Firebird Generator Guide

A guide on how and when to use generators in Firebird

Frank Ingermann

Version 0.3, 27 June 2020

Table of Contents

Chapter 1. Introduction

1.1. What is this article about?

This article explains what Firebird generators are, and how and why you should use them. It is an attempt to collect all relevant information about generators in a single document.

1.2. Who should read it?

Read this article if you…

- are not familiar with the concept of generators;
- have questions on using them;
- want to make an Integer column behave like an “AutoInc” field as found in other RDBMSs;
- are looking for examples on how to use generators for IDs or other tasks;
- want to know the Firebird word for a “sequence” in Oracle.

Chapter 2. Generator Basics

2.1. What is a generator?

Think of a generator as a “thread-safe” integer counter that lives inside a Firebird database. You can create one by giving it a name:

```
CREATE GENERATOR GenTest;
```

Then you can get its current value and increase or decrease it just like a “`var i:integer`” in Delphi, but it is not always easy to “predictably” set it directly to a certain value and then obtain that same value — it’s inside the database, but *outside of transaction control.*

2.2. What is a sequence?

“Sequence” is the official SQL term for what Firebird calls a generator. Because Firebird is constantly striving for better SQL compliance, the term `SEQUENCE` can be used as a synonym for `GENERATOR` in Firebird 2 and up. In fact it is recommended that you use the `SEQUENCE` syntax in new code.

Although the word “sequence” puts the emphasis on the series of values generated whereas “generator” seems to refer primarily to the factory that produces these values, there is *no difference at all* between a Firebird generator and a sequence. They are just two words for the same database object. You can create a generator and access it using the sequence syntax, and vice versa.

This is the preferred syntax for creating a generator/sequence in Firebird 2:

```
CREATE SEQUENCE SeqTest;
```

2.3. Where are generators stored?

Generator *declarations* are stored in the `RDB$GENERATORS` system table. Their *values* however are stored in special reserved pages inside the database. You never touch those values directly; you access them by means of built-in functions and statements which will be discussed later on in this guide.

The information provided in this section is for educational purposes only. As a general rule, you should leave system tables alone. Don’t attempt to create or alter generators by writing to `RDB$GENERATORS`. (A `SELECT` won’t hurt though.)

The structure of the `RDB$GENERATORS` system table is as follows:

- `RDB$GENERATOR_NAME CHAR(31)`
- `RDB$GENERATOR_ID SMALLINT`

- RDB$SYSTEM_FLAG SMALLINT

And, as from Firebird 2.0:

- RDB$DESCRIPTION BLOB subtype TEXT

Note that the GENERATOR_ID is — as the name says — an IDentifier for each generator, *not* its value. Also, don't let your applications store the ID for later use as a handle to the generator. Apart from this making no sense (the *name* is the handle), the ID may be changed after a backup-restore cycle. The SYSTEM_FLAG is 1 for generators used internally by the engine, and NULL or 0 for all those you created.

Now let's have a look at the RDB$GENERATORS table, here with a single self-defined generator:

RDB$GENERATOR_NAME	RDB$GENERATOR_ID	RDB$SYSTEM_FLAG
RDB$SECURITY_CLASS	1	1
SQL$DEFAULT	2	1
RDB$PROCEDURES	3	1
RDB$EXCEPTIONS	4	1
RDB$CONSTRAINT_NAME	5	1
RDB$FIELD_NAME	6	1
RDB$INDEX_NAME	7	1
RDB$TRIGGER_NAME	8	1
MY_OWN_GENERATOR	9	NULL

Firebird 2 notes

- Firebird 2 saw the introduction of an additional system generator, called RDB$BACKUP_HISTORY. It is used for the new NBackup facility.
- Even though the SEQUENCE syntax is preferred, the RDB$GENERATORS system table and its columns have not been renamed in Firebird 2.

2.4. What is the maximum value of a generator?

Generators store and return 64-bit values in all versions of Firebird. This gives us a value range of:

-2^{63} .. 2^{63}
-1 or -9,223,372,036,854,775,808 .. 9,223,372,036,854,775,807

So if you use a generator with starting value 0 to feed a NUMERIC(18) or BIGINT column (both types represent 64-bit integers), and you would insert 1000 rows per second, it would take around 300 million years (!) before it rolls over. As it is pretty unlikely mankind will still walk on this planet by then (and still use Firebird databases), that's nothing to be really worried about.

A word of warning though. Firebird speaks two SQL "dialects": dialect 1 and dialect 3. New databases should always be created with dialect 3, which is more powerful in a number of respects. Dialect 1 is a compatibility dialect, to be used only for legacy databases that were first created

under InterBase 5.6 or earlier.

One of the differences between the two is that dialect 1 has no native 64-bit integer type available. `NUMERIC(18)` columns for instance are stored internally as `DOUBLE PRECISION`, which is a floating point type. The biggest integer type in dialect 1 is the 32-bit `INTEGER`.

In dialect 1 as in dialect 3, generators are 64-bit. But if you assign the generated values to an `INTEGER` column in a dialect 1 database, they are truncated to the lower 32 bits, giving an effective range of:

-2^{31} .. 2^{31}
-1 or -2,147,483,648 .. 2,147,483,647

Although the generator itself would go on from 2,147,483,647 to 2,147,483,648 and beyond, the truncated value would wrap around at this point, giving the *impression* of a 32-bit generator.

In the situation described above, with 1000 inserts per second, the generator-fed column would now roll over after 25 *days* (!!!) and that is indeed something to have an eye on. 2^{31} is a lot, but then again not that much depending on the situation.

In dialect 3, if you assign generator values to an `INTEGER` field, all goes well as long as the values lie within the 32-bit range. But as soon as that range is exceeded, you get a numeric overflow error: dialect 3 is much stricter on range checking than dialect 1!

2.4.1. Client dialects and generator values

Clients talking to a Firebird server can set their dialect to 1 or 3, regardless of the database they are connected to. It is the client dialect, *not* the database dialect, that determines how Firebird passes generator values to the client:

- If the client dialect is 1, the server returns generator values as truncated 32-bit integers to the client. But inside the database they remain 64-bit values and they do not wrap after reaching 2^{31} -1 (even though it may look that way to the client). This is true both for dialect 1 and dialect 3 databases.
- If the client dialect is 3, the server passes the full 64-bit value to the client. Again, this holds whether the database dialect is 1 or 3.

2.5. How many generators are available in one database?

Since Firebird version 1.0, the number of generators you can have in a single database is limited only by the maximum assignable ID in the `RDB$GENERATORS` system table. Being a `SMALLINT`, this maximum is 2^{15} -1 or 32767. The first ID is always 1, so the total number of generators cannot exceed 32767. As discussed before, there are 8 or 9 system generators in the database, leaving room for at least 32758 of your own. This should be amply enough for any practical application. And since the number of generators you declare has no effect on performance, you can feel free to use as many generators as you like.

2.5.1. Older InterBase and Firebird versions

In the earliest pre-1.0 Firebird versions, as well as in InterBase, only one database page was used to store the generator values. Therefore, the number of available generators was limited by the page size of the database. The following table lists how many generators — including system generators — you can have in various InterBase and Firebird versions (thanks to Paul Reeves for providing the initial information):

Version	Page size			
	1K	*2K*	*4K*	*8K*
InterBase < v.6	247	503	1015	2039
IB 6 and early pre-1.0 Firebird	123	251	507	1019
All later Firebird versions	32767			

In InterBase versions prior to 6, generators were only 32 bits wide. This explains why these older versions could store roughly twice the number of generators on the same page size.

InterBase, at least up to and including version 6.01, would happily let you “create” generators until the total number reached 32767. What happened if you accessed generators with an ID higher than the number given in the table above depended on the version:

- InterBase 6 would generate an “invalid block type” error because the calculated location lay outside the one page that was allocated to generators.
- In earlier versions, if the calculated location lay outside the database, an error would be returned. Otherwise, if the generator was only *read* (without increment), the value that just “happened to be” on the calculated spot was returned. If it was *written to*, it would overwrite data. This could sometimes lead to an immediate error, but most of the time it would just silently corrupt your database.

2.6. Generators and transactions

As said, generators live outside of transaction control. This simply means you cannot safely “rollback” generators inside a transaction. There may be other transactions executing at the same time that change the value while your transaction runs. So once you have requested a generator value, consider it as “gone forever”.

When you start a transaction and then call a generator and get a value of — let’s say — 5, it will remain at that value **even if you roll back the transaction (!)**. Don’t even *think* of something like “OK, when I rollback, I can just do `GEN_ID(mygen,-1)` afterwards to set it back to 4”. This may work most of the time, but is *unsafe* because other concurrent transactions may have changed the value in between. For the same reason it doesn’t make sense to get the current value with `GEN_ID(mygen,0)` and then increment the value on the client side.

Chapter 3. SQL statements for generators

3.1. Statement overview

The name of a generator must be a usual DB meta-identifier: 31 chars maximum, no special characters except the underscore '_' (unless you use quoted identifiers). The SQL commands and statements that apply to generators are listed below. Their use will be discussed in some detail in the section Use of generator statements.

DDL (Data Definition Language) statements:

```
CREATE GENERATOR name;
SET GENERATOR name TO value;
DROP GENERATOR name;
```

DML (Data Manipulation Language) statements in client SQL:

```
SELECT GEN_ID( GeneratorName, increment ) FROM RDB$DATABASE;
```

DML statements in PSQL (Procedural SQL, available in stored procedures and triggers):

```
intvar = GEN_ID( GeneratorName, increment );
```

3.1.1. Firebird 2 recommended syntax

Although the traditional syntax is still fully supported in Firebird 2, these are the recommended DDL equivalents:

```
CREATE SEQUENCE name;
ALTER SEQUENCE name RESTART WITH value;
DROP SEQUENCE name;
```

And for the DML statements:

```
SELECT NEXT VALUE FOR SequenceName FROM RDB$DATABASE;
```

```
intvar = NEXT VALUE FOR SequenceName;
```

Currently the new syntax does not support an increment other than 1. This limitation will be lifted in a future version. In the meantime, use `GEN_ID` if you need to apply another increment value.

3.2. Use of generator statements

The availability of statements and functions depends on whether you use them in:

- Client SQL — The language you use when you, as a client, talk to a Firebird server.
- PSQL — The server-side programming language used in Firebird stored procedures and triggers.

3.2.1. Creating a generator ("Insert")

Client SQL

```
CREATE GENERATOR GeneratorName;
```

Preferred for Firebird 2 and up:

```
CREATE SEQUENCE SequenceName;
```

PSQL

Not possible. Since you cannot change database metadata inside SPs or triggers, you cannot create generators there either.

In FB 1.5 and up, you can circumvent this limitation with the `EXECUTE STATEMENT` feature.

3.2.2. Getting the current value ("Select")

Client SQL

```
SELECT GEN_ID( GeneratorName, 0 ) FROM RDB$DATABASE;
```

This syntax is still the only option in Firebird 2.

In Firebird's command-line tool *isql* there are two additional commands for retrieving current generator values:

```
SHOW GENERATOR GeneratorName;
SHOW GENERATORS;
```

The first reports the current value of the named generator. The second does the same for all non-system generators in the database.

The preferred Firebird 2 equivalents are, as you could guess:

```
SHOW SEQUENCE SequenceName;
```

```
SHOW SEQUENCES;
```

Please notice again that these SHOW … commands are only available in the Firebird isql tool. Unlike GEN_ID, you can't use them from within other clients (unless these clients are isql frontends).

PSQL

```
intvar = GEN_ID( GeneratorName, 0 );
```

Firebird 2: same syntax.

3.2.3. Generating the next value ("Update" + "Select")

Just like getting the current value, this is done with GEN_ID, but now you use an increment value of 1. Firebird will:

1. get the current generator value;
2. increment it by 1;
3. return the incremented value.

Client SQL

```
SELECT GEN_ID( GeneratorName, 1 ) FROM RDB$DATABASE;
```

The new syntax, which is preferred for Firebird 2, is entirely different:

```
SELECT NEXT VALUE FOR SequenceName FROM RDB$DATABASE;
```

PSQL

```
intvar = GEN_ID( GeneratorName, 1 );
```

Preferred for Firebird 2 and up:

```
intvar = NEXT VALUE FOR SequenceName;
```

3.2.4. Setting a generator directly to a certain value ("Update")

Client SQL

```
SET GENERATOR GeneratorName TO NewValue;
```

This is useful to preset generators to a value other than 0 (which is the default value after you

created it) in e.g. a script to create the database. Just like `CREATE GENERATOR`, this is a DDL (not DML) statement.

Preferred syntax for Firebird 2 and up:

```
ALTER SEQUENCE SequenceName RESTART WITH NewValue;
```

PSQL

```
GEN_ID( GeneratorName, NewValue - GEN_ID( GeneratorName, 0 ) );
```

This is more of a dirty little trick to do what you normally cannot and should not do in SPs and triggers: *setting* generators. They are for *getting*, not *setting* values.

3.2.5. Dropping a generator ("Delete")

Client SQL

```
DROP GENERATOR GeneratorName;
```

Preferred for Firebird 2 and up:

```
DROP SEQUENCE SequenceName;
```

PSQL

Not possible, unless... (Same explanation as with Create: you can't — or rather, shouldn't — change metadata in PSQL.)

Dropping a generator does not free the space it occupied for use by a new generator. In practice this rarely hurts, because most databases don't have the tens of thousands of generators that Firebird allows, so there's bound to be room for more anyway. But if your database *does* risk to hit the 32767 ceiling, you can free up dead generator space by performing a backup-restore cycle. This will neatly pack the `RDB$GENERATORS` table, re-assigning a contiguous series of IDs. Depending on the situation, the restored database may also need less pages for the generator values.

Dropping generators in old IB and Firebird versions

InterBase 6 and earlier, as well as early pre-1.0 Firebird versions, do not have a `DROP GENERATOR` command. The only way to drop a generator in these versions is:

```
DELETE FROM RDB$GENERATORS WHERE RDB$GENERATOR_NAME = 'GeneratorName';
```

...followed by a backup and restore.

In these versions, with the maximum number of generators typically a couple of hundred, it is much more likely that the need will arise to reuse space from deleted generators.

Chapter 4. Using generators to create unique row IDs

4.1. Why row IDs at all?

The answer to this question would go far beyond the scope of this article. If you see no need to have a generic, unique “handle” for every row inside a table, or don’t like the idea of “meaningless” or “surrogate” keys in general, you should probably skip this section…

4.2. One for all or one for each?

OK, so you want row IDs. { author’s note: congratulations! :-) }

A major, basic decision to take is whether we’ll use one single generator for all the tables, or one generator for each table. This is up to you — but take the following considerations into account.

With the “one for all” approach, you:

- + need only a single generator for all your IDs;
- + have one integer number that does not only identify your row within its *table*, but within the *entire database*;
- - have less possible ID values per table (this shouldn’t really be a problem with 64bit generators…);
- - will soon have to deal with large ID values even in e.g. lookup tables with only a handful of records;
- - will likely see gaps in a per-table ID sequence, since generator values are spread throughout all tables.

With the “one for each” approach you:

- - have to create a generator for every single “ID’d” table in your database;
- - always need the combination of ID and table name to uniquely identify any row in any table;
- + have a simple and robust “insert counter” per table;
- + have a chronological sequence per table: if you find a gap in the ID sequence of a table, then it’s caused either by a `DELETE` or by a failed `INSERT`.

4.3. Can you re-use generator values?

Well — yes, technically you *can*. But — NO, you shouldn’t. Never. Never ever. Not only that this would destroy the nice chronological sequence (you can’t judge a row’s “age” by just looking at the ID any more), the more you think about it the more headaches it’ll give you. Moreover it is an absolute contradiction to the entire concept of unique row identifiers.

So unless you have good reasons to re-use generator values, and a well-thought-out mechanism to

make this work safely in multi-user/multi-transaction environments, JUST DON'T DO IT!

4.4. Generators for IDs or auto-increment fields

Giving a newly inserted record an ID (in the sense of a unique "serial number") is easily done with generators and Before Insert triggers, as we'll see in the following subsections. We start with the assumption that we have a table called `TTEST` with a column `ID` declared as Integer. Our generator's name is `GIDTEST`.

4.4.1. Before Insert trigger, version 1

```
CREATE TRIGGER trgTTEST_BI_V1 for TTEST
active before insert position 0
as
begin
  new.id = gen_id( gidTest, 1 );
end
```

Problems with trigger version 1:

This one does the job all right — but it also "wastes" a generator value in cases where there is already an ID supplied in the `INSERT` statement. So it would be more efficient to only assign a value when there was none in the `INSERT`:

4.4.2. Before Insert trigger, version 2

```
CREATE TRIGGER trgTTEST_BI_V2 for TTEST
active before insert position 0
as
begin
  if (new.id is null) then
  begin
    new.id = gen_id( gidTest, 1 );
  end
end
```

Problems with trigger version 2:

Some access components have the "bad habit" to auto-fill all the columns in an `INSERT`. Those not explicitly set by you get default values — usually `0` for integer columns. In that case, the above trigger would not work: it would find that the ID column does not have the *state* `NULL`, but the *value* `0`, so it would not generate a new ID. You could post the record, though — but only one… the second one would fail. It is anyway a good idea to "ban" `0` as a normal ID value, to avoid any confusion with `NULL` and `0`. You could e.g. use a special row with an ID of 0 to store a default record in each table.

4.4.3. Before Insert trigger, version 3

```
CREATE TRIGGER trgTTEST_BI_V3 for TTEST
active before insert position 0
as
begin
  if ((new.id is null) or (new.id = 0)) then
  begin
    new.id = gen_id( gidTest, 1 );
  end
end
```

Well, now that we have a robust, working ID trigger, the following paragraphs will explain to you why mostly you won't need it at all:

The basic problem with IDs assigned in Before Insert triggers is that they are generated on the server side, *after* you send the Insert statement from the client. This plainly means there is no safe way to know from the client side which ID was generated for the row you just inserted.

You could grab the generator value from the client side after the Insert, but in multi-user environments you cannot be really sure that what you get is your own row's ID (because of the transaction issue).

But if you get a new value from the generator *before*, and post the Insert with that value, you can simply fetch the row back with a "`Select ⋯ where ID=genvalue`" to see what defaults were applied or whether columns were affected by Insert triggers. This works especially well because you usually have a unique Primary Key index on the ID column, and those are about the fastest indexes you can have — they're unbeatable in selectivity, and mostly smaller than indexes on `CHAR(n)` cols (for n>8, depending on character set and collation).

The bottom line to this is:

You should create a Before Insert trigger to make absolutely sure every row gets a unique ID, even if no ID value was supplied from the client side in the Insert statement.

If you have an "SQL-closed" database (that is, your own application code is the only source for newly inserted records), then you can leave out the trigger, but then you should *always* obtain a new generator value from the database before issuing the Insert statement and include it there. The same, of course, goes for inserts from within triggers and stored procedures.

Chapter 5. What else to do with generators

Here you can find some ideas for usages of generators other than generating unique row IDs.

5.1. Using generators to give e.g. transfer files unique numbers

A “classic” usage of generators is to ensure unique, sequential numbers for — well, anything in your application other than the row IDs discussed above. When you have an application that is transferring data to some other system, you can use generators to safely identify a single transfer by labeling it with a generated value. This greatly helps tracking down problems with interfaces between two systems (and, unlike most of the following, this does work safely and exactly).

5.2. Generators as “usage counters” for SPs to provide basic statistics

Imagine you just built a fantastic new feature into your database with a stored procedure. Now you update your customer’s systems and some time later you’d like to know if the users really *use* this feature and how often. Simple: make a special generator that only gets incremented in that SP and you’re there… with the restriction that you can’t know the number of transactions that were rolled back after or while your SP executed. So in this case you at least know how often users *tried* to use your SP :-)

You could further refine this method by using two generators: One gets incremented at the very start of the SP, another at the very end just before the `EXIT`. This way you can count how many attempts to use the SP were successful: if both generators have the same value, then none of the calls to the SP failed etc. Of course you then still don’t know how many times the transaction(s) invoking your SP were actually committed.

5.3. Generators to simulate “Select count(*) from…”

There is the known problem with InterBase and Firebird that a `SELECT COUNT(*)` (with no Where clause) from a really large table can take quite a while to execute, since the server must count "by hand" how many rows there are in the table at the time of the request. In theory, you could easily solve this problem with generators:

- Create a special “row counter” generator;
- Make a Before Insert trigger that increments it;
- Make an After Delete trigger that decrements it.

This works beautifully and makes a “full” record count needless — just get the current generator value. I stressed the “_in theory_” here because the whole thing goes down the drain when any Insert statements fail, because as said those generators are *beyond transaction control*. Inserts can fail because of constraints (Unique Key violations, `NOT NULL` fields being `NULL`, etc.) or other metadata restrictions, or simply because the transaction that issued the Insert gets rolled back. You

have no rows in the table and still your Insert counter climbs.

So it depends — when you know the rough percentage of Inserts that fail (you can kinda get a "feeling" for this), and you're only interested in an *estimation* of the record count, then this method can be useful even though it's not exact. From time to time you can do a "normal" record count and set the generator to the exact value ("re-synchronize" the generator), so the error can be kept rather small.

There are situations when customers can happily live with an info like "there are *about* 2.3 million records" instantly at a mouseclick, but would shoot you if they have to wait 10 minutes or more to see that there are precisely 2.313.498.229 rows...

5.4. Generators to monitor and/or control long-running Stored Procedures

When you have SPs that e.g. generate report outputs on large tables and/or complex joins, they can take quite a while to execute. Generators can be helpful here in two ways: they can provide you with a "progress counter" which you can poll periodically from the client side while the SP runs, and they can be used to stop it:

```
CREATE GENERATOR gen_spTestProgress;
CREATE GENERATOR gen_spTestStop;

set term ^;

CREATE PROCEDURE spTest (...)
AS
BEGIN
  (...)
  for select <lots of data taking lots of time>
  do begin
    GEN_ID(gen_spTestProgress,1);

    IF (GEN_ID(gen_spTestStop,0)>0) THEN Exit;

    (...normal processing here...)
  end
END^
```

Just a rough sketch, but you should get the idea. From the client, you can do a `GEN_ID(gen_spTestProgress,0)` asynchronously to the actual row fetching (e.g. in a different thread), to see how many rows were processed, and display the value in some sort of progress window. And you can do a `GEN_ID(gen_spTestStop,1)` to cancel the SP at any time from the "outside".

Although this can be very handy, it has a strong limitation: *it's not multi-user safe*. If the SP would run simultaneously in two transactions, they would mess up the progress generator — they would both increment the same counter at the same time so the result would be useless. Even worse, incrementing the stop generator would immediately stop the SP in *both* transactions. But for e.g.

monthly reports that are generated by a single module run in batch mode, this can be acceptable — as usual, it depends on your needs.

If you want to use this technique and allow *users* to trigger the SP at any time, you must make sure by other means that the SP can not be run twice. Thinking about this, I had the idea to use another generator for that: let's call this one `gen_spTestLocked` (assuming the initial value of `0` of course):

```
CREATE GENERATOR gen_spTestProgress;
CREATE GENERATOR gen_spTestStop;
CREATE GENERATOR gen_spTestLocked;

set term ^;

CREATE PROCEDURE spTest (...)
AS
DECLARE VARIABLE lockcount INTEGER;
BEGIN
  lockcount = GEN_ID(gen_spTestLocked,1);
    /* very first step: increment the locking generator */

  if (lockcount=1) then /* _we_ got the lock, continue */
  begin
    (..."normal" procedure body here...)
  end

  lockcount = GEN_ID(gen_spTestLocked,-1); /* undo the increment */

  /* make sure the gen is reset at the very end even when an exception
     happens inside the normal procedure body: */

  WHEN ANY DO
    lockcount = GEN_ID(spTestLocked,-1); /* undo the increment */
  exit;
END^
```

I'm not yet 100% sure this is absolutely multi-user safe, but it looks rock solid — as long as no `EXIT` occurs in the `normal` procedure body, for then the SP would stop and quit, leaving the generator incremented. The `WHEN ANY` clause handles exceptions, but not normal `EXIT`s. Then you'd have to decrement it by hand — but you could decrement the generator just before the `EXIT` to avoid this. Given the right precautions, I can't make up any situation where this mechanism could fail… If you can — let us know!

Appendix A: Document history

The exact file history is recorded in the firebird-documentation git repository; see https://github.com/FirebirdSQL/firebird-documentation

Revision History

0.1	4 Apr 2006	FI	First edition.
0.2	7 May 2006	PV	Added SEQUENCE syntax and other Firebird 2 info. Added information on: the importance of client dialects; the SHOW GENERATOR statement and friends; dropping generators and packing generator space. Edited and extended the following sections more or less heavily: *Where are generators stored?*, *What is the maximum value of a generator?*, *How many generators...?*, *Use of generator statements*. Further editing, additions and corrections to various sections, mainly in the first half of the document. Light editing in second half (starting at *Using generators to create unique row IDs*).
0.3	27 Jun 2020	MR	Conversion to AsciiDoc, minor copy-editing

Appendix B: License notice

The Original Documentation is titled *Firebird Generator Guide*.

The Initial Writer of the Original Documentation is: Frank Ingermann.

Contributor: Paul Vinkenoog – see document history.