Firebird Future Development : What to Expect?

by Dimitry Yemanov, 2005

1. Review of Firebird 2.0

Some time ago, many of our users were surprised by the number of features released in Firebird 1.5. Honestly, before preparing this paper, I didn't count new features in Firebird 2.0 and I don't have a clue whether it's superior to the version 1.5 in this aspect or not. But, in my opinion, the major benefit of the version 2.0 is not its feature set. "What then?", - you ask. I'd describe Firebird 2.0 as "the version which removes the annoying limits". Sounds not so modest, I know. Let me explain better. No doubt that Firebird has a nice multi-generational architecture and a rich SQL language, an embedded usage and a good performance. But I'm sure almost everyone stepped on some internal limitations that worried or even shocked you. To name a few (in no particular order):

- Undocumented table limit of about 35GB, an overflow may cause data corruption
- Removing (garbage collecting) nodes from a non-selective index is extremely slow
- Bigger page cache often means slower performance
- Optimizer is not able to choose a good plan in many cases
- International support is weak, a lot of bugs in the Unicode/MBCS handling
- Lack of a fast backup/restore mechanism
- Weak security and many known vulnerabilities
- Requirement of exclusive database access for referential integrity declarations
- Too few built-in functions
- Unreliable database shutdown

Some of them could be of critical priority for your business, others are just annoying. Anyway, I'm glad to report that Firebird 2.0 eliminates most of the aforementioned issues and significantly reduces the impact of others. As for me, this is more important than declaring new language features. But considering also tons of bugfixes and enough new features, Firebird 2.0 definitely represents a major release of your favourite RDBMS. More robust, more feature complete, faster and much more friendly to non-ASCII users.

Of course, there are limits that still exist and many features we don't support yet. But we should have some room for future improvements, shouldn't we? We'll talk about the future development a bit later.

Well, for those who're interested in numbers, let's read the WhatsNew document or Release Notes and make a summary of total changes per version:

- Version 1.0: 32 improvements, 55 bugfixes
- Version 1.5: 58 improvements, 94 bugfixes
- Version 2.0: 82 improvements, 140 bugfixes

Note: the version 2.0 statistics represents its current state, i.e. Beta 1 release.

Impressive, don't you think so? Obviously, Firebird 1.5 was developed longer than Firebird 1.0 and the same appears to be true for Firebird 2.0. But at least you see what the development time is spent for.

2. Roadmap of future versions

Speaking about the short-term plans, our primary goal is to merge two codebases (Firebird 2.0 and Vulcan) in order to release Firebird version 3.0. It will be based on the Vulcan tree and will contain its modular architecture and new features, as well as all improvements made in Firebird 2.0. The key features of the Vulcan codebase are:

- Globally refactured code
- Fine-grained multithreading
- Unified provider based architecture
- Flexible configuration mechanism
- Database level authentication and enhanced security management
- Internal DSQL implementation

As both Firebird 2.0 and Firebird Vulcan releases are going to co-exist in the next year, you may ask why version 3.0 is numbered as a major release and what else (except of features already done in both codebases) it will contain. A good question. As we want to shorten the 3.0 release cycle as much as possible, no completely new development is expected to happen in that version. But we need to keep our users interested, so something new should be introduced. Solution is simple: the 3.0 release is going to incorporate all the work done inside independent branches. As you perhaps know, there are some improvements done by various Firebird developers that didn't go into the 2.0 release due to time constraints. Some of them are included and being tested in Fyracle, others are still in private trees. Also, we still have a few features in Yaffil that requires backporting into Firebird. Everything mentioned above is exactly the new stuff you'll see in version 3.0. Let's see what has been already done:

- Common tables expressions and recursive queries (SQL-99 compliant)
  Developed by: Paul Ruizendaal
  Current state: Completed
- Global temporary tables (SQL-99 compliant)
  Developed by: Vlad Horsun
  Current state: Completed

- External procedures / functions (SQL-99 compliant)
  Developed by: Eugene Putilin, Roman Rokytskyy, Vlad Horsun
  Current state: Partially done, requires the callback API discussion
- New built-in functions (string, math, binary, date/time)
  Developed by: Oleg Loa
  Current state: Completed, although requires some changes

These features are the major candidates to be included into Firebird 3.0, but there are others (less important) as well.

As soon as version 3.0 will be released for public testing, development of the next version will begin. We don't have a decision about version numbering yet, so it could be 3.5 or 4.0 or whatever else. For the duration of this talk, I'll be calling it "version 3.0+", where the plus sign simply means "the next version". Version 3.0+ is going to have major ODS changes as well as a lot of administration, tracing, security, performance and SQL improvements. Most probably, it will also contain an updated remote protocol implementation. Now it's a bit early to say what exactly will be included in that release, but you'll find some hints a bit later.

If you'd ask me to outline the generic development priorities, they would be:

1. Reliability and safety (bugfixing, guaranteed recovery, security improvements)
2. Administration and monitoring facilities
3. Compliance with the SQL specification
4. Performance (both algorithmical and optimizer decisions)
5. Language enhancements

Now a few words about the 2.0 point releases. First, our usual maintenance schedule will definitely cover the 2.0 product line, so please expect 2.0.1, 2.0.2, etc releases (containing bugfixes) every few months. As always, these point versions will consist of changes backported from the active development branch. So, if you see your "favourite" bug fixed in version 3.0, feel free to ask developers about porting it into the next maintenance release of 2.0.x. Second, I'd expect some of the scheduled 3.0 features (e.g. GTT implementation or WITH [RECURSIVE] code) to be ported into the 2.0 HEAD branch before the merge in order to make a 2.1 minor release possible in the case of noticable delay with the 3.0 development.

Recalling everything said above, the timetable should look like:

2005:

- Release 2.0 RC and fork the 2.0 HEAD to create the release branch
- Port some changes from independent trees to HEAD
- Fork the Vulcan HEAD to create the 3.0 development branch

2006, 1st quarter:

- Release Firebird 2.0 Final and Firebird Vulcan Final

2006, 2nd quarter:

- Release Firebird 3.0 Beta
- Fork the 3.0 HEAD to create the 3.0+ development branch

2006, 3rd quarter:

- Release Firebird 3.0 Final

2006, 4th quarter:

- Release Firebird 3.0+ Beta

The key point of this roadmap is when we're able to release Firebird 2.0 and Firebird Vulcan. And this is exactly the point where your help with testing/feedback allows us to move faster.

3. What features to expect?

This part of our talk is dedicated to the project activities that we'd expect to see in the not so distant future. To follow the list easily, they're grouped by category, similar to our RFE tracker.

Every work has two associated parameters: priority and complexity. The priority value is based on user wishes, it's set up after looking at various polls and forum/newsgroup discussions. In other words, it shows how much our users want a particular feature. Also, this value depends on our analysis of the features offered by our competitors. The complexity value is based on time/effort estimates made by the core team. "None" means that all the required work is already done (in some code branch) and we just need to backport and test it. "Implementation" means that the feature has already been discussed and agreed on, but it still requires some minor discussions and the actual coding. "Design" means that we have a basic agreement and some vision of the things, but the work hasn't been discussed in depth yet and hence we don't have any implementation plan. "Research" means that the work requires serious analysis before discussing its design and implementation specifications.

Administration / Tracing / Monitoring

- Monitoring via API and/or special tables

  Perform a "snapshot" monitoring (i.e. at the given moment) of the internal activities inside the engine. Obvious objects of such a monitoring are: databases, attachments, transactions, active requests, resource (memory, CPU) usage, etc.

  priority = high
  complexity = design/implementation
- Asynchronous statement cancellation / timeouts

  Force any of the following actions: cancel a running statement, rollback a transaction, kill an attachment. Allow setting timeouts for SQL statements. Should be available at least at the API level.

  priority = high
  complexity = implementation
- Detailed logging/audit

  Allow to log some events happening on the server. These could be: successful/rejected authentification attempts (containing client host info), prepared/executed SQL statements, committed/rolled back transactions, etc. We need API to set up the required events and to retrieve the audit log.

  priority = medium
  complexity = design
- Detailed SQL tracing/profiling

  Show detailed access path (at the RSB tree level) for every retrieval, count rows (profile CPU time, etc) per every node. Available runtime statistics should be extended.

  priority = medium
  complexity = design/implementation
- DDL level and global triggers

  Allow triggers ON CREATE/ALTER/DROP. Implement triggers attachment and transaction level triggers running in autonomous transactions.

  priority = medium
  complexity = design
- PSQL debugging extentions/hooks

  Allow PSQL debugging via introducing: looper breakpoints, handler callbacks, retrieval of context data, etc.

  priority = low
  complexity = research

Security

- Embedded users / SQL users management

  Allow in-database users management.

  priority = high
  complexity = none (done in Vulcan)
- User permissions for metadata

  Protect all metadata with security classes. Implement metadata-level permissions. Add database-level permissions like BACKUP, DROP, etc.

  priority = high
  complexity = design
- Pluggable authentication modules

  Enable using of custom authentication mechanisms (e.g. native OS ones).

  priority = medium
  complexity = design
- Security groups

  Design group-based security as an alternative to the existing role-based one.

  priority = medium
  complexity = research
- Database encryption

  Allow optional encryption of database files. Keys management is an open question here.

  priority = medium

complexity = research

Language extentions

- Schemas/namespaces

  First, it significantly reduces a cost of the issue with short metadata names. Second, it simplifies administration as a number of different databases could be united into a single file. Third, it finally allows us to be fully SQL-92 (entry level) compliant.

  priority = high
  complexity = research
- Native long numeric data type

  Implement long exact numeric data type (with precision longer than 30 decimal digits) and appropriate BCD arithmetics.

  priority = high
  complexity = research
- More built-in functions

  The SQL-99 (or later) ones (those of major importance for us) must be implemented first. Then we need users feedback about other ones.

  priority = high
  complexity = design/implementation
- Temporary tables / transient datasets

  Implement temporary schema objects and/or datasets. SQL-99 compliance is required, extentions are welcome.

  priority = high
  complexity = design/implementation (partially done in Fyracle)
- Longer metadata names

  Up to 128 unicode characters.

  priority = medium
  complexity = design
- SQL functions

  CREATE/ALTER/DROP FUNCTION as per SQL-99.

  priority = medium
  complexity = implementation
- Domains everywhere

  Allow usage of domains in PSQL parameters and variables, as well as in the CAST function.

  priority = medium
  complexity = design
- Recursive queries

  Implement SQL for recursive retrievals. Make it consistent with the SQL specification.

  priority = medium
  complexity = none (done in Fyracle)
- Regular expressions

  Allow usage of regular expressions in search conditions. Add some special syntax (a new predicate) for this purpose.

  priority = medium
  complexity = design/implementation
- TEXT BLOB compatible with [VAR]CHAR

  Allow BLOB SUB_TYPE TEXT to be compatible/interchangable with string data types. Allow text blobs in all built-in functions.

  priority = medium
  complexity = design/implementation
- Deferred constraints

  Implement commit-time constraint checking as per SQL specification.

  priority = low

complexity = design

Performance / Optimizer

- Faster outer joins

  Implement the merge algorithm for outer joins.

  priority = high
  complexity = implementation
- Optimizer improvements

  Fix known bugs/limitations, better optimizer decisions, more data statistics

  priority = medium
  complexity = design/implementation
- More access paths

  Consider implementation of hash join / hash aggregate and other retrieval algorithms used in the competitor RDBMS.

  priority = medium
  complexity = research
- More effective sorting

  Implement partial sorting to spead up FIRST-limited retrievals. Consider sorting recno's instead of entire rows.

  priority = medium
  complexity = design
- Optimized network protocol

  Avoid sending a lot of unneeded data (buffer tails). Consider implementing protocol batches (e.g. prepare + info). Compress spaces more effectively.

  priority = medium
  complexity = research

Maintenance / Recovery

- Reliable logical backup

  The only case of unrestorable backup should be a physically corrupted backup. Primitive objects (generators, UDFs, etc) must be restored in the beginning. Computed columns and validation constraints must be restored at the end. The engine should reject inconsistent data instead of transform them when reading (e.g. no value -> NULL). GBAK should allow partial restore, driven by switches or interactively.

  priority = high
  complexity = design
- Point-in-time recovery

  The engine must have an optional ability to maintain a redo log in order to roll it over the last logical backup. No data loss is acceptable.

  priority = high
  complexity = research

Generic / Architecture

- SMP support in SS

  Support effective fine grained multi-threading in the SS architecture.

  priority = high
  complexity = none (done in Vulcan)
- Compiled statements cache

  Support caching/reusing of compiled statements.

  priority = high
  complexity = implementation (partially done in Vulcan)
- Statement/transaction consistency

  Solve the known inconsistencies in verbs/transactions. Mostly, this covers blr_for behaviour in INSERT/UPDATE/DELETE statements. Make read-committed transactions compliant with the SQL specification.

priority = high
complexity = design

- External data sources / database links / cross-database SQL

  Allow retrievals from external data sources. Provide a few drivers (native FB, JDBC, ODBC) in the distros. Add DDL to declare and DML to use such sources. Implement optimization of retrievals for native data sources.

  priority = high
  complexity = research
- External functions/procedures

  Allow to create procedures/functions written in non-PSQL languages. Provide a few drivers (cdecl, Java, .NET) in the distros.

  priority = high
  complexity = design/implementation (partially done in Fyracle)
- Full-text search

  Implement FTS features inside the engine or add API to plug the external FTS engines in. FB seems to be the only one RDBMS that don't have this feature yet.

  priority = medium
  complexity = research
- Clustering

  MySQL supports clusters (AFAIK, only shared-memory so far), PostgreSQL shared-disk implementation is in the way. More and more buzz about this feature around the world. We should compete.

  priority = medium
  complexity = research
- Bi-directional indices

  Allow reversed index navigation to use ASC-indices for DESC sorting and vice versa.

  priority = medium
  complexity = implementation
- Referential integrity without indices

  Implement (optionally) foreign keys that are not enforced by indices. Also provide an ability to reuse the existing index for a constraint.

  priority = medium
  complexity = design
- Bulk load/import

  Implement effective massive load ability. Provide utility/syntax to use different input formats (csv, xml schema, etc) for import.

  priority = medium
  complexity = design
- Bi-directional cursors

  Consider implementation of scrollable cursors inside the engine or provide a thin layer at the top of the RSB hierarchy to implement the feature via the caching.

  priority = low
  complexity = research
- XML integration

  Provide at least fetch to XML and insert from XML abilities. Consider having BLOB SUB_TYPE XML and implementing XPath queries.

  priority = low
  complexity = research

Obviously, the aforementioned list is not complete, it includes only the changes that we consider mostly important. If this list misses your favourite wish, speak now!

Now let's create a matrix where most preferrable and easy-to-do features are placed in the top-left corner and most hard-to-implement and/or less wanted ones are in the bottom-right corner. If you'll be moving from one of these corners to another you'll see a most probable feature implementation roadmap. Recalling what has been said before, we could imagine a more detailed roadmap:

Firebird 3.0 (the merged version):

- Monitoring
- Asynchronous statement cancellation

- Embedded users / SQL users management
- More built-in functions
- Temporary tables
- SQL functions
- Recursive queries
- Faster outer joins
- SMP support in SS
- Compiled statements cache
- External functions/procedures

As you can see (and as it has been stated earlier), version 3.0 is expected to include the work already done and a few features that are highly wanted and relatively easy to implement. Everything else tends to slow the development down and hence is excluded from the above list.

Firebird 3.0+ (the next major version):

- Detailed logging/audit
- SQL tracing/profilingUser permissions for metadata
- Pluggable authentication modules
- Security groups
- Long exact numeric implementation
- Domains everywhere
- Regular expressions
- TEXT BLOB compatible with [VAR]CHAR
- Reliable logical backup
- Optimizer improvements
- Statement consistency/atomicy, read committed compliance
- Optimized network protocol
- Bi-directional indices
- Referential integrity without indices
- Bulk load/import

It would also be excellent to design schemas/namespaces and longer metadata names for 3.0+, but no promises here. The same for external data sources and deferred constraints.

Firebird 3.0++ (something we don't have a schedule for yet):

- PSQL debugging extentions/hooks
- Database encryption
- More access paths
- Full-text search
- Clustering
- Bi-directional cursors
- XML integration

Of course, some intermediate or minor releases may happen in the meantime, as we'll try to make the release cycles shorter. As soon as the details are discussed and agreed on among the project admins, you'll see both an actual short-term roadmap and an expected long-term roadmap on our site.