAMP	ZONE_OFFSET	DST_OFFSET
2016-10-16 03:00:00.0000 GMT	2017-02-19 01:59:59.9999 GMT	-180	60
2017-02-19 02:00:00.0000 GMT	2017-10-15 02:59:59.9999 GMT	-180	0
2017-10-15 03:00:00.0000 GMT	2018-02-18 01:59:59.9999 GMT	-180	60
2018-02-18 02:00:00.0000 GMT	2018-10-21 02:59:59.9999 GMT	-180	0
2018-10-21 03:00:00.0000 GMT	2019-02-17 01:59:59.9999 GMT	-180	60

Updating the Time Zone Database

Time zones are often changed: of course, when it happens, it is desirable to update the time zone database as soon as possible.

Firebird stores WITH TIME ZONE values translated to UTC time. Suppose a value is created with one time zone database and a later update of that database changes the information in the range of our stored value. When that value is read, it will be returned as different to the value that was stored initially.

Firebird uses the [IANA time zone database](#) through the ICU library. The ICU library presented in the Firebird kit (Windows), or installed in a POSIX operating system, can sometimes have an outdated time zone database.

An updated database can be found on [this page on the FirebirdSQL GitHub](#). Filename `le.zip` stands for little-endian and is the necessary file for most computer architectures (Intel/AMD compatible x86 or x64), while `be.zip` stands for big-endian architectures and is necessary mostly for RISC computer architectures. The content of the zip file must be extracted in the `/tzdata` sub-directory of the Firebird installation, overwriting existing `*.res` files belonging to the database.



`/tzdata` is the default directory where Firebird looks for the time zone database. It can be overridden with the `ICU_TIMEZONE_FILES_DIR` environment variable.

Firebird Replication

Dmitry Yemanov; Roman Simakov

Tracker ticket [CORE-2021](#)

Firebird 4 introduces built-in support for uni-directional (“primary-replica”) logical replication. Logical here means record-level replication, as opposed to physical (page-level) replication. Implementation is primarily directed towards providing for high availability, but it can be used for other tasks as well.

Events that are tracked for replication include

- inserted/updated/deleted records
- sequence changes
- DDL statements

Replication is transactional and commit order is preserved. Replication can track changes either in all tables, or in a customized subset of tables. Any table that is to be replicated must have a primary key or, at least, a unique key.

Replication Modes

Both *synchronous* and *asynchronous* modes are available.

Synchronous Mode

In synchronous replication, the primary (master) database is permanently connected to the replica (slave) database(s) and changes are replicated immediately. Effectively the databases are in sync after every commit, which could have an impact on performance due to additional network traffic and round-trips.



Although some recent uncommitted changes may be buffered, they are not transmitted until committed.

More than one synchronous replica can be configured, if necessary.

Asynchronous Mode

In asynchronous replication, changes are written into local journal files that are transferred over the wire and applied to the replica database. The impact on performance is much lower, but imposes a delay — *replication lag* — while changes wait to be applied to the replica database; i.e. the replica database is always “catching up” the master database.

Access Modes

There are two access modes for replica databases: *read-only* and *read-write*.

- With a read-only replica, only queries that do not modify data are allowed. Modifications are limited to the replication process only.



Global temporary tables can be modified, as they are not replicated.

- A read-write replica allows execution of any query. In this access mode, potential conflicts must be resolved by users or database administrators.

Journaling

Asynchronous replication is implemented with journaling. Replicated changes are written into the journal which consists of multiple files, known as *replication segments*. The Firebird server writes

segments continuously, one after another. Every segment has a unique number which is generated sequentially. This number, known as a *segment sequence*, is combined with the database UUID to provide globally unique identification of journal segments. The global sequence counter is stored inside the replicated database and is reset only when the database is restored from backup.

Segments are rotated regularly, a process that is controlled by either *maximum segment size* or *timeout*. Both thresholds are configurable. Once the active segment reaches the threshold, it is marked as “full” and writing switches to the next available segment.

Full segments are archived and then reused for subsequent writes. Archiving consists of copying the segment in preparation for transferring it to the replica host and applying it there. Copying can be done by the Firebird server itself or, alternatively, by a user-specified custom command.

On the replica side, journal segments are applied in the replication sequence order. The Firebird server periodically scans for new segments appearing in the configured directory. Once the next segment is found, it gets replicated. For each replication source, the replication state is stored in a local file named for the UUID and the replication source. It contains markers for

- latest segment sequence (LSS)
- oldest segment sequence (OSS)
- a list of active transactions started between the OSS and the LSS

About the LSS and OSS

LSS refers to the last replicated segment. OSS refers to the segment that started the earliest transaction that was incomplete at the time LSS was processed.

These markers control two things:

1. what segment must be replicated next and
2. when segment files can be safely deleted

Segments with numbers between the OSS and the LSS are preserved in case the journal needs replaying after the replicator disconnects from the replica database; for example, due to a replication error or an idle timeout.

If there are no active transactions pending and the LSS was processed without errors, all segments up to and including the LSS are deleted.

If a critical error occurs, replication is temporarily suspended and will be retried after the timeout.

Error Reporting

All replication errors and warnings (such as detected conflicts) are written into the `replication.log` file. It may also include detailed descriptions of the operations performed by the replicator.



Log file location

The `replication.log` file is stored in the *Firebird log directory*. By default, the Firebird log directory is the root directory of the Firebird installation.

Setting Up Replication

Setup involves tasks on both the primary and replica sides.

Setting Up the Primary Side

Replication is configured using a single configuration file, `replication.conf`, on the host serving the primary database. Both global and per-database settings are possible within the same file. The available options are listed inside `replication.conf`, along with commented descriptions of each.



Per-database configurations

When configuring options at per-database level, the full database path must be specified within the `{database}` section. Aliases and wildcards are not accepted.

Inside the database, replication should be enabled using the following DDL statement:

```
ALTER DATABASE ENABLE PUBLICATION
```

Defining a Custom Replication Set

Optionally, the replication set (aka publication) should be defined. It includes tables that should be replicated. This is done using the following DDL statements:

```
-- to replicate all tables (including the ones created later)
ALTER DATABASE INCLUDE ALL TO PUBLICATION

-- to replicate specific tables
ALTER DATABASE INCLUDE TABLE T1, T2, T3 TO PUBLICATION
```

Tables may later be excluded from the replication set:

```
-- to disable replication of all tables (including the ones created later)
ALTER DATABASE EXCLUDE ALL FROM PUBLICATION

-- to disable replication of specific tables
ALTER DATABASE EXCLUDE TABLE T1, T2, T3 FROM PUBLICATION
```

Tables enabled for replication inside the database can be additionally filtered using two settings in `replication.conf`: `include_filter` and `exclude_filter`. They are regular expressions that are applied to table names, defining the rules for including or excluding them from the replication set. The regular expression syntax used to match table names is the same as in `SIMILAR TO` Boolean expressions.

Synchronous/Asynchronous Modes

Synchronous Mode

Synchronous replication can be turned on by setting the `sync_replica` specifying a connection string to the replica database, prefixed with username and password. Multiple entries are allowed.

In the SuperServer and SuperClassic architectures, the replica database is attached internally when the first user gets connected to the primary database and is detached when the last user disconnects from the primary database.

In the Classic Server architecture, each server process keeps its own active connection to the replica database.

Asynchronous Mode

For asynchronous replication the journaling mechanism must be set up. The primary parameter is `log_directory` which defines location of the replication journal. Specifying this location turns on asynchronous replication and tells the Firebird server to start producing the journal segments.

A Minimal Configuration

A minimal primary-side configuration would look like this:

```
database = /data/mydb.fdb
{
    log_directory = /dblogs/mydb/
    log_archive_directory = /shiplogs/mydb/
}
```

Archiving is performed by the Firebird server copying the segments from `/dblogs/mydb/` to `/shiplogs/mydb/`.

The same setup, but with user-defined archiving:

```
database = /data/mydb.fdb
{
    log_directory = /dblogs/mydb/
    log_archive_directory = /shiplogs/mydb/
    log_archive_command = "test ! -f $(archpathname) && cp $(logpathname)
$(archpathname)"
}
```

—where `$(logpathname)` and `$(archpathname)` are built-in macros that are expanded to full path names when running the specified custom shell command.

About custom archiving

Custom archiving, through use of the setting `log_archive_command` allows use of any system shell command, including scripts or batch files, to deliver segments to the replica side. It could use compression, FTP, or whatever else is available on the server.



The actual transport implementation is up to the DBA: Firebird just produces segments on the primary side and expects them to appear at the replica side. If the replica storage can be remotely attached to the primary host, it becomes just a matter of copying the segment files. In other cases, some transport solution is required.

If custom archiving is used, the setting `log_archive_directory` can be omitted, unless `log_archive_command` mentions the `$(archpathname)` macro.

The same setup, with archiving performed every 10 seconds:

```
database = /data/mydb.fdb
{
    log_directory = /dblogs/mydb/
    log_archive_directory = /shiplogs/mydb/
    log_archive_command = "test ! -f $(archpathname) && cp $(logpathname)
$(archpathname)"
    log_archive_timeout = 10
}
```

Read `replication.conf` for other possible settings.

Applying the Primary Side Settings

To take into effect changes applied to the primary-side settings, all users connected to a database must be disconnected (or a database must be shutdown). After that, all users connected again would use an updated configuration.

Setting Up the Replica Side

`replication.conf` file is also used for setting up the replica side. Setting the parameter `log_source_directory` specifies the location that the Firebird server scans for the transmitted segments. In addition, the DBA may specify explicitly which source database is accepted for replication, by setting the parameter `source_guid`.

A Sample Replica Setup

A configuration for a replica could look like this:

```

database = /data/mydb.fdb
{
  log_source_directory = /incominglogs/
  source_guid = {6F9619FF-8B86-D011-B42D-00CF4FC964FF}
}

```

Read `replication.conf` for other possible settings.

Applying the Replica Side Settings

To take into effect changes applied to replica-side settings, the Firebird server must be restarted.

Creating a Replica Database

Task 1 — Make the initial replica

Any physical copying method can be used to create an initial replica of the primary database:

- File-level copy while the Firebird server is shut down
- `ALTER DATABASE BEGIN BACKUP + file-level copy + ALTER DATABASE END BACKUP`
- `nbackup -l + file-level copy + nbackup -n`
- `nbackup -b 0 + nbackup -f -seq`

Task 2 — Activate the *replica* access mode

Activating the access mode—for the copied database involves the command-line utility *gfix* with the new `-replica` switch and either `read-only` or `read-write` as the argument:

- To set the database copy as a read-only replica

```
gfix -replica read-only <database>
```

If the replica is read-only then only the replicator connection can modify the database. This is mostly intended for high-availability solutions, as the replica database is guaranteed to match the primary one and can be used for fast recovery. Regular user connections may perform any operations allowed for read-only transactions: select from tables, execute read-only procedures, write into global temporary tables, etc. Database maintenance such as sweeping, shutdown, monitoring is also allowed.

A read-only replica can be useful for distributing read-only load, for example, analytics, away from the master database.



Read-only connections have the potential to conflict with replication if DDL statements that are performed on the master database are of the kind that requires an exclusive lock on metadata.

- To set the database copy as a read-write replica

```
gfix -replica read-write <database>
```

Read-write replicas allow both the replicator connection and regular user connections to modify the database concurrently. With this mode, there is no guarantee that the replica database will be in sync with the master one. Therefore, use of a read-write replica for high availability conditions is not recommended unless user connections on the replica side are limited to modifying only tables that are excluded from replication.

Task 3 — Converting the replica to a regular database

A third `gfix -replica` argument is available for “switching off” replication to a read-write replica when conditions call for replication flow to be discontinued for some reason. Typically, it would be used to promote the replica to become the primary database after a failure; or to make physical backup copies from the replica.

```
gfix -replica none <database>
```

Pooling of External Connections

Vlad Khorsun

Tracker ticket [CORE-5990](#)

To avoid delays when external connections are being established frequently, the external data source (EDS) subsystem has been augmented by a pool of external connections. The pool retains unused external connections for a period to reduce unnecessary overhead from frequent connections and disconnections by clients using the same connection strings.

Key Characteristics of Connection Pooling

The implementation of connection pooling in Firebird 4 eliminates the problem of interminable external connections by controlling and limiting the number of idle connections. The same pool is used for all external connections to all databases and all local connections handled by a given Firebird process. It supports a quick search of all pooled connections using four parameters, described below in [New Connections](#).

Terminology

Two terms recur in the management of the connection pool, in configuration, by DDL ALTER statements during run-time and in new context variables in the SYSTEM namespace:

Connection life time

The time interval allowed from the moment of the last usage of a connection to the moment after which it will be forcibly closed. SQL parameter LIFETIME, configuration parameter ExtConnPoolLifeTime, context variable EXT_CONN_POOL_LIFETIME.

Pool size

The maximum allowed number of idle connections in the pool. SQL parameter SIZE, configuration parameter ExtConnPoolSize, context variable EXT_CONN_POOL_SIZE.

How the Connection Pool Works

Every successful connection is associated with a pool, which maintains two lists—one for idle connections and one for active connections. When a connection in the “active” list has no active requests and no active transactions, it is assumed to be “unused”. A reset of the unused connection is attempted using an ALTER SESSION RESET statement and,

- if the reset succeeds (no errors occur) the connection is moved into the “idle” list;
- if the reset fails, the connection is closed;
- if the pool has reached its maximum size, the oldest idle connection is closed.
- When the *lifetime* of an idle connection expires, it is deleted from the pool and closed.

New Connections

When the engine is asked to create a new external connection, the pool first looks for a candidate in the “idle” list. The search, which is case-sensitive, involves four parameters:

1. connection string
2. username
3. password
4. role

If suitable connection is found, it is tested to check that it is still alive.

- If it fails the check, it is deleted and the search is repeated, without reporting any error to the client
- Otherwise, the live connection is moved from the “idle” list to the “active” list and returned to the caller
- If there are multiple suitable connections, the most recently used one is chosen
- If there is no suitable connection, a new one is created and added to the “active” list.

Managing the Connection Pool

A new SQL statement has been introduced to manage the pool during run-time from any connection, between Firebird restarts, i.e., changes made with `ALTER EXTERNAL CONNECTIONS POOL` are not persistent.

This is the syntax pattern:

```
ALTER EXTERNAL CONNECTIONS POOL { <parameter variants> }
```

Syntax Variants Available

```
ALTER EXTERNAL CONNECTIONS POOL SET SIZE <int>
```

Sets the maximum number of idle connections

```
ALTER EXTERNAL CONNECTIONS POOL SET LIFETIME <int> <time_part>
```

Sets the lifetime of an idle connection, from 1 second to 24 hours. The `<time_part>` can be `SECOND` | `MINUTE` | `HOUR`.

```
ALTER EXTERNAL CONNECTIONS POOL CLEAR ALL
```

Closes all idle connections and instigates dissociation of all active connections so they are immediately closed when they become unused

```
ALTER EXTERNAL CONNECTIONS POOL CLEAR OLDEST
```

Closes expired idle connections

For a full descriptions and examples of the variants, see [ALTER EXTERNAL CONNECTIONS POOL Statement](#) in the chapter [Management Statements](#).

Querying the Connection Pool

The state of the external connections pool can be queried using a set of new context variables in the 'SYSTEM' namespace:

<code>EXT_CONN_POOL_SIZE</code>	Pool size
<code>EXT_CONN_POOL_LIFETIME</code>	Idle connection lifetime, in seconds
<code>EXT_CONN_POOL_IDLE_COUNT</code>	Count of currently inactive connections
<code>EXT_CONN_POOL_ACTIVE_COUNT</code>	Count of active connections associated with the pool

Parameters for Configuring the Connection Pool

Two new parameters, for `firebird.conf` only, are for configuring the connection pool at process start. Follow the links for details.

ExtConnPoolSize

Configures the maximum number of idle connections allowed in the pool

ExtConnPoolLifetime

Configures the number of seconds a connection should stay available after it has gone idle

Timeouts at Two levels

Vlad 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. A database administrator can use it to enforce closure of old connections that have become inactive, to reduce unnecessary consumption of resources. It can 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 and reset to zero
- If the maximum idle time is exceeded, the engine immediately closes the connection in the same way as with asynchronous 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.

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



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.



Take note of the difference between the time units at each level. At database level, in the conf file, the unit for `SessionTimeout` is minutes. In SQL, the default unit is minutes but can be expressed in hours or seconds explicitly. At the API level, the unit is seconds.

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 allows execution of a statement to be stopped automatically when it has been running longer than a given timeout period. It gives the database administrator an instrument for limiting excessive resource consumption from heavy queries.

Statement timeouts can also 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.



A 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



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`. `StatementTimeout` is 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.



Take note of the difference between the time units at each level. At database level, in the conf file, the unit for StatementTimeout is seconds. In SQL, the default unit is seconds but can be expressed in hours, minutes or milliseconds explicitly. At the API level, the unit is milliseconds.

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 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 the 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_dsqli_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 function 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_dsqli_info()` API function 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_dsqli_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 Statement-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-value
```

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

Commit Order for Capturing the Database Snapshot

Nickolay Samofatov; Roman Simakov; Vlad Khorsun

Tracker ticket [CORE-5953](#)

Traditionally, a SNAPSHOT (“concurrency”) transaction takes a private copy of the transaction inventory page (TIP) at its start and uses it to refer to the state of the latest committed versions of all records in the database, right up until it commits or rolls back its own changes. Thus, by definition, a SNAPSHOT transaction sees the database state only as it was at the moment it started.

In the traditional model, a READ COMMITTED transaction does not use a stable snapshot view of database state and does not keep a private copy of the TIP. Instead, it asks the TIP for the most recent state of a record committed by another transaction. In Super (“SuperServer”) mode, the TIP cache is shared to provide optimal access to it by READ COMMITTED transactions.

The 'Commit Order' Approach

Firebird 4 takes a new approach to establishing a consistent view of the database state visible to running transactions. This new approach uses the concept of *commit order*.

It is sufficient to know the *order of commits* in order to capture the state of any transaction at the moment when a snapshot is created.

Commit Order for Transactions

The elements for establishing and utilising commit order are:

- Initialize a *Commit Number (CN)* for each database when the database is first opened
- Each time a transaction is committed, the Commit Number for that database is incremented and the new CN is associated with the specific transaction
- This specific transaction and commit number combination — “transaction CN” are stored in memory and can be queried subsequently while the database remains active
- A *database snapshot* is identified by the value stored for the global CN at moment when the database snapshot was created

Special Values for the Transaction CN

Possible values for the transaction Commit Number include some special CN values that signify whether the transaction is active or dead, viz.:

CN_ACTIVE = 0

Transaction is active

CN_PREHISTORIC = 1

Transaction was committed before the database started (i.e., older than OIT)

CN_PREHISTORIC < CN < CN_DEAD

Transaction was committed while the database was working

CN_DEAD = MAX_TRA_NUM - 2

Dead transaction

CN_LIMBO = MAX_TRA_NUM - 1

Transaction is in limbo

The Rule for Record Visibility

Supposing *database snapshot* is the current snapshot in use by the current transaction and *other transaction* is the transaction that created the given record version, the rule for determining the visibility of the record version works like this:

- If the state of *other transaction* is 'active', 'dead' or 'in limbo' then the given record version is not visible to the current transaction
- If the state of *other transaction* is 'committed' then the visibility of the given record version depends on the timing of the creation of *database snapshot*, so
 - if it was committed before *database snapshot* was created, it is visible to the current transaction;
 - if it was committed after *database snapshot* was created, it is not visible to the current transaction.

Thus, as long as a maintained list of all known transactions with their associated Commit Numbers is in existence, it is enough to compare the CN of *other transaction* with the CN of *database snapshot* to decide whether the given record version should be visible within the scope of *database snapshot*.



The status of an association between a transaction and its CN can be queried using a new built-in function, [RDB\\$GET_TRANSACTION_CN](#).

SNAPSHOT transactions now use the *database snapshot* described above. Instead of taking a private copy of TIP when started it just remembers value of global Commit Number at that moment.

Implementation details

The list of all known transactions with associated Commit Numbers is maintained in shared memory. It is implemented as an array whose index is a transaction ID and its item value is the corresponding Commit Number.

The whole array is split into fixed-size blocks containing the CN's for all transactions between the OIT and Next Transaction markers. When Next Transaction moves out of the scope of the highest block, a new block is allocated. An old block is released when the OIT moves out of the scope of the lowest block.

Block Size

The default size of a TIP cache block is 4MB, providing capacity for 512 * 1024 transactions. It is

configurable in `firebird.conf` and `databases.conf` using the new parameter `TipCacheBlockSize`.

Read Consistency for Statements in Read-Committed Transactions

The existing implementation of READ COMMITTED isolation for transactions suffers from an important problem: a single statement, such as a SELECT, could see different views of the same data during execution.

For example, imagine two concurrent transactions, where the first inserts 1000 rows and commits, while the second runs `SELECT COUNT(*)` over the same table.

If the isolation level of the second transaction is READ COMMITTED, its result is hard to predict. It could be any of:

1. the number of rows in the table before the first transaction started, or
2. the number of rows in the table after the first transaction committed, or
3. any number between those two numbers.

Which of those results is actually returned depends on how the two transactions interact:

- CASE 1 would occur if the second transaction finished counting before the first transaction was committed, since the uncommitted inserts at that point are visible only to the first transaction.
- CASE 2 would occur if the second transaction started after the first had committed all of the inserts.
- CASE 3 occurs in any other combination of the conditions: the second transaction sees some, but not all, of the inserts during the commit sequence of the first transaction.

CASE 3 is the problem referred to as *inconsistent read at the statement level*. It matters because, by definition, each *statement* in a READ COMMITTED transaction has its own distinct view of database state. In the existing implementation, the statement's view is not certain to remain stable for the duration of its execution: it could change between the start of execution and the completion.

Statements running in a SNAPSHOT transaction do not have this problem, since every statement runs against a consistent view of database state. Also, different statements that run within the same READ COMMITTED transaction could see different views of database state but this is “as designed” and is not a source of statement-level inconsistency.

Solving the Inconsistent Read Problem

See Tracker ticket [CORE-5954](#).

The obvious solution to the inconsistent read problem is to have the read-committed transaction use a stable database snapshot during execution of a statement. Each new top-level statement creates its own database snapshot that sees the most recently committed data. With snapshots based on commit order, this is a very cheap operation. Let this snapshot be called a *statement-level snapshot* for further references. Nested statements (triggers, nested stored procedures and functions, dynamic statements, etc.) use the same statement-level snapshot that was created by the top-level statement.

New Isolation Sub-Level for READ COMMITTED

A new sub-level for transactions in *READ COMMITTED* isolation is introduced: *READ COMMITTED READ CONSISTENCY*.

The existing sub-levels for *READ COMMITTED* isolation, namely *RECORD VERSION* and *NO RECORD VERSION*, are still supported and operate as before (without using statement-level snapshots), but they are now deprecated and may be removed in future Firebird versions.

In summary, the three variants for transactions in *READ COMMITTED* isolation are now:

- *READ COMMITTED READ CONSISTENCY*
- *READ COMMITTED NO RECORD VERSION*
- *READ COMMITTED RECORD VERSION*

Handling of Update Conflicts

When a statement executes in a *READ COMMITTED READ CONSISTENCY* transaction, its database view is retained in a fashion similar to a *SNAPSHOT* transaction. This makes it pointless to wait for the concurrent transaction to commit, in the hope of being able to read the newly-committed record version. So, when a *READ COMMITTED READ CONSISTENCY* transaction reads data, it behaves similarly to *READ COMMITTED RECORD VERSION* transaction: walks the back versions chain looking for a record version visible to the current snapshot.

When an update conflict occurs, the behaviour of a *READ COMMITTED READ CONSISTENCY* transaction is different to that of one in *READ COMMITTED RECORD VERSION*. The following actions are performed:

1. Transaction isolation mode is temporarily switched to *READ COMMITTED NO RECORD VERSION*.
2. A write-lock is taken for the conflicting record.
3. Remaining records of the current *UPDATE/DELETE* cursor are processed, and they are write-locked too.
4. Once the cursor is fetched, all modifications performed since the top-level statement was started are undone, already taken write-locks for every updated/deleted/locked record are preserved, all inserted records are removed.
5. Transaction isolation mode is restored to *READ COMMITTED READ CONSISTENCY*, new statement-level snapshot is created, and the top-level statement is restarted.

This algorithm ensures that already updated records remain locked after restart, they are visible to the new snapshot, and could be updated again with no further conflicts. Also, due to *READ CONSISTENCY* nature, the modified record set remains consistent.

Notes

- This restart algorithm is applied to UPDATE, DELETE, SELECT WITH LOCK and MERGE statements, with or without the RETURNING clause, executed directly by a client application or inside some PSQL object (stored procedure/function, trigger, EXECUTE BLOCK, etc).
- If an UPDATE/DELETE statement is positioned on some explicit cursor (using the WHERE CURRENT OF clause), then the step (3) above is skipped, i.e. remaining cursor records are not fetched and write-locked.
- If the top-level statement is selectable and update conflict happens after one or more records were returned to the client side, then an update conflict error is reported as usual and restart is not initiated.
- Restart does not happen for statements executed inside autonomous blocks (IN AUTONOMOUS TRANSACTION DO ...).
- After 10 unsuccessful attempts the restart algorithm is aborted, all write locks are released, transaction isolation mode is restored to *READ COMMITTED* *READ CONSISTENCY*, and an update conflict error is raised.
- Any error not handled at step (3) above aborts the restart algorithm and statement execution continues normally.
- UPDATE/DELETE triggers fire multiple times for the same record if the statement execution was restarted and record is updated/deleted again.
- Statement restart is usually fully transparent to client applications and no special actions should be taken by developers to handle it in any way. The only exception is the code with side effects that are outside the transactional control, for example:
 - usage of external tables, sequences or context variables
 - sending e-mails using UDF
 - usage of autonomous transactions or external queries
 and so on. Beware that such code could be executed more than once if update conflict happens.
- There is no way to detect whether a restart happened, but it could be done manually using code with side effects as described above, for example using a context variable.
- Due to historical reasons, error *isc_update_conflict* is reported as the secondary error code, with the primary error code being *isc_deadlock*.

**Read Committed Read-Only Transactions**

In the existing implementation, *READ COMMITTED* transactions in *READ ONLY* mode are marked as committed when the transaction starts. This provides a benefit in that record versions in such transactions are never “interesting”, thus not inhibiting the regular garbage collection and not delaying the advance of the OST marker.

READ CONSISTENCY READ ONLY transactions are still started as pre-committed, but in order to avoid the regular garbage collection breaking future statement-level snapshots, it delays the advance of the OST marker in the same way as it happens for *SNAPSHOT* transactions.



This delays only the *regular* (traditional) garbage collection, the *intermediate* garbage collection (see below) is not affected.

Syntax and Configuration

Support for the new *READ COMMITTED READ CONSISTENCY* isolation level is found in SQL syntax, in the API and in configuration settings.

Where `SET TRANSACTION` is available in SQL, the new isolation sub-level is set as follows:

```
SET TRANSACTION READ COMMITTED READ CONSISTENCY
```

To start a *READ COMMITTED READ CONSISTENCY* transaction via the ISC API, use the new constant `isc_tpb_read_consistency` in the Transaction Parameter Buffer.

Starting with Firebird 4, usage of the legacy *READ COMMITTED* modes (*RECORD VERSION* and *NO RECORD VERSION*) is discouraged and *READ CONSISTENCY* mode is recommended to be used instead. For now, existing applications can be tested with the new *READ COMMITTED READ CONSISTENCY* isolation level by setting the new configuration parameter [ReadConsistency](#) described in the Configuration Additions and Changes chapter.



Please pay attention that the `ReadConsistency` configuration setting is enabled by default, thus forcing all *READ COMMITTED* transactions to be executed in the *READ CONSISTENCY* mode. Consider disabling this setting if the legacy behaviour of *READ COMMITTED* transactions must be preserved.

Garbage Collection

The *record version visibility rule* provides the following logic for identifying record versions as garbage:

- If snapshot *CN* can see some record version (*RV_X*) then all snapshots with numbers greater than *CN* can also see *RV_X*.
- If all existing snapshots can see *RV_X* then all its back-versions can be removed, OR
- If the oldest active snapshot can see *RV_X* then all its back-versions can be removed.

The last part of the rule reproduces the legacy rule, whereby all record versions at the tail of the versions chain start from some “mature” record version. The rule allows that mature record version to be identified so that the whole tail after it can be cut.

However, with snapshots based on commit-order, version chains can be further shortened because it enables some record versions located in intermediate positions in the versions chain to be identified as eligible for GC. Each record version in the chain is marked with the value of the oldest

active snapshot that can see it. If several consecutive versions in a chain are marked with the same oldest active snapshot value, then all those following the first one can be removed.

The engine performs garbage collection of intermediate record versions during the following processes:

- sweep
- table scan during index creation
- background garbage collection in SuperServer
- in every user attachment after an updated or delete record is committed



Regular (traditional) garbage collection mechanism is not changed and still works the same way as in prior Firebird versions.

To make it work, the engine maintains in shared memory an array of all active database snapshots. When it needs to find the oldest active snapshot that can see a given record version, it just searches for the CN of the transaction that created that record version.

The default initial size of this shared memory block is 64KB but it will grow automatically when required. The initial block can be set to a custom size in `firebird.conf` and/or `databases.conf` using the new parameter `SnapshotsMemSize`.

Precision Improvement for Calculations Involving NUMERIC and DECIMAL

Alex Peshkov

Tracker ticket [CORE-4409](#)

As a side-effect of implementing the internal 128-bit integer data type, some improvements were made to the way Firebird handles the precision of intermediate results from calculations involving long NUMERIC and DECIMAL data types. In prior Firebird versions, numerics backed internally by the BIGINT data type (i.e. with precision between 10 and 18 decimal digits) were multiplied/divided using the same BIGINT data type for the result, which could cause overflow errors due to limited precision available. In Firebird 4, such calculations are performed using 128-bit integers, thus reducing possibilities for unexpected overflows.

Increased Number of Formats for Views

Adriano dos Santos Fernandes

Tracker ticket [CORE-5647](#)

Views are no longer limited to 255 formats (versions) before the database requires a backup and restore. The new limit is 32,000 versions.



This change does not apply to tables.

Optimizer Improvement for GROUP BY

Dmitry Yemanov

Tracker ticket [CORE-4529](#)

The improvement allows the use of a DESCENDING index on a column that is specified for GROUP BY.

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 now 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.

Support for RISC v.64 Platform

Richard Jones

Tracker ticket [CORE-5779](#)

A patch was introduced to compile Firebird 4.0 on the RISC v.64 platform.

Virtual table RDB\$CONFIG

Vlad Khorsun

Tracker ticket [CORE-3708](#)

A virtual table enumerating configuration settings actual for the current database. Columns:

RDB\$CONFIG_ID type INTEGER	Unique row identifier, no special meaning
RDB\$CONFIG_NAME type VARCHAR(63)	Setting name (e.g. "TempCacheLimit")
RDB\$CONFIG_VALUE type VARCHAR(255)	Actual value of setting
RDB\$CONFIG_DEFAULT type VARCHAR(255)	Default value of setting (defined in the Firebird code)
RDB\$CONFIG_IS_SET type BOOLEAN	TRUE if value was set by user, FALSE otherwise

RDB\$CONFIG_SOURCE type VARCHAR(255)	Name of configuration file (relative to the Firebird root directory) where this setting was taken from, or special value "DPB" if the setting was specified by the client application via API
---	---

Table RDB\$CONFIG is populated from in-memory structures upon request and its instance is preserved for the SQL query lifetime. For security reasons, access to this table is allowed to administrators only. Non-privileged users see no rows in this table (and no error is raised).

Chapter 4. Changes to the Firebird API and ODS

since Firebird 3.0 release

ODS (On-Disk Structure) Changes

New ODS Number

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

New System Tables

System tables added in ODS13:

<code>RDB\$TIME_ZONES</code>	Virtual table that enumerates supported time zones
<code>RDB\$PUBLICATIONS</code>	Publications defined in the database
<code>RDB\$PUBLICATION_TABLES</code>	Tables enabled for publication
<code>RDB\$CONFIG</code>	Virtual table that enumerates actual configuration settings



In Firebird 4.0, there's a single (pre-defined) publication named `RDB$DEFAULT`. User-defined publications will be available in future Firebird releases.

New Columns in System Tables

Column `RDB$SQL_SECURITY` was added to the following system tables in ODS13:

- `RDB$DATABASE`
- `RDB$FUNCTIONS`
- `RDB$PACKAGES`
- `RDB$PROCEDURES`
- `RDB$RELATIONS`
- `RDB$TRIGGERS`

For `RDB$DATABASE`, it defines the default SQL SECURITY mode (*DEFINER* or *INVOKER*) applied to the newly created objects. For other system tables, it defines the SQL SECURITY mode active for the appropriate objects.

Also, column `RDB$SYSTEM_PRIVILEGES` is added to the system table `RDB$ROLES`. It stores system privileges granted to a role.

Application Programming Interfaces

The wire protocol version for the Firebird 4.0 API is 16. Additions and changes are described in the sections below.

Services Cleanup

Alex Peshkov

Apart from the widely-known Services Manager (`service_mgr`), Firebird has a group of so-called “version 1” service managers. Backup and `gsec` are examples, along with a number of other services related to shared cache control and the unused journaling feature. Since at least Firebird 3 they seem to be in a semi-working state at best, so they have undergone a cleanup.

A visible effect is that the constant `service_mgr` is no longer required in the connection string for a service request. The request call will ignore anything in that field, including an empty string. The remote client will do the right thing just by processing the host name, such as `localhost:`, `inet://localhost/` or `inet://localhost`.

Services API Extensions

Support for `nbackup -fixup`

Alex Peshkov

Added support to `fixup` (i.e. change the physical backup mode to *normal*) databases after file-system copy.

The following action was added: `isc_action_svc_nfix:: fixup database`

Samples of use of new parameter in `fbsvcmgr` utility (supposing login and password are set using some other method):

```
fbsvcmgr -action_nfix dbname /tmp/ecopy.fdb
```

Timeouts for Sessions & Statements

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](#).

New Isolation Sub-level for **READ COMMITTED** Transactions

Provides API support for the new **READ COMMITTED READ CONSISTENCY** isolation sub-level for **READ COMMITTED** transactions. To start a **READ COMMITTED READ CONSISTENCY** transaction via the ISC API, use the new constant `isc_tpb_read_consistency` in the Transaction Parameter Buffer.

Support for Batch Insert and Update Operations in the API

Alex Peshkov

The OO-API in Firebird 4 supports execution of statements with more than a single set of parameters — *batch execution*. The primary purpose of the batch interface design is to satisfy JDBC requirements for batch processing of prepared statements, but it has some fundamental differences:

- As with all data operations in Firebird, it is oriented on messages, not on single fields
- An important extension of our batch interface is support for inline use of BLOBs, which is especially efficient when working with small BLOBs
- The `execute()` method returns not a plain array of integers but the special `BatchCompletionState` interface which, depending on the batch creation parameters, can contain both the information about the updated records and the error flag augmented by detailed status vectors for the messages that caused execution errors

The methods described below illustrate how to implement everything needed for JDBC-style prepared statement batch operations. Almost all of the methods described are used in `11.batch.cpp`. Please refer to it to see a live example of batching in Firebird.

Creating a Batch

As with `ResultSet` a batch may be created in two ways—using either the `Statement` or the `Attachment` interface. In both cases, the `createBatch()` method of appropriate interface is called.

For the `Attachment` case, the text of the SQL statement to be executed in a batch is passed directly to `createBatch()`.

Tuning of the batch operation is performed using the `Batch Parameters Block (BPB)` whose format is similar to `DPB v.2`: beginning with the tag (`IBatch::CURRENT_VERSION`) and followed by the set of wide clumplets: 1-byte tag, 4-byte length, length-byte value. Possible tags are described in batch interface.

The recommended (and simplest) way to create a BPB for batch creation is to use the appropriate `XpbBuilder` interface:

```
IXpbBuilder* pb = utl->getXpbBuilder(&status, IXpbBuilder::BATCH, NULL, 0);
pb->insertInt(&status, IBatch::RECORD_COUNTS, 1);
```

This usage of the BPB directs the batch to account for a number of updated records on per-message basis.

Creating the Batch Interface

To create the batch interface with the desired parameters, pass the BPB to a `createBatch()` call:

```
IBatch* batch = att->createBatch(&status, tra, 0, sqlStmtText, SQL_DIALECT_V6, NULL,
pb->getBufferLength(&status), pb->getBuffer(&status));
```

In this sample, the batch interface is created with the default message format because NULL is passed instead of the input metadata format.

Getting the Message Format

To proceed with the created batch interface, we need to get the format of the messages it contains, using the `getMetadata()` method:

```
IMessageMetadata* meta = batch->getMetadata(&status);
```

If you have passed your own format for messages to the batch, of course you can simply use that.

We assume here that some function is present that can fill the buffer “data” according to the passed format “metadata”. For example,

```
fillNextMessage(unsigned char* data, IMessageMetadata* metadata)
```

A Message Buffer

To work with the messages we need a buffer for our “data”:

```
unsigned char* data = new unsigned char[meta->getMessageLength(&status)];
```

Now we can add some messages full of data to the batch:

```
fillNextMessage(data, meta);
batch->add(&status, 1, data);
fillNextMessage(data, meta);
batch->add(&status, 1, data);
```



An alternative way to work with messages is to use the `FB_MESSAGE` macro. An example of this method can be found in the batch interface example, `11.batch.cpp`.

Executing the Batch

The batch is now ready to be executed:

```
IBatchCompletionState* cs = batch->execute(&status, tra);
```

We requested accounting of the number of modified records (inserted, updated or deleted) per

message. The interface `BatchCompletionState` is used to print it. The total number of messages processed by the batch could be less than the number of messages passed to the batch if an error happened and the option enabling multiple errors during batch processing was not turned on. To determine the number of messages processed:

```
unsigned total = cs->getSize(&status);
```

Now to print the state of each message:

```
for (unsigned p = 0; p < total; ++p)
    printf("Msg %u state %d\n", p, cs->getState(&status, p));
```

A complete example of printing the contents of `BatchCompletionState` is in the function `print_cs()` in sample `11.batch.cpp`.

Cleaning Up

Once analysis of the completion state is finished, remember to dispose of it:

```
cs->dispose();
```

If you want to empty the batch's buffers without executing it for some reason, such as preparing for a new portion of messages to process, use the `cancel()` method:

```
batch->cancel(&status);
```

Being reference-counted, the batch does not have special method to close it—just a standard `release()` call:

```
batch->release();
```

Multiple Messages per Call

More than a single message can be added in one call to the batch. It is important to remember that messages should be appropriately aligned for this feature to work correctly. The required alignment and aligned size of the message should be obtained from the interface `MessageMetadata`. For example:

```
unsigned aligned = meta->getAlignedLength(&status);
```

Later that size will be useful when allocating an array of messages and working with it:

```

unsigned char* data = new unsigned char[aligned * N];
    // N is the desired number of messages
for (int n = 0; n < N; ++n) fillNextMessage(&data[aligned * n], meta);
batch->add(&status, N, data);

```

After that, the the batch can be executed or the next portion of messages can be added to it.

Passing In-line BLOBs in Batch Operations

As a general rule, BLOBs are not compatible with batches. Batching is efficient when a lot of small data are to be passed to the server in single step. BLOBs are treated as large objects so, as a rule, it makes no sense to use them in batches.

Nevertheless, in practice it often happens that BLOBs are not too big. When that is the case, use of the traditional BLOB API (create BLOB, pass segments to the server, close BLOB, pass BLOB's ID in the message) kills performance, especially over a WAN. Firebird's batching therefore supports passing BLOBs to the server *in-line*, along with other messages.

BLOB usage policy

To use the in-line BLOB feature, first a *BLOB usage policy* has to be set up as an option in the BPB for the batch being created:

```
pb->insertInt(&status, IBatch::BLOB_IDS, IBatch::BLOB_IDS_ENGINE);
```

In this example, for the simplest and fairly common usage scenarios, the Firebird engine generates the temporary BLOB IDs needed to keep a link between a BLOB and the message where it is used. Imagine that the message is described as follows:

```

FB_MESSAGE(Msg, ThrowStatusWrapper,
(FB_VARCHAR(5), id)
(FB_VARCHAR(10), name)
(FB_BLOB, desc)
) project(&status, master);

```

Something like the following will send a message to the server containing the BLOB:

```

project->id = ++idCounter;
project->name.set(currentName);
batch->addBlob(&status, descriptionSize, descriptionText, &project->desc);
batch->add(&status, 1, project.getData());

```

Over-sized BLOBs

If some BLOB happens to be too big to fit into your existing buffer, then, instead of reallocating the buffer, you can use the `appendBlobData()` method to append more data to the last added BLOB:

```
batch->addBlob(&status, descriptionSize, descriptionText, &project->desc, bpbLength,
bpb);
```

After adding the first part of the BLOB, get the next portion of data into `descriptionText`, update `descriptionSize` and then do:

```
batch->appendBlobData(&status, descriptionSize, descriptionText);
```

You can do this work in a loop but take care not to overflow the internal batch buffers. Its size is controlled by the `BUFFER_BYTES_SIZE` option when creating the batch interface. The default size is 10MB, but it cannot exceed 40MB. If you need to process a BLOB that is too big, having chosen to use batching on the basis of data involving a lot of small BLOBs, just use the standard BLOB API and the `registerBlob` method of the Batch interface.

User-Supplied BLOB IDs

Another possible choice in the BLOB policy is `BLOB_IDS_USER`, to supply a temporary `BLOB_ID` instead of having one generated by Firebird.

Usage is not substantially different. Before calling `addBlob()`, place the correct execution ID, which is unique per batch, into the memory referenced by the last parameter. Exactly the same ID should be passed in the data message for the BLOB.

Considering that generation of BLOB IDs by the engine is very fast, such a policy may seem useless. However, imagine a case where you get BLOBs and other data in relatively independent streams (blocks in a file, for example) and some good IDs are already present in them. Supplying the BLOB IDs can greatly simplify your code for such cases.

Streams vs Segments

Be aware that BLOBs created by the Batch interface are by default streamed, not segmented like BLOBs created by means of `createBlob()`. Segmented BLOBs provide nothing interesting compared with streamed ones—we support that format only for backward compatibility and recommend avoiding them in new development.

Overriding to Use Segmented BLOBs

If you really must have segmented BLOBs, you can override the default by calling:

```
batch->setDefaultBpb(&status, bpbLength, bpb);
```



Of course, the passed BPB could contain other BLOB creation parameters, too. You could also pass the BPB directly to `addBlob()` but, if most of the BLOBs you are going to add have the same non-default format, it is slightly more efficient to use `setDefaultBpb()`.

A call to `addBlob()` will add the first segment to the BLOB; successive calls to `appendBlobData()` will add more segments.



Segment size limit!

Keep in mind that segment size is limited to 64KB -1. Attempting to pass more data in a single call will cause an error.

Multiple BLOBs Using Streams

Using the method `addBlobStream()`, it is possible to add more than one BLOB to the batch in a single call.

A blob stream is a sequence of BLOBs, each starting with a BLOB header which needs to be appropriately aligned. The Batch interface provides a special call for this purpose:

```
unsigned alignment = batch->getBlobAlignment(&status);
```

It is assumed that all components of a BLOB stream in a batch will be aligned, at least at the alignment boundary. This includes the size of stream portions passed to `addBlobStream()`, which should be a multiple of this alignment.

The header contains three fields: an 8-byte BLOB ID (must be non-zero), a 4-byte total BLOB size and a 4 byte BPB size. The total BLOB size includes the enclosed BPB, i.e. the next BLOB in the stream will always be found in the BLOB-size bytes after the header, taking the alignment into account.

The BPB is present if the BPB size is not zero and is placed immediately after the header. The BLOB data goes next, its format depending upon whether the BLOB is streamed or segmented:

- For a stream BLOB it is a plain sequence of bytes whose size is (BLOB-size - BPB-size)
- For a segmented BLOB, things are a bit more complicated: the BLOB data is a set of segments where each segment has the format: 2-bytes for the size of the segment, aligned at `IBatch::BLOB_SEGHDR_ALIGN` boundary, followed by as many bytes as are accounted for by this 2-byte segment size

Bigger BLOBs in the Stream

When a big BLOB is added to the stream, its size is not always known in advance. To avoid having too large a buffer for that BLOB (recalling that the size has to be provided in the BLOB header, before the BLOB data) a *BLOB continuation record* may be used. In the BLOB header, you leave BLOB size at a value known when creating that header and add a continuation record. The format of the continuation record is identical to the BLOB header, except that both the BLOB ID and the BPB size must always be zero.

Typically, you will want to have one continuation record per `addBlobStream()` call.

An example of this usage can be found in `sample 11.batch.cpp`.

Registering a Standard BLOB

The last method used to work with BLOBs stands apart from the first three that pass BLOB data inline with the rest of the batch data. It is required for registering in a batch the ID of a BLOB created using the standard BLOB API. This may be unavoidable if a really big BLOB has to be passed to the batch.

The ID of such BLOB cannot be used in the batch directly without causing an invalid BLOB ID error during batch execution. Instead do:

```
batch->registerBlob(&status, &realId, &msg->desc);
```

If the BLOB policy is making the Firebird engine generate BLOB IDs then this code is enough to correctly register an existing BLOB in a batch. In other cases you will have to assign to `msg->desc` the ID that is correct from the point of view of the batch.

Batch Ops in the Legacy (ISC) API

A few words about access to batches from the ISC API: a prepared ISC statement can be executed in batch mode. The main support for it is present in ISC API functions: `fb_get_transaction_interface` and `fb_get_statement_interface`. These methods enable access to the appropriate interfaces in the same way as to existing ISC handles.

An example of this usage can be found in `12.batch_isc.cpp`.

API Support for Time Zones

Structures (structs)

```
struct ISC_TIME_TZ
{
    ISC_TIME utc_time;
    ISC_USHORT time_zone;
};
```

```
struct ISC_TIMESTAMP_TZ
{
    ISC_TIMESTAMP utc_timestamp;
    ISC_USHORT time_zone;
};
```

```
struct ISC_TIME_TZ_EX
{
    ISC_TIME utc_time;
    ISC_USHORT time_zone;
    ISC_SHORT ext_offset;
};
```

```
struct ISC_TIMESTAMP_TZ_EX
{
    ISC_TIMESTAMP utc_timestamp;
    ISC_USHORT time_zone;
    ISC_SHORT ext_offset;
};
```

API Functions: (FirebirdInterface.idl — IUtil interface)

```
void decodeTimeTz(
    Status status,
    const ISC_TIME_TZ* timeTz,
    uint* hours,
    uint* minutes,
    uint* seconds,
    uint* fractions,
    uint timeZoneBufferLength,
    string timeZoneBuffer
);
```

```
void decodeTimeStampTz(
    Status status,
    const ISC_TIMESTAMP_TZ* timeStampTz,
    uint* year,
    uint* month,
    uint* day,
    uint* hours,
    uint* minutes,
    uint* seconds,
    uint* fractions,
    uint timeZoneBufferLength,
    string timeZoneBuffer
);
```

```
void encodeTimeTz(
    Status status,
    ISC_TIME_TZ* timeTz,
    uint hours,
    uint minutes,
    uint seconds,
    uint fractions,
    const string timeZone
);
```

```
void encodeTimeStampTz(
    Status status,
    ISC_TIMESTAMP_TZ* timeStampTz,
    uint year,
    uint month,
    uint day,
    uint hours,
    uint minutes,
    uint seconds,
    uint fractions,
    const string timeZone
);
```

```
void decodeTimeTzEx(
    Status status,
    const ISC_TIME_TZ_EX* timeTzEx,
    uint* hours,
    uint* minutes,
    uint* seconds,
    uint* fractions,
    uint timeZoneBufferLength,
    string timeZoneBuffer
);
```

```

void decodeTimeStampTzEx(
    Status status,
    const ISC_TIMESTAMP_TZ_EX* timeStampTzEx,
    uint* year,
    uint* month,
    uint* day,
    uint* hours,
    uint* minutes,
    uint* seconds,
    uint* fractions,
    uint timeZoneBufferLength,
    string timeZoneBuffer
);

```

API Support for DECFLOAT and Long Numerics

Alex Peshkov

DecFloat16 and DecFloat34 are helper interfaces that simplify working with the DECFLOAT (16-digit and 34-digit respectively) data types. Available methods in the DecFloat16 interface are the following:

```

void toBcd(
    const FB_DEC16* from,
    int* sign,
    uchar* bcd,
    int* exp
);

```

```

void toString(
    Status status,
    const FB_DEC16* from,
    uint bufferLength,
    string buffer
);

```

```

void fromBcd(
    int sign,
    const uchar* bcd,
    int exp,
    FB_DEC16* to
);

```

```

void fromString(
    Status status,
    const string from,
    FB_DEC16* to
);

```

The DecFloat34 interface shares the same methods, just using the FB_DEC34 structure.

Int128 is a helper interface for 128-bit integers (used internally as a base type for INT128, and also for NUMERIC and DECIMAL data types with precision > 18), it contains the following methods:

```

void toString(
    Status status,
    const FB_I128* from,
    int scale,
    uint bufferLength,
    string buffer
);

```

```

void fromString(
    Status status,
    int scale,
    const string from,
    FB_I128* to
);

```

Structures used by the aforementioned interfaces are defined below:

```

struct FB_DEC16
{
    ISC_UINT64 fb_data[1];
};

```

```

struct FB_DEC34
{
    ISC_UINT64 fb_data[2];
};

```

```

struct FB_I128
{
    ISC_UINT64 fb_data[2];
};

```

In order to work with these new interfaces, the Util interface has been extended with the following methods:

```
DecFloat16 getDecFloat16(Status status);
DecFloat34 getDecFloat34(Status status);
Int128 getInt128(Status status);
```

Additions to Other Interfaces

Alex Peshkov

A number of new methods have been added to the following interfaces.

Attachment

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

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

```
Batch createBatch(
    Status status,
    Transaction transaction,
    uint stmtLength,
    const string sqlStmt,
    uint dialect,
    MessageMetadata inMetadata,
    uint parLength,
    const uchar* par
);
```

Statement

```
uint getTimeout(Status status);
void setTimeout(Status status, uint timeout);
```

```
Batch createBatch(
    Status status,
    MessageMetadata inMetadata,
    uint parLength,
    const uchar* par
);
```

ClientBlock

```
AuthBlock getAuthBlock(Status status);
```

Server

```
void setDbCryptCallback(Status status, CryptKeyCallback cryptCallback);
```

MessageMetadata

```
uint getAlignment(Status status);
uint getAlignedLength(Status status);
```

MetadataBuilder

```
void setField(Status status, uint index, const string field);
void setRelation(Status status, uint index, const string relation);
void setOwner(Status status, uint index, const string owner);
void setAlias(Status status, uint index, const string alias);
```

FirebirdConf

```
uint getVersion(Status status);
```

ConfigManager

```
const string getDefaultSecurityDb();
```

Extensions to various getInfo() Methods

Attachment::getInfo()

The following actions were added:

<code>fb_info_protocol_version</code>	Version of the remote protocol used by the current connection
<code>fb_info_crypt_plugin</code>	Name of the used database encryption plugin
<code>fb_info_wire_crypt</code>	Name of the connection encryption plugin
<code>fb_info_statement_timeout_db</code>	Statement execution timeout set in the configuration file
<code>fb_info_statement_timeout_att</code>	Statement execution timeout set at the connection level

<code>fb_info_ses_idle_timeout_db</code>	Idle connection timeout set in the configuration file
<code>fb_info_ses_idle_timeout_att</code>	Idle connection timeout set at the connection level
<code>fb_info_ses_idle_timeout_run</code>	Actual timeout value for the current connection
<code>fb_info_creation_timestamp_tz</code>	Database creation timestamp (with a time zone)
<code>fb_info_features</code>	List of features supported by provider of the current connection
<code>fb_info_next_attachment</code>	Current value of the next attachment ID counter
<code>fb_info_next_statement</code>	Current value of the next statement ID counter
<code>fb_info_db_guid</code>	Database GUID (persistent until restore / fixup)
<code>fb_info_db_file_id</code>	Unique ID of the database file at the filesystem level
<code>fb_info_replica_mode</code>	Database replica mode

Possible provider features (returned for `fb_info_features`) are:

<code>fb_feature_multi_statements</code>	Multiple prepared statements in single attachment
<code>fb_feature_multi_transactions</code>	Multiple concurrent transaction in single attachment
<code>fb_feature_named_parameters</code>	Query parameters can be named
<code>fb_feature_session_reset</code>	ALTER SESSION RESET is supported
<code>fb_feature_read_consistency</code>	Read Consistency transaction isolation mode is supported
<code>fb_feature_statement_timeout</code>	Statement timeout is supported
<code>fb_feature_statement_long_life</code>	Prepared statements are not dropped on transaction end

Possible replica modes (returned for `fb_info_replica_mode`) are:

<code>fb_info_replica_none</code>	Database is not in the replica state
<code>fb_info_replica_read_only</code>	Database is a read-only replica

`fb_info_replica_read_write` Database is a read-write replica

Statement::getInfo()

The following actions were added:

`isc_info_sql_stmt_timeout_user` Timeout value of the current statement

`isc_info_sql_stmt_timeout_run` Actual timeout value of the current statement

`isc_info_sql_stmt_blob_align` Blob stream alignment in the Batch API

Transaction::getInfo()

The following action was added:

`fb_info_tra_snapshot_number` Snapshot number of the current transaction

Additions to the Legacy (ISC) API

Alex Peshkov

A few functions have been added to the ISC API.

```
ISC_STATUS fb_get_transaction_interface(ISC_STATUS*, void*, isc_tr_handle*);
ISC_STATUS fb_get_statement_interface(ISC_STATUS*, void*, isc_stmt_handle*);
```

They can be used to get an OO API object from the corresponding ISC API handle.

Chapter 5. Reserved Words and Changes

New Keywords in Firebird 4.0

Reserved

BINARY	DECFLOAT	INT128
LATERAL	LOCAL	LOCALTIME
LOCALTIMESTAMP	PUBLICATION	RDB\$GET_TRANSACTION_CN
RDB\$ERROR	RDB\$ROLE_IN_USE	RDB\$SYSTEM_PRIVILEGE
RESETTING	TIMEZONE_HOUR	TIMEZONE_MINUTE
UNBOUNDED	VARBINARY	WINDOW
WITHOUT		

Non-reserved

BASE64_DECODE	BASE64_ENCODE	BIND
CLEAR	COMPARE_DECFLOAT	CONNECTIONS
CONSISTENCY	COUNTER	CRYPT_HASH
CTR_BIG_ENDIAN	CTR_LENGTH	CTR_LITTLE_ENDIAN
CUME_DIST	DEFINER	DISABLE
ENABLE	EXCESS	EXCLUDE
EXTENDED	FIRST_DAY	FOLLOWING
HEX_DECODE	HEX_ENCODE	IDLE
INCLUDE	INVOKER	IV
LAST_DAY	LEGACY	LIFETIME
LPARAM	MAKE_DBKEY	MESSAGE
MODE	NATIVE	NORMALIZE_DECFLOAT
NTILE	NUMBER	OLDEST
OTHERS	OVERRIDING	PERCENT_RANK
POOL	PRECEDING	PRIVILEGE
QUANTIZE	RANGE	RESET
RSA_DECRYPT	RSA_ENCRYPT	RSA_PRIVATE
RSA_PUBLIC	RSA_SIGN_HASH	RSA_VERIFY_HASH
SALT_LENGTH	SECURITY	SESSION
SIGNATURE	SQL	SYSTEM
TIES	TOTALORDER	TRAPS

Chapter 6. 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 in 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 in the Firebird Engine](#).

Parameters for External Connection Pooling

These parameters enable customization of aspects of pooling external connections.

ExtConnPoolSize

Configures the maximum number of idle connections allowed in the pool. It is an integer, from 0 to 1000. The installation default is 0, which disables the connection pool.

ExtConnPoolLifetime

Configures the number of seconds a connection should stay available after it has gone idle. The installation default is 7200 seconds.

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.



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.

Parameters Supporting Read Consistency in Transactions

Firebird 4 takes a [new approach to read consistency within transaction snapshots](#), enabling, amongst other benefits, a sustained consistent read for statements within `READ COMMITTED` transactions. This group of parameters allows for some customisation of the elements involved.

ReadConsistency

For now, existing applications can be tested with and without the new `READ COMMITTED READ CONSISTENCY` isolation level by setting this parameter. Possible values are 1 and 0.

ReadConsistency = 1

(Default) The engine ignores the specified `[NO] RECORD VERSION` sub-level and forces all read-committed transactions to be `READ COMMITTED READ CONSISTENCY`.

ReadConsistency = 0

Allows the legacy engine behaviour, with the RECORD VERSION and NO RECORD VERSION sub-levels working as before. READ COMMITTED READ CONSISTENCY is available but needs to be specified explicitly.

This behaviour can be defined in `firebird.conf` and/or `databases.conf`.

TipCacheBlockSize

The list of all known transactions with associated Commit Numbers is maintained in shared memory. It is implemented as an array whose index is a transaction ID and its item value is the corresponding Commit Number.

The whole array is split into fixed-size blocks containing the CN's for all transactions between the OIT and Next Transaction markers. When the "Next Transaction" marker moves out of the scope of the highest block, a new block is allocated. An old block is released when the "Oldest [Interesting] Transaction" (OIT) marker moves out of the scope of the lowest block.

The default size for a TIP cache block is 4MB, providing capacity for 512 * 1024 transactions. Use this parameter to configure a custom TIP cache block size in `firebird.conf` and/or `databases.conf`.

SnapshotsMemSize

To handle garbage collection of record versions younger than the Oldest Snapshot, ("intermediate record versions") the engine maintains in shared memory an array that it can search for the Commit Number (CN) of a particular record version. See the [Garbage Collection](#) topic the chapter [Changes in the Firebird Engine](#).

The default initial size of this shared memory block is 64KB but it will grow automatically when required. The initial block can be set to a custom size in `firebird.conf` and/or `databases.conf`.

Other Parameters

ClientBatchBuffer

Defines the buffer size used by the client connection for batch-mode transmission to the server (when Batch API is used). See the [Support for Batch Insert and Update Operations in the API](#) topic for more details.

DataTypeCompatibility

Specifies the compatibility level that defines what SQL data types can be exposed to the client API. Currently two options are available: "3.0" and "2.5". The "3.0" emulation mode hides data types introduced after Firebird 3.0 release, in particular DECIMAL/NUMERIC with precision 19 or higher, DECFLOAT, TIME/TIMESTAMP WITH TIME ZONE. The corresponding values are returned via data types already supported by Firebird 3.0. The "2.5" emulation mode also converts the BOOLEAN data type. See the [Native to Legacy Coercion Rules table](#) for details. This setting allows legacy client applications to work with Firebird 4.0 without recompiling and adjusting them to understand the

new data types.

DefaultTimeZone

Defines the time zone used when the client session does not specify it explicitly. If left empty, the default is the operating system time zone. When set at the server side, it's the default session time zone for attachments. When set at the client side, it's the default time zone used with client-side API functions.

OutputRedirectionFile

Allows to (optionally) redirect server's stdout/stderr streams to some user-defined file. By default, these streams are opened by the server but the output is discarded. Available as a global setting inside `firebird.conf`.

Srp256 becomes the default authentication method

See Tracker ticket [CORE-5788](#)

The Secure Remote Password authentication plugin now uses the SHA-256 algorithm to calculate the client's proof for both server and client sides (see `AuthServer` and `AuthClient` settings in `firebird.conf`). For backward compatibility, the client is configured to use the old `Srp` plugin (which implements the SHA-1 algorithm) as a fallback. This setup allows to communicate with Firebird 3 servers that are not configured to use `Srp256` (available since v3.0.4).

ChaCha is added as a default wire encryption method

`WireCryptPlugin` setting now defaults to `ChaCha#20` as a wire encryption algorithm. If the appropriate plugin is missing, then Alleged RC4 (aka ARC4) algorithm is used.

TempCacheLimit at database level

See Tracker ticket [CORE-5718](#)

`TempCacheLimit`, for setting the maximum amount of temporary space that can be cached in memory, can now be configured at database level, i.e., in `databases.conf`. Previously, it was available only as a global setting for all databases.

UseFileSystemCache is added as a replacement for FileSystemCacheThreshold

See Tracker ticket [CORE-6332](#)

New boolean setting `UseFileSystemCache` provides an explicit control whether the OS filesystem cache is used for the database. The value is customizable at the database level. The old setting `FileSystemCacheThreshold` is preserved, but it is taken into account only if value for `UseFileSystemCache` is not specified explicitly. Setting `FileSystemCacheThreshold` becomes deprecated and will be removed in future Firebird versions.

InlineSortThreshold

See Tracker ticket [CORE-2650](#)

Controls how non-key fields are processed during sorting: stored inside the sort block or refetched from data pages after the sorting.

Historically, when the external sorting is performed, Firebird writes both key fields (those specified in the `ORDER BY` or `GROUP BY` clause) and non-key fields (all others referenced inside the query) to the sort blocks, either stored in memory or swapped to temporary files. Once the sorting is completed, these fields are read back from the sort blocks. This approach is generally considered being faster, because records are read in storage order instead of randomly fetching data pages corresponding to the sorted records. However, if non-key fields are large (e.g. long `VARCHARs` are involved), this increases the size of the sort blocks and thus causes earlier swapping and more I/O for temporary files. Firebird 4 provides an alternative approach, when only key fields and record `DBKEYs` are stored inside the sort blocks and non-key fields are refetched from data pages after the sorting. This improves sorting performance in the case of longish non-key fields.

The value specified for `InlineSortThreshold` defines the maximum sort record size (in bytes) that can be stored inline, i.e. inside the sort block. Zero means that records are always refetched.

Chapter 7. Security

Security enhancements 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
- Run [management statements](#)

The implementation defines a set of *system privileges*, analogous to object privileges, from which lists of privileged tasks can be assigned to roles.

List of Valid System Privileges

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

USER_MANAGEMENT	Manage users
READ_RAW_PAGES	Read pages in raw format using <code>Attachment::getInfo()</code>
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	Instruct 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 statements
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
MODIFY_EXT_CONN_POOL	Use command ALTER EXTERNAL CONNECTIONS POOL
REPLICATE_INTO_DATABASE	Use replication API to load change sets into database

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

Tracker ticket [CORE-2557](#)

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.



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 is 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 of system privileges 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.

Syntax

```
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.



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];
```



Above syntax is a simplified version, the full GRANT and REVOKE has more options.

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.

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 property 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 of arbitrary length and returns a Boolean result.

Syntax

```
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 available to the *CURRENT_USER* or the calling object, 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 from functions, procedures and packages, 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 and some database object

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 SECURITY] [AS ...]
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 overridden 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 *t*. If it were set for *INVOKER*, the user would also need 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 *f*. If it were set for *INVOKER*, the user would also need 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 *p*. If it were set for *INVOKER*, either the user or the procedure would also need 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_ins*, user *US* needs only the *INSERT* privilege on *tr*. If it were set for *INVOKER*, either the user or the trigger would also need 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 *pk*. If it were set for *INVOKER*, either the user or the package would also need 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;

```

Built-in Cryptographic Functions

Alex Peshkov

Tracker ticket [CORE-5970](#)

Firebird 4 introduces eight new built-in functions supporting cryptographic tasks.

ENCRYPT() and DECRYPT()

For encrypting/decrypting data using a symmetric cipher.

Syntax

```
{ENCRYPT | DECRYPT} ( <string | blob> USING <algorithm> [MODE <mode>] KEY <string>
                    [IV <string>] [<endianness>] [CTR_LENGTH <smallint>] [COUNTER <bigint>])

<algorithm> ::= { <block_cipher> | <stream_cipher> }
<block_cipher> ::= { AES | ANUBIS | BLOWFISH | KHAZAD | RC5 | RC6 | SAFER+ | TWOFISH |
XTEA }
<stream_cipher> ::= { CHACHA20 | RC4 | SOBER128 }
<mode> ::= { CBC | CFB | CTR | ECB | OFB }
<endianness> ::= { CTR_BIG_ENDIAN | CTR_LITTLE_ENDIAN }
```



- Mode should be specified for block ciphers
- Initialization vector (IV) should be specified for block ciphers in all modes except ECB and all stream ciphers except RC4
- Endianness may be specified only in CTR mode, default is little endian counter
- Counter length (CTR_LENGTH, bytes) may be specified only in CTR mode, default is the size of IV
- Initial counter value (COUNTER) may be specified only for CHACHA20 cipher, default is 0
- Sizes of data strings passed to these functions are in accordance with the selected algorithm and mode requirements
- Functions return BLOB when the first argument is blob and varbinary for all text types.

Examples

```
select encrypt('897897' using sober128 key 'AbcdAbcdAbcdAbcd' iv '01234567')
       from rdb$database;
select decrypt(x'0154090759DF' using sober128 key 'AbcdAbcdAbcdAbcd' iv '01234567')
       from rdb$database;
select decrypt(secret_field using aes mode ofb key '0123456701234567' iv init_vector)
       from secure_table;
```

RSA_PRIVATE()

Returns an RSA private key of specified length (in bytes) in PKCS#1 format as a VARBINARY string.

Syntax

```
RSA_PRIVATE ( <smallint> )
```

Example

```
select rdb$set_context('USER_SESSION', 'private_key', rsa_private(256))
from rdb$database;
```



Putting private keys in the context variables is not secure. SYSDBA and users with the role RDB\$ADMIN or the system privilege MONITOR_ANY_ATTACHMENT can see all context variables from all attachments.

RSA_PUBLIC()

Returns the RSA public key for a specified RSA private key. Both keys are in PKCS#1 format.

Syntax

```
RSA_PUBLIC ( <private key> )
```



Run your samples one by one from the RSA_PRIVATE function forward.

Example

```
select rdb$set_context('USER_SESSION', 'public_key',
rsa_public(rdb$get_context('USER_SESSION', 'private_key'))) from rdb$database;
```

RSA_ENCRYPT()

Pads data using [OAEP padding](#) and encrypts it using an RSA public key. Normally used to encrypt short symmetric keys which are then used in block ciphers to encrypt a message.

Syntax

```
RSA_ENCRYPT ( <string> KEY <public key> [LPARAM <string>] [HASH <hash>] )
```

KEY should be a value returned by the RSA_PUBLIC function. LPARAM is an additional system-specific tag that can be applied to identify which system encrypted the message. Its default value is NULL.

```
<hash> ::= { MD5 | SHA1 | SHA256 | SHA512 }
```

Default is SHA256.



Run your samples one by one from the RSA_PRIVATE function forward.

Example

```
select rdb$set_context('USER_SESSION', 'msg', rsa_encrypt('Some message'
  key rdb$get_context('USER_SESSION', 'public_key'))) from rdb$database;
```

RSA_DECRYPT()

Decrypts using the RSA private key and OAEP de-pads the resulting data.

Syntax

```
RSA_DECRYPT ( <string> KEY <private key> [LPARAM <string>] [HASH <hash>] )
```

KEY should be a value returned by the RSA_PRIVATE function. LPARAM is the same variable passed to RSA_ENCRYPT. If it does not match what was used during encryption, RSA_DECRYPT will not decrypt the packet.

```
<hash> ::= { MD5 | SHA1 | SHA256 | SHA512 }
```

Default is SHA256.



Run your samples one by one from the RSA_PRIVATE function forward.

Example

```
select rsa_decrypt(rdb$get_context('USER_SESSION', 'msg')
  key rdb$get_context('USER_SESSION', 'private_key')) from rdb$database;
```

RSA_SIGN_HASH()

Performs PSS encoding of the message digest to be signed and signs using the RSA private key.

PSS encoding

Probabilistic Signature Scheme (PSS) is a cryptographic signature scheme specifically developed to allow modern methods of security analysis to prove that its security directly relates to that of the RSA problem. There is no such proof for the traditional PKCS#1 v1.5 scheme.

Syntax

```
RSA_SIGN_HASH ( <message digest> KEY <private key>
  [HASH <hash>] [SALT_LENGTH <smallint>] )
```

message digest is an already hashed message.

KEY should be a value returned by the RSA_PRIVATE function.

```
<hash> ::= { MD5 | SHA1 | SHA256 | SHA512 }
```

Default is SHA256. *hash* should match the hash algorithm used to generate the message hash value.

SALT_LENGTH indicates the length of the desired salt, and should typically be small. A good value is between 8 and 16.



Run your samples one by one from the RSA_PRIVATE function forward.

Example

```
select rdb$set_context('USER_SESSION', 'msg',
  rsa_sign_hash(crypt_hash('Test message' using sha256)
  key rdb$get_context('USER_SESSION', 'private_key')) from rdb$database;
```

RSA_VERIFY_HASH()

Performs PSS encoding of message digest to be signed and verifies its digital signature using the RSA public key.

Syntax

```
RSA_VERIFY_HASH (
  <message digest> SIGNATURE <string>
  KEY <public key> [HASH <hash>] [SALT_LENGTH <smallint>] )
```

message digest is an already hashed message.

SIGNATURE should be a value returned by the RSA_SIGN_HASH function.

KEY should be a value returned by RSA_PUBLIC function.

```
<hash> ::= { MD5 | SHA1 | SHA256 | SHA512 }
```

Default is SHA256. The hash should match the hash algorithm used to generate the *message digest* value and the original signature.

SALT_LENGTH indicates the length of the desired salt, and should typically be small. A good value is

between 8 and 16.



Run your samples one by one from the `RSA_PRIVATE` function forward.

Example

```
select rsa_verify_hash(
  crypt_hash('Test message' using sha256)
  signature rdb$get_context('USER_SESSION', 'msg')
  key rdb$get_context('USER_SESSION', 'public_key'))
from rdb$database;
```

Improvements to Security Features

The following improvements were made to existing security features:

User Managing Other Users

Alex Peshkov

Tracker ticket [CORE-5770](#)

A user that was created with user account administration privileges in the security database (via the `... GRANT ADMIN ROLE` clause) no longer has to hold the `RDB$ADMIN` role in the connected database and pass it explicitly in order to create, alter or drop other users.



This improvement is also backported to Firebird 3.0.5.

Chapter 8. Management Statements

Over the more recent releases of Firebird a new class of DSQL statement has emerged in Firebird's SQL lexicon, usually for administering aspects of the client/server environment. Typically, such statements commence with the verb SET, especially those introduced in Firebird 4.



Some statements of this class, introduced earlier, use the verb ALTER, although management statements should not be confused with DDL ALTER statements that modify database objects like tables, views, procedures, roles, et al.

Management statements can run anywhere DSQL can run but, typically, the developer will want to run a management statement in a database trigger. In past releases, management statements were treated in PSQL like DDL, precluding them from running directly inside a PSQL module. From Firebird 4 forward, a pre-determined set of them can be used directly in PSQL modules without the need to wrap them in an EXECUTE STATEMENT block. For more details of the current set, see [Allow Management Statements in PSQL Blocks](#) in the PSQL chapter.

Most of the management statements introduced in Firebird 4 affect the current connection (“session”) only, and do not require any authorization over and above the login privileges of a current user without elevated privileges.

Some management statements operate beyond the scope of the current session. Examples are the ALTER DATABASE ... statements to control *nBackup* or the ALTER EXTERNAL CONNECTIONS POOL statements introduced in Firebird 4 to manage connection pooling. A new set of *system privileges*, analogous with SQL privileges granted for database objects, is provided for assignment to a role, to enable the required authority to run a specific management statement in this category. For details, refer to [Enhanced System Privileges](#) in the [Security](#) chapter.

Connections Pooling Management

A group of management statements for use with connections pooling.



Authorization

A role carrying the new system privilege MODIFY_EXT_CONN_POOL is required to run the statements.

ALTER EXTERNAL CONNECTIONS POOL

The new statement ALTER EXTERNAL CONNECTIONS POOL has been added to the repertoire for managing the external connections pool.

The syntax is:

```
ALTER EXTERNAL CONNECTIONS POOL { <parameter variants> }
```

When prepared it is described like a DDL statement but its effect is immediate—it is executed

immediately and completely, without waiting for transaction commit.

The statements can be issued from any connection, and changes are applied to the in-memory instance of the pool in the current Firebird process. If the process is a Classic one, a change submitted there does not affect other Classic processes.

Changes made with `ALTER EXTERNAL CONNECTIONS POOL` are not persistent: after a restart, Firebird will use the pool settings configured in `firebird.conf` by `ExtConnPoolSize` and `ExtConnPoolLifeTime`.

Full Syntax

Full syntax for the variants follows.

To set the maximum number of idle connections:

```
ALTER EXTERNAL CONNECTIONS POOL SET SIZE int_value
```

Valid values are from 0 to 1000. Setting it to zero disables the pool. The default value is set using the parameter `ExtConnPoolSize` in `firebird.conf`.

To set the lifetime of an idle connection:

```
ALTER EXTERNAL CONNECTIONS POOL SET LIFETIME int_value <time_part>
```

```
<time_part> ::= { SECOND | MINUTE | HOUR }
```

Valid values are from 1 `SECOND` to 24 `HOUR`. The default value (in seconds) is set using the parameter `ExtConnPoolLifetime` in `firebird.conf`.

To close all idle connections and instigate dissociation of all active connections so they are immediately closed when they become unused:

```
ALTER EXTERNAL CONNECTIONS POOL CLEAR ALL
```

To close expired idle connections:

```
ALTER EXTERNAL CONNECTIONS POOL CLEAR OLDEST
```

ALTER SESSION RESET

Syntax

```
ALTER SESSION RESET
```

This statement is used to reset the current user session to its initial state. It could be useful for re-

using the session by a client application (for example, by a client-side connection pool). In order to reuse a session, all its user context variables and contents of temporary tables should be cleared and all its session-level settings should be reset to their default values.

This statement is executed the following way:

- Error *isc_ses_reset_err* is raised if any transaction remains active in the current session, except of current transaction and two-phase-commit transactions in the prepared state.
- System variable `RESETTING` is set to `TRUE`.
- `ON DISCONNECT` database triggers are fired, if present and allowed for current connection.
- The current transaction (the one executing `ALTER SESSION RESET`), if present, is rolled back. A warning is reported if this transaction had modified some data in tables before resetting the session.
- `DECFLOAT` parameters (`TRAP` and `ROUND`) are reset to the initial values defined using `DPB` at connection time, or otherwise the system default
- Session and statement timeouts are reset to zero.
- Context variables defined for the 'USER_SESSION' namespace are removed.
- Global temporary tables defined as `ON COMMIT PRESERVE ROWS` are truncated (their contents is cleared).
- Current role is restored to the initial value defined using `DPB` at connection time, the security classes cache is cleared (if role was changed).
- The session time zone is reset to the initial value defined using `DPB` at connection time, or otherwise the system default.
- The bind configuration is reset to the initial value defined using `DPB` at connection time, or otherwise the database or system default.
- In general, configuration values should revert to values configured using `DPB` at connection time, or otherwise the database or system default.
- `ON CONNECT` database triggers are fired, if present and allowed for current connection.
- A new transaction is implicitly started with the same parameters as the transaction that was rolled back (if it was present).
- System variable `RESETTING` is set to `FALSE`.

Note, `CURRENT_USER` and `CURRENT_CONNECTION` will not be changed.

New system variable `RESETTING` is introduced to detect cases when a database trigger is fired due to session reset. It is available in triggers only and can be used in any place when a boolean predicate is allowed. Its value is `TRUE` if session reset is in progress and `FALSE` otherwise. `RESETTING` is a reserved word now.

Errors handling

Any error raised by `ON DISCONNECT` triggers aborts session reset and leaves the session state unchanged. Such errors are reported using primary error code *isc_session_reset_err* and error text

"Cannot reset user session".

Any error raised after ON DISCONNECT triggers (including the ones raised by ON CONNECT triggers) aborts both session reset statement execution and connection itself. Such errors are reported using primary error code *isc_session_reset_failed* and error text "Reset of user session failed. Connection is shut down.". Subsequent operations on connection (except detach) will fail with *isc_att_shutdown_error*.

Time Zone Management

Statement syntax has been added to support management of the time zone features for the current connection.

SET TIME ZONE

Changes the session time zone.

Syntax

```
SET TIME ZONE { time_zone_string | LOCAL }
```

Examples

```
set time zone '-02:00';
set time zone 'America/Sao_Paulo';
set time zone local;
```

Timeout Management

The timeout periods for session and statement timeouts can be managed at session level using the management statements `SET SESSION IDLE TIMEOUT` and `SET STATEMENT TIMEOUT`, respectively.

Setting DECFLOAT Properties

Syntax:

```
SET DECFLOAT <property-name> [T0] <value>
```

are available for controlling the properties of the DECFLOAT data type for the current session.

Possible properties and their values are the following:

- `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

The default rounding mode is HALF-UP. The initial configuration may be specified via API by using DPB tag `isc_dpb_decfloat_round` followed by the string value.

- `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	—

The initial configuration may be specified via API by using DPB tag `isc_dpb_decfloat_traps` followed by the string value.

Setting Data Type Coercion Rules

Syntax:

```
SET BIND OF { <type-from> | TIME ZONE } TO { <type-to> | LEGACY | NATIVE | EXTENDED }
```

This management statement makes it possible to substitute one data type with another one when performing the client-server interaction. In other words, *type-from* returned by the engine is represented as *type-to* in the client API.



Only fields returned by the database engine in regular messages are substituted according to these rules. Variables returned as an array slice are not affected by the SET BIND statement.

When an incomplete type definition is used (i.e. simply CHAR instead of CHAR(*n*)) in the *FROM* part, the coercion is performed for all CHAR columns. The special incomplete type TIME ZONE stands for all types WITH TIME ZONE (namely TIME and TIMESTAMP). When an incomplete type definition is used in the *TO* part, the engine defines missing details about that type automatically based on source column.

Changing the binding of any NUMERIC or DECIMAL data type does not affect the appropriate underlying integer type. In contrast, changing the binding of an integer data type also affects appropriate NUMERICs/DECIMALs.

The special format LEGACY is used when a data type, missing in previous Firebird version, should be represented in a way, understandable by old client software (possibly with some data loss). The coercion rules applied in this case are shown in the table below.

Table 1. Native to LEGACY coercion rules

Native data type	Legacy data type
BOOLEAN	CHAR(5)
DECFLOAT	DOUBLE PRECISION
INT128	BIGINT
TIME WITH TIME ZONE	TIME WITHOUT TIME ZONE
TIMESTAMP WITH TIME ZONE	TIMESTAMP WITHOUT TIME ZONE

Using EXTENDED in the *TO* part causes the engine to coerce to an extended form of the *FROM* data type. Currently, this works only for TIME/TIMESTAMP WITH TIME ZONE, they are coerced to EXTENDED TIME/TIMESTAMP WITH TIME ZONE. The EXTENDED type contains both the time zone name, and the corresponding GMT offset, so it remains usable if the client application cannot process named time zones properly (e.g. due to the missing ICU library).

Setting a binding to NATIVE resets the existing coercion rule for this data type and returns it in the native format.

Examples:

```

SELECT CAST('123.45' AS DECFLOAT(16)) FROM RDB$DATABASE;    --native

          CAST
=====
          123.45

SET BIND OF DECFLOAT TO DOUBLE PRECISION;
SELECT CAST('123.45' AS DECFLOAT(16)) FROM RDB$DATABASE;    --double

          CAST
=====
          123.45000000000000

SET BIND OF DECFLOAT(34) TO CHAR;
SELECT CAST('123.45' AS DECFLOAT(16)) FROM RDB$DATABASE;    --still double

          CAST
=====
          123.45000000000000

SELECT CAST('123.45' AS DECFLOAT(34)) FROM RDB$DATABASE;    --text

CAST
=====
123.45

```

In the case of missing ICU on the client side:

```

SELECT CURRENT_TIMESTAMP FROM RDB$DATABASE;

          CURRENT_TIMESTAMP
=====
2020-02-21 16:26:48.0230 GMT*

SET BIND OF TIME_ZONE TO EXTENDED;
SELECT CURRENT_TIMESTAMP FROM RDB$DATABASE;

          CURRENT_TIMESTAMP
=====
2020-02-21 19:26:55.6820 +03:00

```

Chapter 9. Data Definition Language (DDL)

Quick Links

- [Extended Length for Object Names](#)
- [Data Type DECFLOAT](#)
- [Increased Precision for Exact Numeric Types](#)
- [Standard Compliance for Data Type FLOAT](#)
- [Data Type Extensions for Time Zone Support](#)
- [Aliases for Binary String Types](#)
- [Extensions to the IDENTITY Type](#)
- [Excess parameters in EXECUTE STATEMENT](#)
- [Replication Management](#)

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.



Double quotes around a column name 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.

New Data Types

New data types implemented in Firebird 4.0:

Data Type INT128

Alex Peshkov

Tracker ticket [CORE-6342](#)

For details, see [Increased Precision for Exact Numeric Types](#) later in this chapter.

Data Types TIME WITH TIME ZONE and TIMESTAMP WITH TIME ZONE

Adriano dos Santos Fernandes

Tracker tickets [CORE-694](#)

For details, see [Data Type Extensions for Time Zone Support](#) later in this chapter.

Data Type DECFLOAT

Alex Peshkov

Tracker ticket [CORE-5525](#)

DECFLOAT is an SQL:2016 standard-compliant numeric type that stores floating-point numbers precisely (decimal floating-point type), unlike FLOAT or DOUBLE PRECISION that provide a binary approximation of the purported precision. Firebird 4 accords with the IEEE 754-1985 standard types `Decimal64` and `Decimal128` by providing both 16-digit and 34-digit precision for this type.

All intermediate calculations are performed with 34-digit values.



16-digit and 34-digit

The “16” and “34” refer to the maximum 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)
DECFLOAT
```

The default precision is 34 digits, i.e., if DECFLOAT is declared with no parameter, it will be defined as 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));
```



The precision of the DECFLOAT column or domain is stored in the system table `RDB$FIELDS`, in `RDB$FIELD_PRECISION`.

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	CEILING	EXP	FLOOR	LN
LOG	LOG10	POWER	SIGN	SQRT

The aggregate functions SUM, AVG, MAX and MIN work with DECFLOAT data, as do all of the statistics aggregates (including 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

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

TOTALORDER

performs an exact comparison on two DECFLOAT values

Detailed descriptions are in the DML chapter, in the topic [Special Functions for DECFLOAT](#).

Session Control Operator SET DECFLOAT

Firebird supports the session control operator SET DECFLOAT that allows to change the DECFLOAT data type properties. For details, see [Setting DECFLOAT Properties](#) in the [Management Statements](#) chapter.

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 as well as other extensions.

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

Security chapter.

Increased Precision for Exact Numeric Types

Alex Peshkov

Fixed types NUMERIC and DECIMAL can now be defined with up to 38 digits precision. Any value with precision higher than 18 digits will be stored as a 38-digit number. There's also an explicit INT128 integer data type with 128-bit (up to 38 decimal digits) storage.

Syntax rules

```
INT128
NUMERIC [( P [, S] )]
DECIMAL [( P [, S] )]
```

where P is precision ($P \leq 38$, previously limited to 18 digits), and the optional S is scale, as previously, i.e., the number of digits after the decimal separator.

Storage for $P \geq 19$ is a 128-bit signed integer.

Examples

1. Declare a variable of 25 digits to behave like an integer:

```
DECLARE VARIABLE VAR1 DECIMAL(25);
```

2. Define a column to accommodate up to 38 digits, with 19 decimal places:

```
CREATE TABLE TABLE1 (FIELD1 NUMERIC(38, 19));
```

3. Define a procedure with input parameter defined as 128-bit integer:

```
CREATE PROCEDURE PROC1 (PAR1 INT128) AS BEGIN END;
```



Numerics with precision less than 19 digits use SMALLINT, INTEGER, BIGINT or DOUBLE PRECISION as the base datatype, depending on the number of digits and SQL dialect. When precision is between 19 and 38 digits a 128-bit integer is used for internal storage, and the actual precision is always extended to the full 38 digits.

For complex calculations, those digits are cast internally to **DECFLOAT(34)**. The result of various mathematical operations, such as LOG(), EXP() and so on, and aggregate functions using a high precision numeric argument, will be DECFLOAT(34).

Standard Compliance for Data Type FLOAT

Mark Rotteveel

FLOAT data type was enhanced to support precision in binary digits as defined in the SQL:2016 specification. The approximate numeric types supported by Firebird are a 32-bit single precision and a 64-bit double precision binary floating-point type. These types are available with the following SQL standard type names:

- REAL : 32-bit single precision (synonym for FLOAT)
- FLOAT : 32-bit single precision
- FLOAT(*P*) where *P* is the precision of the significand in binary digits
 - $1 \leq P \leq 24$: 32-bit single precision (synonym for FLOAT)
 - $25 \leq P \leq 53$: 64-bit double precision (synonym for DOUBLE PRECISION)
- DOUBLE PRECISION : 64-bit double precision

In addition the following non-standard type names are supported:

- LONG FLOAT : 64-bit double precision (synonym for DOUBLE PRECISION)
- LONG FLOAT(*P*) where *P* is the precision of the significand in binary digits ($1 \leq P \leq 53$: synonym for DOUBLE PRECISION)

These non-standard type names are deprecated and they may be removed in a future version.

Compatibility Notes



1. REAL has been available as a synonym for FLOAT since Firebird 1.0 and even earlier, but was never documented.
2. Firebird 3.0 and earlier supported FLOAT(*P*) where *P* was the approximate precision in decimal digits, with $0 \leq P \leq 7$ mapped to 32-bit single precision and $P > 7$ mapped to 64-bit double precision. This syntax was never documented.
3. For *P* in FLOAT(*P*), the values $1 \leq P \leq 24$ are all treated as $P = 24$, values $25 \leq P \leq 53$ are all handled as $P = 53$.
4. Firebird 3.0 and earlier supported LONG FLOAT(*P*) where *P* was the approximate precision in decimal digits, where any value for *P* mapped to 64-bit double precision. This type name and syntax were never documented.
5. For *P* in LONG FLOAT(*P*), the values $1 \leq P \leq 53$ are all handled as $P = 53$.

Data Type Extensions for Time Zone Support

Adriano dos Santos Fernandes

The syntax for declaring the data types `TIMESTAMP` and `TIME` has been extended to include arguments defining whether the column, domain, parameter or variable should be defined with or without time zone adjustments, i.e.,

```
TIME [ { WITHOUT | WITH } TIME ZONE ]
```

```
TIMESTAMP [ { WITHOUT | WITH } TIME ZONE ]
```



For a summary of the effects of time zone support on existing data and application code, refer to [Changes in DDL and DML Due to Timezone Support](#) in the [Compatibility](#) chapter.

Storage

Data of types TIME/TIMESTAMP WITH TIME ZONE are stored respectively with the same storage as TIME/TIMESTAMP WITHOUT TIME ZONE plus two extra bytes for the time zone identifier or displacement.

- The time/timestamp parts, translated from the informed time zone, are stored in UTC.
- Time zone identifiers (from regions) are put directly in the time_zone bytes. They start from 65535, for the GMT code, decreasing as new time zones are added.

The time zone literals, together with their time zone identifiers, can be obtained from the RDB\$TIME_ZONES system table.

- Time zone displacements (+/- HH:MM) are encoded with $(\text{sign} * (\text{HH} * 60 + \text{MM})) + 1439$.

For example, a 00:00 displacement is encoded as $(1 * (0 * 60 + 0)) + 1439 = 1439$ and -02:00 as $(-1 * (2 * 60 + 0)) + 1439 = 1319$.

The default for both TIME and TIMESTAMP is WITHOUT TIME ZONE.

See also [Management Statements Pertaining to Time Zone Support](#) in the [Management Statements](#) chapter.

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 was introduced in Firebird 3 and has undergone a number of extensions in version 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>... ) ] <constraints>

<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($p, 0$) and DECIMAL($p, 0$) with $1 \leq p \leq 18$.
- 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 [Data Manipulation Language](#) 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 or negativestep, i.e., 1 or more or -1 or less:

```
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 ALTER TABLE statement syntax:

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



1. Changing the step value does not affect existing data.
2. It is not necessary to specify SET INCREMENT BY 1 for a new column, nor for one that has not been altered previously, as the default step is 1.

Implementation

Two columns have been added to 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.

Excess parameters in EXECUTE STATEMENT

Vlad Khorsun

Input parameters of the EXECUTE STATEMENT command may be prefixed by the EXCESS keyword. If EXCESS is specified, then the given parameter may be omitted from the query text.

Example

```
CREATE PROCEDURE P_EXCESS (A_ID INT, A_TRAN INT = NULL, A_CONN INT = NULL)
  RETURNS (ID INT, TRAN INT, CONN INT)
AS
DECLARE S VARCHAR(255);
DECLARE W VARCHAR(255) = '';
BEGIN
  S = 'SELECT * FROM TTT WHERE ID = :ID';

  IF (A_TRAN IS NOT NULL)
    THEN W = W || ' AND TRAN = :a';

  IF (A_CONN IS NOT NULL)
    THEN W = W || ' AND CONN = :b';

  IF (W <> '')
    THEN S = S || W;

  -- could raise error if TRAN or CONN is null
  -- FOR EXECUTE STATEMENT (:S) (a := :A_TRAN, b := A_CONN, id := A_ID)

  -- OK in all cases
  FOR EXECUTE STATEMENT (:S) (EXCESS a := :A_TRAN, EXCESS b := A_CONN, id := A_ID)
    INTO :ID, :TRAN, :CONN
    DO SUSPEND;
END
```

Replication Management

Dmitry Yemanov

Once replication is set up in the replication.conf configuration file, it can be enabled/disabled at runtime using the special extension to the ALTER DATABASE statement. Also, the replication set (i.e. tables to be replicated) can be customized using the extensions to the ALTER DATABASE and CREATE/ALTER TABLE statements.

Extended Syntax for Replication Management

```
ALTER DATABASE ... [<database replication management>]
```

```
CREATE TABLE tablename ... [<replication state>]
```

```
ALTER TABLE tablename ... [<replication state>]
```

```
<database replication management> ::=  
  <replication state> |  
  INCLUDE <replication set> TO PUBLICATION |  
  EXCLUDE <replication set> FROM PUBLICATION
```

```
<replication state> ::=  
  ENABLE PUBLICATION |  
  DISABLE PUBLICATION
```

```
<replication set> ::=  
  ALL |  
  TABLE tablename [, tablename ...]
```

Comments

- All replication management commands are DDL statements and thus effectively executed at the transaction commit time.
- ALTER DATABASE ENABLE PUBLICATION allows replication to begin (or continue) with the next transaction started after this transaction commits.
- ALTER DATABASE DISABLE PUBLICATION disables replication immediately after commit.
- If INCLUDE ALL TO PUBLICATION clause is used, then all tables created afterwards will also be replicated, unless overridden explicitly in the CREATE TABLE statement.
- If EXCLUDE ALL FROM PUBLICATION clause is used, then all tables created afterwards will not be replicated, unless overridden explicitly in the CREATE TABLE statement.

Chapter 10. 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 4.0.

Quick Links

- [Lateral Derived Tables](#)
- [DEFAULT Context Value for Inserting and Updating](#)
- [OVERRIDING Clause for IDENTITY Columns](#)
- [Frames for Window Functions](#)
- [Named Windows](#)
- [More Window Functions](#)
- [FILTER Clause for Aggregate Functions](#)
- [Optional AUTOCOMMIT for SET TRANSACTION](#)
- [Sharing Transaction Snapshots](#)
- [Expressions and Built-in Functions](#)
- [UDF Changes](#)
- [Improve Error Message for an Invalid Write Operation](#)
- [Improved Failure Messages for Expression Indexes](#)
- [RETURNING * Now Supported](#)

Lateral Derived Tables

Dmitry Yemanov

Tracker ticket [CORE-3435](#)

A derived table defined with the LATERAL keyword is called a lateral derived table. If a derived table is defined as lateral, then it is allowed to refer to other tables in the same FROM clause, but only those declared before it in the FROM clause.

The feature is defined in (SQL:2011): 7.6 <table reference> (Feature T491).

Examples

```

select dt.population, dt.city_name, c.country_name
from (select distinct country_name from cities) AS c,
     LATERAL (select first 1 city_name, population
              from cities
              where cities.country_name = c.country_name
              order by population desc) AS dt;
--
select salespeople.name,
       max_sale.amount,
       customer_of_max_sale.customer_name
from salespeople,
     LATERAL ( select max(amount) as amount from all_sales
              where all_sales.salesperson_id = salespeople.id
              ) as max_sale,
     LATERAL ( select customer_name from all_sales
              where all_sales.salesperson_id = salespeople.id
              and all_sales.amount = max_sale.amount
              ) as customer_of_max_sale;

```

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

```



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 version 2.1, Firebird has supported the `DEFAULT VALUES` clause. The two clauses are not the same. The `DEFAULT` clause applies to an individual column in the `VALUES` list, while `DEFAULT VALUES` applies to the row to be inserted as a whole. A statement like `INSERT INTO sometable DEFAULT VALUES` is equivalent to `INSERT INTO sometable VALUES (DEFAULT, ...)` with as many `DEFAULT` in the `VALUES` list as there are columns in *sometable*.

OVERRIDING Clause for IDENTITY Columns

Adriano dos Santos Fernandes

Tracker ticket [CORE-5463](#)

Identity columns defined with the `BY DEFAULT` attribute can be overridden 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

The `OVER` clause for Window functions in Firebird now supports not just the sub-clauses `PARTITION` and `ORDER` subclauses but also *frames* and *windows with names* that can be re-used in the same

query.

The pattern for Firebird 4 windowing syntax is as follows:

Syntax Pattern

```

<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 <window frame bound 1> AND <window frame bound 2>

<window frame bound 1> ::=
  {UNBOUNDED PRECEDING | <expr> PRECEDING | <expr> FOLLOWING | CURRENT ROW}

<window frame bound 2> ::=
  {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>
    <from clause>
    [<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.

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

Syntax Elements for 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 <window frame bound 1> AND <window frame bound 2>

<window frame bound 1> ::=
  {UNBOUNDED PRECEDING | <expr> PRECEDING | <expr> FOLLOWING | CURRENT ROW}

<window frame bound 2> ::=
  {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> ] ] )
```

The default frame is RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW which might produce strange results when a frame with these properties is operated on by FIRST_VALUE, NTH_VALUE or, particularly, LAST_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

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



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

In a query with multiple `SELECT` and `WINDOW` clauses (for example, with subqueries), the scope of the window name is confined to its query context. That means a window name from an inner context cannot be used in an outer context, nor vice versa. However, the same window name definition can be used independently in different contexts.

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 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 (?).

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

Simple Example

```

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

FILTER Clause for Aggregate Functions

Adriano dos Santos Fernandes

Tracker ticket [CORE-5768](#)

The FILTER clause extends aggregate functions (sum, avg, count, etc.) with an additional WHERE clause. The set returned is the aggregate of the rows that satisfy the conditions of both the main WHERE clause and those inside the FILTER clause(s).

It can be thought of as a shortcut for situations where one would use an aggregate function with some condition (decode, case, iif) to ignore some of the values that would be considered by the aggregation.

The clause can be used with any aggregate functions in aggregate or windowed (OVER) statements, but not with window-only functions like DENSE_RANK.

Example

Suppose you have a query where you want to count the number of status = 'A' and the number of status = 'E' as different columns. The old way to do it would be:

```
select count(decode(status, 'A', 1)) status_a,
       count(decode(status, 'E', 1)) status_e
from data;
```

The FILTER clause lets you express those conditions more explicitly:

```
select count(*) filter (where status = 'A') status_a,
       count(*) filter (where status = 'E') status_e
from data;
```



You can use more than one FILTER modifier in an aggregate query. You could, for example, use 12 filters on totals aggregating sales for a year to produce monthly figures for a pivot set

Syntax for FILTER Clauses

```
aggregate_function [FILTER (WHERE <condition>)] [OVER (<window>)]
```

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;
```

Sharing Transaction Snapshots

Adriano dos Santos Fernandes

Tracker ticket [CORE-6018](#)

With this feature it's possible to create parallel processes (using different attachments) reading consistent data from a database. For example, a backup process may create multiple threads reading data from the database in parallel. Or a web service may dispatch distributed sub-services doing some processing in parallel.

For this purpose, the SET TRANSACTION statement is extended with the SNAPSHOT [AT NUMBER snapshot_number] option. Alternatively, this feature can also be used via API, new Transaction Parameter Buffer item isc_tpb_at_snapshot_number <snapshot number length> snapshot number is

added for this purpose.

The *snapshot_number* from an active transaction can be obtained with `RDB$GET_CONTEXT('SYSTEM', 'SNAPSHOT_NUMBER')` in SQL or using the transaction information API call with `fb_info_tra_snapshot_number` information tag. Note that the *snapshot_number* passed to the new transaction must be a snapshot of a currently active transaction.

Example

```
SET TRANSACTION SNAPSHOT AT NUMBER 12345;
```

Expressions and Built-in Functions

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

New Functions and Expressions

Built-in functions and expressions added in Firebird 4.0.

Functions & Expressions for Timezone Operations

Adriano dos Santos Fernandes

Expressions and built-in functions for timezone operations.

AT Expression

Translates a time/timestamp value to its corresponding value in another time zone. If `LOCAL` is used, the value is converted to the session time zone.

Syntax

```
<at expr> ::= <expr> AT { TIME ZONE <time zone string> | LOCAL }
```

Examples

```
select time '12:00 GMT' at time zone '-03:00' from rdb$database;
select current_timestamp at time zone 'America/Sao_Paulo' from rdb$database;
select timestamp '2018-01-01 12:00 GMT' at local from rdb$database;
```

LOCALTIME Expression

Returns the current time as a `TIME WITHOUT TIME ZONE`, in the session time zone.

Example

```
select localtime from rdb$database;
```

LOCALTIMESTAMP Expression

Returns the current timestamp as a `TIMESTAMP WITHOUT TIME ZONE`, in the session time zone.

Example

```
select localtimestamp from rdb$database;
```

Two New Date/Time Functions

Adriano dos Santos Fernandes

FIRST_DAY

Returns a date or timestamp (as appropriate) with the first day of the year, month or week of a given date or timestamp value.

Syntax

```
FIRST_DAY( OF { YEAR | MONTH | WEEK } FROM <date_or_timestamp> )
```

- a. The first day of the week is considered as Sunday, following the same rules as for `EXTRACT` with `WEEKDAY`
- b. When a timestamp is passed the return value preserves the time part

Examples

```
select first_day(of month from current_date) from rdb$database;
select first_day(of year from current_timestamp) from rdb$database;
select first_day(of week from date '2017-11-01') from rdb$database;
```

LAST_DAY

Returns a date or timestamp (as appropriate) with the last day of the year, month or week of a given date or timestamp value.

Syntax

```
LAST_DAY( OF { YEAR | MONTH | WEEK } FROM <date_or_timestamp> )
```

- a. The last day of the week is considered as Saturday, following the same rules as for `EXTRACT` with `WEEKDAY`
- b. When a timestamp is passed the return value preserves the time part

Examples

```
select last_day(of month from current_date) from rdb$database;
select last_day(of year from current_timestamp) from rdb$database;
select last_day(of week from date '2017-11-01') from rdb$database;
```

Security 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`

A number of cryptographic functions were also added. See [Built-in Cryptographic Functions](#) in the [Security](#) chapter for syntax and usage details.

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.

There are almost no restrictions on the pattern. However, in almost all usages, sNaN will produce an exception, NULL will make the function return NULL, and so on.

```
SQL> select v, pic, quantize(v, pic) from examples;
```

V	PIC	QUANTIZE
3.16 0.001		3.160
3.16 0.01		3.16
3.16 0.1		3.2
3.16 1		3
3.16 1E+1		0E+1
-0.1 1		-0
0 1E+5		0E+5
316 0.1		316.0
316 1		316
316 1E+1		3.2E+2
316 1E+2		3E+2



If scaling like the example produces a result that would exceed the precision, the error “Decimal float invalid operation” is returned.

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
```

Function RDB\$GET_TRANSACTION_CN: Supporting Snapshots Based on Commit Order

Vlad Khorsun

See Tracker ticket [CORE-5921](#). For the background, see [Commit Order for Capturing the Database Snapshot](#) in the [Engine](#) chapter.

Returns the commit number (“CN”) of the supplied transaction. Result type is BIGINT.

Syntax

```
RDB$GET_TRANSACTION_CN( <transaction number> )
```

If the return value is greater than 1, it is the actual CN of the transaction if it was committed after the database was started.

The function can also return one of the following results, indicating the commit status of the transaction:

- 2 Transaction is dead (rolled back)
- 1 Transaction is in limbo
- 0 Transaction is still active
- 1 Transaction committed before the database started or less than the Oldest Interesting Transaction for the database
- NULL Transaction number supplied is NULL or greater than Next Transaction for the database

Note about the numerics



Internally, the engine uses unsigned 8-byte integers for commit numbers and unsigned 6-byte integers for transaction numbers. Thus, although the SQL language has no unsigned integers and `RDB$GET_TRANSACTION_CN` returns a signed `BIGINT`, a negative commit number will only be returned for the special values returned for uncommitted transactions.

Examples

```
select rdb$get_transaction_cn(current_transaction) from rdb$database;
select rdb$get_transaction_cn(123) from rdb$database;
```

Function MAKE_DBKEY

Vlad Khorsun

Creates a DBKEY value using relation name or ID, record number, and (optionally) logical number of data page and pointer page. Result type is `BINARY(8)`.

Syntax

```
MAKE_DBKEY( relation, recnum [, dpnum [, ppnum>]] )
```

Notes

1. If *relation* is a string expression or literal, then it is treated as a relation name, and the engine searches for the corresponding relation ID. The search is case-sensitive. In the case of string literal, relation ID is evaluated at query preparation time. In the case of expression, relation ID is evaluated at execution time. If the relation could not be found, then error `isc_relnotdef` is raised.
2. If *relation* is a numeric expression or literal, then it is treated as a relation ID and used “as is”, without verification against existing relations. If the argument value is negative or greater than

the maximum allowed relation ID (65535 currently), then NULL is returned.

3. Argument *recnum* represents an absolute record number in the relation (if the next arguments *dpnum* and *ppnum* are missing), or a record number relative to the first record, specified by the next arguments.
4. Argument *dpnum* is a logical number of data page in the relation (if the next argument *ppnum* is missing), or number of data page relative to the first data page addressed by the given *ppnum*.
5. Argument *ppnum* is a logical number of pointer page in the relation.
6. All numbers are zero-based. Maximum allowed value for *dpnum* and *ppnum* is 2^{32} (4294967296). If *dpnum* is specified, then *recnum* could be negative. If *dpnum* is missing and *recnum* is negative, then NULL is returned. If *ppnum* is specified, then *dpnum* could be negative. If *ppnum* is missing and *dpnum* is negative, then NULL is returned.
7. If any of specified arguments has NULL value, the result is also NULL.
8. Argument <relation> is described as INTEGER during query preparation, but it can be overridden by a client application as VARCHAR or CHAR. Arguments *recnum*, *dpnum* and *ppnum* are described as BIGINT.

Examples

```
-- (1) Select record using relation name
--      (note: relation name is uppercased)
select * from rdb$relations where rdb$db_key = make_dbkey('RDB$RELATIONS', 0)

-- (2) Select record using relation ID
select * from rdb$relations where rdb$db_key = make_dbkey(6, 0)

-- (3) Select all records physically residing on the first data page
select * from rdb$relations
  where rdb$db_key >= make_dbkey(6, 0, 0)
     and rdb$db_key < make_dbkey(6, 0, 1)

-- (4) Select all records physically residing on the first data page
--      of 6th pointer page
select * from SOMETABLE
  where rdb$db_key >= make_dbkey('SOMETABLE', 0, 0, 5)
     and rdb$db_key < make_dbkey('SOMETABLE', 0, 1, 5)
```

BASE64_ENCODE() and BASE64_DECODE()

Alex Peshkov

These two functions are for encoding and decoding input data between string and BASE64 representation. They operate with character strings and BLOBs. Considered useful when working with binary objects, for example with keys.

Syntax

```
BASE64_ENCODE( binary_data )  
BASE64_DECODE( base64_data )
```

Example

```
select base64_encode(public_key) from clients;
```

HEX_ENCODE() and HEX_DECODE()

Alex Peshkov

These two functions are for encoding and decoding input data between string and hexadecimal representation. They operate with character strings and BLOBs.

Syntax

```
HEX_ENCODE( binary_data )  
HEX_DECODE( hex_data )
```

Example

```
select hex_encode(binary_string) from clients;
```

CRYPT_HASH()

Alex Peshkov

Accepts an argument than can be a field, variable or expression of any type recognized by DSQL/PSQL and returns a cryptographic hash calculated from the input argument using the specified algorithm.

Syntax

```
CRYPT_HASH( <any value> USING <algorithm> )  
  
<algorithm> ::= { MD5 | SHA1 | SHA256 | SHA512 }
```

Example

```
select crypt_hash(job_title using sha256) from job;
```



- This function returns a VARBINARY string with the length depending on the specified algorithm.
- MD5 and SHA1 algorithms are not recommended due to known severe issues, these algorithms are provided for backward compatibility ONLY.

Changes to Built-in Functions and Expressions

Functions changed or extended in this release:

Changes Arising from Timezone Support

EXTRACT Expressions

Two new arguments have been added to the EXTRACT expression:

TIMEZONE_HOUR extracts the time zone hours displacement

TIMEZONE_MINUTE extracts the time zone minutes displacement

Example

```
select extract(timezone_hour from current_time) from rdb$database;
select extract(timezone_minute from current_timestamp) from rdb$database;
```

Changes in CURRENT_TIME and CURRENT_TIMESTAMP

In version 4.0, CURRENT_TIME and CURRENT_TIMESTAMP are changed: they now return TIME WITH TIME ZONE and TIMESTAMP WITH TIME ZONE, with the time zone set by the session time zone. In previous versions, CURRENT_TIME and CURRENT_TIMESTAMP returned the respective types according to the system clock, i.e. without any time zone.

To ease the transition, LOCALTIME and LOCALTIMESTAMP were added to versions 3.0.4 and 2.5.9, allowing developers to adjust application code without any functional changes, before migrating to Firebird 4.



See also [Changes in DDL and DML Due to Timezone Support](#) in the [Compatibility](#) chapter.

HASH()

Adriano dos Santos Fernandes

Tracker ticket [CORE-4436](#)

Returns a generic hash for the input argument using the specified algorithm.

Syntax

```
HASH( <any value> [ USING <algorithm> ] )

<algorithm> ::= { CRC32 }
```

The syntax with the optional USING clause is introduced in FB 4.0 and returns an integer of appropriate size. CRC32 algorithm implemented by Firebird uses polynomial 0x04C11DB7.

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 crc32) from y;
--
select hash(x) from y; -- not recommended
```

SUBSTRING()

A SUBSTRING start position smaller than 1 is now allowed. It has some properties that need to be taken into consideration for predicting the end of the string value returned.

Examples

```
select substring('abcdef' from 0) from rdb$database
-- Expected result: 'abcdef'

select substring('abcdef' from 0 for 2) from rdb$database
-- Expected result: 'a' (and NOT 'ab', because there is
-- "nothing" at position 0)

select substring('abcdef' from -5 for 2) from rdb$database
-- Expected result: ''
```

Those last two examples might not be what you expect. The for length is considered from the specified from start position, not the start of the string, so the string returned could be shorter than the specified length, or even empty.

UDF Changes

Many of the UDFs in previous versions became built-in functions. The UDF feature itself is heavily

deprecated in Firebird 4—see [External Functions \(UDFs\) Feature Deprecated](#) in the [Engine](#) chapter. Most of the remaining UDFs in the `ib_udf` and `fbudf` libraries now have analogues, either as UDRs in the new library `udf_compat` or as precompiled PSQL functions.

A script in the `/misc/upgrade/v4.0/` sub-directory of your installation provides an easy way to upgrade existing UDF declarations to the safe form that is available for each respective UDF. For details and instructions, see [Deprecation of External Functions \(UDFs\)](#) in the [Compatibility](#) chapter.

New UDR `GetExactTimestampUTC`

The new UDR `GetExactTimestampUTC`, in the `udf_compat` library, takes no input argument and returns the `TIMESTAMP WITH TIME ZONE` value at the moment the function is called.

The older function, `GetExactTimestamp` has been refactored as a stored function, returning, as before, the `TIMESTAMP WITHOUT TIME ZONE` value at the moment the function is called.

Miscellaneous DML Improvements

Improvements to behaviour and performance in DML include:

Improve Error Message for an Invalid Write Operation

Adriano dos Santos Fernandes

See Tracker ticket [CORE-5874](#).

When a read-only column is incorrectly targeted in an `UPDATE ... SET xxx` operation, the error message now provides the name of the affected column.

Improved Failure Messages for Expression Indexes

Adriano dos Santos Fernandes

Tracker ticket [CORE-5606](#)

If computation of an expression index fails, the exception message will now include the name of the index.

RETURNING * Now Supported

Adriano dos Santos Fernandes

Tracker ticket [CORE-3808](#)

The engine now supports `RETURNING *` syntax, and variants, to return a complete set of field values after committing a row that has been inserted, updated or deleted. The syntax and semantics of `RETURNING *` are similar to `SELECT *`.

Examples

```
INSERT INTO T1 (F1, F2) VALUES (:F1, :F2) RETURNING *
```

```
DELETE FROM T1 WHERE F1 = 1 RETURNING *
```

```
UPDATE T1 SET F2 = F2 * 10 RETURNING OLD.*, NEW.*
```

Chapter 11. 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.

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
  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 returns NULL.

The type of the return value depends on the [context](#).

Syntax

```

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



For descriptions of the context variables GDSCODE, SQLCODE and SQLSTATE, refer to the [Context Variables](#) topic in the *Firebird 2.5 Language Reference*.

Example of RDB\$ERROR

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

Allow Management Statements in PSQL Blocks

Adriano dos Santos Fernandes

See Tracker ticket [CORE-5887](#).

In prior Firebird versions, [management statements](#) were not allowed inside PSQL blocks. They were allowed only as top-level SQL statements, or as the top-level statement of an EXECUTE STATEMENT embedded in a PSQL block.

Now they can be used directly in PSQL blocks (triggers, procedures, EXECUTE BLOCK), which is especially helpful for applications that need some management statements to be issued at the start of a session, specifically in ON CONNECT triggers.

The management statements permitted for this usage are:

```
ALTER SESSION RESET
SET BIND
SET DECFLOAT ROUND
SET DECFLOAT TRAPS TO
SET ROLE
SET SESSION IDLE TIMEOUT
SET STATEMENT TIMEOUT
SET TIME_ZONE
SET TRUSTED ROLE
```

Example

```
create or alter trigger on_connect on connect
as
begin
    set bind of decfloat to double precision;
    set time zone 'America/Sao_Paulo';
end
```

Chapter 12. Monitoring & Command-line Utilities

Improvements and additions to the Firebird utilities continue.

Monitoring

Additions to MON\$ATTACHMENTS and MON\$STATEMENTS to report on timeouts and wire status. Refer to [Timeouts at Two levels](#) in the chapter [Changes in the Firebird Engine](#) for details.

New columns in the tables:

In MON\$DATABASE:

MON\$CRYPT_STATE	Current state of database encryption (not encrypted = 0, encrypted = 1, decryption in progress = 2, encryption in progress = 3)
MON\$GUID	Database GUID (persistent until restore / fixup)
MON\$FILE_ID	Unique ID of the database file at the filesystem level
MON\$NEXT_ATTACHMENT	Current value of the next attachment ID counter
MON\$NEXT_STATEMENT	Current value of the next statement ID counter
MON\$REPLICA_MODE	Database replica mode (not a replica = 0, read-only replica = 1, read-write replica = 2)

In MON\$ATTACHMENTS:

MON\$IDLE_TIMEOUT	Connection level idle timeout
MON\$IDLE_TIMER	Idle timer expiration time
MON\$STATEMENT_TIMEOUT	Connection level statement timeout
MON\$WIRE_COMPRESSED	Wire compression (enabled = 1, disabled = 0)
MON\$WIRE_ENCRYPTED	Wire encryption (enabled = 1, disabled = 0)
MON\$WIRE_CRYPT_PLUGIN	Name of the wire encryption plugin used by client

In MON\$STATEMENTS:

MON\$STATEMENT_TIMEOUT Connection level statement timeout

MON\$STATEMENT_TIMER Timeout timer expiration time

In **MON\$RECORD_STATS**:

MON\$RECORD_IMGC Number of records processed by the intermediate garbage collection

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>
```



“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

- a. 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}
```

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

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

c. Apply increment to the standby database:

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

Restore and Fixup for Replica Database

New (optional) command-line option `-sequence` (can be abbreviated to `-seq`) has been added for `-restore` and `-fixup` commands. It preserves the existing GUID and replication sequence of the original database (they are reset otherwise). This option should be used when creating a replica using `nbackup` tool, so that the asynchronous replication could automatically be continued from the point when a physical backup was performed on the primary side.

The syntax pattern is:

```
nbackup -R[ESTORE] <database file> <backup file> -SEQ[EUENCE]
nbackup -F[IXUP] <database file> -SEQ[EUENCE]
```

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.

Better transaction control

A new command has been introduced in *isql* to remember and reuse the last entered transaction parameters. The syntax is:

```
SET KEEP_TRAN_PARAMS [{ ON | OFF}]
```

When set to ON, *isql* keeps the complete SQL text of the following successful SET TRANSACTION statement and new transactions are started using the same SQL text (instead of the default CONCURRENCY WAIT mode). When set to OFF, *isql* starts new transactions as usual. Name KEEP_TRAN can be used as a shorthand for KEEP_TRAN_PARAMS.

Examples

```
-- check current value
SQL> SET;
...
Keep transaction params: OFF

-- toggle value
SQL> SET KEEP_TRAN;
SQL> SET;
...
Keep transaction params: ON
SET TRANSACTION

SQL>commit;

-- start new transaction, check KEEP_TRAN value and actual transaction's parameters
SQL>SET TRANSACTION READ COMMITTED WAIT;
SQL>SET;
...
Keep transaction params: ON
  SET TRANSACTION READ COMMITTED WAIT
SQL> SELECT RDB$GET_CONTEXT('SYSTEM', 'ISOLATION_LEVEL') FROM RDB$DATABASE;

RDB$GET_CONTEXT

=====
READ COMMITTED

SQL> commit;

-- start new transaction, ensure is have parameters as KEEP_TRAN value
SQL> SELECT RDB$GET_CONTEXT('SYSTEM', 'ISOLATION_LEVEL') FROM RDB$DATABASE;

RDB$GET_CONTEXT

=====
READ COMMITTED

-- disable KEEP_TRAN, current transaction is not changed
SQL> SET KEEP_TRAN OFF;
SQL> SELECT RDB$GET_CONTEXT('SYSTEM', 'ISOLATION_LEVEL') FROM RDB$DATABASE;
```

```

RDB$GET_CONTEXT

=====
READ COMMITTED

SQL> commit;

-- start new transaction, ensure is has default parameters (SNAPSHOT)
SQL> SELECT RDB$GET_CONTEXT('SYSTEM', 'ISOLATION_LEVEL') FROM RDB$DATABASE;

RDB$GET_CONTEXT

=====
SNAPSHOT

SQL> SET;
...
Keep transaction params: OFF

```

gbak

Backup and Restore with Encryption

Alex Peshkov

Tracker ticket [CORE-5808](#)

With an encrypted database, sooner or later it will need to be backed up and restored. It is not unreasonable to want the database backup to be encrypted as well. If the encryption key is delivered to the plug-in by some means that does not require input from the client application, it is not a big problem. However, if the server expects the key to be delivered from the client side, that could become a problem.

The introduction of keys to *gbak* in Firebird 4 provides a solution.

Prerequisites

A *keyholder plug-in* is required. This plug-in is able to load keys from some external source, such as a configuration file, and deliver them using the call

```
ICryptKeyCallback* IKeyHolderPlugin::chainHandle(IStatus* status)
```

That key holder and the *dbcrypt* plug-ins that work with it should be installed on the workstation that will be used to perform backups.

New Switches for Encrypted Backups & Restores

With the prerequisites in place, the following new switches are available for use. They are case-

insensitive.

Table 2. Switches for Encrypted Backups/Restores

Switch	What it Does
-KEYHOLDER	This is the main switch necessary for <i>gbak</i> to access an encrypted database.
-KEYNAME	Available to name the key explicitly, in place of the default key specified in the original database (when backing up) or in the backup file (when restoring).
-CRYPT	Available to name the plug-in to use to encrypt the backup file or restored database in place of the default plug-in. It can also be used in combination with the -KEYNAME switch to encrypt the backup of a non-encrypted database or to encrypt a database restored from a non-encrypted backup. See example below.
-ZIP	Only for a backup, to compress the backup file before encrypting it. The switch is necessary because the usual approach of compressing the backup file with some favoured compression routine after <i>gbak</i> , perhaps using pipe, does not work with encrypted backups because they are not compressible. The -ZIP switch is unnecessary for a restore because the format is detected automatically.

Usage and Examples

To back up an encrypted database do something like this:

```
gbak -b -keyholder MyKeyHolderPlugin host:dbname backup_file_name
```

The backup file will be encrypted using the same crypt plug-in and key that are used for database encryption. This ensures that it will not be any easier to steal data from your backup file than from the database.

To restore a database that was previously backed up encrypted:

```
gbak -c -keyholder MyKeyHolderPlugin backup_file_name host:dbname
```

The restored database will be encrypted using the same plug-in and key as the backup file. Using the backup example above, of course this means the same plug-in and key as the original database.



The database is first encrypted right after creation and only after the encryption data are restored into the header. This is a bit faster than a “restore-then-encrypt” approach but, mainly, it is to avoid having non-encrypted data on the server during the restore process.

The next example will either:

- restore the database from a backup file made using non-default Crypt and Keyholder plug-ins, using the same key name as was used for the backup; OR
- restore a non-encrypted backup as an encrypted database

```
gbak -c -keyholder MyKeyHolderPlugin -crypt MyDbCryptPlugin
      -keyname SomeKey non_encrypted_backup_file host:dbname
```

The restored database will be encrypted by MyDbCryptPlugin using SomeKey.

To make an encrypted backup of a non-encrypted database:

```
gbak -b -keyholder MyKeyHolderPlugin -crypt MyDbCryptPlugin
      -keyname SomeKey host:dbname encrypted_backup_file
```



Take note:

Attempts to create a non-encrypted backup of an encrypted database or to restore an encrypted backup to a non-encrypted database will fail. Such operations are intentionally disallowed to avoid foolish operator errors that would expose critical data in non-encrypted form.

To create a compressed, encrypted backup:

```
gbak -b -keyholder MyKeyHolderPlugin -zip host:dbname backup_file_name
```

The backup file will be compressed before being encrypted using the same crypt plug-in and same key that are used for the database encryption. ZLib is used to compress the backup file content and the appropriate record is added to its header.



Compressing Non-Encrypted Databases

The `-ZIP` switch is also available for compressing a non-encrypted database. It is important to understand that the format of a backup file thus created is not the same as one created by compressing a backup file with a utility such as 7Zip. It can be decompressed only by a *gbak* restore.

Enhanced Restore Performance

Alex Peshkov

Tracker ticket [CORE-5952](#)

The new Batch API is used to enhance the performance of restoring from backup.

Friendlier “-fix_fss_*” Messages

Alex Peshkov

Tracker ticket [CORE-5741](#)

The messages in the verbose output from a restore using the “-fix_fss_*” switches now use the word “adjusting” instead of “fixing”.

The same change was backported to version 3.0.5.

Ability to Backup/Restore Only Specified Tables

Dimitry Sibiryakov

Tracker ticket [CORE-5538](#)

A new command-line switch has been added to *gbak*: -INCLUDE(_DATA). Similarly to the existing -SKIP(_DATA) switch, it accepts one parameter which is a regular expression pattern used to match table names. If specified, it defines tables to be backed up or restored. The regular expression syntax used to match table names is the same as in SIMILAR TO Boolean expressions. Interaction between both switches is described in the following table.

Table 3. Interaction between -INCLUDE(_DATA) and -SKIP(_DATA) switches

	INCLUDE_DATA		
SKIP_DATA	NOT SET	MATCHED	NOT MATCHED
NOT SET	included	included	excluded
MATCHED	excluded	excluded	excluded
NOT MATCHED	included	included	excluded

gfix

Configuring and managing replication

The *gfix* repertoire now includes the new -replica switch for configuring and managing [Firebird replication](#). For more detail, see the topic [Creating a Replica Database](#).

It takes one of three arguments (case-insensitive):

read_only

Sets the database copy as a read-only replica, usually for a high-availability solution.

read_write

Sets the database copy as a read-write replica, e.g. for merging external changes into a database.

none

Converts the replica to a regular database, “switching off” replication to a read-write replica when conditions call for replication flow to be discontinued for some reason. Typically, it would be used to promote the replica to become the master database after a failure; or to make physical backup copies from the replica.

Chapter 13. Compatibility Issues

In this section are features and modifications that might affect the way you have installed and used Firebird in earlier releases.

SQL

Changes that may affect existing SQL code:

Deprecation of Legacy SQL Dialect 1

Starting with Firebird 4, *Dialect 1* is declared deprecated. Its support will be removed in future Firebird versions, with *Dialect 3* becoming the only dialect supported. Please consider migrating to *Dialect 3* as soon as possible.

Read Consistency for READ COMMITTED transactions Used By Default

Firebird 4 not only introduces [Read Consistency for Statements in Read-Committed Transactions](#), but also makes it a default mode for all READ COMMITTED transactions, regardless of their RECORD VERSION or NO RECORD VERSION properties. This is done to provide users with better behaviour — both compliant with the SQL specification and less conflict-prone. However, this new behaviour may also have unexpected side effects, please read the aforementioned chapter carefully. If specifics of the READ CONSISTENCY mode are undesirable for some reasons, the configuration setting `ReadConsistency` may be changed to allow the legacy behaviour. See more details about the [ReadConsistency](#) setting in the Configuration Additions and Changes chapter.

Deprecation of External Functions (UDFs)

Support for the external function (UDF) feature is deprecated in Firebird 4. Its immediate effect, out of the box, is that UDFs cannot be used with the default configuration, where the parameter `UdfAccess` in `firebird.conf` is set to `None`) and the UDF libraries `ib_udf` and `fbudf` are withdrawn from the distribution.

Most of the functions in those libraries were already deprecated in previous Firebird versions and replaced with built-in analogues. Safe replacements for a few of the remaining functions are now available, either in a new library of user-defined routines (UDRs) named `[lib]udf_compat.[dll/so/dylib]`, or as scripted conversions to PSQL stored functions. They are listed below; those marked with asterisks (*) are the UDR conversions.

ADDDAY()	*DOW()	ROUND()
ADDDAY2()	DPOWER()	RTRIM()
ADDDHOUR()	GETEXACTTIMESTAMP	*SDOW()
ADDMILLISECOND()	*GETEXACTTIMESTAMPUTC	SNULLIF()
ADDMINUTE()	I64NULLIF()	SNVL()
ADDMONTH()	I64NVL()	SRIGHT()
ADDSECOND()	I64ROUND()	STRING2BLOB()

ADDWEEK()	I64TRUNCATE()	STRLEN()
ADDDATE()	INULLIF()	SUBSTR()
*DIV()	INVL()	SUBSTRLEN()
DNULLIF()	ISLEAPYEAR()	TRUNCATE()
DNVL()	LTRIM()	*UDF_FRAC() or *FRAC()

The Firebird 4 distribution contains a script to migrate all (or any) of those UDF declarations. You can edit and extract from it to suit, if you wish, but you must keep the respective re-declarations and conversions intact as scripted.

The UDF Migration Script

The SQL script that you can use to upgrade the declarations for the UDFs listed above to the analogue UDRs or stored functions is located beneath the Firebird root, in `misc/upgrade/v4.0/udf_replace.sql`.

How to Work with the Script

During the restore of your Firebird 3 backup, *gbak* will issue warnings about any UDFs that are affected, but the restore will proceed. It would be useful to output the `-verbose` reporting to a file if you want a list of the affected function declarations. You will note items like

```
gbak: WARNING:function UDF_FRAC is not defined
gbak: WARNING:  module name or entrypoint could not be found
```

It means you have a UDF that is declared in the database but whose library is missing — which, of course, we know is true.

Running the Script

From the command shell:

```
isql -user sysdba -pas masterkey -i udf_replace.sql {your-database}
```



REMINDER

This script will have no effect on declarations for UDFs from third-party libraries!

What If You MUST Use a UDF?

In the short term, if you absolutely cannot avoid retaining the use of a UDF, you must configure the `UdfAccess` parameter to `Restrict <path-list>`. The default `<path-list>` points to the UDF sub-directory beneath the Firebird root. The (uncommented!) line in `firebird.conf` should be:

```
UdfAccess = Restrict UDF
```

The libraries `[lib]ib_udf.[dll/so/dylib]` and `[lib]fbudf.[dll/so/dylib]` that were distributed with Firebird 3 were tested to work with Firebird 4. No tests were done for any third-party or custom UDF libraries but, considering that nothing changed in the way Firebird works with UDFs, other than the default value for `UdfAccess`, they should also work.



The recommended long-term solution for any UDFs which you absolutely *must* use is to replace them with UDRs or stored functions.

Changes in DDL and DML Due to Timezone Support

Timezone support introduces some changes in DDL and DML which could affect compatibility with existing databases and applications.

Changes to Data Types `TIMESTAMP` and `TIME`

The syntax for declaring the data types `TIMESTAMP` and `TIME` has been extended to include arguments defining whether the column, domain, parameter or variable should be defined with or without time zone adjustments, i.e.:

```
TIME [ { WITHOUT | WITH } TIME ZONE ]
```

```
TIMESTAMP [ { WITHOUT | WITH } TIME ZONE ]
```

The default in both cases is `WITHOUT TIME ZONE`. If you are shifting migrated databases and/or applications to use the zoned date/time features, it is advisable to run reality checks on any calculations, computed fields, domains, query sets ordered or grouped by dates or timestamps, etc.

For more details, see [Data Type Extensions for Time Zone Support](#) in the `DDL` chapter.

`CURRENT_TIME` and `CURRENT_TIMESTAMP`

In version 4.0, `CURRENT_TIME` and `CURRENT_TIMESTAMP` are changed: they now return `TIME WITH TIME ZONE` and `TIMESTAMP WITH TIME ZONE`, with the time zone set by the session time zone. In previous versions, `CURRENT_TIME` and `CURRENT_TIMESTAMP` returned the respective types according to the system clock, i.e. without any time zone.

The expressions `LOCALTIMESTAMP` and `LOCALTIME` now replace the former functionality of `CURRENT_TIMESTAMP` and `CURRENT_TIME`, respectively.



Firebird 3.0.4 `LOCALTIME` and `LOCALTIMESTAMP`

To ease the transition, `LOCALTIME` and `LOCALTIMESTAMP` were added to versions 3.0.4 and 2.5.9, allowing developers to adjust application and PSQL code without any functional changes, before migrating to Firebird 4.

Prefixed Implicit Date/Time Literals Now Rejected

The literal date/time syntax (`DATE`, `TIME` or `TIMESTAMP` prefixing the quoted value) used together with the implicit date/time literal expressions (`'NOW'`, `'TODAY'`, etc.) was known to evaluate those

expressions in ways that would produce unexpected results, often undetected:

- In stored procedures and functions, evaluation would occur at compile time but not during the procedure or function call, storing the result in BLR and retrieving that stale value at runtime
- In DSQL, this style of usage in DSQL causes the evaluation to occur at prepare time, not at each iteration of the statement as would be expected with correct usage of the implicit date/time literals. The time difference between statement preparation and execution may be too small to discover the issue, particularly with 'NOW', which is a timestamp. Users could have been misled thinking the expression was evaluated at each iteration of the statement at runtime, when in fact it happened at prepare time.

If something like `TIMESTAMP 'NOW'` has been used in DSQL queries in application code or in migrated PSQL, there will be a compatibility issue with Firebird 4.

The behaviour was considered undesirable — the Firebird 4.0 engine and above will now reject such expressions in both PSQL and DSQL.

Example of such usage that will now be rejected:

```
..
DECLARE VARIABLE moment TIMESTAMP;
..
SELECT TIMESTAMP 'NOW' FROM RDB$DATABASE INTO :moment;
/* here, the variable :moment will 'frozen' as the timestamp at the moment
   the procedure or function was last compiled */
..
```

`TIMESTAMP '<constant>'` is for explicit date/time literals, e.g. `DATE '2019-02-20'` is legal. The implicit date/time literals, such as 'NOW' or 'YESTERDAY' are for use in expressions. Enforcement of the appropriate usage means that attempting to combine both becomes explicitly invalid syntax.

Existing code where usage does not break the rule remains unaffected. Both 'NOW' and `CAST('NOW' AS TIMESTAMP)` continue to work as before, as well as code that correctly uses the date/time prefixes with explicit literals, like `DATE '2019-02-20'`.

Starting Value of Sequences

Before Firebird 4.0 a sequence was created with its current value set to its starting value (or zero by default). So a sequence with starting value = 0 and increment = 1 starts at 1. While such a sequence has the same result in Firebird 4.0 (i.e. also starts at 1), the underlying implementation is different, thus making other cases different.

Now a sequence is created (or restarted) with its current value set to its starting value minus its increment. And the default starting value is changed to 1. Then a sequence with starting value = 100 and increment = 10 has its first `NEXT VALUE` equal to 100 now, while it was 110 before. Likewise, this sequence has its first `GEN_ID(SEQ, 1)` equal to 91 now, while it was 101 before.

INSERT ... RETURNING Now Requires a SELECT privilege

If some INSERT statement contains a RETURNING clause that refers columns of the underlying table, the appropriate SELECT privilege must be granted to the caller.

Utilities

Deprecation of QLI

Starting with Firebird 4, command-line utility *QLI* is declared deprecated. It will be removed in future Firebird versions.

Chapter 14. Bugs Fixed

Firebird 4.0 Release: Bug Fixes

The following bug-fixes since the Release Candidate 1 are noted:

Core Engine

(#6816) — Illegal output length in BASE64/HEX ENCODE/DECODE functions.

fixed by A. Peshkov

(#6812) — BASE64_ENCODE and HEX_ENCODE can exceed the maximum length for VARCHAR.

fixed by A. Peshkov

(#6805) — RDB\$TYPES has incorrect entries for RDB\$TYPE 28 and 29 in RDB\$TYPE_NAME column.

fixed by A. Peshkov

(#6804) — Assertion in tomcrypt when the key length for RC4 is too small.

fixed by A. Peshkov

(#6800) — Client config setting *DefaultTimeZone* is not passed to server when *isc_dpb_session_time_zone* is not set.

fixed by A. dos Santos Fernandes

(#6797) — Functions DECRYPT and RSA_DECRYPT return VARCHAR CHARACTER SET NONE instead of VARBINARY (VARCHAR CHARACTER SET OCTETS).

fixed by A. Peshkov

(#6795) — Replication gets stuck due to "Blob xxx.xx is not found for table xxx" error.

fixed by D. Yemanov

(#6790) — MON\$ATTACHMENTS.MON\$TIMESTAMP is incorrect when *DefaultTimeZone* is configured with time

zone different from the server's default.

fixed by A. dos Santos Fernandes

(#6787) — MON\$ATTACHMENTS.MON\$TIMESTAMP should use its session original time zone.

fixed by A. dos Santos Fernandes

(#6785) — Problem when restoring the database on Firebird 4 RC1.

fixed by V. Khorsun

(#6782) — Cannot get "records fetched" for selectable procedures in trace.

fixed by V. Khorsun

(#6778) — Inconsistent cursor-driven deletion.

fixed by D. Yemanov

(#6768) — Cannot restore backup on a raw device (*error during "open O_CREAT" operation for file "/dev/sda1"*).

fixed by A. Peshkov

(#6761) — Hash join cannot match records using some TIME_ZONE / DECFLOAT keys.

fixed by D. Yemanov

(#6759) — Results of concatenation with blob has no info about collation of source columns (which are declared with such info).

fixed by V. Khorsun

(#6758) — COMPUTED BY column loses charset and collate of source field <F> when <F> is either of type BLOB or VARCHAR casted to BLOB.

fixed by V. Khorsun

(#6756) — Error "no current record for fetch operation" when sorting by an international string.

fixed by D. Yemanov

(#6750) — CAST of Infinity values to FLOAT doesn't work.

fixed by A. Peshkov

(#6749) — Error "Invalid time zone (+08). Falling back to displacement" in firebird.log.

fixed by A. dos Santos Fernandes

(#6747) — Wrong message when connecting to tiny trash database file.

fixed by A. Peshkov

(#6746) — Regression: CREATE DATABASE fails with 'Token unknown' error when DB name is enclosed in double quotes and 'DEFAULT CHARACTER SET' is specified after DB name.

fixed by A. dos Santos Fernandes

(#6734) — Provide same results for date arithmetics when date is changed by values near +/- max(bigint).

fixed by A. Peshkov

(#6733) — Attempt to create database with page_size >= 65536 makes DB with actual page size = 4KB or 8KB, but not 32KB as it should.

fixed by A. Peshkov

(#6727) — Synchronous replication to localhost hangs on disconnect.

fixed by D. Yemanov

(#6724) — Inconsistent translation "string → timestamp → string → timestamp" in Dialect 1.

fixed by A. dos Santos Fernandes

(#6719)— User without ALTER ANY ROLE privilege can use COMMENT ON ROLE.

fixed by A. Peshkov

(#6717)— FETCH ABSOLUTE and RELATIVE beyond bounds of cursor should always position immediately before-first or after-last.

fixed by D. Yemanov

(#6716)— FETCH RELATIVE has an off by one error for the first row.

fixed by D. Yemanov

(#6710)— COMMENT ON USER can only apply comment on user defined by the default user manager plugin.

fixed by A. Peshkov

(#6700)— Wire compression causes sporadic "Error reading data from the connection" errors.

fixed by A. Peshkov

(#6698)— Comments before the first line of code are removed.

fixed by A. dos Santos Fernandes

(#3810)— Wrong or missing IS NULL optimization.

fixed by V. Khorsun

(#3106)— Many indexed reads in a compound index with NULLs.

fixed by V. Khorsun

(#2469)— Stored procedure recursively called by calculated field fails after reconnect.

fixed by V. Khorsun

Server Crashes/Hang-ups

(#6808) — Segfault in ENCRYPT/DECRYPT functions when their first argument is *NULL*.

fixed by A. Peshkov

(#6781) — Crashing (due to UDF exception) process hangs.

fixed by V. Khorsun

(#6777) — AV when the engine shuts down and cancels an attachment waiting in the lock manager.

fixed by V. Khorsun

(#6766) — Replication plugin can crash the engine by returning *nullptr* from *startTransaction()* method.

fixed by D. Yemanov

(#6754) — Connect to database that contains broken pages can lead to a server crash.

fixed by A. Peshkov

(#6753) — AV in the engine when *StatementTimeout* is active for user statement and some internal DSQL statement was executed as part of overall execution process.

fixed by V. Khorsun

(#6752) — Segfaults in *fbclient* when receiving invalid / unexpected data from the server.

fixed by A. Peshkov

(#6751) — Various segfaults in *fbclient*.

fixed by A. Peshkov

(#6738) — Segfault when *GFIX* requests for database page buffer more memory than available from OS.

fixed by A. Peshkov

(#6731)— Segfault when shutting down database which got encrypted by another process.

fixed by A. Peshkov

(#6708)— Rare race condition in Plugin Manager could lead to the server crash.

fixed by V. Khorsun

(#6265)— Segfault when using expression index with complex expression.

fixed by V. Khorsun

(#5784)— When 32-bit and 64-bit Firebird 3 servers run on a single Windows machine concurrently, Firebird services freeze several minutes after first disconnect.

fixed by V. Khorsun

API/Remote Interface

(#6718)— Event delivery could be missed when local (XNET) protocol is used.

fixed by V. Khorsun

(#6679)— *CLOOP* envelopes are wrong regarding IStatus.

fixed by A. Peshkov

Build Issues

(#6780)— `firebird.msg` is missing in Firebird Android builds.

fixed by A. Peshkov

(#6745)— Protect included tomcrypt library from being overwritten by a system package.

fixed by A. Peshkov

Utilities

isql

(#6796) — Buffer overflow when padding line with national characters causes *ISQL* to crash.

fixed by A. dos Santos Fernandes

gbak

(#6709) — *GBAK* discards replica mode during backup/restore.

fixed by D. Yemanov

gstat

(#6729) — Regression: *GSTAT* with switch *-t* executed via services fails with "found unknown switch" error.

fixed by A. Peshkov

Firebird 4.0 Release Candidate 1: Bug Fixes

The following bug-fixes since the Beta 2 release are noted:

Core Engine

(*CORE-6475*) — Memory leak when running *EXECUTE STATEMENT* with named parameters.

fixed by V. Khorsun

(*CORE-6472*) — Wrong byte order for *UUIDs* reported by *GSTAT* and monitoring tables.

fixed by D. Sibiryakov

(*CORE-6460*) — Incorrect query result when using named window.

fixed by V. Khorsun

(*CORE-6453*) — *EXECUTE STATEMENT* fails on FB 4.x if containing time/timestamp with time zone parameters.

fixed by A. dos Santos Fernandes

([CORE-6447](#))—Unexpectedly different text of message for parameterized expression starting from second run. Same fix was backported to Firebird 3.0.8.

fixed by V. Khorsun

([CORE-6441](#))—Srp plugin keeps connection after database has been removed for ~10 seconds. Same fix was backported to Firebird 3.0.8.

fixed by A. Peshkov

([CORE-6440](#))—Expression indexes containing COALESCE inside cannot be matched by the optimizer after migration from v2.5 to v3.0. Same fix was backported to Firebird 3.0.8.

fixed by D. Yemanov

([CORE-6437](#))—GFIX cannot set big value for page buffers. Same fix was backported to Firebird 3.0.8.

fixed by V. Khorsun

([CORE-6427](#))—Whitespace as date separator causes conversion error.

fixed by A. dos Santos Fernandes

([CORE-6421](#))—Parameter in offset expression in LAG, LEAD, NTH_VALUE window functions requires explicit cast to BIGINT or INTEGER.

fixed by A. dos Santos Fernandes

([CORE-6419](#))—Truncation of strings to put in MON\$ tables do not work correctly.

fixed by A. dos Santos Fernandes

([CORE-6415](#))—Error "malformed string" is raised instead of "expected: N, actual: M" when UTF-8 charset is used and default value is longer than the column length.

fixed by A. dos Santos Fernandes

([CORE-6414](#)) — Error "expected length N, actual M" contains wrong value of M when UTF-8 charset is used in the field declaration.

fixed by A. dos Santos Fernandes

([CORE-6408](#)) — RETURNING clause in the MERGE statement cannot reference column in aliased target table using qualified reference (alias.column) if DELETE action present. Same fix was backported to Firebird 3.0.8.

fixed by A. dos Santos Fernandes

([CORE-6403](#)) — Some PSQL statements may lead to exceptions report wrong line/column.

fixed by A. dos Santos Fernandes

([CORE-6398](#)) — Error converting string with hex representation of INTEGER to SMALLINT.

fixed by A. Peshkov

([CORE-6397](#)) — Message length error with COALESCE and TIME / TIMESTAMP WITHOUT TIME ZONE and WITH TIME ZONE.

fixed by A. dos Santos Fernandes

([CORE-6389](#)) — Using binary string literal to assign to user-defined blob sub-types yield conversion error.

fixed by A. dos Santos Fernandes

([CORE-6386](#)) — ALTER SEQUENCE RESTART WITH <n> should not change the initial sequence START value.

fixed by A. dos Santos Fernandes

([CORE-6385](#)) — Wrong line and column information after IF statement.

fixed by A. dos Santos Fernandes

([CORE-6379](#)) — Bugcheck 179 (decompression overran buffer).

fixed by V. Khorsun

([CORE-6376](#)) — IDENTITY column with explicit START WITH or INCREMENT BY starts with wrong value.

fixed by A. dos Santos Fernandes

([CORE-6357](#)) — LEAD() and LAG() do not allow to specify 3rd argument of INT128 datatype.

fixed by A. Peshkov

([CORE-6356](#)) — ROUND() does not allow second argument ≥ 1 when its first argument is more than MAX_BIGINT / 10.

fixed by A. Peshkov

([CORE-6355](#)) — TRUNC() does not accept second argument = -128 (but shows it as required boundary in error message).

fixed by A. Peshkov

([CORE-6353](#)) — INT128 data type has problems with some PSQL objects.

fixed by A. Peshkov

([CORE-6344](#)) — Invalid return type for functions with INT128 / NUMERIC(38) argument.

fixed by A. Peshkov

([CORE-6337](#)) — Sub-type information is lost when calculating arithmetic expressions.

fixed by A. Peshkov

([CORE-6336](#)) — Error "Implementation of text subtype <NNNN> not located" on attempt to use some collations defined in fbintl.conf.

fixed by A. dos Santos Fernandes

([CORE-6335](#)) — INSERT ... RETURNING does not require a SELECT privilege.

fixed by D. Yemanov

([CORE-6328](#))—FB4 Beta 2 may still be using the current date for TIME WITH TIME ZONE and extended wire protocol.

fixed by A. dos Santos Fernandes

([CORE-6325](#))—NTILE/RANK/PERCENT_RANK may cause problems in big/complex statements.

fixed by A. dos Santos Fernandes

([CORE-6318](#))—CAST('NOW' as TIME) raises a conversion error.

fixed by A. dos Santos Fernandes

([CORE-6316](#))—Unable to specify new 32KB page size in CREATE DATABASE statement.

fixed by A. Peshkov

([CORE-6303](#))—Error writing to TIMESTAMP / TIME WITH TIME ZONE array.

fixed by A. Peshkov

([CORE-6302](#))—Error writing an array of NUMERIC(24,6) to the database.

fixed by A. Peshkov

([CORE-6084](#))—CREATE SEQUENCE START WITH has wrong initial value.

fixed by A. dos Santos Fernandes

([CORE-6023](#))—FB4 is unable to overwrite older ODS database.

fixed by A. Peshkov

([CORE-5838](#))—Rotated trace files are locked by the engine.

fixed by V. Khorsun

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

fixed by D. Yemanov

([CORE-2274](#)) — MERGE has a non-standard behaviour, accepts multiple matches.

fixed by V. Khorsun

Server Crashes/Hang-ups

([CORE-6450](#)) — Races in the security databases cache could lead to the server crash. Same fix was backported to Firebird 3.0.8.

fixed by A. Peshkov

([CORE-6433](#)) — Server could crash during a daily maintenance / set statistics index. Same fix was backported to Firebird 3.0.8.

fixed by A. Peshkov

([CORE-6412](#)) — Firebird was freezing when trying to manage users via triggers. Same fix was backported to Firebird 3.0.8.

fixed by A. Peshkov

([CORE-6387](#)) — Client process was aborting due to bugs inside the ChaCha plugin.

fixed by A. Peshkov

API/Remote Interface

([CORE-6432](#)) — Possible buffer overflow in client library in Attachment::getInfo() call. Same fix was backported to Firebird 3.0.8.

fixed by A. Peshkov

([CORE-6426](#)) — Assertion when the batch is executed without a BLOB field.

fixed by A. Peshkov

(*CORE-6425*)—Exception in client library in `IAttachment::createBatch()`.

fixed by A. Peshkov

Build Issues

(*CORE-6305*)—Android port build failure.

fixed by A. Peshkov

Utilities

isql

(*CORE-6438*)—Bad headers when text columns has ≥ 80 characters.

fixed by A. dos Santos Fernandes

gbak

(*CORE-6377*)—Unable to restore database with tables using `GENERATED ALWAYS AS IDENTITY` columns.

fixed by A. Peshkov

Firebird 4.0 Beta 2 Release: Bug Fixes

The following bug-fixes since the Beta 1 release are noted:

Core Engine

(*CORE-6290*)—Hex number used at the end of statement could read invalid memory and produce wrong values or exceptions. Same fix was backported to Firebird 3.0.6.

fixed by A. dos Santos Fernandes

(*CORE-6282*)—Data type of `MON$ATTACHMENTS.MON$IDLE_TIMER` and `MON$STATEMENTS.MON$STATEMENT_TIMER` was defined as `TIMESTAMP WITHOUT TIME ZONE`, now it's changed to `TIMESTAMP WITH TIME ZONE`.

fixed by A. dos Santos Fernandes

([CORE-6281](#))—Invalid timestamp errors could happen when working with the `RDB$TIME_ZONE_UTIL.TRANSITIONS` procedure.

fixed by A. dos Santos Fernandes

([CORE-6280](#))—`MERGE` statement could lose parameters in the “`WHEN [NOT] MATCHED`” clause that will never be matched. This could also cause server crashes in some situations. Same fix was backported to Firebird 3.0.6.

fixed by A. dos Santos Fernandes

([CORE-6272](#))—Failed attach to a database was not traced.

fixed by A. Peshkov

([CORE-6266](#))—Deleting records from `MON$ATTACHMENTS` using the `ORDER BY` clause didn’t close the corresponding attachments. Same fix was backported to Firebird 3.0.6.

fixed by D. Yemanov

([CORE-6251](#))—`UNIQUE CONSTRAINT` violation could be possible. Same fix was backported to Firebird 3.0.6.

fixed by V. Khorsun

([CORE-6250](#))—Signature mismatch error could be raised when creating package body on identical packaged procedure header. Same fix was backported to Firebird 3.0.6.

fixed by A. dos Santos Fernandes

([CORE-6248](#))—A number of errors could happen when database name is longer than 255 characters.

fixed by A. Peshkov

([CORE-6243](#))—v4 Beta 1 regression happened: the engine rejects `POSITION` element of the `SQL:2003 CREATE TRIGGER` syntax.

fixed by A. dos Santos Fernandes

([CORE-6241](#))— Values greater than number of days between 01.01.0001 and 31.12.9999 (=3652058) could be added or subtracted from DATE.

fixed by A. dos Santos Fernandes

([CORE-6238](#))— DECFLOAT: subtraction (“Num1 - Num2”) would lead to the “Decimal float overflow” error if Num2 is specified in scientific notation and is less than max double (1.7976931348623157e308).

fixed by A. Peshkov

([CORE-6236](#))— RDB\$TIME_ZONE_UTIL package had wrong privilege defined for PUBLIC.

fixed by A. dos Santos Fernandes, D. Yemanov

([CORE-6230](#))— It was impossible to connect to a database if security.db reference was removed from databases.conf. Same fix was backported to Firebird 3.0.6.

fixed by A. Peshkov

([CORE-6221](#))— Incorrect implementation of allocFunc() for zlib1: memory leak was possible. Same fix was backported to Firebird 3.0.6.

fixed by A. Peshkov

([CORE-6214](#))— tzdata database version was outdated and required an update.

fixed by A. dos Santos Fernandes

([CORE-6206](#))— VARCHAR of insufficient length was used for command SET BIND OF DECFLOAT TO VARCHAR.

fixed by V. Khorsun

([CORE-6205](#))— Improper error was raised for UNION DISTINCT with more than 255 columns.

fixed by A. dos Santos Fernandes

([CORE-6186](#))— Original contents of the column used with ENCRYPT() looked as distorted after this

call.

fixed by A. Peshkov

([CORE-6181](#))— Usage of “SET DECFLOAT BIND BIGINT, n” with result of 11+ digits, would fail with the “Decimal float invalid operation” error.

fixed by A. Peshkov

([CORE-6166](#))— Some problems could appear for long object names (> 255 bytes).

fixed by A. dos Santos Fernandes

([CORE-6160](#))— SUBSTRING of non-text/blob was described to return NONE character set in DSQL.

fixed by A. dos Santos Fernandes

([CORE-6159](#))— SUBSTRING SIMILAR was described with wrong data type in DSQL.

fixed by A. dos Santos Fernandes

([CORE-6110](#))— 64-bit transaction IDs were not stored properly inside the status vector.

fixed by I. Eremin

([CORE-6080](#))— Attempt to drop an existing user could randomly fail with error “336723990 : record not found for user”.

fixed by V. Khorsun

([CORE-6046](#))— Incorrect time zone parsing could read garbage in memory.

fixed by A. dos Santos Fernandes

([CORE-6034](#))— The original time zone was not set to the current time zone at the routine invocation.

fixed by A. dos Santos Fernandes

([CORE-6033](#)) — SUBSTRING(CURRENT_TIMESTAMP ...) would fail with a “string truncation” error.

fixed by A. dos Santos Fernandes

([CORE-5957](#)) — Adding a numeric quantifier as a bound for repetition of expression inside SIMILAR TO could lead to an empty resultset.

fixed by A. dos Santos Fernandes

([CORE-5931](#)) — SIMILAR TO did not return the result when an invalid pattern was used.

fixed by A. dos Santos Fernandes

([CORE-5892](#)) — SQL SECURITY DEFINER context was not properly evaluated for monitoring tables.

fixed by R. Simakov

([CORE-5697](#)) — Conversion from numeric literals to DECFLOAT would add the precision that is not originally present.

fixed by A. Peshkov

([CORE-5696](#)) — Conversion from zero numeric literals to DECFLOAT would lead to the incorrect result.

fixed by A. Peshkov

([CORE-5664](#)) — SIMILAR TO was substantially (500-700x) slower than LIKE on trivial pattern matches with VARCHAR data.

fixed by A. dos Santos Fernandes

([CORE-4874](#)) — Server could perform a SIMILAR TO matching infinitely.

fixed by A. dos Santos Fernandes

([CORE-4739](#)) — Accent insensitive comparison: diacritical letters with diagonal crossing stroke failed for non-equality conditions with their non-accented forms.

fixed by A. dos Santos Fernandes

([CORE-3858](#)) — Very poor performance of `SIMILAR TO` for some arguments.

fixed by A. dos Santos Fernandes

([CORE-3380](#)) — It was possible to read from the newly created BLOB. It's prohibited now.

fixed by A. dos Santos Fernandes

Server Crashes/Hang-ups

([CORE-6254](#)) — Server could crash when using `SET TRANSACTION` and `ON TRANSACTION START` trigger uses `EXECUTE STATEMENT` against current transaction. Same fix was backported to Firebird 3.0.6.

fixed by V. Khorsun

([CORE-6253](#)) — Locked `fb_lock` file could cause a server crash. Same fix was backported to Firebird 3.0.6.

fixed by V. Khorsun

([CORE-6251](#)) — Server would crash when built-in function `LEFT` or `RIGHT` is missing its 2nd argument. Same fix was backported to Firebird 3.0.6.

fixed by A. dos Santos Fernandes

([CORE-6231](#)) — Server would crash during shutdown of XNET connection to a local database when events have been registered. Same fix was backported to Firebird 3.0.6.

fixed by V. Khorsun

([CORE-6224](#)) — Server could crash due to double destruction of the `rem_port` object. Same fix was backported to Firebird 3.0.6.

fixed by D. Kovalenko, A. Peshkov

([CORE-6218](#)) — `COUNT(DISTINCT DECFLOAT_FIELD)` could cause the server to crash when there are duplicate values in this field.

fixed by A. Peshkov

([CORE-6217](#)) — Dangerous (possibly leading to a crash) work with pointer: `delete ptr; ptr=new ;`.

fixed by D. Kovalenko, A. Peshkov

([CORE-5972](#)) — External engine trigger would crash the server if the table has computed fields. Same fix was backported to Firebird 3.0.6.

fixed by A. dos Santos Fernandes

([CORE-4893](#)) — `SIMILAR TO` would cause a server crash when matching a blob with size >2GB to a string literal.

fixed by A. dos Santos Fernandes

API/Remote Interface

([CORE-6283](#)) — Result of `isNullable()` in message metadata, returned by metadata builder, did not match datatype set by `setType()` in metadata builder. Same fix was backported to Firebird 3.0.6.

fixed by A. Peshkov

([CORE-6227](#)) — `isc_info_svc_user_dbpath` was always returning an alias of the main security database. Same fix was backported to Firebird 3.0.6.

fixed by A. Peshkov

([CORE-6212](#)) — Authentication plugin on the server could read garbage data from the client instead of the empty packet.

fixed by A. Peshkov

([CORE-6207](#)) — It was impossible to compile `Firebird.pas` with FPC.

fixed by A. Peshkov

Build Issues

([CORE-6174](#)) — `ibase.h` was missing from the nightly builds.

fixed by A. dos Santos Fernandes

([CORE-6170](#))— Installation on CentOS 8 failed because of the mismatched version of LibTomMath and LibNCurses libraries.

fixed by A. Peshkov

([CORE-6061](#))— It was impossible to build the server with the `--with-builtin-tommath` option.

fixed by A. Peshkov

([CORE-6056](#))— Overflow warnings appeared when building some collations.

fixed by A. dos Santos Fernandes

([CORE-6019](#))— Wire compression did not work without the MSVC 2010 runtime package installed.

fixed by V. Khorsun

([CORE-5691](#))— File description of the Firebird executables was not specific.

fixed by V. Khorsun

([CORE-5445](#))— Installation failed on Debian Stretch/Testing due to incorrect version of the LibTomMath library.

fixed by A. Peshkov

Utilities

isql

([CORE-6262](#))— `SHOW DOMAIN/TABLE` did not display the character set of system objects.

fixed by A. dos Santos Fernandes

([CORE-6260](#))— Warnings were not always displayed in ISQL. Same fix was backported to Firebird 3.0.6.

fixed by A. Peshkov

([CORE-6211](#))—Command “`isql -X`” could not extract the ROLE name when using a multi-byte charset for the connection.

fixed by A. dos Santos Fernandes

([CORE-6116](#))—Metadata script extracted with ISQL from a database restored from a v2.5 backup was invalid if some table has COMPUTED BY fields. Same fix was backported to Firebird 3.0.6.

fixed by A. dos Santos Fernandes

([CORE-6044](#))—Some issues were noticed due to the increased SQL identifier length.

fixed by A. dos Santos Fernandes

gbak

([CORE-6265](#))—Existing mapping rules were removed by the backup/restore cycle. Same fix was backported to Firebird 3.0.6.

fixed by A. Peshkov

([CORE-6233](#))—Wrong dependencies of stored function on view were created after backup/restore. Same fix was backported to Firebird 3.0.6.

fixed by A. dos Santos Fernandes

([CORE-6208](#))—CREATE DATABASE permission would disappear from security database after the backup/restore cycle. Same fix was backported to Firebird 3.0.6.

fixed by A. Peshkov

([CORE-6130](#))—Creating backup to STDOUT using the service manager was broken. Same fix was backported to Firebird 3.0.6.

fixed by A. Peshkov

([CORE-6071](#))—Restoring an encrypted backup of a SQL dialect 1 database would fail.

fixed by A. Peshkov

(*CORE-5976*)—GBAK multi-database file restore used wrong minimum number of pages for the first database file.

fixed by M. Rotteveel

(*CORE-2251*)—GBAK doesn't return the error code in some cases. Same fix was backported to Firebird 3.0.6.

fixed by A. Peshkov

gfix

(*CORE-5364*)—`gfix -online normal` did not raise an error when there was another SYSDBA-owned session open. Same fix was backported to Firebird 3.0.6.

fixed by A. Peshkov

Firebird 4.0 Beta 1 Release: Bug Fixes

The following bug-fixes since the Alpha release are noted:

Core Engine

(*CORE-5986*)—Evaluation of `NULL IS [NOT] FALSE | TRUE` was incorrect. Same fix was backported to Firebird 3.0.5.

fixed by A. dos Santos Fernandes

(*CORE-5985*)—Regression: `ROLE` was not being passed to ES/EDS: specifying it in the statement was ignored. Same fix was backported to Firebird 3.0.5.

fixed by A. Peshkov

(*CORE-5982*)—An error involving read permission for a BLOB field was being thrown when the BLOB was an input or output parameter for a procedure. Same fix was backported to Firebird 3.0.5.

fixed by D. Starodubov

(*CORE-5974*)—`SELECT DISTINCT` with a `decfloat/timezone/collated` column was producing wrong results.

fixed by A. dos Santos Fernandes

([CORE-5973](#))—Improvement: Fixed-point overflow in a DOUBLE PRECISION value converted from DECFLOAT is now handled properly.

fixed by A. Peshkov

([CORE-5965](#))—The optimizer was choosing a less efficient plan in FB4 and FB3 than the FB2.5 optimizer. Same fix was backported to Firebird 3.0.5.

fixed by D. Yemanov

([CORE-5959](#))—Firebird would return the wrong time after a change of time zone. Same fix was backported to Firebird 3.0.5.

fixed by V. Khorsun

([CORE-5950](#))—Deadlock could occur when attaching to a bugchecked database. Same fix was backported to Firebird 3.0.5.

fixed by A. Peshkov

([CORE-5949](#))—Bugcheck could happen when a read-only database with non-zero linger was set to read-write mode. Same fix was backported to Firebird 3.0.5.

fixed by V. Khorsun

([CORE-5935](#))—Bugcheck 165 (cannot find TIP page). Same fix was backported to Firebird 3.0.5.

fixed by V. Khorsun

([CORE-5930](#))—Bugcheck with message “incorrect snapshot deallocation - too few slots”.

fixed by V. Khorsun

([CORE-5918](#))—Memory pool statistics were inaccurate. Same fix was backported to Firebird 3.0.5.

fixed by A. Peshkov

([CORE-5896](#)) — A NOT NULL constraint was not being synchronized after the column was renamed.

fixed by A. dos Santos Fernandes

([CORE-5785](#)) — An ORDER BY clause on a compound index could disable usage of other indices. Same fix was backported to Firebird 3.0.5.

fixed by D. Yemanov

([CORE-5871](#)) — Incorrect caching of the result of a subquery result in a procedure call from a SELECT query.

fixed by A. dos Santos Fernandes

([CORE-5862](#)) — RDB\$CHARACTER_LENGTH in RDB\$FIELDS was not being populated when the column was a computed VARCHAR without an explicit type.

fixed by A. dos Santos Fernandes

([CORE-5750](#)) — Date-time parsing needed strengthening.

fixed by A. dos Santos Fernandes

([CORE-5728](#)) — The field subtype of DEC_FIXED columns was not returned by isc_info_sql_sub_type.

fixed by A. Peshkov

([CORE-5726](#)) — The error message when inserting a value exceeding the maximum value of DEC_FIXED decimal was unclear.

fixed by A. Peshkov

([CORE-5717](#)) — The literal date/time prefix syntax (DATE, TIME or TIMESTAMP prefix before the quoted value) used together with the implicit date/time literal expressions ('NOW', 'TODAY', etc.) was known to evaluate those expressions in ways that would produce unexpected results, often undetected. This behaviour was considered undesirable — the Firebird 4.0 engine and above will now reject them everywhere.

For details, see [Prefixed Implicit Date/Time Literals Now Rejected](#) in the Compatibility chapter.

fixed by A. dos Santos Fernandes

([CORE-5710](#))—Data type declaration DECFLOAT without precision should be using a default precision.

fixed by A. Peshkov

([CORE-5700](#))—DECFLOAT underflow should yield zero instead of an error.

fixed by A. Peshkov

([CORE-5699](#))—DECFLOAT should not throw exceptions when +/-NaN, +/-sNaN and +/-Infinity is used in comparisons.

fixed by A. Peshkov

([CORE-5646](#))—A parse error when compiling a statement would cause a memory leak until the attachment was disconnected.

fixed by A. dos Santos Fernandes

([CORE-5612](#))—View operations (create, recreate or drop) were exhibiting gradual slow-down.

fixed by D. Yemanov

([CORE-5611](#))—Memory consumption for prepared statements was higher.

fixed by A. dos Santos Fernandes

([CORE-5593](#), [CORE-5518](#))—The system function RDB\$ROLE_IN_USE could not take long role names.

fixed by A. Peshkov

([CORE-5480](#))—A SUBSTRING start position smaller than 1 should be allowed.

fixed by A. dos Santos Fernandes

([CORE-1592](#))—Altering procedure parameters could lead to an unrestorable database.

fixed by A. dos Santos Fernandes

Server Crashes/Hang-ups

([CORE-5980](#))— Firebird would crash due to concurrent operations with expression indices. Same fix was backported to Firebird 3.0.5.

fixed by V. Khorsun

([CORE-5972](#))— External engine trigger could crash the server if the table had a computed field.

fixed by A. dos Santos Fernandes

([CORE-5943](#))— The server could crash while preparing a query with both DISTINCT/ORDER BY and a non-field expression in the select list. Same fix was backported to Firebird 3.0.5.

fixed by D. Yemanov

([CORE-5936](#))— The server could segfault at the end of a database backup.

fixed by V. Khorsun

Security

([CORE-5927](#))— With some non-standard authentication plugins, traffic would remain unencrypted despite providing the correct crypt key. Same fix was backported to Firebird 3.0.5.

fixed by A. Peshkov

([CORE-5926](#))— An attempt to create a mapping with a non-ASCII user name that was encoded in a **single-byte** codepage (e.g. WIN1251) would lead to a “Malformed string” error. Same fix was backported to Firebird 3.0.5.

fixed by A. Peshkov

([CORE-5861](#))— New objects and some old objects in a database could not be granted the GRANT OPTION via role privileges.

fixed by R. Simakov

([CORE-5657](#))— Attended to various UDF-related security vulnerabilities, resulting in aggressive deprecation of support for the use of UDFs as external functions. See also [External Functions \(UDFs\) Feature Deprecated](#) in the the chapter [Changes to the Firebird Engine](#) and [Deprecation of External Functions \(UDFs\)](#) in the [Compatibility](#) chapter.

fixed by A. Peshkov

([CORE-5639](#))— Mapping rule using WIN_SSPI plugin: Windows user group conversion to Firebird role was not working.

fixed by A. Peshkov

([CORE-5518](#))— Firebird UDF string2blob() could allow remote code execution.

fixed by A. Peshkov

Utilities

gbak

([CORE-5855](#))— A database with generators containing space characters in their names could not be backed up.

fixed by A. Peshkov

([CORE-5800](#))— After backup/restore, expression indexes on computed fields would not work properly. Same fix was backported to Firebird 3.0.5.

fixed by D. Yemanov

([CORE-5637](#))— A string right truncation error was occurring on restore of the security database.

fixed by A. Peshkov

gpre

([CORE-5834](#))— gpre_boot was failing to link using cmake, giving undefined reference 'dladdr' and 'dlerror'. Same fix was backported to Firebird 3.0.5.

fixed by A. Peshkov

trace

(*CORE-5907*)—Regression: Trace could not be launched if its 'database' section contained a regular expression pattern with curly brackets to enclose a quantifier. Same fix was backported to Firebird 3.0.5.

fixed by A. Peshkov

Build Issues

(*CORE-5989*)—Some build issues involving iconv / libiconv 1.15 vs libc / libiconv_open | common/isc_file.cpp. Same fix was backported to Firebird 3.0.5.

fixed by A. Peshkov

(*CORE-5955*)—Static linking problem with ld >= 2.31. Same fix was backported to Firebird 3.0.5.

fixed by R. Simakov

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-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 15. Firebird 4.0 Project Teams

Table 4. 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; Firebird Butler coordinator; Python driver developer
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 implementer and co-coordinator; Documentation writer
Jiri Cincura	Czech Republic	Developer and coordinator of .NET providers
Martin Koeditz	Germany	Developer and coordinator of PHP driver Documentation translator
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
Tomneko Hayashi	Japan	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 <https://www.firebirdsql.org/pdfmanual/pdl.pdf> (PDF) and <https://www.firebirdsql.org/manual/pdl.html> (HTML).

The Original Documentation is entitled *Firebird 4.0 Release Notes*.

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

Copyright © 2004-2020. All Rights Reserved. Initial Writer contact: helebor at users dot sourceforge dot net.