Data types and subtypes

BIGINT data type

Added in: 1.5

Description

BIGINT is the SQL99-compliant 64-bit signed integer type. It is available in Dialect 3 only.

BIGINT numbers range from -263 .. 263-1, or -9,223,372,036,854,775,808 .. 9,223,372,036,854,775,807.

Since Firebird 2.5, BIGINT numbers may be entered in hexadecimal form, with 9–16 hex digits. Shorter hex numerals are interpreted as INTEGERs.

Examples

```
create table WholeLottaRecords (
id bigint not null primary key,
description varchar(32)
)

insert into MyBigints values (
-236453287458723,
328832607832,
22,
-56786237632476,
0x6F55A09D42,         -- 478177959234
0X7FFFFFFFFFFFFFFF,   -- 9223372036854775807
0xffffffffffffffff,   -- -1
0x80000000,           -- -2147483648, an INTEGER
0x080000000,          -- 2147483648, a BIGINT
0XFFFFFFFF,           -- -1, an INTEGER
0X0FFFFFFFF           -- 4294967295, a BIGINT
)
```

The hexadecimal INTEGERs in the second example will be automatically cast to BIGINT before insertion into the table. However, this happens after the numerical value has been established, so 0x80000000 (8 digits) and 0x080000000 (9 digits) will be stored as different values. For more information on this difference, see Hexadecimal notation for numerals, in particular the paragraph Value ranges.

See also: BIGINT

back to top of page

BLOB data type

Text BLOB support in functions and operators

Changed in: 2.1, 2.1.5, 2.5.1

Description

Text BLOBs of any length and character set (including multi-byte sets) are now supported by practically every internal text function and operator. In a few cases there are limitations or bugs.

Level of support

- Full support for:
  - = (assignment);
  - =, <>, «/color>, <color #c3c3c3>⇐, >, >= and synonyms (comparison);
  - || (concatenation);
  - BETWEEN, IS [NOT] DISTINCT FROM, IN, ANY | SOME and ALL.
- Support for STARTING [WITH], LIKE and CONTAINING:
  - In versions 2.1–2.1.4 and 2.5, an error is raised if the second operand is 32 KB or longer, or if the first operand is a BLOB with character set NONE and the second operand is a BLOB of any length and character set.
  - In versions 2.5.1 and up (as well as 2.1.5 and up in the 2.1 branch), each operand can be a BLOB of any length and character set.
- SELECT DISTINCT, ORDER BY and GROUP BY work on the BLOB ID, not the contents. This makes them as good as useless, except that SELECT DISTINCT weeds out multiple NULLs, if present. GROUP BY behaves oddly in that it groups together equal rows if they are adjacent, but not if they are apart.
- Any issues with BLOBs in internal functions and aggregate functions are discussed in their respective sections.

back to top of page

Various enhancements

Changed in: 2.0

Description

In Firebird 2.0, several enhancements have been implemented for text BLOBs:

- DML COLLATE clauses are now supported.
- Equality comparisons can be performed on the full BLOB contents.
- Character set conversions are possible when assigning a BLOB to a BLOB or a string to a BLOB.

When defining binary BLOBs, the mnemonic binary can now be used instead of the integer 0.

Examples

```
select NameBlob from MyTable
  where NameBlob collate pt_br = 'João'
create table MyPictures (
  id int not null primary key,
  title varchar(40),
  description varchar(200),
  picture blob sub_type binary
)
```

See also:

- BLOB
- Blob filter

back to top of page

SQL_NULL data type

Added in: 2.5

Description

The SQL_NULL data type is of little or no interest to end users. It can hold no data, only a state: NULL or NOT NULL. It is also not possible to declare columns, variables or PSQL parameters of type SQL_NULL. At present, its only purpose is to support the "? IS NULL" syntax in SQL statements with positional parameters. Application developers can make use of this when constructing queries that contain one or more optional filter terms.

Syntax

If a statement containing the following predicate is prepared:

```
? <op> NULL
```

Firebird will describe the parameter ('?') as being of type SQL_NULL. <op> can be any comparison operator, but the only one that makes sense in practice is "IS" (and possibly, in some rare cases, "NOT IS").

back to top of page

Rationale

In itself, having a query with a "WHERE ? IS NULL" clause doesn't make a lot of sense. You could use such a parameter as an on/off switch, but that hardly warrants inventing a whole new datataype. After all, such switches can also be constructed with a CHAR, SMALLINT or other parameter type. The reason for adding the SQL_NULL type is that developers of applications, connectivity toolsets, drivers etc. want to be able to support queries with optional filters like these:

```
select make, model, weight, price, in_stock from automobiles
```

```
  where (make = :make or :make is null)
    and (model = :model or :model is null)
    and (price <= :maxprice or :maxprice is null)
```

The idea is that the end user can optionally enter choices for the parameters :make, :model and :maxprice. Wherever a choice is entered, the corresponding filter should be applied. Wherever a parameter is left unset (NULL), there should be no filtering on that attribute. If all are unset, the entire table AUTOMOBILES should be shown.

Unfortunately, named parameters like :make and :model only exist on the application level. Before the query is passed to Firebird for preparation, it must be converted to this form:

```
select make, model, weight, price, in_stock from automobiles
  where (make = ? or ? is null)
    and (model = ? or ? is null)
    and (price <= ? or ? is null)
```

Instead of three named parameters, each occurring twice, we now have six positional parameters. There is *no way* that Firebird can tell whether some of them actually refer to the same application-level variable. (The fact that, in this example, they happen to be within the same pair of parentheses doesn't mean anything.) This in turn means that Firebird also cannot determine the data type of the “? is null” parameters. This last problem could be solved by casting:

```
select make, model, weight, price, in_stock from automobiles
  where (make = ? or cast(? as type of column automobiles.make) is null)
    and (model = ? or cast(? as type of column automobiles.model) is null)
    and (price <= ? or cast(? as type of column automobiles.price) is null)
```

… but this is rather cumbersome. And there is another issue: wherever a filter term is *not* NULL, its value will be passed twice to the server: once in the parameter that is compared against the table column, and once in the parameter that is tested for NULL. This is a bit of a waste. But the only alternative is to set up no less then eight separate queries (2 to the power of the number of optional filters), which is even more of a headache. Hence the decision to implement a dedicated SQL_NULL datatype.

back to top of page

Use in practice

Notice: The following discussion assumes familiarity with the Firebird API and the passing of parameters via XSQLVAR structures. Readers without this knowledge won't have to deal with the SQL_NULL data type anyway and can skip this section.

As usual, the application passes the parameterized query in ?-form to the server. It is not possible to merge pairs of “identical” parameters into one. So, for e.g. two optional filters, four positional parameters are needed:

```
select size, colour, price from shirts
  where (size = ? or ? is null)
  and (colour = ? or ? is null)
```

After the call to isc_dsql_describe_bind(), the sqltype of the 2nd and 4th parameter will be set to SQL_NULL. As said, Firebird has no knowledge of their special relation with the 1st and 3rd parameter – this is entirely the responsibility of the programmer. Once the values for size and colour have been set (or left unset) by the user and the query is about to be executed, each pair of XSQLVARs must be filled as follows:

User has filled in a value

- First parameter (value compare): set *sqldata to the supplied value and *sqlind to 0 (for NOT NULL);
- Second parameter (NULL test): set sqldata to null (null pointer, not SQL NULL) and *sqlind to 0 (for NOT NULL).

User has left the field blank

- Both parameters: set sqldata to null (null pointer, not SQL NULL) and *sqlind to -1 (indicating NULL).

In other words: The value compare parameter is always set as usual. The SQL_NULL parameter is set the same, except that sqldata remains null at all times.

back to top of page

New character sets

Added in: 1.0, 1.5, 2.0, 2.1, 2.5

The following table lists the character sets added in Firebird.

Table 5.1. Character sets new in Firebird

Name	Max bytes/ch.	Languages	Added in
CP943C	2	Japanese	2.1
DOS737	1	Greek	1.5
DOS775	1	Baltic	1.5
DOS858	1	DOS850 plus € sign	1.5
DOS862	1	Hebrew	1.5
DOS864	1	Arabic	1.5
DOS866	1	Russian	1.5
DOS869	1	Modern Greek	1.5
GB18030	4	Chinese	2.5
GBK	2	Chinese	2.1
ISO8859_2	1	Latin-2, Central European	1.0
ISO8859_3	1	Latin-3, Southern European	1.5
ISO8859_4	1	Latin-4, Northern European	1.5
ISO8859_5	1	Cyrillic	1.5

ISO8859_6	1	Arabic	1.5
ISO8859_7	1	Greek	1.5
ISO8859_8	1	Hebrew	1.5
ISO8859_9	1	Latin-5, Turkish	1.5
ISO8859_13	1	Latin-7, Baltic Rim	1.5
KOI8R	1	Russian	2.0
KOI8U	1	Ukrainian	2.0
TIS620	1	Thai	2.1
UTF8 (*)	4	All	2.0
WIN1255	1	Hebrew	1.5
WIN1256	1	Arabic	1.5
WIN1257	1	Baltic	1.5
WIN1258	1	Vietnamese	2.0
WIN_1258 (alias for WIN1258)	1	Vietnamese	2.5

(*) In Firebird 1.5, UTF8 is an alias for UNICODE_FSS. This character set has some inherent problems. In Firebird 2, UTF8 is a character set in its own right, without the drawbacks of UNICODE_FSS.

See also:

- Character sets
- Default character set
- SET NAMES
- Firebird 2.0 Language Reference Update: Character set NONE
- Firebird 2.1 Release Notes: International language support (INTL)
- Firebird 2.1 Release Notes: Appendix B: International character sets
- Overview of the main character sets in Firebird
- Character sets and Unicode in Firebird
- Convert your Firebird applications to Unicode

back to top of page

Character set NONE handling changed

Changed in: 1.5.1

Description | ''

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. For more details, see the Note at the end of the book.

back to top of page

New collations

Added in: 1.0, 1.5, 1.5.1, 2.0, 2.1, 2.5

The following table lists the collations added in Firebird. The **Details** column is based on what has been reported in the Release Notes and other documents. The information in this column is probably incomplete; some collations with an empty *Details* field may still be case insensitive (*ci*), accent insensitive (*ai*) or dictionary-sorted (*dic*).

Please note that the default – binary – collations for new character sets are not listed here, as doing so would add no meaningful information.

Table 5.2. Collations new in Firebird

Character set	Collation	Language	Details	Added in
CP943C	CP943C_UNICODE	**Japanese**		**2.1**
GB18030	GB18030_UNICODE	**Chinese**		**2.5**
GBK	GBK_UNICODE	**Chinese**		**2.1**
ISO8859_1	ES_ES_CI_AI	**Spanish**	**ci, ai**	**2.0**
	FR_FR_CI_AI	**French**	**ci, ai**	**2.1**
	PT_BR	**Brazilian Portuguese**	**ci, ai**	**2.0**
ISO8859_2	CS_CZ	**Czech**		**1.0**
	ISO_HUN	**Hungarian**		**1.5**
	ISO_PLK	**Polish**		**2.0**
ISO8859_13	LT_LT	**Lithuanian**		**1.5.1**
UTF8	UCS_BASIC	**All**		**2.0**
	UNICODE	**All**	**dic**	**2.0**
	UNICODE_CI	**All**	**ci**	**2.1**
	UNICODE_CI_AI	**All**	**ci, ai**	**2.5**
WIN1250	BS_BA	**Bosnian**		**2.0**
	PXW_HUN	**Hungarian**	**ci**	**1.0**
	WIN_CZ	**Czech**	**ci**	**2.0**
	WIN_CZ_CI_AI	**Czech**	**ci, ai**	**2.0**
WIN1251	WIN1251_UA	**Ukrainian and Russian**		**1.5**
WIN1252	WIN_PTBR	**Brazilian Portuguese**	**ci, ai**	**2.0**
WIN1257	WIN1257_EE	**Estonian**	**dic**	**2.0**
	WIN1257_LT	**Lithuanian**	**dic**	**2.0**
	WIN1257_LV	**Latvian**	**dic**	**2.0**
KOI8R	KOI8R_RU	**Russian**	**dic**	**2.0**
TIS620	TIS620_UNICODE	**Thai**		**2.1**

A note on the *UTF8* collations: The UCS_BASIC collation sorts in Unicode code-point order: A, B, a, b, á... This is exactly the same as UTF8 with no collation specified. UCS_BASIC was added to comply with the SQL standard.

The UNICODE collation sorts using UCA (Unicode Collation Algorithm): a, A, á, b, B...

UNICODE_CI is truly case-insensitive. In a search for e.g. 'Apple', it will also find 'apple', 'APPLE' and 'aPPLe'.

UNICODE_CI_AI is accent-insensitive as well. According to this collation, 'APPEL' equals 'Appèl'.

back to top of page

Unicode collations for all character sets

Added in: 2.1

Firebird now comes with UNICODE collations for all the standard character sets. However, except for the ones listed in the new collations table in the previous section, these collations are not automatically available in your databases. Instead, they must be added with the CREATE COLLATION statement, like this:

```
create collation ISO8859_1_UNICODE for ISO8859_1
```

The new Unicode collations all have the name of their character set with _UNICODE added. (The built-in Unicode collations for UTF8 are the exception to the rule.) They are defined, along with the other collations, in the manifest file fbintl.conf in Firebird's /intl subdirectory.

Collations may also be registered under a user-chosen name, e.g.:

```
create collation LAT_UNI for ISO8859_1 from external ('ISO8859_1_UNICODE')
```

See CREATE COLLATION for the full syntax.

From:
http://ibexpert.com/docu/ - **IBExpert**

Permanent link:
http://ibexpert.com/docu/doku.php?id=01-documentation:01-09-sql-language-references:firebird2.5-language-reference-update:data-types-and-subtypes

Last update: **2023/08/01 16:46**