Usando UUID no Firebird

Sumário

Este material foi criado com o apoio e patrocínio da IBSurgeon ib-aid.com, empresa que desenvolve ferramentas Firebird SQL para empresas e fornece serviços de suporte técnico para Firebird SQL.

Prefácio

Alguns usuários do Firebird manifestaram o desejo de que um novo tipo de dado `UUID` seja adicionado ao Firebird, semelhante ao PostgreSQL. Então vamos entender o que é UUID, como trabalhar com ele no Firebird e se é necessário um tipo de dado separado para isso.

1. O que é UUID?

UUID (Identificadores Universalmente Únicos) é um identificador único universal definido em RFC 9562, ISO/IEC 9834-8:2005 e padrões relacionados. (Em alguns sistemas é chamado de GUID, identificador global único.) Esse identificador é um valor de 128 bits gerado por um algoritmo especial que praticamente garante que não será gerado novamente em nenhum outro lugar do mundo pelo mesmo algoritmo. Assim, esses identificadores são únicos não apenas em um único banco de dados, como os valores de geradores de sequência, mas também em sistemas distribuídos.

1.1. UUID versão 4

Esta versão é gerada usando um gerador de números pseudoaleatórios criptograficamente seguro (CSPRNG) para preencher 122 dos 128 bits. Os 6 bits restantes são reservados: 4 bits codificam a versão (0100), 2 bits codificam a variante (10).

Algumas estimativas de probabilidade de colisão:

- Para atingir uma probabilidade de colisão de cerca de 50%, é necessário gerar aproximadamente 2,71 quadrilhões de UUID v4.
- A uma taxa de geração de 1 bilhão de UUIDs por segundo, para atingir essa quantidade seriam necessários cerca de 86 anos.

Na prática, colisões de UUID v4 são virtualmente inexistentes. No entanto, é importante ter em mente que a probabilidade não é zero e, com o tempo, com uso suficientemente intenso, uma colisão pode ocorrer.

Em uma única máquina, a probabilidade de colisão depende da qualidade do gerador de números aleatórios e da frequência de geração. Se o gerador tiver entropia insuficiente ou houver correlação entre geradores em diferentes nós, a probabilidade de colisão pode aumentar.

Em máquinas diferentes usando geradores independentes, a probabilidade de colisão também é extremamente baixa, desde que os geradores em todos os nós funcionem corretamente e tenham entropia suficiente. No entanto, se o mesmo algoritmo de

geração for usado em máquinas diferentes, isso pode aumentar o risco de colisões se os geradores estiverem correlacionados.

1.2. UUID versão 7

Esta versão foi padronizada na RFC 9562 (2024). Ela inclui um timestamp Unix de 48 bits (em milissegundos) nos bits mais significativos, seguido pelos bits de versão, 12 bits aleatórios, bits de variante e outros 62 bits aleatórios. A aleatoriedade total é de 74 bits.

Probabilidade de colisão para UUID v7:

- Dentro de um único milissegundo, a probabilidade de colisão ao gerar um milhão de IDs nesse intervalo é de cerca de 10^{-10}.
- Em geral, considerando o espaço de valores possíveis (2^{74} = 1,9 × 10^{22} por milissegundo), a chance de uma colisão aleatória entre dois valores UUID v7 permanece extremamente baixa, mesmo com geração paralela em muitos nós.

O padrão permite que as implementações usem um contador monotônico adicional nos bits aleatórios para garantir a ordenação ao gerar vários UUIDs dentro do mesmo milissegundo.

Em uma única máquina, a probabilidade de colisão em um nó depende da qualidade do gerador de números aleatórios, da precisão do relógio do sistema e de outros fatores. Se o gerador for suficientemente confiável, a probabilidade de colisão será baixa. Em máquinas diferentes, a geração distribuída de UUID v7 também não requer coordenação, desde que cada nó tenha um gerador aleatório de boa qualidade e relógios razoavelmente corretos. No entanto, relógios de sistema incorretos podem prejudicar a ordem de geração dos UUID v7.

2. Como armazenar UUID?

Como mencionado acima, um UUID é um valor único de 128 bits, portanto requer 16 bytes para armazenamento. O Firebird não possui um tipo de dados dedicado para armazenar UUID, mas você pode armazená-lo simplesmente como uma matriz de bytes usando o tipo de dados `BINARY(16)`. Se você deseja que suas definições de tabela se pareçam com o PostgreSQL, basta criar um domínio chamado `UUID`.

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

Antes do Firebird 4.0, não existia o tipo `BINARY` para armazenar dados binários; no entanto, você podia armazenar dados binários de comprimento fixo em uma coluna

`CHAR(N)` com o conjunto de caracteres `OCTETS`. Assim, em versões anteriores, o domínio `UUID` deve ser declarado como:

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

No momento, não tenho conhecimento de planos para adicionar um tipo UUID separado ao Firebird ou torná-lo uma palavra-chave; no entanto, você pode tomar algumas precauções, por exemplo, nomear o domínio não como `UUID`, mas como `D_UUID`, para evitar quebrar acidentalmente a compatibilidade com versões futuras do Firebird, caso isso mude.

Agora você pode usar o domínio `UUID` como um tipo de dados para identificadores únicos:

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

Alguns usuários armazenam UUIDs em formato legível por humanos em colunas `CHAR(36) CHARACTER SET ASCII`. Deve-se notar que, nessa forma, um UUID ocupa 20 bytes a mais (isso não é tão significativo). Mas o maior problema está nos índices construídos sobre tais colunas – suas chaves ocupam significativamente mais espaço, o que pode levar a uma maior profundidade do índice. Portanto, recomendamos construir chaves e índices apenas em colunas UUID armazenadas em formato binário. Mais sobre indexação de UUID será escrito abaixo.

3. Gerando UUIDs

A RFC 9562 define várias versões de UUID. Cada versão tem seus próprios requisitos para gerar novos valores UUID e tem suas próprias vantagens e desvantagens. Um UUID pode ser gerado tanto no lado da aplicação quanto no lado do SGBD.

O Firebird possui a função embutida `GEN_UUID()`, que gera UUID versão 4.

No Firebird 6.0, a função `GEN_UUID()` pode aceitar um parâmetro que especifica a versão do algoritmo de geração de UUID. Os valores de versão válidos são 4 e 7. Se o argumento não for fornecido, a versão 4 do UUID é gerada.

A inserção de um novo registro em uma tabela usando a função `GEN_UUID()` será assim:

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

Se você não quiser escrever `GEN_UUID()` em suas consultas toda vez, pode criar um trigger `BEFORE INSERT` que preencherá o campo `ID` se ele não for definido na consulta.

```
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
```

Infelizmente, não é permitido especificar a função `GEN_UUID()` como valor padrão para o campo `ID`.

4. Exibindo UUID em formato legível por humanos

A função `GEN_UUID()` retorna um identificador universalmente único (UUID) como uma string binária de 16 bytes. A exibição de tal identificador em formato legível por humanos pode ser delegada ao aplicativo cliente. No entanto, se você precisar converter um UUID para uma string de caracteres, o Firebird fornece a função `UUID_TO_CHAR()`.

Exemplos de uso de `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
```

Há também a função inversa `CHAR_TO_UUID()`, que converte uma string UUID legível de 36 caracteres para o valor UUID de 16 bytes correspondente.

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

5. Indexação de colunas UUID

Agora é hora de falar sobre índices em colunas que armazenam UUIDs. Como você sabe, os índices são criados automaticamente para chaves primárias, alternativas (exclusivas) e estrangeiras. Vamos comparar o tamanho e a eficiência de índices para colunas UUID de diferentes tipos de armazenamento e índices em colunas numéricas.

Uma chave primária com tipo numérico geralmente está associada a um gerador (sequência) e, portanto, os valores dessas chaves são monotonicamente crescentes.

Chaves primárias com tipo UUID podem ser monotonicamente crescentes ou aleatórias, dependendo da versão do UUID.

Para UUID versão 4, os valores das chaves são distribuídos aleatoriamente, portanto, para tais chaves, não faz sentido usar predicados diferentes de igualdade, desigualdade (<>), `IS [NOT] NULL` e `IS [NOT] DISTINCT`. Também não faz sentido tentar ordenar um conjunto de resultados por uma coluna que contém UUID versão 4, o que significa que para tais colunas não há diferença entre índices `ASCENDING` e `DESCENDING`. O Firebird possui a função embutida `GEN_UUID()` para gerar UUID versão 4.

Para UUID versão 7, os valores das chaves são parcialmente ordenados e monotonicamente crescentes dentro de um único servidor. Para tais chaves, em alguns casos, predicados como < e > podem ser significativos, bem como a ordenação por tais chaves. O Firebird até a versão 6.0 não possui funções embutidas para gerar UUID versão 7, mas você pode obter tais identificadores a partir da aplicação ou escrever sua própria função externa (UDR). A partir do Firebird 6.0, você pode usar a função embutida `GEN_UUID()` para gerar UUID versão 7, passando 7 como argumento.

Decidi fazer um pequeno teste para demonstrar os tamanhos e a eficiência dos índices para diferentes tipos de chave. Executarei os testes em um snapshot do Firebird 6.0, pois ele possui uma função embutida para gerar UUID versão 7. Para UUID versão 4 e para chaves numéricas, não há diferença entre o Firebird 6.0 e o 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
    -- para materialização das colunas obtidas 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. Tempo de criação dos índices

Agora vamos criar índices ASCENDING exclusivos em todas as colunas. O teste será executado no `isql` com o modo AUTODDL ativado, portanto o tempo do `COMMIT` está incluído no tempo de execução do comando de criação do índice.

```
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
```

Comparando os tempos de criação dos índices e o número de páginas gravadas (Writes), já podemos tirar algumas conclusões. Os índices mais rápidos de criar são aqueles em chaves numéricas monotonicamente crescentes, seguidos pelos índices para UUID versão 7 armazenados em formato binário, depois UUID versão 4 armazenados em formato binário. Os índices mais lentos são aqueles para UUID armazenados em formato legível por humanos; eles também exigem a gravação de quase o dobro de páginas no disco, o que indica indiretamente que esses índices são menos compactos.

5.2. Teste de busca por chave

Vamos comparar o desempenho da busca por igualdade nessas chaves. Como a busca por um único valor leva muito pouco tempo, faremos com que a busca por um valor de chave ocorra 100.000 vezes.

```
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
```

Agora vamos buscar uma chave UUID versão 4 em formato binário:

```
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
```

Repita este teste, mas agora buscando pela chave UUID versão 4 em formato de caractere:

```
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
```

Agora vamos buscar uma chave UUID versão 7 em formato binário:

```
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
```

Finalmente, vamos buscar pela chave UUID versão 7 em formato de caractere:

```
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
```

Neste teste, a diferença entre os diferentes tipos de chave é insignificante. A razão será explicada mais tarde.

5.3. Teste de busca do primeiro valor por chave

Se você estiver usando um identificador numérico monotonicamente crescente, é óbvio que quanto menor o valor da chave, mais cedo o registro foi criado (dentro de um único servidor). Portanto, a seguinte consulta faz sentido:

```
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
```

Esta consulta é executada quase instantaneamente.

Como as chaves UUID versão 4 não são monotonicamente crescentes, tal consulta não faz sentido, mas a executaremos para avaliação de desempenho.

```
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
```

A consulta é rápida, mas não tem significado prático. Obtivemos um registro aleatório do meio da tabela.

As chaves UUID versão 7 são parcialmente ordenadas e monotonicamente crescentes

dentro do servidor atual. Tal consulta ainda não faz sentido, porque o UUID versão 7 contém um componente aleatório, mas ele não se manifesta da mesma forma que para o UUID versão 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
```

Esta consulta retornou algum registro do início da tabela, não do meio, como o UUID versão 4.

5.4. Comparação de varredura completa do índice

As consultas anteriores foram executadas o mais rápido possível porque apenas procuravam a primeira chave. Mas e se uma varredura completa do índice (Index Full Scan) for executada?

```
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
```

Agora executamos este teste para UUID versão 4 armazenado em formato binário.

```
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
```

Aqui pode-se ver que uma varredura completa sobre as chaves do índice para UUID versão 4 armazenado em formato binário é muito mais lenta do que para chaves numéricas monotonicamente crescentes.

Agora repetimos este teste para UUID versão 4 armazenado em formato de texto.

```
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
```

O tempo de execução permaneceu aproximadamente o mesmo, mas o número de leituras físicas aumentou.

Agora executamos este teste para UUID versão 7 armazenado em formato binário.

```
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
```

Uma varredura completa sobre as chaves do índice para UUID versão 7 armazenado em formato binário é muito mais rápida do que para chaves UUID versão 4 e é praticamente comparável a uma varredura completa sobre as chaves de um índice monotonicamente crescente em uma coluna numérica.

Agora repetimos este teste para UUID versão 7 armazenado em formato de texto.

```
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
```

O tempo de execução permaneceu aproximadamente o mesmo, mas o número de leituras físicas aumentou.

5.5. Índices DESCENDING

Como mencionado acima, para identificadores numéricos monotonicamente crescentes, faz sentido criar um índice DESCENDING se você deseja obter algo como os últimos registros adicionados (dentro de um único servidor). Para chaves UUID versão 4, isso não faz sentido, mas ainda demonstraremos como os UUIDs funcionam com tais índices.

```
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
```

Não criei índices DESCENDING para UUIDs armazenados em formato de texto; o cenário lá será aproximadamente o mesmo que para índices ASCENDING.

Então, executamos a seguinte consulta para obter os 5 registros adicionados mais recentemente.

```
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
```

A consulta é rápida e a saída corresponde às expectativas – retorna os 5 registros que foram adicionados por último.

Vamos executar uma consulta semelhante para uma chave UUID versão 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
```

A consulta foi executada rapidamente, mas obtivemos um conjunto aleatório de linhas como resultado. Daí a conclusão: a direção do índice para chaves UUID versão 4 não importa, pois não podemos obter um resultado significativo da ordenação por tal chave.

Vamos executar uma consulta semelhante para uma chave UUID versão 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
```

A consulta foi executada rapidamente. Como resultado, obtivemos 5 registros aleatórios adicionados recentemente. Provavelmente não é exatamente o que gostaríamos, embora esse resultado possa ser útil em alguns casos. Assim, se você precisar desse tipo de resultado, faz sentido escolher a direção do índice para tais chaves.

5.6. Teste de inserção em massa

Neste ponto, eu estava prestes a passar para a comparação das estatísticas dos índices e tirar uma conclusão, mas me disseram que, em um teste de inserção em massa, essas chaves se comportam de maneira muito diferente, então decidi adicionar mais um teste.

Neste teste, compararemos inserções em uma tabela composta por apenas uma coluna, que é a chave primária. Para cada tipo de chave, será usada uma tabela separada. Neste teste, mediremos tanto o tempo da instrução `insert .. select` em si quanto o tempo de confirmação da transação, porque as páginas são garantidamente gravadas apenas após o `COMMIT`.

5.6.1. Inserindo chaves do tipo 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. Inserindo chaves 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. Inserindo chaves UUID v4 em representação de string

```
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. Inserindo chaves 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. Inserindo chaves UUID v7 em representação de string

```
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. Comparação dos resultados do teste de inserção em massa

Pelos resultados do teste de inserção em massa, é evidente que as inserções mais rápidas são para chaves BIGINT. Há também uma característica interessante: bastante trabalho permanece para o próprio commit. Para chaves UUID, a inserção é mais lenta, mas menos trabalho permanece no momento do commit. Isso pode ser porque a geração de chaves não é gratuita, ou seja, parte do tempo é gasta na execução da função `gen_uuid()`. O UUID versão 7 é cerca de 1,5 vez mais rápido que o UUID versão 4. Armazenar UUID em formato binário resulta em inserção mais rápida e menos

modificações de página.

5.7. Comparação de estatísticas

Agora vamos olhar as estatísticas dos índices para diferentes tipos de chave.

```
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
```

Os índices do tipo BIGINT são os mais compactos, sua profundidade é 2, enquanto para índices UUID a profundidade é 3. Além disso, para índices em uma coluna BIGINT o fator de clusterização é o melhor, o que significa que a ordenação por tal índice será a mais eficiente, conforme confirmado pelos testes.

Índices em tipos UUID binários são mais compactos do que índices onde o UUID é armazenado em formato de caractere. Índices em colunas para UUID versão 7 ocupam 1,5 vez menos espaço do que UUID versão 4. Eles comprimem melhor com compressão de prefixo e têm um melhor fator de clusterização.

6. Conclusão

O Firebird permite que você trabalhe com identificadores universalmente únicos (UUIDs); não é necessário adicionar um novo tipo ao núcleo do Firebird. Para armazenar UUIDs, use o tipo de dados `BINARY(16)`. Você pode criar um domínio chamado `UUID` e usá-lo ao definir colunas de tabelas ou parâmetros de procedimentos armazenados. Um UUID pode ser gerado tanto no lado da aplicação quanto no servidor Firebird.

Para gerar UUIDs no lado do Firebird, a função `GEN_UUID()` é usada, que gera UUID versão 4. Índices para UUID versão 7 são mais compactos do que para a versão 4 e são parcialmente ordenados por tempo. O Firebird 5.0 não possui funções embutidas para gerar UUID versão 7, mas você pode gerar tais chaves no lado da aplicação ou usando uma função externa. No Firebird 6.0, você pode usar a função `GEN_UUID()` para gerar UUID versão 7.

Para converter um UUID para uma forma legível por humanos, você pode usar a função embutida `UUID_TO_CHAR()`. Para converter um UUID da forma de caractere para binária, você pode usar a função `CHAR_TO_UUID()`.