Firebird Null Guide

`NULL` behaviour and pitfalls in Firebird SQL

Paul Vinkenoog

26 January 2007 – Document version 1.0.1

Table of Contents

What is NULL?

Time and again, support questions pop up on the Firebird mailing lists about “strange things” happening with NULLs. The concept seems difficult to grasp – perhaps partly because of the name, which suggests a kind of “nothing” that won't do any harm if you add it to a number or stick it to the back of a string. In reality, performing such operations will render the entire expression NULL.

This guide explores the behaviour of NULL in Firebird SQL, points out common pitfalls and shows you how to deal safely with expressions that contain NULL or may resolve to NULL.

If you only need a quick reference to refresh your memory, go to the summary at the end of the guide.

So – what is it?

In SQL, NULL is not a value. It is a *state* indicating that an item's value is unknown or nonexistent. It is not zero or blank or an “empty string” and it does not behave like any of these values. Few things in SQL lead to more confusion than NULL, and yet its workings shouldn't be hard to understand as long as you stick to the following simple definition: NULL means *unknown*.

Let me repeat that:

> **NULL means UNKNOWN**

Keep this line in mind as you read through the rest of the guide, and most of the seemingly illogical results you can get with NULL will practically explain themselves.

> **Note**
>
> A few sentences and examples in this guide were taken from the *Firebird Quick Start Guide*, first published by IBPhoenix, now part of the Firebird Project.

NULL support in Firebird SQL

Only a few language elements are purposely designed to give an unambiguous result with NULL (unambiguous in the sense that some specific action is taken and/or a non-NULL result is returned). They are discussed in the following paragraphs.

Disallowing NULL

In a column or domain definition, you can specify that only non-NULL values may be entered by adding NOT NULL to the definition:

```
create table MyTable ( i int not null )
```

```
create domain DTown as varchar( 32 ) not null
```

Special care should be taken when adding a NOT NULL field to an existing table that already contains records. This operation will be discussed in detail in the section *Altering populated tables*.

Testing for NULL

If you want to know whether a variable, field or other expression is NULL, use the following syntax:

> *<expression>* IS [NOT] NULL

Examples:

```
if ( MyField is null ) then YourString = 'Dunno'
```

```
select * from Pupils where PhoneNumber is not null
```

```
select * from Pupils where not ( PhoneNumber is null )
/* does the same as the previous example */
```

```
update Numbers set Total = A + B + C where A + B + C is not null
```

```
delete from Phonebook where PhoneNum is null
```

Do **not** use "... = NULL" to test for nullness. This syntax is illegal in Firebird versions up to 1.5.n, and gives the wrong result in Firebird 2 and up: it returns NULL no matter what you compare. This is by design, incidentally, and in that sense it's not *really* wrong – it just doesn't give you what you want. The same goes for "... <> NULL", so don't use that either; use IS NOT NULL instead.

IS NULL and IS NOT NULL always return true or false; they never return NULL.

Assigning NULL

Setting a field or variable to NULL is done with the "=" operator, just like assigning values. You can also include NULL in an insert list:

```
if ( YourString = 'Dunno' ) then MyField = null
```

```
update Potatoes set Amount = null where Amount < 0
```

```
insert into MyTable values ( 3, '8-May-2004', NULL, 'What?' )
```

Remember:

- You cannot – and should not – use the *comparison operator* "=" to *test* if something *is* NULL...
- ...but you can – and often must – use the *assignment operator* "=" to *set* something *to* NULL.

Testing DISTINCTness (Firebird 2+)

In Firebird 2 and higher only, you can test for the null-encompassing equality of two expressions with "IS [NOT] DISTINCT FROM":

```
if ( A is distinct from B ) then...
```

```
if ( Buyer1 is not distinct from Buyer2 ) then...
```

Fields, variables and other expressions are considered:

- DISTINCT if they have different values or if one of them is `NULL` and the other isn't;
- NOT DISTINCT if they have the same value or if both of them are `NULL`.

[NOT] DISTINCT always returns `true` or `false`, never `NULL` or something else.

With earlier Firebird versions, you have to write special code to obtain the same information. This will be discussed later.

The NULL literal

The ability to use `NULL` literals depends on your Firebird version.

Firebird 1.5 and below

In Firebird 1.5 and below you can only use the literal word “`NULL`” in a few situations, namely the ones described in the previous paragraphs plus a few others such as “cast(`NULL` as `<datatype>`)” and “select `NULL` from MyTable”.

In all other circumstances, Firebird will complain that `NULL` is an unknown token. If you really *must* use `NULL` in such a context, you have to resort to tricks like “cast(`NULL` as int)”, or using a field or variable that you know is `NULL`, etc.

Firebird 2.0 and up

Firebird 2 allows the use of `NULL` literals in every context where a normal value can also be entered. You can e.g. include `NULL` in an IN() list, write expressions like “if (MyField = `NULL`) then...”, and so on. However, as a general rule you should **not** make use of these new possibilities! In almost every thinkable situation, such use of `NULL` literals is a sign of poor SQL design and will lead to `NULL` results where you meant to get `true` or `false`. In that sense the earlier, more restrictive policy was safer, although you could always bypass it with casts etc. – but at least you had to take deliberate steps to bypass it.

NULL in operations

As many of us have found out to our chagrin, `NULL` is contagious: use it in a numerical, string or date/time operation, and the result will invariably be `NULL`. With boolean operators, the outcome depends on the type of operation and the value of the other operand.

Please remember that in Firebird versions prior to 2.0 it is mostly illegal to use the constant `NULL` directly in operations or comparisons. Wherever you see `NULL` in the expressions below, read it as “a field, variable or other expression that resolves to `NULL`”. In Firebird 2 and above this expression may also be a `NULL` literal.

Mathematical and string operations

The operations in this list *always* return `NULL`:

- `1 + 2 + 3 + NULL`
- `5 * NULL - 7`
- `'Home ' || 'sweet ' || NULL`
- `MyField = NULL`
- `MyField <> NULL`
- `NULL = NULL`

If you have difficulty understanding why, remember that `NULL` means “unknown”. You can also look at the following table where per-case explanations are provided. In the table we don't write `NULL` in the expressions (as said, this is often illegal); instead, we use two entities A and B that are both `NULL`. A and B may be fields, variables, or even composite subexpressions – as long as they're `NULL`, they'll all behave the same in the enclosing expressions.

Table 1. Operations on null entities A and B

If A and B are NULL, then:	Is:	Because:
`1 + 2 + 3 + A`	`NULL`	If A is unknown, then 6 + A is also unknown.
`5 * A - 7`	`NULL`	If A is unknown, then 5 * A is also unknown. Subtract 7 and you end up with another unknown.
`'Home ' || 'sweet ' || A`	`NULL`	If A is unknown, 'Home sweet ' || A is unknown.
`MyField = A`	`NULL`	If A is unknown, you can't tell if MyField has the same value...
`MyField <> A`	`NULL`	...but you also can't tell if MyField has a *different* value!
`A = B`	`NULL`	With A and B unknown, it's impossible to know if they are equal.

Here is the complete list of math and string operators that return `NULL` if at least one operand is `NULL`:

- `+`, `-`, `*`, `/`, and `%`
- `!=`, `~=`, and `^=` (synonyms of `<>`)
- `<`, `<=`, `>`, and `>=`
- `!<`, `~<`, and `^<` (low-precedence synonyms of `>=`)
- `!>`, `~>`, and `^>` (low-precedence synonyms of `<=`)
- `||`
- [NOT] BETWEEN
- [NOT] STARTING WITH
- [NOT] LIKE
- [NOT] CONTAINING

The explanations all follow the same pattern: if A is unknown, you can't tell if it's greater than B; if string S1 is unknown, you can't tell if it contains S2; etcetera.

Using LIKE with a `NULL` escape character would crash the server in Firebird versions up to and including 1.5. This bug was fixed in v. 1.5.1. From that version onward, such a statement will yield an empty result set.

Boolean operations

All the operators examined so far return `NULL` if any operand is `NULL`. With boolean operators, things are a bit more complex:

- `not NULL = NULL`
- `NULL or false = NULL`
- `NULL or true = true`
- `NULL or NULL = NULL`
- `NULL and false = false`
- `NULL and true = NULL`
- `NULL and NULL = NULL`

In reality, Firebird SQL doesn't have a boolean data type; nor are `true` and `false` existing constants. In the leftmost column of the explanatory table below, “`true`” and “`false`” represent expressions (fields, variables, composites...) that evaluate to `true`/`false`.

Table 2. Boolean operations on null entity A

If A is `NULL`, then:	Is:	Because:
`not A`	`NULL`	If A is unknown, its inverse is also unknown.
`A or false`	`NULL`	“`A or false`” always has the same value as A – which is unknown.
`A or true`	`true`	“`A or true`” is always `true` – A's value doesn't matter.
`A or A`	`NULL`	“`A or A`” always equals A – which is `NULL`.
`A and false`	`false`	“`A and false`” is always `false` – A's value doesn't matter.
`A and true`	`NULL`	“`A and true`” always has the same value as A – which is unknown.
`A and A`	`NULL`	“`A and A`” always equals A – which is `NULL`.

All these results are in accordance with boolean logic. The fact that you don't need to know X's value to compute “X or `true`” and “X and `false`” is also the basis of a feature found in various programming languages: short-circuit boolean evaluation.

The above results can be generalised as follows for expressions with one type of binary boolean operator (`and` | `or`) and any number of operands:

Disjunctions (“A or B or C or D or ...”)

1. If at least one operand is `true`, the result is `true`.
2. Else, if at least one operand is `NULL`, the result is `NULL`.
3. Else (i.e. if all operands are `false`) the result is `false`.

Conjunctions (“A and B and C and D and ...”)

1. If at least one operand is `false`, the result is `false`.

2. Else, if at least one operand is `NULL`, the result is `NULL`.
3. Else (i.e. if all operands are `true`) the result is `true`.

Or, shorter:

- `TRUE` beats `NULL` in a disjunction (OR-operation);
- `FALSE` beats `NULL` in a conjunction (AND-operation);
- In all other cases, `NULL` wins.

If you have trouble remembering which constant rules which operation, look at the second letter: t**R**ue prevails with o**R** — f**A**lse with **A**nd.

More logic (or not)

The short-circuit results obtained above may lead you to the following ideas:

- 0 times `x` equals 0 for every `x`. Hence, even if `x`'s value is unknown, `0 * x` is 0. (Note: this only holds if `x`'s datatype only contains numbers, not `NaN` or infinities.)

- The empty string is ordered lexicographically before every other string. Therefore, `S >= ''` is true whatever the value of S.

- Every value equals itself, whether it's unknown or not. So, although `A = B` justifiably returns `NULL` if A and B are different `NULL` entities, `A = A` should always return `true`, even if A is `NULL`. The same goes for `A <= A` and `A >= A`.

  By analogous logic, `A <> A` should always be `false`, as well as `A < A` and `A > A`.

- Every string *contains* itself, *starts with* itself and is *like* itself. So, "`S CONTAINING S`", "`S STARTING WITH S`" and "`S LIKE S`" should always return `true`.

How is this reflected in Firebird SQL? Well, I'm sorry I have to inform you that despite this compelling logic – and the analogy with the boolean results discussed above – the following expressions all resolve to `NULL`:

- `0 * NULL`
- `NULL >= ''` and `'' <= NULL`
- `A = A`, `A <= A` and `A >= A`
- `A <> A`, `A < A` and `A > A`
- `S CONTAINING S`, `S STARTING WITH S` and `S LIKE S`

So much for consistency.

Internal functions and directives

Internal functions

The following built-in functions return `NULL` if at least one argument is `NULL`:

- CAST()
- EXTRACT()
- GEN_ID()
- SUBSTRING()
- UPPER()
- LOWER()
- BIT_LENGTH()
- CHAR[ACTER]_LENGTH()
- OCTET_LENGTH()
- TRIM()

Notes

- In 1.0.0, EXTRACT from a `NULL` date would crash the server. Fixed in 1.0.2.
- If the first argument to GEN_ID is a valid generator name and the second argument is `NULL`, the named generator keeps its current value.
- In versions up to and including 2.0, SUBSTRING results are sometimes returned as “false emptystrings”. These strings are in fact `NULL`, but are described by the server as non-nullable. Therefore, most clients show them as empty strings. See the bugs list for a detailed description.

FIRST, SKIP and ROWS

The following two directives **crash** a Firebird 1.5.n or lower server if given a `NULL` argument. In Firebird 2, they treat `NULL` as the value 0:

- FIRST
- SKIP

This new Firebird 2 directive returns an empty set if any argument is `NULL`:

- ROWS

Side note: ROWS complies with the SQL standard. In new code, use ROWS, not FIRST and SKIP.

Predicates

Predicates are statements about objects that return a boolean result: `true`, `false` or `unknown` (= `NULL`). In computer code you typically find predicates in places where as yes/no type of decision has to be taken. For Firebird SQL, that means in WHERE, HAVING, CHECK, CASE WHEN, IF and WHILE clauses.

Comparisons such as “x > y” also return boolean results, but they are generally not called predicates, although this is mainly a matter of form. An expression like Greater(x, y) that does exactly the same would immediately qualify as a predicate. (Mathematicians like predicates to have a *name* – such as “Greater” or just plain “G” – and a pair of *parentheses* to hold the arguments.)

Firebird supports the following SQL predicates: IN, ANY, SOME, ALL, EXISTS and SINGULAR.

Note

It is also perfectly defensible to call “IS [NOT] NULL” and “IS [NOT] DISTINCT FROM” predicates, despite the absence of parentheses. But, predicates or not, they have already been introduced and won't be discussed in this section.

The IN predicate

The IN predicate compares the expression on its left-hand side to a number of expressions passed in the argument list and returns `true` if a match is found. NOT IN always returns the opposite of IN. Some examples of its use are:

```
select RoomNo, Floor from Classrooms where Floor in (3, 4, 5)
```

```
delete from Customers where upper(Name) in ('UNKNOWN', 'NN', '')
```

```
if ( A not in (MyVar, MyVar + 1, YourVar, HisVar) ) then ...
```

The list can also be generated by a one-column subquery:

```
select ID, Name, Class from Students
  where ID in (select distinct LentTo from LibraryBooks)
```

With an empty list

If the list is empty (this is only possible with a subquery), IN always returns `false` and NOT IN always returns `true`, even if the test expression is `NULL`. This makes sense: even if a value is unknown, it's certain not to occur in an empty list.

With a `NULL` test expression

If the list is not empty and the test expression – called “A” in the examples below – is `NULL`, the following predicates will always return `NULL`, regardless of the expressions in the list:

- A IN (Expr1, Expr2, ..., Expr*N*)
- A NOT IN (Expr1, Expr2, ..., Expr*N*)

The first result can be understood by writing out the entire expression as a disjunction (OR-chain) of equality tests:

A=Expr1 or A=Expr2 or ... or A=Expr*N*

which, if A is `NULL`, boils down to

`NULL` or `NULL` or ... or `NULL`

which is `NULL`.

The nullness of the second predicate follows from the fact that “not (`NULL`)” equals `NULL`.

With NULLs in the list

If A has a proper value but the list contains one or more NULL expressions, things become a little more complicated:

- If at least one of the expressions in the list has the same value as A:

  - "A IN(Expr1, Expr2, ..., Expr*N*)" returns true
  - "A NOT IN(Expr1, Expr2, ..., Expr*N*)" returns false

  This is due to the fact that "true or NULL" returns true (see above). Or, more general: a disjunction where at least one of the elements is true, returns true even if some other elements are NULL. (Any falses, if present, are not in the way. In a disjunction, true rules.)

- If none of the expressions in the list have the same value as A:

  - "A IN(Expr1, Expr2, ..., Expr*N*)" returns NULL
  - "A NOT IN(Expr1, Expr2, ..., Expr*N*)" returns NULL

  This is because "false or NULL" returns NULL. In generalised form: a disjunction that has only false and NULL elements, returns NULL.

Needless to say, if neither A nor any list expression is NULL, the result is always as expected and can only be true or false.

IN() results

The table below shows all the possible results for IN and NOT IN. To use it properly, start with the first question in the left column. If the answer is No, move on to the next line. As soon as an answer is Yes, read the results from the second and third columns and you're done.

Table 3. Results for "A [NOT] IN (<list>)"

Conditions	**Results**	
	IN()	**NOT IN()**
Is the list empty?	false	true
Else, is A NULL?	NULL	NULL
Else, is at least one list element equal to A?	true	false
Else, is at least one list element NULL?	NULL	NULL
Else (i.e. all list elements are non-NULL and unequal to A)	false	true

In many contexts (e.g. within IF and WHERE clauses), a NULL result behalves like false in that the condition is not satisfied when the test expression is NULL. On the one hand this is convenient for cases where you might expect false but NULL is returned: you simply won't notice the difference. On the other hand, this may also lead you to expect `true` when the expression is inverted (using NOT) and this is where you'll run into trouble. In that sense, the most "dangerous" case in the above table is when you use an expression of the type "A NOT IN (<list>)", with A indeed not present in the list (so you'd expect a clear `true` result) but the list happens to contain one or more `NULL`s.

Caution

Be especially careful if you use NOT IN with a subselect instead of an explicit list, e.g.

```
A not in ( select Number from MyTable )
```

If A is not present in the Number column, the result is `true` if no Number is `NULL`, but `NULL` if the column does contain a `NULL` entry. Please be aware that even in a situation where A is constant and its value is never contained in the Number column, the result of the expression (and therefore your program flow) may still vary over time according to the absence or presence of `NULL`s in the column. Hours of debugging fun! Of course you can avoid this particular problem simply by adding "where Number is not `NULL`" to the subselect.

Bug alert

All Firebird versions before 2.0 contain a bug that causes [NOT] IN to return the wrong result if an index is active on the subselect and one of the following conditions is true:

- A is `NULL` and the subselect doesn't return any `NULL`s, or
- A is not `NULL` and the subselect result set doesn't contain A but does contain `NULL`(s).

Please realise that an index may be active even if it has not been created explicitly, namely if a key is defined on A.

Example: Table TA has a column A with values { 3, 8 }. Table TB has a column B containing { 2, 8, 1, `NULL` }. The expressions:

```
A [not] in ( select B from TB )
```

should both return `NULL` for A = 3, because of the `NULL` in B. But if B is indexed, IN returns `false` and NOT IN returns `true`. As a result, the query

```
select A from TA where A not in ( select B from TB )
```

returns a dataset with one record – containing the field with value 3 – while it should have returned an empty set. Other errors may also occur, e.g. if you use "NOT IN" in an IF, CASE or WHILE statement.

As an alternative to NOT IN, you can use "<> ALL". The ALL predicate will be introduced shortly.

IN() in CHECK constraints

The IN() predicate is often used in CHECK constraints. In that context, `NULL` expressions have a surprisingly different effect in Firebird versions 2.0 and up. This will be discussed in the section *CHECK constraints*.

The ANY, SOME and ALL quantifiers

Firebird has two quantifiers that allow you to compare a value to the results of a subselect:

- ALL returns `true` if the comparison is true for *every* element in the subselect.

- ANY and SOME (full synonyms) return `true` if the comparison is true for *at least one* element in the subselect.

With ANY, SOME and ALL you provide the comparison operator yourself. This makes it more flexible than IN, which only supports the (implicit) “=” operator. On the other hand, ANY, SOME and ALL only accept a subselect as an argument; you can't provide an explicit list, as with IN.

Valid operators are `=`, `!=`, `<`, `>`, `=<`, `=>` and all their synonyms. You can't use LIKE, CONTAINING, IS DISTINCT FROM, or any other operators.

Some usage examples:

```
select name, income from blacksmiths
  where income > any( select income from goldsmiths )
```

(returns blacksmiths who earn more than at least one goldsmith)

```
select name, town from blacksmiths
  where town != all( select distinct town from goldsmiths )
```

(returns blacksmiths who live in a goldsmithless town)

```
if ( GSIncome !> some( select income from blacksmiths ) )
  then PoorGoldsmith = 1;
  else PoorGoldsmith = 0;
```

(sets PoorGoldsmith to 1 if at least one blacksmith's income is not less than the value of GSIncome)

Result values

If the subselect returns an empty set, ALL returns `true` and ANY|SOME return `false`, even if the left-hand side expression is `NULL`. This follows from the definitions and the rules of formal logic. (Math-heads will already have noticed that ALL is equivalent to the universal (“A”) quantifier and ANY|SOME to the existential (“E”) quantifier.)

For non-empty sets, you can write out “A <*op*> ANY|SOME (<*subselect*>)” as

A <*op*> E1 **or** A <*op*> E2 **or** ... **or** A <*op*> E*n*

with <*op*> the operator used and E1, E2 etc. the items returned by the subquery.

Likewise, “A <*op*> ALL (<*subselect*>)” is the same as

A <*op*> E1 **and** A <*op*> E2 **and** ... **and** A <*op*> E*n*

This should look familiar. The first writeout is equal to that of the IN predicate, except that the operator may now be something other than “=”. The second is different but has the same general form. We can now work out how nullness of A and/or nullness of subselect results affect the outcome of ANY|SOME and ALL. This is done in the same way as earlier with IN, so instead of including all the steps here we will just present the result tables. Again, read the questions in the left column from top to bottom. As soon as you answer a question with “Yes”, read the result from the second column and you're done.

Table 4. Results for “A <op> ANY|SOME (<subselect>)”

Conditions	Result
	ANY | SOME
Does the subselect return an empty set?	`false`
Else, is A `NULL`?	`NULL`
Else, does at least one comparison return `true`?	`true`
Else, does at least one comparison return `NULL`?	`NULL`
Else (i.e. all comparisons return `false`)	`false`

If you think these results look a lot like what we saw with IN(), you're right: with the “=” operator, ANY is the same as IN. In the same way, “<> ALL” is equivalent to NOT IN.

Bug alert (revisited)

In versions before 2.0, “= ANY” suffers from the same bug as IN. Under the “right” circumstances, this can lead to wrong results with expressions of the type “NOT A = ANY(...)”.

On the bright side, “<> ALL” is not affected and will always return the right result.

Table 5. Results for “A <op> ALL (<subselect>)”

Conditions	Result
	ALL
Does the subselect return an empty set?	`true`
Else, is A `NULL`?	`NULL`
Else, does at least one comparison return `false`?	`false`
Else, does at least one comparison return `NULL`?	`NULL`
Else (i.e. all comparisons return `true`)	`true`

ALL bug

Although “<> ALL” always works as it should, ALL should nevertheless be considered broken in all pre-2.0 versions of Firebird: with every operator other than “<>”, wrong results may be returned if an index is active on the subselect – with or without `NULL`s around.

Note

Strictly speaking, the second question in both tables ("is A `NULL`?") is redundant and can be dropped. If A is `NULL`, all the comparisons return `NULL`, so that situation will be caught a little later. And while we're at it, we could drop the first question too: the "empty set" situation is just a special case of the final "else". The whole thing then once again boils down to "`true` beats `NULL` beats `false`" in disjunctions (ANY|SOME) and "`false` beats `NULL` beats `true`" in conjunctions (ALL).

The reason we included those questions is convenience: you can see if a set is empty at a glance, and it's also easier to check if the left-hand side expression is `NULL` than to evaluate each and every comparison result. But do feel free to skip them, or to skip just the second. Do *not*, however, skip the first question and start with the second: this will lead to a wrong conclusion if the set is empty!

EXISTS and SINGULAR

The EXISTS and SINGULAR predicates return information about a subquery, usually a correlated subquery. You can use them in WHERE, HAVING, CHECK, CASE, IF and WHILE clauses (the latter two are only available in PSQL, Firebird's stored procedure and trigger language).

EXISTS

EXISTS tells you whether a subquery returns at least one row of data. Suppose you want a list of farmers who are also landowners. You could get one like this:

```
SELECT Farmer FROM Farms WHERE EXISTS
  (SELECT * FROM Landowners
   WHERE Landowners.Name = Farms.Farmer)
```

This query returns the names of all farmers who also figure in the Landowners table. The EXISTS predicate returns `true` if the result set of the subselect contains at least one row. If it is empty, EXISTS returns `false`. EXISTS never returns `NULL`, because a result set always either has rows, or hasn't. Of course the subselect's search condition may evolve to `NULL` for certain rows, but that doesn't cause any uncertainty: such a row won't be included in the subresult set.

Note

In reality, the subselect doesn't return a result set at all. The engine simply steps through the Landowners records one by one and applies the search condition. If it evolves to `true`, EXISTS returns `true` immediately and the remaining records aren't checked. If it evolves to `false` or `NULL`, the search continues. If all the records have been searched and there hasn't been a single `true` result, EXISTS returns `false`.

NOT EXISTS always returns the opposite of EXISTS: `false` or `true`, never `NULL`. NOT EXISTS returns `false` immediately if it gets a `true` result on the subquery's search condition. Before returning `true` it must step through the entire set.

SINGULAR

SINGULAR is an InterBase/Firebird extension to the SQL standard. It is often described as returning `true` if exactly one row in the subquery meets the search condition. By analogy with EXISTS this would make you expect that SINGULAR too will only ever return `true` or `false`. After all, a result set has either exactly 1 row or a different number of rows. Unfortunately, all versions of Firebird up to and including 2.0 have a bug that causes `NULL` results in a number of cases. The behaviour is pretty inconsistent, but at the same time fully reproducible. For instance, on a column A containing (1, `NULL`, 1), a SINGULAR test with subselect "`A=1`" returns `NULL`, but the same test on a column with (1, 1, `NULL`) returns `false`. Notice that only the insertion order is different here!

To make matters worse, all versions prior to 2.0 sometimes return `NULL` for NOT SINGULAR where `false` or `true` is returned for SINGULAR. In 2.0, this at least doesn't happen anymore: it's either `false` vs. `true` or twice `NULL`.

The code has been fixed for Firebird 2.1; from that version onward SINGULAR will return:

- `false` if the search condition is never `true` (this includes the empty-set case);
- `true` if the search condition is `true` for exactly 1 row;
- `false` if the search condition is `true` for more than 1 row.

Whether the other rows yield `false`, `NULL` or a combination thereof, is irrelevant.

NOT SINGULAR will always return the opposite of SINGULAR (as is already the case in 2.0).

In the meantime, if there's *any* chance that the search condition may evolve to `NULL` for one or more rows, you should always add an IS NOT NULL condition to your [NOT] SINGULAR clauses, e.g. like this:

```
... SINGULAR( SELECT * from MyTable
              WHERE MyField > 38
              AND MyField IS NOT NULL )
```

Searches

If the search condition of a SELECT, UPDATE or DELETE statement resolves to `NULL` for a certain row, the effect is the same as if it had been `false`. Put another way: if the search expression is `NULL`, the condition is not met, and consequently the row is not included in the output set (or is not updated/deleted).

Note

The *search condition* or *search expression* is the WHERE clause minus the WHERE keyword itself.

Some examples (with the search condition in boldface):

```
SELECT Farmer, Cows FROM Farms WHERE Cows > 0 ORDER BY Cows
```

The above statement will return the rows for farmers that are known to possess at least one cow. Farmers with an unknown (`NULL`) number of cows will not be included, because the expression "`NULL > 0`" returns `NULL`.

```
SELECT Farmer, Cows FROM Farms WHERE NOT (Cows > 0) ORDER BY Cows
```

Now, it's tempting to think that this will return "all the other records" from the Farms table, right? But it won't – not if the Cows column contains any `NULL`s. Remember that `not(NULL)` is itself `NULL`. So for any row where Cows is `NULL`, "`Cows > 0`" will be `NULL`, and "`NOT (Cows > 0)`" will be `NULL` as well.

```
SELECT Farmer, Cows, Sheep FROM Farms WHERE Cows + Sheep > 0
```

On the surface, this looks like a query returning all the farms that have at least one cow and/or sheep (assuming that neither Cows nor Sheep can be a negative number). However, if farmer Fred has 30 cows and an unknown number of sheep, the sum `Cows + Sheep` becomes `NULL`, and the entire search expression boils down to "`NULL > 0`", which is... you got it. So despite his 30 cows, our friend Fred won't make it into the result set.

As a last example, we shall rewrite the previous statement so that it *will* return any farm which has at least one animal of a known kind, even if the other number is `NULL`. To do that, we exploit the fact that "`NULL or true`" returns `true` – one of the rare occasions where a `NULL` operand doesn't render the entire expression `NULL`:

```
SELECT Farmer, Cows, Sheep FROM Farms WHERE Cows > 0 OR Sheep > 0
```

This time, Fred's thirty cows will make the first comparison `true`, while the sheep bit is still `NULL`. So we have "`true or NULL`", which is `true`, and the row will be included in the output set.

Caution

If your search condition contains one or more IN predicates, there is the additional complication that some of the list elements (or subselect results) may be `NULL`. The implications of this are discussed in *The IN() predicate*.

Sorts

In Firebird 2, `NULL`s are considered "smaller" than anything else when it comes to sorting. Consequently, they come first in ascending sorts and last in descending sorts. You can override this default placement by adding a NULLS FIRST or NULLS LAST directive to the ORDER BY clause.

In earlier versions, `NULL`s were always placed at the end of a sorted set, no matter whether the order was ascending or descending. For Firebird 1.0, that was the end of the story: `NULL`s would always come last in any sorted set, period. Firebird 1.5 introduced the NULLS FIRST/LAST syntax, so you could force them to the top or bottom.

To sum it all up:

Table 6. `NULL` placement in ordered sets

Ordering	NULLs placement		
	Firebird 1	Firebird 1.5	Firebird 2
order by Field [asc]	bottom	bottom	top
order by Field desc	bottom	bottom	bottom
order by Field [asc | desc] nulls first	—	top	top
order by Field [asc | desc] nulls last	—	bottom	bottom

Specifying NULLS FIRST on an ascending or NULLS LAST on a descending sort in Firebird 2 is of course rather pointless, but perfectly legal. The same is true for NULLS LAST on any sort in Firebird 1.5.

Notes

- If you override the default NULLs placement, no index will be used for sorting. In Firebird 1.5, that is the case with NULLS FIRST. In 2.0 and higher, with NULLS LAST on ascending and NULLS FIRST on descending sorts.
- If you open a pre-2.0 database with Firebird 2, it will show the *old* NULL ordering behaviour (that is: at the bottom, unless overridden by NULLS FIRST). A backup-restore cycle will fix this, provided that at least the restore is executed with Firebird 2's gbak!
- Firebird 2.0 has a bug that causes the NULLS FIRST|LAST directive to fail under certain circumstances with SELECT DISTINCT. See the bugs list for more details.

Warning

Don't be tempted into thinking that, because NULL is the "smallest thing" in sorts since Firebird 2, an expression like "NULL < 3" will now also return true. It won't. Using NULL in this kind of expression will always give a NULL outcome.

Aggregate functions

The aggregate functions – COUNT, SUM, AVG, MAX, MIN and LIST – don't handle NULL in the same way as ordinary functions and operators. Instead of returning NULL as soon as a NULL operand is encountered, they only take non-NULL fields into consideration while computing the outcome. That is, if you have this table:

MyTable

ID	Name	Amount
1	John	37
2	Jack	NULL
3	Jim	5
4	Joe	12
5	Josh	NULL

...the statement **select sum(Amount) from MyTable** returns 54, which is 37 + 5 + 12. Had all five fields been summed, the result would have been NULL. For AVG, the non-NULL fields are summed and the sum divided by the number of non-NULL fields.

There is one exception to this rule: COUNT(*) returns the count of all rows, even rows whose fields are all NULL. But COUNT(*FieldName*) behaves like the other aggregate functions in that it only counts rows where the specified field is not NULL.

Another thing worth knowing is that COUNT(*) and COUNT(*FieldName*) never return NULL: if there are no rows in the set, both functions return 0. COUNT(*FieldName*) also returns 0 if all *FieldName* fields in the set are NULL. The other aggregate functions return NULL in such cases. Be warned that SUM even returns NULL if used on an empty set, which is contrary to common logic (if there are no rows, the average, maximum and minimum are undefined, but the sum is *known* to be zero).

Now let's put all that knowledge in a table for your easy reference:

Table 7. Aggregate function results with different column states

Function	Results		
	Empty set	**All-null set or column**	**Other sets or columns**
COUNT(*)	0	Total number of rows	Total number of rows
COUNT(Field)	0	0	Number of rows where Field is not `NULL`
MAX, MIN	`NULL`	`NULL`	Max or min value found in the column
SUM	`NULL`	`NULL`	Sum of non-`NULL` values in the column
AVG	`NULL`	`NULL`	Average of non-`NULL` values in the column. This equals SUM(Field) / COUNT(Field).[a]
LIST[b]	`NULL`	`NULL`	Comma-separated string concatenation of non-`NULL` values in the column

[a]If Field is of an integer type, AVG is always rounded towards 0. For instance, 6 non-null INT records with a sum of -11 yield an average of -1, not -2.
[b]LIST was added in Firebird 2.1

The GROUP BY clause

A GROUP BY clause doesn't change the aggregate function logic described above, except that it is now applied to each group individually rather than to the result set as a whole. Suppose you have a table Employee, with fields Dept and Salary which both allow `NULL`s, and you run this query:

```
SELECT Dept, SUM(Salary) FROM Employee GROUP BY Dept
```

The result may look like this (the row where Dept is `<null>` may be at the top or bottom, depending on your Firebird version):

```
DEPT                      SUM
====== =====================
<null>               219465.19
000                  266643.00
100                  155262.50
110                  130442.81
115                13480000.00
120                     <null>
121                  110000.00
123                  390500.00
```

First notice that the people whose department is unknown (`NULL`) are grouped together, although you can't say that they have the same *value* in the Dept field. But the alternative would have been to give each of those records a “group” of their own. Not only would this possibly add a huge number of lines to the output, but it would also defeat the purpose of *group*ing: those lines wouldn't be aggregates, but simple “SELECT Dept, Salary” rows. So it makes sense to group the `NULL` depts by their state and the rest by their value.

Anyway, the Dept field is not what interests us most. What does the aggregate SUM column tell us? That all salaries are non-`NULL`, except in department 120? No. All we can say is that in every department except 120, there is at least one employee with a known salary in the database. Each department *may* contain `NULL` salaries; in dept. 120 *all* the salaries are `NULL`.

You can find out more by throwing in one or more COUNT() columns. For instance, if you want to know the number of `NULL` salaries in each group, add a column “COUNT(*) – COUNT(Salary)”.

Counting frequencies

A GROUP BY clause can be used to report the frequencies with which values occur in a table. In that case you use the same field name several times in the query statement. Let's say you have a table TT with a column A whose contents are { 3, 8, `NULL`, 6, 8, -1, `NULL`, 3, 1 }. To get a frequencies report, you could use:

```
SELECT A, COUNT(A) FROM TT GROUP BY A
```

which would give you this result:

```
           A        COUNT
============ ============
          -1            1
           1            1
           3            2
           6            1
           8            2
      <null>            0
```

Oops – something went wrong with the `NULL` count, but what? Remember that `COUNT(`*`FieldName`*`)` skips all `NULL` fields, so with `COUNT(`*`A`*`)` the count of the `<null>` group can only ever be 0. Reformulate your query like this:

```
SELECT A, COUNT(*) FROM TT GROUP BY A
```

and the correct value will be returned (in casu 2).

The HAVING clause

HAVING clauses can place extra restrictions on the output rows of an aggregate query – just like WHERE clauses do in record-by-record queries. A HAVING clause can impose conditions on any output column or combination of columns, aggregate or not.

As far as `NULL` is concerned, the following two facts are worth knowing (and hardly surprising, I would guess):

- Rows for which the HAVING condition evaluates to `NULL` won't be included in the result set. (“Only `true` is good enough.”)
- “HAVING *`<col>`* IS [NOT] NULL” is a legal and often useful condition, whether *`<col>`* is aggregate or not. (But if *`<col>`* is non-aggregate, you may save the engine some work by changing HAVING to WHERE and placing the condition before the “GROUP BY” clause. This goes for any condition on non-aggregate columns.)

For instance, adding the following clause to the example query from the “GROUP BY” paragraph:

```
...HAVING Dept IS NOT NULL
```

will prevent the first row from being output, whereas this one:

```
...HAVING SUM(Salary) IS NOT NULL
```

suppresses the sixth row (the one with Dept = 120).

Conditional statements and loops

IF statements

If the test expression of an IF statement resolves to `NULL`, the THEN clause is skipped and the ELSE clause – if present – executed. In other words, `NULL` and `false` have the same effect in this context. So in situations where you would logically expect `false` but `NULL` is returned, no harm will be done. However, we've already seen examples of `NULL` being returned where you would expect `true`, and that *does* affect the flow of the code!

Below are some examples of the seemingly paradoxical (but perfectly correct) results you can get if `NULL`s creep into your IF statements.

> **Tip**
>
> If you use Firebird 2 or higher, you can avoid all the pitfalls discussed here, simply by using [NOT] DISTINCT instead of the “=” and “<>” operators!

-
  ```
  if (a = b) then
    MyVariable = 'Equal';
  else
    MyVariable = 'Not equal';
  ```

  If `a` and `b` are both `NULL`, `MyVariable` will yet be “`Not equal`” after executing this code. The reason is that the expression “`a = b`” yields `NULL` if at least one of them is `NULL`. With a `NULL` test expression, the THEN block is skipped and the ELSE block executed.

-
  ```
  if (a <> b) then
    MyVariable = 'Not equal';
  else
    MyVariable = 'Equal';
  ```

  Here, `MyVariable` will be “`Equal`” if `a` is `NULL` and `b` isn't, or vice versa. The explanation is analogous to that of the previous example.

So how should you set up equality tests that *do* give the logical result under all circumstances, even with `NULL` operands? In Firebird 2 you can use DISTINCT, as already shown (see *Testing DISTINCTness*). With earlier versions, you'll have to write some more code. This is discussed in the section *Equality tests*, later on in this guide. For now, just remember that you have to be very careful with IF conditions that may resolve to `NULL`.

Another aspect you shouldn't forget is the following: a `NULL` test expression may *behave* like `false` in an IF condition, but it doesn't have the *value* `false`. It's still `NULL`, and that means that its inverse will also be `NULL`

– not “`true`”. As a consequence, inverting the test expression and swapping the THEN and ELSE blocks may change the behaviour of the IF statement. In binary logic, where only `true` and `false` can occur, such a thing could never happen.

To illustrate this, let's refactor the last example:

-
  ```
  if (not (a <> b)) then
    MyVariable = 'Equal';
  else
    MyVariable = 'Not equal';
  ```

  In the original version, if one operand was `NULL` and the other wasn't (so they were intuitively unequal), the result was “`Equal`”. Here, it's “`Not equal`”. The explanation: one operand is `NULL`, therefore “`a <> b`” is `NULL`, therefore “`not(a <> b)`” is `NULL`, therefore ELSE is executed. While this result is correct where the original had it wrong, there's no reason to rejoice: in the refactored version, the result is also “`Not equal`” if both operands are `NULL` – something that the original version “got right”.

Of course, as long as no operand in the test expression can ever be `NULL`, you can happily formulate your IF statements like above. Also, refactoring by inverting the test expression and swapping the THEN and ELSE blocks will always preserve the functionality, regardless of the complexity of the expressions – as long as they aren't `NULL`. What's especially treacherous is when the operands are *almost always* non-`NULL`, so in the vast majority of cases the results will be correct. In such a situation those rare `NULL` cases may go unnoticed for a long time, silently corrupting your data.

CASE statements

Firebird introduced the CASE construct in version 1.5, with two syntactic variants. The first one is called the *simple syntax*:

```
case <expression>
  when <exp1> then <result1>
  when <exp2> then <result2>
  ...
  [else <defaultresult>]
end
```

This one works more or less like a Pascal `case` or a C `switch` construct: *`<expression>`* is compared to *`<exp1>`*, *`<exp2>`* etc., until a match is found, in which case the corresponding result is returned. If there is no match and there is an ELSE clause, *`<defaultresult>`* is returned. If there is no match and no ELSE clause, `NULL` is returned.

It is important to know that the comparisons are done with the “`=`” operator, so a null *`<expression>`* will *not* match a null *`<expN>`*. If *`<expression>`* is `NULL`, the only way to get a non-`NULL` result is via the ELSE clause.

It is OK to specify `NULL` (or any other valid `NULL` expression) as a result.

The second, or *searched syntax* is:

```
case
  when <condition1> then <result1>
  when <condition2> then <result2>
  ...
  [else <defaultresult>]
end
```

Here, the *<conditionN>*s are tests that give a ternary boolean result: `true`, `false`, or `NULL`. Once again, only `true` is good enough, so a condition like "A = 3" – or even "A = null" – is not satisfied when A is `NULL`. Remember though that "IS [NOT] NULL" never returns `NULL`: if A is `NULL`, the condition "A is null" returns `true` and the corresponding *<resultN>* will be returned. In Firebird 2+ you can also use "IS [NOT] DISTINCT FROM" in your conditions – this operator too will never return `NULL`.

WHILE loops

When evaluating the condition of a WHILE loop, `NULL` has the same effect as in an IF statement: if the condition resolves to `NULL`, the loop is not (re)entered – just as if it were `false`. Again, watch out with inversion using NOT: a condition like

```
while ( Counter > 12 ) do
```

will skip the loop block if Counter is `NULL`, which is probably what you want. But

```
while ( not Counter > 12 ) do
```

will also skip if Counter is `NULL`. Maybe this is also exactly what you want – just be aware that these seemingly complementary tests both exclude `NULL` counters.

FOR loops

To avoid any possible confusion, let us emphasise here that FOR loops in Firebird PSQL have a totally different function than WHILE loops, or **`for`** loops in general programming languages. Firebird FOR loops have the form:

```
for <select-statement> into <var-list> do <code-block>
```

and they will keep executing the code block until all the rows from the result set have been retrieved, unless an exception occurs or a BREAK, LEAVE or EXIT statement is encountered. Fetching a `NULL`, or even row after row filled with `NULL`s, does *not* terminate the loop!

Keys and unique indices

Primary keys

`NULL`s are never allowed in primary keys. A column can only be (part of) a PK it has been defined as NOT NULL, either in the column definition or in a domain definition. Note that a "CHECK (XXX IS NOT NULL)" constraint won't do: you need a NOT NULL specifier right after the data type.

Warning

Firebird 1.5 has a bug that allows primary keys to be defined on a NOT NULL column with `NULL` entries. How these `NULL`s can exist in such a column will be explained later.

Unique keys and indices

Firebird 1.0

In Firebird 1.0, unique *keys* are subject to the same restrictions as primary keys: the column(s) involved must be defined as NOT NULL. For unique *indices*, this is not necessary. However, when a unique index is created the table may not contain any `NULL`s or duplicate values, or the creation will fail. Once the index is in place, insertion of `NULL`s or duplicate values is no longer possible.

Firebird 1.5 and higher

In Firebird 1.5 and up, unique keys and unique indices allow `NULL`s, and what's more: they even allow multiple `NULL`s. With a single-column key or index, you can insert as many `NULL`s as you want in that column, but you can insert each non-`NULL` value only once.

If the key or index is defined on multiple columns in Firebird 1.5 and higher:

- You can insert multiple rows where all the key columns are `NULL`;
- But as soon as one or more key columns are non-`NULL`, each combination of non-`NULL` values must be unique in the table. Of course with the understanding that (1, `NULL`) is not the same as (`NULL`, 1).

Foreign keys

Foreign keys as such impose no restrictions with respect to `NULL`s. Foreign key columns must always reference a column (or set of columns) that is a primary key or a unique key. A unique index on the referenced column(s) is not enough.

> **Note**
>
> In versions up to and including 2.0, if you try to create a foreign key referencing a target that is neither a primary nor a unique key, Firebird complains that no unique *index* can been found on the target – even if such an index does exist. In 2.1, the message correctly states that no unique or primary *key* could be found.

Even if `NULL`s are absolutely forbidden in the target key (for instance if the target is a PK), the foreign key column may still contain `NULL`s, unless this is prevented by additional constraints.

CHECK constraints

It has been said several times in this guide that if test expressions return `NULL`, they have the same effect as `false`: the condition is not satisfied. Starting at Firebird 2, this is **no longer true** for the CHECK constraint. To comply with SQL standards, a CHECK is now **passed** if the condition resolves to `NULL`. Only an unambiguous `false` outcome will cause the input to be rejected.

In practice, this means that checks like

```
check ( value > 10000 )
```

```
check ( upper( value ) in ( 'A', 'B', 'X' ) )
```

```
check ( value between 30 and 36 )
```

```
check ( ColA <> ColB )
```

```
check ( Town not like 'Amst%' )
```

...will reject `NULL` input in Firebird 1.5, but let it pass in Firebird 2. Existing database creation scripts will have to be carefully examined before being used under Firebird 2. If a domain or column has no NOT NULL constraint, and a CHECK constraint may resolve to `NULL` (which usually – but not exclusively – happens because the input is `NULL`), the script has to be adapted. You can extend your check constraints like this:

```
check ( value > 10000 and value is not null )
```

```
check ( Town not like 'Amst%' and Town is not null )
```

However, it's easier and clearer to add NOT NULL to the domain or column definition:

```
create domain DCENSUS int not null check ( value > 10000 )
```

```
create table MyPlaces
(
  Town varchar(24) not null check ( Town not like 'Amst%' ),
  ...
)
```

If your scripts and/or databases should function consistently under both old and new Firebird versions, make sure that no CHECK constraint can ever resolve to `NULL`. Add “`or ... is null`” if you want to allow `NULL` input in older versions. Add NOT NULL constraints or “`and ... is not null`” restrictions to disallow it explicitly in newer Firebird versions.

SELECT DISTINCT

A SELECT DISTINCT statement considers all `NULL`s to be equal (NOT DISTINCT FROM each other), so if the select is on a single column it will return at most one `NULL`.

As mentioned earlier, Firebird 2.0 has a bug that causes the NULLS FIRST|LAST directive to fail under certain circumstances with SELECT DISTINCT. For more details, see the bugs list.

User-Defined Functions (UDFs)

*UDF*s (*User-Defined Functions*) are functions that are not internal to the engine, but defined in separate modules. Firebird ships with two UDF libraries: `ib_udf` (a widely used InterBase library) and `fbudf`. You can add more libraries, e.g. by buying or downloading them, or by writing them yourself. UDFs can't be used out of the box; they have to be “declared” to the database first. This also applies to the UDFs that come with Firebird itself.

NULL <–> non-NULL conversions you didn't ask for

Teaching you how to declare, use, and write UDFs is outside the scope of this guide. However, we must warn you that UDFs can occasionally perform unexpected `NULL` conversions. This will sometimes result in `NULL` input being converted to a regular value, and other times in the nullification of valid input like `''` (an empty string).

The main cause of this problem is that with “old style” UDF calling (inherited from InterBase), it is not possible to pass `NULL` as input to the function. When a UDF like `LTRIM` (left trim) is called with a `NULL` argument, the argument is passed to the function as an empty string. (Note: in Firebird 2 and up, it *can* also be passed as a null pointer. We'll get to that later.) From inside the function there is *no way* of telling if this argument represents a real empty string or a `NULL`. So what does the function implementor do? He has to make a choice: either take the argument at face value, or assume it was originally a `NULL` and treat it accordingly.

If the function result type is a pointer, returning `NULL` is possible even if receiving `NULL` isn't. Thus, the following unexpected things can happen:

- You call a UDF with a `NULL` argument. It is passed as a value, e.g. 0 or `''`. Within the function, this argument is not changed back to `NULL`; a non-`NULL` result is returned.
- You call a UDF with a valid argument like 0 or `''`. It is passed as-is (obviously). But the function code supposes that this value really represents a `NULL`, treats it as a black hole, and returns `NULL` to the caller.

Both conversions are usually unwanted, but the second probably more so than the first (better validate something `NULL` than wreck something valid). To get back to our `LTRIM` example: in Firebird 1.0, this function returns `NULL` if you feed it an empty string. This is wrong. In 1.5 it never returns `NULL`: even `NULL` strings (passed by the engine as `''`) are “trimmed” to empty strings. This is also wrong, but it's considered the lesser of two evils. Firebird 2 has finally got it right: a `NULL` string gives a `NULL` result, an empty string is trimmed to an empty string – at least if you declare the function in the right way.

Descriptors

As early as in Firebird 1.0, a new method of passing UDF arguments and results was introduced: “by descriptor”. Descriptors allow `NULL` signalling no matter the type of data. The `fbudf` library makes ample use of this technique. Unfortunately, using descriptors is rather cumbersome; it's more work and less fun for the UDF implementor. But they do solve all the traditional `NULL` problems, and for the caller they're just as easy to use as old-style UDFs.

Improvements in Firebird 2

Firebird 2 comes with a somewhat improved calling mechanism for 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:

```
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.

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 `NULL`s 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 file name; 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.

“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.

Being prepared for undesired conversions

The unsolicited `NULL` <-> non-`NULL` conversions described earlier usually only happen with legacy UDFs, but there are a lot of them around (most notably in `ib_udf`). Also, nothing will stop a careless implementor from doing the same in a descriptor-style function. So the bottom line is: if you use a UDF and you don't know how it behaves with respect to `NULL`:

1. Look at its declaration to see how values are passed and returned. If it says “by descriptor”, it should be safe (though it never hurts to make sure). Ditto if arguments are followed by a NULL keyword. In all other cases, walk through the rest of the steps.

2. If you have the source and you understand the language it's written in, inspect the function code.

3. Test the function both with `NULL` input and with input like 0 (for numerical arguments) and/or `''` (for string arguments).

4. If the function performs an undesired `NULL` <-> non-`NULL` conversion, you'll have to anticipate it in your code before calling the UDF (see also *Testing for `NULL` – if it matters*, elsewhere in this guide).

The declarations for the shipped UDF libraries can be found in the Firebird subdirectory `examples` (v. 1.0) or `UDF` (v. 1.5 and up). Look at the files with extension `.sql`

More on UDFs

To learn more about UDFs, consult the *InterBase 6.0 Developer's Guide* (free at http://www.ibphoenix.com/downloads/60DevGuide.zip), *Using Firebird* and the *Firebird Reference Guide* (both on CD), or the *Firebird Book*. CD and book can be purchased via http://www.ibphoenix.com.

Converting to and from NULL

Substituting NULL with a value

The COALESCE function

The `COALESCE` function in Firebird 1.5 and higher can convert `NULL` to most anything else. This enables you to perform an on-the-fly conversion and use the result in your further processing, without the need for “`if (MyExpression is null) then`” or similar constructions. The function signature is:

```
COALESCE( Expr1, Expr2, Expr3, ... )
```

`COALESCE` returns the value of the first non-`NULL` expression in the argument list. If all the expressions are `NULL`, it returns `NULL`.

This is how you would use `COALESCE` to construct a person's full name from the first, middle and last names, assuming that some middle name fields may be `NULL`:

```
select FirstName
       || coalesce( ' ' || MiddleName, '' )
       || ' ' || LastName
from Persons
```

Or, to create an as-informal-as-possible name from a table that also includes nicknames, and assuming that both nickname and first name may be `NULL`:

```
select coalesce ( Nickname, FirstName, 'Mr./Mrs.' )
       || ' ' || LastName
from OtherPersons
```

`COALESCE` will only help you out in situations where `NULL` can be treated in the same way as some allowed value for the datatype. If `NULL` needs special handling, different from any other value, your only option is to use an IF or CASE construct after all.

Firebird 1.0: the *NVL functions

Firebird 1.0 doesn't have `COALESCE`. However, you can use four UDFs that provide a good part of its functionality. These UDFs reside in the `fbudf` lib and they are:

- `iNVL`, for integer arguments
- `i64NVL`, for bigint arguments
- `dNVL`, for double precision arguments
- `sNVL`, for strings

The `*NVL` functions take exactly two arguments. Like `COALESCE`, they return the first argument if it's not `NULL`; otherwise, they return the second. Please note that the Firebird 1.0 `fbudf` lib – and therefore, the `*NVL` function set – is only available for Windows.

Converting values to `NULL`

Sometimes you want certain values to show up as `NULL` in the output (or intermediate output). This doesn't happen often, but it may for instance be useful if you want to exclude certain values from summing or averaging. The `NULLIF` functions can do this for you, though only for one value at the time.

Firebird 1.5 and up: the `NULLIF` function

The NULLIF internal function takes two arguments. If their values are equal, the function returns `NULL`. Otherwise, it returns the value of the first argument.

A typical use is e.g.

```
select avg( nullif( Weight, -1 ) ) from FatPeople
```

which will give you the average weight of the FatPeople population, without counting those with weight -1. (Remember that aggregate functions like AVG exclude all `NULL` fields from the computation.)

Elaborating on this example, suppose that until now you have used the value -1 to indicate “weight unknown” because you weren't comfortable with `NULL`s. After reading this guide, you may feel brave enough to give the command:

```
update FatPeople set Weight = nullif( Weight, -1 )
```

Now unknown weights will *really* be unknown.

Firebird 1.0: the `*nullif` UDFs

Firebird 1.0.x doesn't have the `NULLIF` internal function. Instead, it has four user-defined functions in the `fbudf` lib that serve the same purpose:

- `inullif`, for integer arguments
- `i64nullif`, for bigint arguments
- `dnullif`, for double precision arguments
- `snullif`, for strings

Please note that the Firebird 1.0 `fbudf` lib – and therefore, the `*nullif` function set – is only available for Windows.

Warning

The Firebird 1 Release Notes state that, because of an engine limitation, these UDFs return a zero-equivalent if the arguments are equal. This is incorrect: if the arguments have the same value, the functions all return a true `NULL`.

However – they also return `NULL` if the first argument is a real value and the second argument is `NULL`. This is a wrong result. The Firebird 1.5 internal `NULLIF` function correctly returns the first argument in such a case.

Altering populated tables

If your table already contains data, and you want to add a non-nullable column or change the nullability of an existing column, there are some consequences that you should know about. We'll discuss the various possibilities in the sections below.

Adding a non-nullable field to a populated table

Suppose you have this table:

Table 8. Adventures table

Name	Bought	Price
Maniac Mansion	12-Jun-1995	$ 49,--
Zak McKracken	9-Oct-1995	$ 54,95

You have already entered some adventure games in this table when you decide to add a non-nullable ID field. There are two ways to go about this, both with their own specific problems.

Adding a NOT NULL field

This is by far the preferred method in general, but it causes some special problems if used on a populated table, as you will see in a moment. First, add the field with this statement:

```
alter table Adventures add id int not null
```

After committing, the new ID fields that have been added to the existing rows will all be `NULL`. In this special case, Firebird allows invalid data to be present in a NOT NULL column. It will also back them up without complaining, but it will refuse to restore them, precisely because of this violation of the NOT NULL constraint.

Note

Firebird 1.5 (but not 1.0 or 2.0) even allows you to make such a column the primary key!

False reporting of `NULL`s as zeroes

To make matters worse, Firebird lies to you when you retrieve data from the table. With isql and many other clients, “SELECT * FROM ADVENTURES” will return this dataset:

Table 9. Result set after adding a NOT NULL column

Name	Bought	Price	ID
Maniac Mansion	12-Jun-1995	$ 49,--	0
Zak McKracken	9-Oct-1995	$ 54,95	0

Of course this will make most people think “OK, cool: Firebird used a default value of 0 for the new fields – nothing to worry about”. But you can verify that the ID fields are really `NULL` with these queries:

- SELECT * FROM ADVENTURES WHERE ID = 0 (returns empty set)
- SELECT * FROM ADVENTURES WHERE ID IS NULL (returns set shown above, with false 0's)
- SELECT * FROM ADVENTURES WHERE ID IS NOT NULL (returns empty set)

Another type of query hinting that something fishy is going on is the following:

- SELECT NAME, ID, ID+3 FROM ADVENTURES

Such a query will return 0 in the “ID+3” column. With a true 0 ID it should have been 3. The *correct* result would be `NULL`, of course!

With a (VAR)CHAR column, you would have seen phoney emptystrings (''). With a DATE column, phoney “zero dates” of 17 November 1858 (epoch of the Modified Julian Day). In all cases, the true state of the data is `NULL`.

Explanation

What's going on here?

When a client application like isql queries the server, the conversation passes through several stages. During one of them – the “describe” phase – the engine reports type and nullability for each column that will appear in the result set. It does this in a data structure which is later also used to retrieve the actual row data. For columns flagged as NOT NULL by the server, there is no way to return `NULL`s to the client — unless the client flips back the flag before entering the data retrieval stage. Most client applications don't do this. After all, if the server assures you that a column can't contain `NULL`s, why would you think you know better, override the server's decision and check for `NULL`s anyway? And yet that's exactly what you should do if you want to avoid the risk of reporting false values to your users.

FSQL

Firebird expert Ivan Prenosil has written a free command-line client that works almost the same as isql, but – among other enhancements – reports `NULL`s correctly, even in NOT NULL columns. It's called FSQL and you can download it here:

http://www.volny.cz/iprenosil/interbase/fsql.htm

Ensuring the validity of your data

This is what you should do to make sure that your data are valid when adding a NOT NULL column to a populated table:

- To prevent the nulls-in-not-null-columns problem from occurring at all, provide a default value when you add the new column:

  ```
  alter table Adventures add id int default -1 not null
  ```

  Default values are normally not applied when adding fields to existing rows, but with NOT NULL fields they are.

- Else, explicitly set the new fields to the value(s) they should have, right after adding the column. Verify that they are all valid with a “SELECT ... WHERE ... IS NULL” query, which should return an empty set.

- If the damage has already been done and you find yourself with an unrestorable backup, use gbak's `-n` switch to ignore validity constraints when restoring. Then fix the data and reinstate the constraints manually. Again, verify with a “WHERE ... IS NULL” query.

  Important

  Firebird versions up to and including 1.5 have an additional bug that causes gbak to restore NOT NULL constraints even if you specify `-n`. With those versions, if you have backed up a database with `NULL` data in NOT NULL fields, you are really up the creek. Solution: install 1.5.1 or higher, restore with gbak `-n` and fix your data.

Adding a CHECKed column

Using a CHECK constraint is another way to disallow `NULL` entries in a column:

```
alter table Adventures add id int check (id is not null)
```

If you do it this way, a subsequent SELECT will return:

Table 10. Result set after adding a CHECKed field

Name	Bought	Price	ID
Maniac Mansion	12-Jun-1995	$ 49,--	<null>
Zak McKracken	9-Oct-1995	$ 54,95	<null>

Well, at least now you can *see* that the fields are `NULL`! Firebird does not enforce CHECK constraints on existing rows when you add new fields. The same is true if you add checks to existing fields with ADD CONSTRAINT or ADD CHECK.

This time, Firebird not only tolerates the presence and the backing up of the `NULL` entries, but it will also restore them. Firebird's gbak tool does restore CHECK constraints, but doesn't apply them to the existing data in the backup.

> **Note**
>
> Even with the *-n* switch, gbak restores CHECK constraints. But since they are not used to validate backed-up data, this will never lead to a failed restore.

This restorability of your NULL data despite the presence of the CHECK constraint is consistent with the fact that Firebird allows them to be present in the first place, and to be backed up as well. But from a pragmatical point of view, there's a downside: you can now go through cycle after cycle of backup and restore, and your “illegal” data will survive without you even receiving a warning. So again: make sure that your existing rows obey the new rule immediately after adding the constrained column. The “default” trick won't work here; you'll just have to remember to set the right value(s) yourself. If you forget it now, chances are that your outlawed NULLs will survive for a long time, as there won't be any wake-up calls later on.

Adding a non-nullable field using domains

Instead of specifying data types and constraints directly, you can also use domains, e.g. like this:

```
create domain icnn as int check (value is not null);
alter table Adventures add id icnn;
```

For the presence of NULL fields, returning of false 0's, effects of default values etc., it makes *no difference at all* whether you take the domain route or the direct approach. However, a NOT NULL constraint that came with a domain can later be removed; a direct NOT NULL on the column will stay forever.

Making existing columns non-nullable

Making an existing column NOT NULL

You cannot add NOT NULL to an existing column, but there's a simple workaround. Suppose the current type is int, then this:

```
create domain intnn as int not null;
alter table MyTable alter MyColumn type intnn;
```

will change the column type to “int not null”.

If the table already had records, any NULLs in the column will remain NULL, and again Firebird will report them as 0 to the user when queried. The situation is almost exactly the same as when you add a NOT NULL column (see *Adding a NOT NULL field*). The only difference is that if you give the domain (and therefore the column) a default value, this time you can't be sure that it will be applied to the existing NULL entries. Tests show that sometimes the default is applied to all NULLs, sometimes to none, and in a few cases to *some* of the existing entries but not to others! Bottom line: if you change a column's type and the new type includes a default, double-check the existing entries – especially if they “seem to be” 0 or zero-equivalents.

Adding a CHECK constraint to an existing column

There are two ways to add a CHECK constraint to a column:

```
alter table Stk add check (Amt is not null)
alter table Stk add constraint AmtNotNull check (Amt is not null)
```

The second form is preferred because it gives you an easy handle to drop the check, but the constraints themselves function exactly the same. As you might have expected, existing NULLs in the column will remain, can be backed up and restored, etc. etc. – see *Adding a CHECKed column*.

Making non-nullable columns nullable again

If you used a CHECK constraint to make the column non-nullable, you can simply drop it again:

```
alter table Stk drop constraint AmtNotNull
```

If you haven't named the constraint yourself but added the CHECK directly to the column or table, you must first find out its name before you can drop it. This can be done with the isql “SHOW TABLE” command (in this case: SHOW TABLE STK).

In the case of a NOT NULL constraint, if you know its name you can just drop it:

```
alter table Stk drop constraint NN_Amt
```

If you don't know the name you can try isql's “SHOW TABLE” again, but this time it will *only* show the constraint name if it is user-defined. If the name was generated by the engine, you have to use this SQL to dig it up:

```
select rc.rdb$constraint_name
from   rdb$relation_constraints rc
       join rdb$check_constraints cc
       on rc.rdb$constraint_name = cc.rdb$constraint_name
where  rc.rdb$constraint_type   = 'NOT NULL'
       and rc.rdb$relation_name = '<TableName>'
       and cc.rdb$trigger_name  = '<FieldName>'
```

Don't break your head over some of the table and field names in this statement; they are illogical but correct. Make sure to uppercase the names of your table and field if they were defined case-insensitively. Otherwise, match the case exactly.

If the NOT NULL constraint came with a domain, you can also remove it by changing the column type to a nullable domain or built-in datatype:

```
alter table Stk alter Amt type int
```

Any concealed NULLs, if present, will now become visible again.

No matter how you removed the NOT NULL constraint, commit your work and *close all connections to the database*. After that, you can reconnect and insert NULLs in the column.

Testing for NULL and equality in practice

This section contains some practical tips and examples that may be of use to you in your everyday dealings with NULLs. It concentrates on testing for NULL itself and testing the (in)equality of two things when NULLs may be involved.

Testing for NULL – if it matters

Quite often, you don't need to take special measures for fields or variables that may be `NULL`. For instance, if you do this:

```
select * from Customers where Town = 'Ralston'
```

you probably don't want to see the customers whose town is unspecified. Likewise:

```
if (Age >= 18) then CanVote = 'Yes'
```

doesn't include people of unknown age, which is also defensible. But:

```
if (Age >= 18) then CanVote = 'Yes';
else CanVote = 'No';
```

seems less justified: if you don't know a person's age, you shouldn't explicitly deny her the right to vote. Worse, this:

```
if (Age < 18) then CanVote = 'No';
else CanVote = 'Yes';
```

won't have the same effect as the previous. If some of the `NULL` ages are in reality under 18, you're now letting minors vote!

The right approach here is to test for `NULL` explicitly:

```
if      (Age is null) then CanVote = 'Unsure';
else if (Age >= 18  ) then CanVote = 'Yes';
else                       CanVote = 'No';
```

Since this code covers more than two possibilities, it's more elegant to use the CASE syntax (available in Firebird 1.5 and up):

```
CanVote = case
            when Age is null then 'Unsure'
            when Age >= 18   then 'Yes'
            else 'No'
          end;
```

Or, prettier:

```
CanVote = case
            when Age >= 18 then 'Yes'
            when Age <  18 then 'No'
            else 'Unsure'
          end;
```

Equality tests

Sometimes you want to find out if two fields or variables are the same and you want to consider them equal if they are both `NULL`. The way to do this depends on your Firebird version.

Firebird 2.0 and up

In Firebird 2 and higher, you test for null-encompassing equality with DISTINCT. This has already been discussed, but here's a quick recap. Two expressions are considered:

- DISTINCT if they have different values or if one of them is `NULL` and the other isn't;
- NOT DISTINCT if they have the same value or if both of them are `NULL`.

[NOT] DISTINCT always returns `true` or `false`, never `NULL` or something else. Examples:

```
if (A is distinct from B) then...
```

```
if (Buyer1 is not distinct from Buyer2) then...
```

Skip the next section if you're not interested in the pre-Firebird-2 stuff.

Earlier Firebird versions

Pre-2.0 versions of Firebird don't support this use of DISTINCT. Consequently, the tests are a little more complicated and there are some pitfalls to avoid.

The correct equality test for pre-2.0 Firebird versions is:

```
if (A = B or A is null and B is null) then...
```

or, if you want to make the precedence of the operations explicit:

```
if ((A = B) or (A is null and B is null)) then...
```

A word of warning though: if exactly one of A and B is `NULL`, the test expression becomes `NULL`, not `false`! This is OK in an `if` statement, and we can even add an `else` clause which will be executed if A and B are not equal (including when one is `NULL` and the other isn't):

```
if (A = B or A is null and B is null)
  then ...stuff to be done if A equals B...
  else ...stuff to be done if A and B are different...
```

But don't get the bright idea of inverting the expression and using it as an inequality test:

```
/* Don't do this! */
if (not(A = B or A is null and B is null))
  then ...stuff to be done if A differs from B...
```

The above code will work correctly if A and B are both `NULL` or both non-`NULL`. But it will fail to execute the `then` clause if exactly one of them is `NULL`.

If you only want something to be done if A and B are different, either use one of the correct expressions shown above and put a dummy statement in the `then` clause (starting at 1.5, an empty `begin..end` block is also allowed), or use this longer test expression:

```
/* This is a correct inequality test for pre-2 Firebird: */
if (A <> B
    or A is null and B is not null
    or A is not null and B is null) then...
```

Remember, all this is only necessary in pre-2.0 Firebird versions. From version 2 onward, the inequality test is simply “`if (A is distinct from B)`”.

Summary of (in)equality tests

Table 11. Testing (in)equality of A and B in different Firebird versions

Test type	Firebird version	
	<= 1.5.x	>= 2.0
Equality	`A = B or A is null and B is null`	`A is not distinct from B`

Inequality	`A <> B`
	`or A is null and B is not null`
	`or A is not null and B is null`
		`A is distinct from B`

Please keep in mind that with Firebird 1.5.x and earlier:

- the equality test returns `NULL` if exactly one operand is `NULL`;
- the inequality test returns `NULL` if both operands are `NULL`.

In an IF or WHERE context, these `NULL` results act as `false` – which is fine for our purposes. But remember that an inversion with NOT() will also return `NULL` – not “`true`”. Also, if you use the 1.5-and-earlier tests within CHECK constraints in Firebird 2 or higher, be sure to read the section *CHECK constraints*, if you haven't done so already.

Tip

Most JOINs are made on equality of fields in different tables, and use the “=” operator. This will leave out all `NULL`-`NULL` pairs. If you want `NULL` to match `NULL`, pick the equality test for your Firebird version from the table above.

Finding out if a field has changed

In triggers you often want to test if a certain field has changed (including: gone from `NULL` to non-`NULL` or vice versa) or stayed the same (including: kept its `NULL` state). This is nothing but a special case of testing the (in)equality of two fields, so here again our approach depends on the Firebird version.

In Firebird 2 and higher we use this code:

```
if (New.Job is not distinct from Old.Job)
  then ...Job field has stayed the same...
  else ...Job field has changed...
```

And in older versions:

```
if (New.Job = Old.Job or New.Job is null and Old.Job is null)
  then ...Job field has stayed the same...
  else ...Job field has changed...
```

Summary

NULL in a nutshell:

- NULL means *unknown*.
- To exclude NULLs from a domain or column, add “NOT NULL” after the type name.
- To find out if A is NULL, use “A IS [NOT] NULL”.
- Assigning NULL is done like assigning values: with “A = NULL” or an insert list.
- To find out if A and B are the same, with the understanding that all NULLs are the same and different from anything else, use “A IS [NOT] DISTINCT FROM B” in Firebird 2 and up. In earlier versions the tests are:

  ```
  // equality:
  A = B or A is null and B is null
  ```

  ```
  // inequality:
  A <> B
  or A is null and B is not null
  or A is not null and B is null
  ```

- In Firebird 2 and up you can use NULL literals in just about every situation where a regular value is also allowed. In practice this mainly gives you a lot more rope to hang yourself.
- Most of the time, NULL operands make the entire operation return NULL. Noteworthy exceptions are:

  - “NULL or true” evaluates to true;
  - “NULL and false” evaluates to false.

- The IN, ANY|SOME and ALL predicates may (but do not always) return NULL if either the left-hand side expression or a list/subresult element is NULL.
- The [NOT] EXISTS predicate never returns NULL. The [NOT] SINGULAR predicate never returns NULL in Firebird 2.1 and up. It is broken in all previous versions.
- In aggregate functions only non-NULL fields are involved in the computation. Exception: COUNT(*).
- In ordered sets, NULLs are placed...

  - 1.0: At the bottom;
  - 1.5: At the bottom, unless NULLS FIRST specified;
  - 2.0: At the “small end” (top if ascending, bottom if descending), unless overridden by NULLS FIRST/LAST.

- If a WHERE or HAVING clause evaluates to NULL, the row is not included in the result set.
- If the test expression of an IF statement is NULL, the THEN block is skipped and the ELSE block executed.

- A CASE statement returns `NULL`:

  - If the selected result is `NULL`.
  - If no matches are found (simple CASE) or no conditions are `true` (searched CASE) and there is no ELSE clause.

- In a simple CASE statement, “CASE *`<null_expr>`*” does *not* match “WHEN *`<null_expr>`*”.

- If the test expression of a WHILE statement evaluates to `NULL`, the loop is not (re)entered.

- A FOR statement is not exited when `NULL`s are received. It continues to loop until either all the rows have been processed or it is interrupted by an exception or a loop-breaking PSQL statement.

- In Primary Keys, `NULL`s are never allowed.

- In Unique Keys and Unique Indices, `NULL`s are

  - *not allowed* in Firebird 1.0;
  - *allowed* (even multiple) in Firebird 1.5 and higher.

- In Foreign Key columns, multiple `NULL`s are allowed.

- If a CHECK constraint evaluates to `NULL`, the input is

  - *rejected* under Firebird 1.5 and earlier;
  - *accepted* under Firebird 2.0 and higher.

- SELECT DISTINCT considers all `NULL`s equal: in a single-column select, at most one is returned.

- UDFs sometimes convert `NULL` `<–>` non-`NULL` in a seemingly random manner.

- The `COALESCE` and `*NVL` functions can convert `NULL` to a value.

- The `NULLIF` family of functions can convert values to `NULL`.

- If you add a NOT NULL column without a default value to a populated table, all the entries in that column will be `NULL` upon creation. Most clients however – including Firebird's isql tool – will falsely report them as zeroes (`0` for numerical fields, `''` for string fields, etc.)

- If you change a column's datatype to a NOT NULL domain, any existing `NULL`s in the column will remain `NULL`. Again most clients – including isql – will show them as zeroes.

Remember, this is how `NULL` works *in Firebird SQL*. There may be (at times subtle) differences with other RDBMSes.

Appendix A: NULL-related bugs in Firebird

Attention: both historic and current bugs are listed in the sections below. Always look if and when a bug has been fixed before assuming that it exists in your version of Firebird.

Bugs that crash the server

EXECUTE STATEMENT with NULL argument

EXECUTE STATEMENT with a NULL argument crashed Firebird 1.5 and 1.5.1 servers. Fixed in 1.5.2.

EXTRACT from NULL date

In 1.0.0, EXTRACT from a NULL date would crash the server. Fixed in 1.0.2.

FIRST and SKIP with NULL argument

FIRST and SKIP crash a Firebird 1.5.n or lower server if given a NULL argument. Fixed in 2.0.

LIKE with NULL escape

Using LIKE with a NULL escape character would crash the server. Fixed in 1.5.1.

Other bugs

NULLs in NOT NULL columns

NULLs can exist in NOT NULL columns in the following situations:

- If you add a NOT NULL column to a populated table, the fields in the newly added column will all be NULL.
- If you make an existing column NOT NULL, any NULLs already present in the column will remain in that state.

Firebird allows these NULLs to stay, also backs them up, but refuses to restore them with gbak. See *Adding a NOT NULL field* and *Making an existing column NOT NULL*.

Illegal `NULL`s returned as `0`, `''`, etc.

If a NOT NULL column contains `NULL`s (see previous bug), the server will still describe it as non-nullable to the client. Since most clients don't question this assurance from the server, they will present these `NULL`s as 0 (or equivalent) to the user. See *False reporting of NULLs as zeroes*.

Primary key with `NULL` entries

The following bug appeared in Firebird 1.5: if you had a table with some rows and you added a NOT NULL column (which automatically creates `NULL` entries in the existing rows – see above), you could make that column the primary key even though it had `NULL` entries. In 1.0 this didn't work because of the stricter rules for UNIQUE indices. Fixed in 2.0.

SUBSTRING results described as non-nullable

The engine describes SUBSTRING result columns as non-nullable in the following two cases:

- If the first argument is a string literal, as in “SUBSTRING('Ootchie-coo' FROM 5 FOR 2)”.
- If the first argument is a NOT NULL column.

This is incorrect because even with a known string, substrings may be `NULL`, namely if the one of the other arguments is `NULL`. In versions 1.* this bug didn't bite: the FROM and FOR args had to be literal values, so they could never be `NULL`. But as from Firebird 2, any expression that resolves to the required data type is allowed. And although the engine correctly returns `NULL` whenever any argument is `NULL`, it *describes* the result column as non-nullable, so most clients show the result as an empty string.

This bug seems to be fixed in 2.1.

Gbak -n restoring NOT NULL

Gbak *`-n[o_validity]`* restored NOT NULL constraints in early Firebird versions. Fixed in 1.5.1.

IN, =ANY and =SOME with indexed subselect

Let *`A`* be the expression on the left-hand side and *`S`* the result set of the subselect. In versions prior to 2.0, “IN”, “=ANY” and “=SOME” return false instead of `NULL` if an index is active on the subselect column and:

- either *`A`* is `NULL` and *`S`* doesn't contain any `NULL`s;
- or *`A`* is not `NULL`, *`A`* is not found in *`S`*, and *`S`* contains at least one `NULL`.

See the warnings in the IN and ANY sections. Workaround: use “<> ALL” instead. Fixed in 2.0.

ALL with indexed subselect

With every operator except “<>”, ALL may return wrong results if an index is active on the subselect column. This can happen with our without `NULL`s involved. See the ALL bug warning. Fixed in 2.0.

SELECT DISTINCT with wrong NULLS FIRST|LAST ordering

Firebird 2.0 has the following bug: if a SELECT DISTINCT is combined with an [ASC] NULLS LAST or DESC NULLS FIRST ordering, and the ordering field(s) form(s) the beginning (but not the whole) of the select list, every field in the ORDER BY clause that is followed by a field with a different (or no) ordering gets the `NULL`s placed at the default relative location, ignoring the NULLS XXX directive. Fixed in 2.0.1 and 2.1.

UDFs returning values when they should return `NULL`

This should definitely be considered a bug. If an angle is unknown, *don't* tell me that its cosine is 1! Although the history of these functions is known and we can understand why they behave like they do (see *User-Defined Functions*), it's still wrong. Incorrect results are returned and this should not happen. Most of the math functions in `ib_udf`, as well as some others, have this bug.

UDFs returning `NULL` when they should return a value

This is the complement of the previous bug. `LPAD` for instance returns `NULL` if you want to pad an empty string with 10 dots. This function and others are fixed in 2.0, with the annotation that you must explicitly declare them with the NULL keyword or they'll show the old – buggy – behaviour. `LTRIM` and `RTRIM` trim empty strings to `NULL` in Firebird 1.0.n. This is fixed in 1.5 at the expense of returning `''` when trimming a `NULL` string, and only fully fixed in 2.0 (if declared with the NULL keyword).

SINGULAR inconsistent with `NULL` results

NOT SINGULAR sometimes returns `NULL` where SINGULAR returns `true` or `false`. Fixed in 2.0.

SINGULAR may wrongly return `NULL`, in an inconsistent but reproducible manner. Fixed in 2.1.

See the section on SINGULAR.

Appendix B: Document history

The exact file history is recorded in the `manual` module in our CVS tree; see http://sourceforge.net/cvs/?group_id=9028

Revision History

0.1	8 Apr 2005	PV	First edition.
0.2	15 Apr 2005	PV	Mentioned that Fb 2.0 legalises "A = NULL" comparisons. Changed text in "Testing if something is `NULL`". Slightly altered "Dealing with `NULL`s" section.
1.0	24 Jan 2007	PV	This is a major update, with so much new material added that the document has grown to around 4 times its former size. In addition, much of the existing text has been reorganised and thoroughly reworked. It's not feasible to give a summary of all the changes here. Consider this a new guide with 15–25% old material. The most important additions are: • `NULL` literals • IS [NOT] DISTINCT FROM • Internal functions and directives • Predicates: IN, ANY, SOME, ALL, EXISTS, SINGULAR • Searches (WHERE) • Sorts (ORDER BY) • GROUP BY and HAVING • CASE, WHILE and FOR • Keys and indices • CHECK constraints • SELECT DISTINCT • Converting values to `NULL` with `NULLIF` • Altering populated tables • Bugs list • Alphabetical index
1.0.1	26 Jan 2007	PV	*Making non-nullable columns nullable again*: Provisory fix of error regarding removal of NOT NULL constraints.

Appendix C: License notice

Alphabetical index