Appendix A: Notes

Character set NONE data accepted "as is"

In Firebird 1.5.1 and up

Firebird 1.5.1 has improved the way character set NONE data are moved to and from fields or variables with another character set, resulting in fewer transliteration errors.

In Firebird 1.5.0, from a client connected with character set NONE, you could read data in two incompatible character sets – such as SJIS (Japanese) and WIN1251 (Russian) – even though you could not read one of those character sets while connected from a client with the other character set. Data would be received “as is” and be stored without raising an exception.

However, from this character set NONE client connection, an attempt to update any Russian or Japanese data columns using either parameterized queries or literal strings without introducer syntax would fail with transliteration errors; and subsequent queries on the stored NONE data would similarly fail.

In Firebird 1.5.1, both problems have been circumvented. Data received from the client in character set NONE are still stored “as is” but what is stored is an exact, binary copy of the received string. In the reverse case, when stored data are read into this client from columns with specific character sets, there will be no transliteration error. When the connection character set is NONE, no attempt is made in either case to resolve the string to wellformed characters, so neither the write nor the read will throw a transliteration error.

This opens the possibility for working with data from multiple character sets in a single database, as long as the connection character set is NONE. The client has full responsibility for submitting strings in the appropriate character set and converting strings returned by the engine, as needed.

Abstraction layers that have to manage this can read the low byte of the sqlsubtype field in the XSQLVAR structure, which contains the character set identifier.

While character set NONE literals are accepted and implicitly stored in the character set of their context, the use of introducer syntax to coerce the character sets of literals is highly recommended when the application is handling literals in a mixture of character sets. This should avoid the string's being misinterpreted when the application shifts the context for literal usage to a different character set.

Note: Coercion of the character set, using the introducer syntax or casting, is still required when handling heterogeneous character sets from a client context that is anything other than NONE. Both methods are shown below, using character set ISO8859_1 as an example target. Notice the “_” prefix in the introducer syntax.

Introducer syntax:

```
_ISO8859_1 mystring
```

Casting:

```
CAST (mystring AS VARCHAR(n) CHARACTER SET ISO8859_1)
```

See also:

- Default character set
- Field and domain character sets
- SET NAMES
- New character sets

back to top of page

Understanding the WITH LOCK clause

This note looks a little deeper into explicit locking and its ramifications. The WITH LOCK feature, added in Firebird 1.5, provides a limited explicit pessimistic locking capability for cautious use in conditions where the affected row set is:

a. extremely small (ideally, a singleton), *and*

b. precisely controlled by the application code.

Pessimistic locks are rarely needed in Firebird. This is an expert feature, intended for use by those who thoroughly understand its consequences. Knowledge of the various levels of transaction isolation is essential. WITH LOCK is available in DSQL and PSQL, and only for top-level, single-table SELECTs. As stated in the reference part of this guide, WITH LOCK is not available:

- in a subquery specification;
- for joined sets;
- with the DISTINCT operator, a GROUP BY clause or any other aggregating operation;
- with a view;
- with the output of a selectable stored procedure;
- with an external table.

Syntax and behaviour

```
SELECT ... FROM single_table
  [WHERE ...]
  [FOR UPDATE [OF ...]]
  [WITH LOCK]
```

If the WITH LOCK clause succeeds, it will secure a lock on the selected rows and prevent any other transaction from obtaining write access to any of those rows, or their dependants, until your transaction ends.

If the FOR UPDATE clause is included, the lock will be applied to each row, one by one, as it is fetched

into the server-side row cache. It becomes possible, then, that a lock which appeared to succeed when requested will nevertheless fail subsequently, when an attempt is made to fetch a row which becomes locked by another transaction.

As the engine considers, in turn, each record falling under an explicit lock statement, it returns either the record version that is the most currently committed, regardless of database state when the statement was submitted, or an exception.

Wait behaviour and conflict reporting depend on the transaction parameters specified in the TPB block:

Table A.1. How TPB settings affect explicit locking

TPB mode	Behaviour
isc_tpb_consistency	**Explicit locks are overridden by implicit or explicit table-level locks and are ignored.**
isc_tpb_concurrency + isc_tpb_nowait	**If a record is modified by any transaction that was committed since the transaction attempting to get explicit lock started, or an active transaction has performed a modification of this record, an update conflict exception is raised immediately.**
isc_tpb_concurrency + isc_tpb_wait	**If the record is modified by any transaction that has committed since the transaction attempting to get explicit lock started, an update conflict exception is raised immediately. If an active transaction is holding ownership on this record (via explicit locking or by a normal optimistic write-lock) the transaction attempting the explicit lock waits for the outcome of the blocking transaction and, when it finishes, attempts to get the lock on the record again. This means that, if the blocking transaction committed a modified version of this record, an update conflict exception will be raised.**
isc_tpb_read_committed + isc_tpb_nowait	**If there is an active transaction holding ownership on this record (via explicit locking or normal update), an update conflict exception is raised immediately.**
isc_tpb_read_committed + isc_tpb_wait	**If there is an active transaction holding ownership on this record (via explicit locking or by a normal optimistic write-lock), the transaction attempting the explicit lock waits for the outcome of blocking transation and when it finishes, attempts to get the lock on the record again. Update conflict exceptions can never be raised by an explicit lock statement in this TPB mode.**

back to top of page

How the engine deals with WITH LOCK

When an UPDATE statement tries to access a record that is locked by another transaction, it either raises an update conflict exception or waits for the locking transaction to finish, depending on TPB mode. Engine behaviour here is the same as if this record had already been modified by the locking transaction.

No special gdscodes are returned from conflicts involving pessimistic locks.

The engine guarantees that all records returned by an explicit lock statement are actually locked and do meet the search conditions specified in WHERE clause, as long as the search conditions do not depend on any other tables, via joins, subqueries, etc. It also guarantees that rows not meeting the search conditions will not be locked by the statement. It can not guarantee that there are no rows which, though meeting the search conditions, are not locked.

Note: This situation can arise if other, parallel transactions commit their changes during the course of the locking statement's execution.

The engine locks rows at fetch time. This has important consequences if you lock several rows at once. Many access methods for Firebird databases default to fetching output in packets of a few hundred rows (“buffered fetches”). Most data access components cannot bring you the rows contained in the last-fetched packet, where an error occurred.

back to top of page

The optional OF <column-names> sub-clause

The FOR UPDATE clause provides a technique to prevent usage of buffered fetches, optionally with the OF <column-names> subclause to enable positioned updates.

Tip: Alternatively, it may be possible in your access components to set the size of the fetch buffer to 1. This would enable you to process the currently-locked row before the next is fetched and locked, or to handle errors without rolling back your transaction.

Caveats using WITH LOCK

- Rolling back of an implicit or explicit savepoint releases record locks that were taken under that savepoint, but it doesn't notify waiting transactions. Applications should not depend on this behaviour as it may get changed in the future.
- While explicit locks can be used to prevent and/or handle unusual update conflict errors, the volume of deadlock errors will grow unless you design your locking strategy carefully and control it rigorously.
- Most applications do not need explicit locks at all. The main purposes of explicit locks are (1) to prevent expensive handling of update conflict errors in heavily loaded applications and (2) to maintain integrity of objects mapped to a relational database in a clustered environment. If your use of explicit locking doesn't fall in one of these two categories, then it's the wrong way to do the task in Firebird.
- Explicit locking is an advanced feature; do not misuse it! While solutions for these kinds of problems may be very important for web sites handling thousands of concurrent writers, or for ERP/CRM systems operating in large corporations, most application programs do not need to work in such conditions.

Examples using explicit locking

i. Simple:

```
SELECT * FROM DOCUMENT WHERE ID=? WITH LOCK
```

ii. Multiple rows, one-by-one processing with DSQL cursor:

```
SELECT * FROM DOCUMENT WHERE PARENT_ID=?
  FOR UPDATE WITH LOCK
```

back to top of page

A note on CSTRING parameters

External functions involving strings often use the type CSTRING(n) in their declarations. This type represents a zero-terminated string of maximum length n. Most of the functions handling CSTRINGs are programmed in such a way that they can accept and return zero-terminated strings of any length. So why the n? Because the Firebird engine has to set up space to process the input an output parameters, and convert them to and from SQL data types. Most strings used in databases are only dozens to hundreds of bytes long; it would be a waste to reserve 32 KB of memory each time such a string is processed. Therefore, the standard declarations of most CSTRING functions – as found in the file ib_udf.sql – specifiy a length of 255 bytes. (In Firebird 1.5.1 and below, this default length is 80 bytes.) As an example, here's the SQL declaration of lpad:

```
DECLARE EXTERNAL FUNCTION lpad
  CSTRING(255), INTEGER, CSTRING(1)
  RETURNS CSTRING(255) FREE_IT
  ENTRY_POINT 'IB_UDF_lpad' MODULE_NAME 'ib_udf'
```

Once you've declared a CSTRING parameter with a certain length, you cannot call the function with a longer input string, or cause it to return a string longer than the declared output length. But the standard declarations are just reasonable defaults; they're not cast in concrete, and you can change them if you want to. If you have to leftpad strings of up to 500 bytes long, then it's perfectly OK to change both 255's in the declaration to 500 or more.

A special case is when you usually operate on short strings (say less then 100 bytes) but occasionally have to call the function with a huge (VAR)CHAR argument. Declaring CSTRING(32000) makes sure that all the calls will be successful, but it will also cause 32000 bytes per parameter to be reserved, even in that majority of cases where the strings are under 100 bytes. In that situation you may consider declaring the function twice, with different names and different string lengths:

```
DECLARE EXTERNAL FUNCTION lpad
  CSTRING(100), INTEGER, CSTRING(1)
  RETURNS CSTRING(100) FREE_IT
  ENTRY_POINT 'IB_UDF_lpad' MODULE_NAME 'ib_udf';

DECLARE EXTERNAL FUNCTION lpadbig
  CSTRING(32000), INTEGER, CSTRING(1)
  RETURNS CSTRING(32000) FREE_IT
```

```
  ENTRY_POINT 'IB_UDF_lpad' MODULE_NAME 'ib_udf';
```

Now you can call lpad() for all the small strings and lpadbig() for the occasional monster. Notice how the declared names in the first line differ (they determine how you call the functions from within your SQL), but the entry point (the function name in the library) is the same in both cases.

See also:

- SET TRANSACTION
- SELECT [WITH LOCK]

back to top of page

Passing NULL to UDFs in Firebird 2

If a pre-2.0 Firebird engine must pass an SQL NULL argument to a user-defined function, it always converts it to a zero-equivalent, e.g. a numerical 0 or an empty string. The only exception to this rule are UDFs that make use of the BY DESCRIPTOR mechanism introduced in Firebird 1. The fbudf library uses descriptors, but the vast majority of UDFs, including those in Firebird's standard ib_udf library, still use the old style of parameter passing, inherited from InterBase.

As a consequence, most UDFs can't tell the difference between NULL and zero input.

Firebird 2 comes with a somewhat improved calling mechanism for these old-style UDFs. The engine will now pass NULL input as a null pointer to the function, if the function has been declared to the database with a NULL keyword after the argument(s) in question, e.g. like this:

```
declare external function ltrim
  cstring(255) null
  returns cstring(255) free_it
  entry_point 'IB_UDF_ltrim' module_name 'ib_udf';
```

This requirement ensures that existing databases and their applications can continue to function like before. Leave out the NULL keyword and the function will behave like it did under Firebird 1.5 and earlier.

Please note that you can't just add NULL keywords to your declarations and then expect every function to handle NULL input correctly. Each function has to be (re)written in such a way that NULLs are dealt with correctly. Always look at the declarations provided by the function implementor. For the functions in the ib_udf library, consult ib_udf2.sql in the Firebird UDF directory. Notice the 2 in the filename; the old-style declarations are in ib_udf.sql.

These are the ib_udf functions that have been updated to recognise NULL input and handle it properly:

- ascii_char
- lower
- lpad and rpad
- ltrim and rtrim

- substr and substrlen

Most ib_udf functions remain as they were; in any case, passing NULL to an old-style UDF is never possible if the argument isn't of a referenced type.

On a side note: don't use lower, trim and substr* in new code; use the internal functions LOWER, TRIM and SUBSTRING instead.

back to top of page

"Upgrading" ib_udf functions in an existing database

If you are using an existing database with one or more of the functions listed above under Firebird 2, and you want to benefit from the improved NULL handling, run the script ib_udf_upgrade.sql against your database. It is located in the Firebird misc\upgrade\ib_udf directory.

See also:

- Expressions involving NULL
- External functions (UDFs)
- User-defined function (UDF)
- UDFs callable as void functions
- DECLARE EXTERNAL FUNCTION (incorporating a new UDF library)
- ALTER EXTERNAL FUNCTION
- DECLARE EXTERNAL FUNCTION
- DROP EXTERNAL FUNCTION
- Threaded Server and UDFs
- Using descriptors with UDFs

back to top of page

Maximum number of indices in different Firebird versions

Between Firebird 1.0 and 2.0 there have been quite a few changes in the maximum number of indices per database table. The table below sums them all up.

Table A.2. Max. indices per table in Firebird 1.0 – 2.0

Page	Firebird version(s)	Firebird version(s)	Firebird version(s)	Firebird version(s)	Firebird version(s)	Firebird version(s)	Firebird version(s)	Firebird version(s)	Firebird version(s)	Firebird version(s)	Firebird version(s)	Firebird version(s)
size	**1.0, 1.0.2**	**1.0, 1.0.2**	**1.0, 1.0.2**	**1.0.3**	**1.0.3**	**1.0.3**	**1.5.x**	**1.5.x**	**1.5.x**	**2.0.x**	**2.0.x**	**2.0.x**
	1 col	**2 cols**	**3 cols**	**1 col**	**2 cols**	**3 cols**	**1 col**	**2 cols**	**3 cols**	**1 col**	**2 cols**	**3 cols**
1024	62	50	41	62	50	41	62	50	41	50	35	27
2048	65	65	65	126	101	84	126	101	84	101	72	56
4096	65	65	65	254	203	169	254	203	169	203	145	113
8192	65	65	65	510	408	340	257	257	257	408	291	227
16384	65	65	65	1022	818	681	257	257	257	818	584	454