Использование UUID в Firebird

Содержание

Этот материал был создан при поддержке и спонсорстве компании iBase.ru, которая разрабатывает инструменты Firebird SQL для предприятий и предоставляет сервис технической поддержки для Firebird SQL.

Предисловие

Некоторые пользователи Firebird высказывают пожелания, чтобы в Firebird добавили новый тип данных `UUID` по аналогии с PostgreSQL. Что же давайте разберёмся, что такое UUID, как с ним работать в Firebird и нужен ли для этого отдельный тип данных.

1. Что такое UUID?

UUID (Universally Unique Identifiers) — это универсальный уникальный идентификатор, который определён в RFC 9562, ISO/IEC 9834-8:2005 и связанных стандартах. (В некоторых системах это называется GUID, глобальным уникальным идентификатором.) Этот идентификатор представляет собой 128-битное значение, генерируемое специальным алгоритмом, практически гарантирующим, что этим же алгоритмом оно не будет получено больше нигде в мире. Таким образом, эти идентификаторы будут уникальными и в распределённых системах, а не только в единственной базе данных, как значения генераторов последовательностей.

1.1. UUID версии 4

Эта версия генерируется с использованием криптографически стойкого генератора псевдослучайных чисел (CSPRNG) для заполнения 122 из 128 бит. Оставшиеся 6 бит зарезервированы: 4 бита кодируют версию (0100), 2 бита — вариант (10).

Некоторые расчёты вероятности коллизий:

- Чтобы достичь вероятности коллизии около 50%, нужно сгенерировать примерно 2.71 квадриллиона UUID v4.
- При скорости генерации 1 миллиард UUID в секунду для достижения такого количества потребуется около 86 лет.

На практике коллизии UUID v4 практически не возникают. Однако важно учитывать, что вероятность не равна нулю, и со временем коллизия может произойти при достаточно активном использовании.

На одной машине вероятность коллизии зависит от качества генератора случайных чисел и частоты генерации. Если генератор недостаточно энтропийный или есть корреляция между генераторами на разных узлах, вероятность коллизий может увеличиться.

На разных машинах при использовании независимых генераторов вероятность

коллизии также крайне мала, если генераторы на всех узлах работают корректно и имеют достаточный уровень энтропии. Однако если на разных машинах используется один и тот же алгоритм генерации, это может повысить риск коллизий, если генераторы коррелируют.

1.2. UUID версии 7

Эта версия стандартизирована в RFC 9562 (2024 год). Она включает 48 бит временной метки Unix (в миллисекундах) в старших битах, затем идут биты версии, 12 случайных бит, биты варианта и ещё 62 случайных бита. Суммарно случайность составляет 74 бита.

Вероятность коллизии для UUID v7:

- В пределах одной миллисекунды вероятность коллизии при генерации миллиона ID за этот интервал — около 10^{-10}.
- В целом, учитывая пространство возможных значений ($2^{74} = 1.9 \times 10^{22}$ на миллисекунду), вероятность случайной коллизии между двумя UUID v7 также остаётся крайне низкой, даже при параллельной генерации на множестве узлов.

Стандарт позволяет реализациям использовать дополнительный монотонный счётчик в случайных битах для гарантии упорядоченности при генерации нескольких UUID в пределах одной миллисекунды.

На одной машине вероятность коллизии на одном узле зависит от качества генератора случайных чисел, точности системных часов и других факторов. Если генератор достаточно надёжен, вероятность коллизий будет низкой. На разных машинах распределённая генерация UUID v7 также не требует координации, если каждый узел имеет достаточно качественный генератор случайностей и относительно правильные часы. Однако некорректные системные часы могут нарушить порядок генерации UUID v7.

2. Как хранить UUID?

Как было сказано выше, UUID представляет собой 128-битное уникальное значение, таким образом для его хранения требуется 16 байт. В Firebird нет специального типа данных для хранения UUID, но его можно сохранить просто как набор байт в типе данных `BINARY(16)`. Если вы хотите, чтобы определение ваших таблиц было похоже на PostgreSQL, то просто создайте домен с именем `UUID`.

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

В Firebird до версии 4.0 не было специального типа `BINARY` для хранения бинарных данных, тем не менее вы могли хранить бинарные данные ограниченной длины в типе `CHAR(N)` с набором символов `OCTETS`. Таким образом, в предыдущих версиях домен `UUID` должен быть объявлен так:

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

На данный момент мне неизвестно о планах добавить в Firebird отдельный тип UUID или сделать его ключевым словом, тем не менее вы можете принять некоторые меры предосторожности, например, назвать домен не UUID, а D_UUID, чтобы случайно не сломать совместимость с будущими версиями Firebird, если это изменится.

Теперь вы можете использовать домен `UUID` как тип данных для уникальных идентификаторов:

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

Некоторые пользователи хранят UUID в человеко-читаемой форме в столбцах типа `CHAR(36) CHARACTER SET ASCII`. Хотелось бы отметить, что в таком виде UUID занимает на 20 байт больше (это не так уж существенно). Но самая большая проблема в индексах, построенных на таких полях. Их ключи занимают существенно больше места, что может приводить к большей глубине индекса, поэтому рекомендуем строить ключи и индексы только на столбцах UUID, хранимых в бинарной форме. Подробнее об индексации UUID будет написано ниже.

3. Генерация UUID

RFC 9562 определяет несколько версий UUID. Каждая версия имеет свои требования к генерации новых значений UUID и имеет свои преимущества и недостатки. Значение UUID может быть генерировано как на стороне

приложения, так и на стороне СУБД.

В Firebird имеется встроенная функция `GEN_UUID()`, которая генерирует UUID версии 4.

В Firebird 6.0 функция `GEN_UUID()` может принимать параметр, в который передаётся версия алгоритма генерации UUID. Допустимыми значениями версии являются 4 и 7. Если значение аргумента не задано, то генерируется UUID версии 4.

Добавление новой записи в таблицу с использованием функции `GEN_UUID()` будет выглядеть следующим образом:

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

Если вы не хотите каждый раз писать `GEN_UUID()` в ваших запросах, то можно создать `BEFORE INSERT` триггер, в котором будет заполняться значение поля `ID`, если оно не задано в запросе.

```
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
```

К сожалению, указание функции `GEN_UUID()` в качестве значения по умолчанию для поля `ID` не допускается.

4. Отображение UUID в человеко-читаемой форме

Функция `GEN_UUID()` возвращает универсальный уникальный идентификатор (UUID) в виде 16-байтовой двоичной строки. Отображение такого идентификатора в человеко-читаемой форме можно возложить на клиентское приложение, но если необходимо преобразовать UUID в символьную строку, то для этого в Firebird существует функция `UUID_TO_CHAR()`.

Примеры использования `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
```

Также существует обратная функция `CHAR_TO_UUID()`, которая преобразует читаемую 36-символьную строку UUID к соответствующему 16-ти байтовому значению UUID.

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

5. Индексация столбцов типа UUID

Теперь настало время поговорить об индексах для столбцов, хранящих UUID. Как известно, индексы создаются автоматически для первичного, альтернативного (уникального) и внешнего ключа. Сравним размер и эффективность индексов для полей, хранящих UUID различных типов и индексов на числовые поля.

Первичный ключ с числовым типом обычно связан с генератором (последовательностью), а потому значения таких ключей являются монотонно возрастающими.

Первичные ключи с типом UUID могут быть как монотонно возрастающими, так и случайными в зависимости от версии UUID.

Для UUID версии 4 значения ключей распределены случайным образом, поэтому для таких ключей не имеет смысл выполнять любые предикаты, кроме равенства, неравенства (<>), `IS [NOT] NULL` и `IS [NOT] DISTINCT`. Также не имеет смысла пытаться упорядочить выборку по полю, содержащему UUID версии 4, а это обозначает, что для таких полей нет никакой разницы между `ASCENDING` и `DESCENDING` индексом. Для генерации UUID версии 4 в Firebird есть встроенная функция `GEN_UUID()`.

Для UUID версии 7 значения ключей являются частично упорядоченными и монотонно возрастающими в пределах одного сервера. Для таких ключей в некоторых случаях могут иметь смысл предикаты < и >, а также сортировка по таким ключам. В Firebird до версии 6.0 нет встроенных функций для генерации UUID версии 7, однако вы можете получать такие идентификаторы из приложения или написать свою внешнюю функцию (UDR). Начиная с Firebird 6.0 для генерации UUID версии 7 вы можете использовать встроенную функцию `GEN_UUID()`, передав в неё в качестве параметра значение 7.

Я решил сделать небольшой тест для демонстрации размеров индексов и их эффективности для различных типов ключей. Я буду выполнять тесты на снапшоте Firebird 6.0, поскольку он имеет встроенную функцию для генерации UUID версии 7. Для UUID версии 4 и для числового ключа между Firebird 6.0 и 5.0 нет никакой разницы.

```
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
    -- для материализации столбцов полученных через 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. Время создания индексов

Теперь создадим уникальные ASCENDING индексы для всех полей. Тест будет выполняться в `isql` с включенным режимом AUTODDL, поэтому время `COMMIT` входит во время выполнения команды создания индекса.

```
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
```

Сравнивая время создания индексов и количество записанных страниц (Writes) уже можно сделать некоторые выводы. Быстрее всего создаются индексы для монотонно возрастающих числовых ключей, следом идут индексы для UUID версии 7, хранимые в бинарном виде, затем - UUID версии 4, хранимые в бинарном виде. Медленнее всего создаются индексы для хранения UUID в

человеко-читаемом виде, они же требуют записать почти в 2 раза больше страниц на диск, что косвенно говорит о том, что эти индексы менее компактные.

5.2. Тест поиска по ключу

Попробуем сравнить производительность поиска на равенство по этим ключам. Поскольку поиск одного значения занимает очень мало времени, мы сделаем так, чтобы значение по ключу искалось 100000 раз.

```
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
```

Теперь будем искать ключ типа UUID версии 4 в бинарном виде:

```
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
```

Повторим этот тест, но будем делать поиск по ключу UUID версии 4 в символьном виде:

```
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
```

Теперь будем искать ключ типа UUID версии 7 в бинарном виде:

```
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
```

И наконец сделаем поиск по ключу UUID версии 7 в символьном виде:

```
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
```

В данном тесте разница между различными типами ключа незначительна. Причина будет объяснена позднее.

5.3. Тест поиска первого значения по ключу

Если у вас используется монотонно возрастающий числовой идентификатор, то очевидно, что чем меньше значение ключа, тем раньше была создана запись (в пределах одного сервера). Поэтому следующий запрос имеет смысл:

```
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
```

Этот запрос выполняется практически мгновенно.

Поскольку ключи UUID версии 4 не являются монотонно возрастающими, то подобный запрос не имеет смысла, но мы выполним его для оценки производительности.

```
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
```

Запрос выполняется быстро, но не имеет практического смысла. Мы получили какую-то запись из середины таблицы.

Ключи UUID версии 7 являются частично упорядоченными и монотонно возрастающими в пределах текущего сервера. Такой запрос всё ещё не имеет смысла, потому что в UUID версии 7 присутствует случайная составляющая, но она проявляется не так, как для UUID версии 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
```

Этот запрос вернул какую-то запись из начала таблицы, а не из середины, как UUID версии 4.

5.4. Сравнение полной навигации по индексу

Предыдущие запросы выполнялись максимально быстро, потому что просто искали первый ключ, но что, если будет выполняться Index Full Scan?

```
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
```

Теперь выполним этот тест для UUID версии 4, хранимого в бинарном виде.

```
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
```

Здесь видно, что полное сканирование по ключам индекса для ключей UUID версии 4 хранимого в бинарном виде намного медленнее, чем для монотонно возрастающих числовых ключей.

Теперь повторим этот тест для UUID версии 4, хранимого в текстовом виде.

```
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
```

Время выполнения осталось приблизительно тем же, но увеличилось количество физических чтений.

Теперь выполним этот тест для UUID версии 7, хранимого в бинарном виде.

```
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
```

Полное сканирование по ключам индекса для ключей UUID версии 7 хранимого в бинарном виде намного быстрее, чем для ключей UUID версии 4, и практически сравнимо с полным сканированием по ключам монотонно возрастающего индекса по числовому столбцу.

Теперь повторим этот тест для UUID версии 7, хранимого в текстовом виде.

```
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
```

Время выполнения осталось приблизительно тем же, но увеличилось количество физических чтений.

5.5. DESCENDING индексы

Как говорилось выше, для монотонно возрастающих числовых идентификаторов имеет смысл создавать DESCENDING индекс, если вы хотите получать что-то вроде последних добавленных записей (в пределах одного сервера). Для ключей типа UUID версии 4 это не имеет смысла, но мы всё же продемонстрируем, как работают UUID с такими индексами.

```
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
```

Я не стал создавать DESCENDING индексы для UUID, хранимых в текстовом виде, тут картина будет примерно такая же, как для ASCENDING индексов.

Итак, выполним следующий запрос для получения 5 последних добавленных записей.

```
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
```

Запрос выполняется быстро и вывод совпадает с ожиданием - выводятся 5 записей, добавленных последними.

Выполним подобный запрос для ключа с UUID версии 4.

```
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
```

Запрос выполнился быстро, но в результате мы получили случайный набор строк. Отсюда вывод: направление индекса для ключей UUID версии 4 не играет никакой роли, поскольку мы не можем получить осмысленный результат от сортировки по такому ключу.

Выполним подобный запрос для ключа с UUID версии 7.

```
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
```

Запрос выполнился быстро. В результате выполнения запроса мы получили 5 случайных недавно добавленных записей. Наверное, это не совсем то, чего мы хотели бы, хотя для такого результата можно найти применение. Таким образом, если вам может потребоваться такой результат, то имеет смысл выбрать направление индекса для таких ключей.

5.6. Тест массовой вставки

В этом месте я хотел было уже перейти к сравнению статистики индексов и сделать заключение, но мне подсказали, что на тесте массовой вставки эти

ключи ведут себя сильно по-разному, поэтому я решил добавить ещё один тест.

В этом тесте будет сравниваться вставка в таблицу, состоящую всего из одного столбца, являющегося первичным ключом. Для каждого типа ключа будет использована отдельная таблица. В этом тесте будет измеряться как время самого оператора `insert .. select` так и время подтверждения транзакции, поскольку гарантировано, что все страницы после вставки записей будут записаны только после `COMMIT`;

5.6.1. Вставка ключей типа bigint

```
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. Вставка ключей типа uuid v4

```
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. Вставка ключей типа uuid v4 в строковом представлении

```
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. Вставка ключей типа uuid v7

```
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. Вставка ключей типа uuid v7 в строковом представлении

```
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. Сравнение результатов теста массовой вставки

По результатам тестирования массовой вставки видно, что быстрее всего вставляются ключи с типом bigint. Тут же видна интересная особенность, а именно довольно много работы остаётся на само подтверждение транзакции. Для ключей типа UUID вставка происходит медленней, но на момент commit остаётся меньше работы. Возможно это связано с тем, что сама генерация ключей не бесплатна, то есть часть времени уходит на выполнение функции `gen_uuid()`. UUID версии 7 примерно в 1.5 раза быстрее UUID версии 4. При хранении UUID в бинарном виде вставка происходит быстрее, и необходимо меньше модификаций страниц.

5.7. Сравнение статистики

Теперь посмотрим на статистику индексов для различных типов ключей.

```
    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
```

Индексы с типом BIGINT являются наиболее компактными, их глубина составляет 2, в то время как для индексов с UUID глубина индекса составляет 3. Также для индексов по полю типа BIGINT наилучший фактор кластеризации, то есть сортировка по такому индексу будет наиболее эффективной, что подтверждается тестами.

Индексы с бинарным типом UUID компактнее индексов, в которых UUID хранится в символьном виде. Индексы по столбцам для UUID для версии 7 занимают в 1.5 раза меньше места, чем UUID версии 4. Они лучше сжимаются префиксной компрессией и для них лучше фактор кластеризации.

6. Заключение

Firebird позволяет работать с универсальными уникальными идентификаторами (UUID), добавлять новый тип в ядро Firebird не требуется. Для хранения UUID используйте тип данных `BINARY(16)`. Вы можете создать домен с именем `UUID` и использовать его при определении столбцов таблиц или параметров хранимых процедур. UUID может быть генерирован как на стороне приложения, так и на сервере Firebird.

Для генерации UUID на стороне Firebird используется функция `GEN_UUID()`, которая генерирует UUID версии 4. Индексы для UUID версии 7 компактнее, чем для версии 4, и являются частично упорядоченными по времени. Firebird 5.0 не имеет встроенных функций для генерации UUID версии 7, но вы можете

генерировать такие ключи на стороне приложения или с помощью внешней функции. В Firebird 6.0 вы сможете использовать функцию `GEN_UUID()` для генерации UUID версии 7.

Для преобразования UUID в человеко-читаемую форму вы можете воспользоваться встроенной функцией `UUID_TO_CHAR()`. Для преобразования UUID из символьной формы в бинарную вы можете воспользоваться функцией `CHAR_TO_UUID()`.