Using UUID in Firebird

Table of Contents

This material was created with the support and sponsorship of IBSurgeon ib-aid.com, a company that develops Firebird SQL tools for enterprises and provides technical support services for Firebird SQL.

This material is released under the Public Documentation License https://www.firebirdsql.org/file/documentation/html/en/licenses/pdl/public-documentation-license.html

Preface

Some Firebird users have expressed a desire to have a new `UUID` data type added to Firebird, similar to PostgreSQL. So let's figure out what UUID is, how to work with it in Firebird, and whether a separate data type is needed for that.

1. What is UUID?

UUID (Universally Unique Identifiers) is a universal unique identifier defined in RFC 9562, ISO/IEC 9834-8:2005 and related standards. (In some systems it is called GUID, globally unique identifier.) This identifier is a 128-bit value generated by a special algorithm that practically guarantees that it will not be generated again anywhere else in the world by the same algorithm. Thus, these identifiers are unique not only in a single database, like sequence generator values, but also in distributed systems.

1.1. UUID version 4

This version is generated using a cryptographically secure pseudo-random number generator (CSPRNG) to fill 122 of the 128 bits. The remaining 6 bits are reserved: 4 bits encode the version (0100), 2 bits encode the variant (10).

Some collision probability estimates:

- To reach a collision probability of about 50%, one needs to generate approximately 2.71 quadrillion UUID v4 values.
- At a generation rate of 1 billion UUIDs per second, reaching that quantity would take about 86 years.

In practice, collisions of UUID v4 are virtually non-existent. However, it is important to keep in mind that the probability is not zero, and over time, with sufficiently heavy usage, a collision could occur.

On a single machine, the collision probability depends on the quality of the random number generator and the generation frequency. If the generator has insufficient entropy or there is correlation between generators on different nodes, the collision probability may increase.

On different machines using independent generators, the collision probability is also extremely low, provided that the generators on all nodes work correctly and have sufficient entropy. However, if the same generation algorithm is used on different machines, it may increase the risk of collisions if the generators are correlated.

1.2. UUID version 7

This version was standardized in RFC 9562 (2024). It includes a 48-bit Unix timestamp (in milliseconds) in the most significant bits, followed by version bits, 12 random bits, variant bits, and another 62 random bits. The total randomness is 74 bits.

Collision probability for UUID v7:

- Within a single millisecond, the collision probability when generating one million IDs in that interval is about 10^{-10}.
- Overall, considering the space of possible values ($2^{74} = 1.9 \times 10^{22}$ per millisecond), the chance of a random collision between two UUID v7 values remains extremely low, even with parallel generation on many nodes.

The standard allows implementations to use an additional monotonic counter in the random bits to guarantee ordering when generating multiple UUIDs within the same millisecond.

On a single machine, the collision probability on one node depends on the quality of the random number generator, system clock accuracy, and other factors. If the generator is reliable enough, the collision probability will be low. On different machines, distributed generation of UUID v7 also does not require coordination, provided each node has a sufficiently good random generator and reasonably correct clocks. However, incorrect system clocks may disrupt the generation order of UUID v7.

2. How to store UUID?

As mentioned above, a UUID is a 128-bit unique value, so it requires 16 bytes for storage. Firebird does not have a dedicated data type for storing UUID, but you can store it simply as a byte array using the `BINARY(16)` data type. If you want your table definitions to look like PostgreSQL, just create a domain named `UUID`.

```
CREATE DOMAIN UUID AS BINARY(16);
```

Before Firebird 4.0, there was no `BINARY` type for storing binary data; nevertheless, you could store fixed-length binary data in a `CHAR(N)` column with the `OCTETS` character set. Thus, in earlier versions the `UUID` domain should be declared as:

```
CREATE DOMAIN UUID AS CHAR(16) CHARACTER SET OCTETS;
```

At the moment I am not aware of any plans to add a separate UUID type to Firebird or make it a keyword; nevertheless, you can take some

precautions, for example, name the domain not `UUID` but `D_UUID`, to avoid accidentally breaking compatibility with future versions of Firebird if that changes.

Now you can use the `UUID` domain as a data type for unique identifiers:

```
CREATE TABLE T (
   ID UUID NOT NULL,
   NAME VARCHAR(30) NOT NULL,
   ...
   CONSTRAINT PK_T_ID PRIMARY KEY (ID)
);
```

Some users store UUIDs in human-readable form in `CHAR(36) CHARACTER SET ASCII` columns. It should be noted that in this form a UUID takes 20 bytes more (this is not so significant). But the biggest problem is in indexes built on such columns — their keys take significantly more space, which can lead to greater index depth. Therefore, we recommend building keys and indexes only on UUID columns stored in binary form. More about UUID indexing will be written below.

3. Generating UUIDs

RFC 9562 defines several versions of UUID. Each version has its own requirements for generating new UUID values and has its own advantages and disadvantages. A UUID can be generated on the application side as well as on the DBMS side.

Firebird has a built-in function `GEN_UUID()`, which generates UUID version 4.

In Firebird 6.0, the `GEN_UUID()` function can accept a parameter that specifies the version of the UUID generation algorithm. Valid version values are 4 and 7. If the argument is not provided, UUID version 4 is generated.

Inserting a new record into a table using the `GEN_UUID()` function will look like this:

```
INSERT INTO T (ID, NAME, ...)
VALUES (GEN_UUID(), ?, ...)
RETURNING ID
```

If you do not want to write `GEN_UUID()` in your queries every time, you can create a `BEFORE INSERT` trigger that will populate the `ID` field if it is not set in the query.

```
CREATE OR ALTER TRIGGER TR_T_BI FOR T
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
  IF (NEW.ID IS NULL) THEN
    NEW.ID = GEN_UUID();
END
```

Unfortunately, specifying the `GEN_UUID()` function as a default value for the `ID` field is not allowed.

4. Displaying UUID in human-readable form

The `GEN_UUID()` function returns a universally unique identifier (UUID) as a 16-byte binary string. Displaying such an identifier in human-readable form can be delegated to the client application. However, if you need to convert a UUID to a character string, Firebird provides the `UUID_TO_CHAR()` function.

Examples of using `UUID_TO_CHAR()`:

```
select uuid_to_char(gen_uuid()) from rdb$database;
-- A4D82B77-BFF8-4D05-BF28-16A24B6ACE2C

select uuid_to_char(id) as s_uuid from t;
-- A4D82B77-BFF8-4D05-BF28-16A24B6ACE2C
```

There is also the inverse function `CHAR_TO_UUID()`, which converts a readable 36-character UUID string to the corresponding 16-byte UUID value.

```
SELECT CHAR_TO_UUID('A0bF4E45-3029-2a44-D493-4998c9b439A3') FROM rdb$database;
```

5. Indexing UUID columns

Now it is time to talk about indexes on columns storing UUIDs. As you know, indexes are created automatically for primary, alternate (unique), and foreign keys. Let us compare the size and efficiency of indexes for UUID columns of different storage types and indexes on numeric columns.

A primary key with a numeric type is usually associated with a generator (sequence), and therefore such key values are monotonically increasing.

Primary keys with a UUID type can be either monotonically increasing or random, depending on the UUID version.

For UUID version 4, the key values are distributed randomly, so for such keys there is no point in using any predicates other than equality, inequality (<>), `IS [NOT] NULL`, and `IS [NOT] DISTINCT`. Also, it makes no sense to try to order a result set by a column containing UUID version 4, which means that for such columns there is no difference between `ASCENDING` and `DESCENDING` indexes. Firebird has a built-in function `GEN_UUID()` for generating UUID version 4.

For UUID version 7, the key values are partially ordered and monotonically increasing within a single server. For such keys, in some cases predicates like < and > may be meaningful, as well as ordering by such keys. Firebird up to version 6.0 does not have built-in functions for generating UUID version 7, but you can obtain such identifiers from the application or write your own external function (UDR). Starting with Firebird 6.0, you can use the built-in function `GEN_UUID()` to generate UUID version 7 by passing 7 as an argument.

I decided to do a small test to demonstrate index sizes and efficiency for different key types. I will run the tests on a snapshot of Firebird 6.0, as it has a built-in function for generating UUID version 7. For UUID version 4 and for numeric keys, there is no difference between Firebird 6.0 and 5.0.

```
create table test (
  id bigint not null,
  uuid_v4 binary(16) not null,
  uuid_v4s char(36) character set ascii not null,
  uuid_v7 binary(16) not null,
  uuid_v7s char(36) character set ascii not null
);

insert into test(id, uuid_v4, uuid_v4s, uuid_v7, uuid_v7s)
with
  t as (
    select
      n,
      gen_uuid() as uuid_4,
      gen_uuid(7) as uuid_7
    from generate_series (1, 1000000) as S(n)

    union all
    -- for materialization of columns obtained via gen_uuid
    select null, null, null
    from rdb$database
    where false
  )
select
  n,
  uuid_4,
  uuid_to_char(uuid_4) as uuid_4s,
  uuid_7,
  uuid_to_char(uuid_7) as uuid_7s
from t;
```

```
commit;
```

5.1. Index creation time

Now we create unique ASCENDING indexes on all columns. The test will be run in `isql` with the AUTODDL mode enabled, so the time of `COMMIT` is included in the execution time of the index creation command.

```
CREATE UNIQUE INDEX IDX_TEST_ID ON TEST(ID);
```

```
Elapsed time = 0.912 sec
Buffers = 51200
Reads = 0
Writes = 761
Fetches = 1043284
```

```
CREATE UNIQUE INDEX IDX_TEST_UUID_V4 ON TEST(UUID_V4);
```

```
Elapsed time = 1.319 sec
Buffers = 51200
Reads = 1
Writes = 1257
Fetches = 1043989
```

```
CREATE UNIQUE INDEX IDX_TEST_UUID_V4S ON TEST(UUID_V4S);
```

```
Elapsed time = 2.141 sec
Buffers = 51200
Reads = 0
Writes = 2356
Fetches = 1046191
```

```
CREATE UNIQUE INDEX IDX_TEST_UUID_V7 ON TEST(UUID_V7);
```

```
Elapsed time = 1.172 sec
Buffers = 51200
Reads = 0
Writes = 950
Fetches = 1043377
```

```
CREATE UNIQUE INDEX IDX_TEST_UUID_V7S ON TEST(UUID_V7S);
```

```
Elapsed time = 1.470 sec
Buffers = 51200
Reads = 0
Writes = 1614
Fetches = 1044709
```

Comparing index creation times and the number of written pages (Writes), we can already draw some conclusions. The fastest to create are indexes on monotonically increasing numeric keys, followed by indexes for UUID version 7 stored in binary form, then UUID version 4 stored in binary form. The slowest to create are indexes for UUID stored in human-readable form; they also require writing almost twice as many pages to disk, which indirectly indicates that these indexes are less compact.

5.2. Search by key test

Let's compare the performance of equality search on these keys. Since searching for a single value takes very little time, we will make the search for a key value happen 100,000 times.

```
execute block
returns (n bigint)
as
  declare i int = 0;
  declare find_id bigint = 500000;
begin
  while (i < 100000) do
  begin
    i = i + 1;

    select count(*)
    from test
    where id = :find_id
    into n;
  end
  suspend;
end
```

```
                     N
=====================
                     1

Elapsed time = 0.266 sec
Buffers = 51200
Reads = 0
Writes = 0
```

```
Fetches = 400000
```

Now we will search for a UUID version 4 key in binary form:

```
execute block
returns (n bigint)
as
  declare i int = 0;
  declare find_uuid_v4 binary(16);
begin
  find_uuid_v4 = CHAR_TO_UUID('4D1EB968-3CA5-41F5-A519-2FAF15A79EFA');

  while (i < 100000) do
  begin
    i = i + 1;

    select count(*)
    from test
    where uuid_v4 = :find_uuid_v4
    into n;
  end
  suspend;
end
```

```
                     N
======================
                     1

Elapsed time = 0.256 sec
Buffers = 51200
Reads = 0
Writes = 0
Fetches = 400000
```

Repeat this test, but this time search by UUID version 4 key in character form:

```
execute block
returns (n bigint)
as
  declare i int = 0;
  declare find_uuid_v4s char(36) character set ascii = '4D1EB968-3CA5-41F5-A519-2FAF15A79EFA';
begin
  while (i < 100000) do
  begin
    i = i + 1;

    select count(*)
    from test
    where uuid_v4s = :find_uuid_v4s
    into n;
  end
```

```
  suspend;
end
```

```
                     N
=====================
                     1

Elapsed time = 0.272 sec
Buffers = 51200
Reads = 0
Writes = 0
Fetches = 400000
```

Now we will search for a UUID version 7 key in binary form:

```
execute block
returns (n bigint)
as
  declare i int = 0;
  declare find_uuid_v7 binary(16);
begin
  find_uuid_v7 = CHAR_TO_UUID('019EAC8E-77AE-7076-BFB1-0805300BC670');

  while (i < 100000) do
  begin
    i = i + 1;

    select count(*)
    from test
    where uuid_v7 = :find_uuid_v7
    into n;
  end
  suspend;
end
```

```
                     N
=====================
                     1

Elapsed time = 0.168 sec
Buffers = 51200
Reads = 0
Writes = 0
Fetches = 400012
```

Finally, we search by UUID version 7 key in character form:

```
execute block
returns (n bigint)
as
```

```
  declare i int = 0;
  declare find_uuid_v7s char(36) character set ascii = '019EAC8E-77AE-7076-BFB1-0805300BC670';
begin
  while (i < 100000) do
  begin
    i = i + 1;

    select count(*)
    from test
    where uuid_v7s = :find_uuid_v7s
    into n;
  end
  suspend;
end
```

```
                     N
=====================
                     1

Elapsed time = 0.175 sec
Buffers = 51200
Reads = 0
Writes = 0
Fetches = 400012
```

In this test, the difference between the different key types is insignificant. The reason will be explained later.

5.3. Test of fetching the first value by key

If you are using a monotonically increasing numeric identifier, it is obvious that the smaller the key value, the earlier the record was created (within a single server). Therefore, the following query makes sense:

```
select
  id, UUID_V4S, UUID_V7S
from test
order by id
fetch first row only;
```

```
                   ID UUID_V4S                             UUID_V7S
===================== ===================================== =====================================
                    1 4488D022-40B2-4CAC-A3FE-DA983BF5DCE1 019EAC8E-6D28-754D-9029-84C131947DEF

Elapsed time = 0.002 sec
Buffers = 51200
Reads = 0
Writes = 0
```

```
Fetches = 4
```

This query executes almost instantly.

Since UUID version 4 keys are not monotonically increasing, such a query does not make sense, but we will run it for performance evaluation.

```
select
  id, UUID_V4S, UUID_V7S
from test
order by uuid_v4
fetch first row only;
```

```
                  ID UUID_V4S                              UUID_V7S
==================== ===================================== ====================================
              574504 000005C9-F71F-4435-A9D4-FB0F9F232753 019EAC8E-7946-71F1-82B9-396AA08C773D

Elapsed time = 0.002 sec
Buffers = 51200
Reads = 0
Writes = 0
Fetches = 5
```

The query runs fast, but has no practical meaning. We got some random record from the middle of the table.

UUID version 7 keys are partially ordered and monotonically increasing within the current server. Such a query still has no meaning, because UUID version 7 contains a random component, but it does not manifest itself in the same way as for UUID version 4.

```
select
  id, UUID_V4S, UUID_V7S
from test
order by uuid_v7
fetch first row only;
```

```
                  ID UUID_V4S                              UUID_V7S
==================== ===================================== ====================================
                  27 45682B35-DB42-47B8-91ED-A3B245809576 019EAC8E-6D28-705C-AB23-6A7FACBB40F1

Elapsed time = 0.002 sec
Buffers = 51200
Reads = 0
Writes = 0
Fetches = 5
```

This query returned some record from the beginning of the table, not from the middle like UUID version 4.

5.4. Full index scan comparison

The previous queries ran as fast as possible because they just looked for the first key. But what if an Index Full Scan is performed?

```
select count(*)
from (
  select
    id, UUID_V4S, UUID_V7S
  from test
  order by id
);
```

```
                COUNT
=====================
              1000000

Elapsed time = 0.472 sec
Buffers = 51200
Reads = 0
Writes = 0
Fetches = 2011053
```

Now we run this test for UUID version 4 stored in binary form.

```
select count(*)
from (
  select
    id, UUID_V4S, UUID_V7S
  from test
  order by uuid_v4
);
```

```
                COUNT
=====================
              1000000

Elapsed time = 1.186 sec
Buffers = 51200
Reads = 0
Writes = 0
Fetches = 2996687
```

Here it can be seen that a full scan over the index keys for UUID version 4 stored in

binary form is much slower than for monotonically increasing numeric keys.

Now we repeat this test for UUID version 4 stored in text form.

```
select count(*)
from (
  select
    id, UUID_V4S, UUID_V7S
  from test
  order by uuid_v4s
);
```

```
                COUNT
=====================
              1000000

Elapsed time = 1.243 sec
Buffers = 51200
Reads = 0
Writes = 0
Fetches = 2997783
```

The execution time remained approximately the same, but the number of physical reads increased.

Now we run this test for UUID version 7 stored in binary form.

```
select count(*)
from (
  select
    id, UUID_V4S, UUID_V7S
  from test
  order by uuid_v7
);
```

```
                COUNT
=====================
              1000000

Elapsed time = 0.570 sec
Buffers = 51200
Reads = 0
Writes = 0
Fetches = 2011242
```

A full scan over the index keys for UUID version 7 stored in binary form is much faster than for UUID version 4 keys, and is practically comparable to a full scan over the keys of a monotonically increasing index on a numeric column.

Now we repeat this test for UUID version 7 stored in text form.

```
select count(*)
from (
  select
    id, UUID_V4S, UUID_V7S
  from test
  order by uuid_v7s
);
```

```
                COUNT
=====================
              1000000

Elapsed time = 0.587 sec
Buffers = 51200
Reads = 0
Writes = 0
Fetches = 2011906
```

The execution time remained approximately the same, but the number of physical reads increased.

5.5. DESCENDING indexes

As mentioned above, for monotonically increasing numeric identifiers it makes sense to create a DESCENDING index if you want to get something like the last added records (within a single server). For UUID version 4 keys this does not make sense, but we will still demonstrate how UUIDs work with such indexes.

```
CREATE UNIQUE DESC INDEX IDX_TEST_ID_DESC ON TEST(ID);
```

```
Elapsed time = 0.863 sec
Buffers = 51200
Reads = 0
Writes = 760
Fetches = 1044718
```

```
CREATE UNIQUE DESC INDEX IDX_TEST_UUID_V4_DESC ON TEST(UUID_V4);
```

```
Elapsed time = 1.442 sec
Buffers = 51200
Reads = 0
Writes = 1255
```

```
Fetches = 1043993
```

```
CREATE UNIQUE DESC INDEX IDX_TEST_UUID_V7_DESC ON TEST(UUID_V7);
```

```
Elapsed time = 1.144 sec
Buffers = 51200
Reads = 0
Writes = 948
Fetches = 1043380
```

I did not create DESCENDING indexes for UUIDs stored in text form; the picture there will be about the same as for ASCENDING indexes.

So, we execute the following query to get the 5 most recently added records.

```
select
  id, UUID_V4S, UUID_V7S
from test
order by id desc
fetch first 5 rows only;
```

```
                  ID UUID_V4S                              UUID_V7S
==================== ====================================== ====================================
             1000000 71180161-A8BF-46EB-B62B-FF00D4F6DB2C 019ECA02-7471-7329-AA81-AE366DD5146B
              999999 4EFFB2DE-5CE2-4C5D-8BD1-BB168B5A0CED 019ECA02-7471-77E9-8D44-D6DC369B9674
              999998 FE694C51-DF96-4CC3-95CF-C989DCD3AE43 019ECA02-7471-795C-8274-74EE5715AD34
              999997 37AF0083-0D4D-412B-9FEF-D1E160D548A9 019ECA02-7471-7EBC-8472-659390764AD5
              999996 DFE2C353-F27D-4EF3-964C-D1FBE80406E9 019ECA02-7471-7BD2-8896-B1011B4C17DF

Elapsed time = 0.004 sec
Buffers = 51200
Reads = 0
Writes = 0
Fetches = 24
```

The query runs fast and the output matches expectations – it returns the 5 records that were added last.

Let’s run a similar query for a UUID version 4 key.

```
select
  id, UUID_V4S, UUID_V7S
from test
order by UUID_V4 desc
fetch first 5 rows only;
```

```
                  ID UUID_V4S                              UUID_V7S
==================== ===================================== =====================================
              729466 FFFFFC2F-E706-40AC-9E46-26FC50E2C5C5 019ECA02-6EB3-76F7-B20C-3D0A4E9F4565
              191645 FFFFF897-DA23-4480-822D-0B949B05766B 019ECA02-6392-7409-A07D-9B817634F643
              468619 FFFFDF10-A187-4FEA-929C-61F4AD42A4ED 019ECA02-69BB-7F70-8443-B444A9FBF5BB
              147745 FFFFDB65-17CA-412A-A5B2-D07ACA857911 019ECA02-62BB-7883-9C5F-B3422354FECC
               44583 FFFF9D89-BC3B-447B-B424-2A4F49EFE80D 019ECA02-60BF-79D3-9DDB-66AC3B6C4B96

Elapsed time = 0.004 sec
Buffers = 51200
Reads = 0
Writes = 0
Fetches = 30
```

The query executed quickly, but we got a random set of rows as a result. Hence the conclusion: the direction of the index for UUID version 4 keys does not matter at all, because we cannot get a meaningful result from ordering by such a key.

Let's run a similar query for a UUID version 7 key.

```
select
  id, UUID_V4S, UUID_V7S
from test
order by UUID_V7 desc
fetch first 5 rows only;
```

```
                  ID UUID_V4S                              UUID_V7S
==================== ===================================== =====================================
              999984 3C67C2F0-D16A-477A-893E-605AA5EBA180 019ECA02-7471-7FE9-B944-57411F3C2B1B
              999997 37AF0083-0D4D-412B-9FEF-D1E160D548A9 019ECA02-7471-7EBC-8472-659390764AD5
              999981 3C3DC5B6-3631-4696-B452-2F7393C010D3 019ECA02-7471-7E85-B5EB-129BCC0EF65F
              999975 6B619B12-EB5B-4B2A-B045-262754B3B480 019ECA02-7471-7CD4-93D6-453B2DB102BF
              999976 DC1FB864-58D6-465B-8863-250157775E90 019ECA02-7471-7CC2-9D85-B2310914DB35

Elapsed time = 0.004 sec
Buffers = 51200
Reads = 0
Writes = 0
Fetches = 25
```

The query executed quickly. As a result of the query, we got 5 random recently added records. This is probably not exactly what we would have wanted, although such a result can be useful in some cases. Thus, if you may need such a result, it makes sense to choose the index direction for such keys.

5.6. Bulk insert test

At this point I was about to move on to comparing index statistics and draw a

conclusion, but I was told that in a bulk insert test these keys behave very differently, so I decided to add one more test.

In this test, we will compare inserts into a table consisting of only one column, which is the primary key. For each key type, a separate table will be used. In this test, we will measure both the time of the `insert .. select` statement itself and the time of transaction commit, because pages are guaranteed to be written only after `COMMIT`.

5.6.1. Inserting BIGINT keys

```
create table test_bigint (
  id bigint not null,
  constraint pk_test_bigint primary key(id)
);
```

```
insert into test_bigint (id)
select n
from generate_series(1, 1000000) as s(n);
```

```
Elapsed time = 6.172 sec
Buffers = 51200
Reads = 0
Writes = 2208
Fetches = 5026174
```

```
commit;
```

```
Elapsed time = 1.620 sec
Buffers = 51200
Reads = 0
Writes = 3920
Fetches = 2
```

5.6.2. Inserting UUID v4 keys

```
create table test_uuid_v4 (
  id binary(16) not null,
  constraint pk_test_uuid_v4 primary key(id)
);
```

```
insert into test_uuid_v4 (id)
select gen_uuid()
```

```
from generate_series(1, 1000000) as s(n);
```

```
Elapsed time = 12.014 sec
Buffers = 51200
Reads = 0
Writes = 5425
Fetches = 5641769
```

```
commit;
```

```
Elapsed time = 0.635 sec
Buffers = 51200
Reads = 0
Writes = 5956
Fetches = 2
```

5.6.3. Inserting UUID v4 keys in string representation

```
create table test_uuid_v4s (
  id char(36) character set ascii not null,
  constraint pk_test_uuid_v4s primary key(id)
);
```

```
insert into test_uuid_v4s (id)
select uuid_to_char(gen_uuid())
from generate_series(1, 1000000) as s(n);
```

```
Elapsed time = 12.454 sec
Buffers = 51200
Reads = 0
Writes = 10397
Fetches = 5936384
```

```
commit;
```

```
Elapsed time = 0.885 sec
Buffers = 51200
Reads = 0
Writes = 8835
Fetches = 2
```

5.6.4. Inserting UUID v7 keys

```
create table test_uuid_v7 (
  id binary(16) not null,
  constraint pk_test_uuid_v7 primary key(id)
);
```

```
insert into test_uuid_v7 (id)
select gen_uuid(7)
from generate_series(1, 1000000) as s(n);
```

```
Elapsed time = 7.389 sec
Buffers = 51200
Reads = 0
Writes = 5109
Fetches = 5542995
```

```
commit;
```

```
Elapsed time = 0.421 sec
Buffers = 51200
Reads = 0
Writes = 4161
Fetches = 2
```

5.6.5. Inserting UUID v7 keys in string representation

```
create table test_uuid_v7s (
  id char(36) character set ascii not null,
  constraint pk_test_uuid_v7s primary key(id)
);
```

```
insert into test_uuid_v7s (id)
select uuid_to_char(gen_uuid(7))
from generate_series(1, 1000000) as s(n);
```

```
Elapsed time = 10.070 sec
Buffers = 51200
Reads = 0
Writes = 8931
Fetches = 5871645
```

```
commit;
```

```
Elapsed time = 0.541 sec
Buffers = 51200
Reads = 1
Writes = 5395
Fetches = 2
```

5.6.6. Comparison of bulk insert test results

From the results of the bulk insert test, it is evident that the fastest inserts are for `BIGINT` keys. There is also an interesting feature: quite a lot of work remains for the commit itself. For UUID keys, insertion is slower, but less work remains at commit time. This may be because key generation is not free, i.e. part of the time is spent on executing the `gen_uuid()` function. UUID version 7 is about 1.5 times faster than UUID version 4. Storing UUID in binary form results in faster insertion and fewer page modifications.

5.7. Statistics comparison

Now let's look at the index statistics for different key types.

```
    Index "IDX_TEST_ID" (1)
        Root page: 11316, depth: 2, leaf buckets: 741, nodes: 1000000
        Average node length: 11.99, total dup: 0, max dup: 0
        Average key length: 9.04, compression ratio: 1.00
        Average prefix length: 2.96, average data length: 6.04
        Clustering factor: 10310, ratio: 0.01
        Fill distribution:
             0 - 19% = 0
            20 - 39% = 1
            40 - 59% = 0
            60 - 79% = 0
            80 - 99% = 740

    Index "IDX_TEST_ID_DESC" (0)
        Root page: 4093, depth: 2, leaf buckets: 741, nodes: 1000000
        Average node length: 11.99, total dup: 0, max dup: 0
        Average key length: 9.04, compression ratio: 1.00
        Average prefix length: 2.96, average data length: 6.04
        Clustering factor: 10310, ratio: 0.01
        Fill distribution:
             0 - 19% = 0
            20 - 39% = 1
            40 - 59% = 0
            60 - 79% = 0
            80 - 99% = 740
```

```
Index "IDX_TEST_UUID_V4" (2)
    Root page: 12760, depth: 3, leaf buckets: 1236, nodes: 1000000
    Average node length: 19.98, total dup: 0, max dup: 0
    Average key length: 17.03, compression ratio: 0.94
    Average prefix length: 1.96, average data length: 14.03
    Clustering factor: 999891, ratio: 1.00
    Fill distribution:
         0 - 19% = 0
        20 - 39% = 0
        40 - 59% = 0
        60 - 79% = 1
        80 - 99% = 1235

Index "IDX_TEST_UUID_V4S" (3)
    Root page: 13685, depth: 3, leaf buckets: 2332, nodes: 1000000
    Average node length: 37.64, total dup: 0, max dup: 0
    Average key length: 34.69, compression ratio: 1.04
    Average prefix length: 4.31, average data length: 31.69
    Clustering factor: 999891, ratio: 1.00
    Fill distribution:
         0 - 19% = 0
        20 - 39% = 0
        40 - 59% = 0
        60 - 79% = 1
        80 - 99% = 2331

Index "IDX_TEST_UUID_V4_DESC" (6)
    Root page: 18841, depth: 3, leaf buckets: 1236, nodes: 1000000
    Average node length: 19.98, total dup: 0, max dup: 0
    Average key length: 17.03, compression ratio: 0.94
    Average prefix length: 1.97, average data length: 14.03
    Clustering factor: 999891, ratio: 1.00
    Fill distribution:
         0 - 19% = 0
        20 - 39% = 0
        40 - 59% = 0
        60 - 79% = 1
        80 - 99% = 1235

Index "IDX_TEST_UUID_V7" (4)
    Root page: 16489, depth: 3, leaf buckets: 929, nodes: 1000000
    Average node length: 15.01, total dup: 0, max dup: 0
    Average key length: 12.06, compression ratio: 1.33
    Average prefix length: 6.93, average data length: 9.06
    Clustering factor: 603272, ratio: 0.60
    Fill distribution:
         0 - 19% = 0
        20 - 39% = 1
        40 - 59% = 0
        60 - 79% = 0
        80 - 99% = 928

Index "IDX_TEST_UUID_V7S" (5)
    Root page: 17062, depth: 3, leaf buckets: 1593, nodes: 1000000
    Average node length: 25.70, total dup: 0, max dup: 0
    Average key length: 22.75, compression ratio: 1.58
```

```
        Average prefix length: 16.25, average data length: 19.75
        Clustering factor: 603272, ratio: 0.60
        Fill distribution:
             0 - 19% = 0
            20 - 39% = 1
            40 - 59% = 0
            60 - 79% = 0
            80 - 99% = 1592

    Index "IDX_TEST_UUID_V7_DESC" (7)
        Root page: 20218, depth: 3, leaf buckets: 929, nodes: 1000000
        Average node length: 15.01, total dup: 0, max dup: 0
        Average key length: 12.06, compression ratio: 1.41
        Average prefix length: 7.93, average data length: 9.06
        Clustering factor: 603272, ratio: 0.60
        Fill distribution:
             0 - 19% = 0
            20 - 39% = 1
            40 - 59% = 0
            60 - 79% = 0
            80 - 99% = 928
```

Indexes of type `BIGINT` are the most compact, their depth is 2, while for UUID indexes the depth is 3. Also, for indexes on a `BIGINT` column the clustering factor is the best, meaning that ordering by such an index will be the most efficient, as confirmed by the tests.

Indexes on binary UUID types are more compact than indexes where UUID is stored in character form. Indexes on columns for UUID version 7 take 1.5 times less space than UUID version 4. They compress better with prefix compression and have a better clustering factor.

6. Conclusion

Firebird allows you to work with universally unique identifiers (UUIDs); adding a new type to the Firebird kernel is not required. For storing UUIDs, use the `BINARY(16)` data type. You can create a domain named `UUID` and use it when defining table columns or stored procedure parameters. A UUID can be generated either on the application side or on the Firebird server.

For generating UUIDs on the Firebird side, the `GEN_UUID()` function is used, which generates UUID version 4. Indexes for UUID version 7 are more compact than for version 4 and are partially ordered by time. Firebird 5.0 does not have built-in functions for generating UUID version 7, but you can generate such keys on the application side or using an external function. In Firebird 6.0, you can use the `GEN_UUID()` function to generate UUID version 7.

To convert a UUID to a human-readable form, you can use the built-in function `UUID_TO_CHAR()`. To convert a UUID from character form to binary, you can use the function `CHAR_TO_UUID()`.