MySQL Server Blog

News from the MySQL Server Team

Storing UUID Values in MySQL Tables

October 14, 2015 Datatypes, MySQL, UUID Guilhem Bichot

After seeing that several blogs discuss storage of UUID values into MySQL, and that this topic is recurrent on forums, I thought I would compile some sensible ideas I have seen, and also add a couple new ones.

See also follow up post called Mysql 8.0: UUID support , which explains the 8.0 solution.

Different techniques

Say we have a table of users, and each user has a UUID. MySQL has a UUID() function, which makes MySQL generate a UUID value and return it in a human-readable form, of type VARCHAR(36). Let's try it on MySQL 5.7.8:

```
mysql> select uuid();
+--------------------------------------+
| uuid()                               |
+--------------------------------------+
| aab5d5fd-70c1-11e5-a4fb-b026b977eb28 |
+--------------------------------------+
```

So the first idea would be to simply do this:

```
create table users(id varchar(36), name varchar(200));
insert into users values(uuid(), 'Andromeda');
```

But this human-readable form of UUID isn't compact; let's observe that:

- the fours dashes are superfluous
- each pair of characters is actually a hexadecimal number in the range 00-FF; that makes 16 numbers in total (above: 0xAA, 0xB5, etc), each of them can be stored in a byte.

So we can use REPLACE() to remove dashes, and UNHEX() to transform each two-char pair into a byte:

```
create table users(id_bin binary(16), name varchar(200));
insert into users values(unhex(replace(uuid(),'-','')), 'Andromeda');
```

This binary form uses 16 bytes which is much smaller than VARCHAR(36) used by the human-readable form (which I'll call the "text" form now). If the UUID has to be a primary key, the gain is even greater, as in InnoDB the primary key value is copied into all secondary index values.

BINARY(16) is… well… just binary! No character set, no collation, just sixteen bytes. Perfect for our need.

Perhaps the text form is still necessary in some application, so let's keep it as an additional column in the table; but, to minimize disk occupation, let's make the text form a *virtual generated column* (that's a new feature of MySQL 5.7, described in the documentation of CREATE TABLE). This column will be calculated by a formula of the binary-form column: we convert the binary form back to hexadecimal digits and insert dashes.

```
create table users(
id_bin binary(16),
id_text varchar(36) generated always as
 (insert(
    insert(
      insert(
        insert(hex(id_bin),9,0,'-'),
        14,0,'-'),
      19,0,'-'),
    24,0,'-')
 ) virtual,
name varchar(200));

insert into users (id_bin,name)
  values(unhex(replace(uuid(),'-','')), 'Andromeda');

select id_text, name from users;
+--------------------------------------+-----------+
| id_text                              | name      |
+--------------------------------------+-----------+
| C2770D2E-70E6-11E5-A4FB-B026B977EB28 | Andromeda |
+--------------------------------------+-----------+
```

I didn't include id_bin in the SELECT because it would come out as cryptic characters (of ASCII code 0xC2, 0x77, etc: generally not in the human-readable range of characters). There's no reason we should need to look at the content of id_bin; but, if you do, you can visualize its hexadecimal codes by using HEX(id_bin).

Note that id_text is declared VIRTUAL, so takes no space in the table on disk.

Another benefit of making id_text a *generated* column, is that it eliminates any risk of inconsistency between the two columns. Indeed, if id_text were a plain column, one could do

```
update users set id_bin = <something>;
```

without updating id_text, accidentally. But as a generated column, id_text is never updatable directly: instead, it is automatically updated when one updates id_bin. In other words, information is in only one place (id_bin) and the database guarantees consistency.

Then, what about queries? for example, we could want to find a user by UUID:

```
select * from users where <it has UUID XYZ>;
```

Should the WHERE clause specify the binary or the text form? It depends:

- If we create an index over the binary form:

  ```
  alter table users add unique(id_bin);
  ```

  then, for this index to be used, WHERE should specify the binary form:

  ```
  WHERE id_bin = binary_form_of_XYZ
  ```

- If instead we create an index over the text form:

```
1 alter table users add unique(id_text);
```

then, WHERE should specify the text form:

```
1 WHERE id_text = text_form_of_XYZ
```

Even though id_text is a virtual column, it is possible, as above, to add an index on it (in that case, the index does occupy disk space). That is a new feature introduced in MySQL 5.7.8.

However, if we have a choice, as the binary form is shorter, it looks more logical to index it, not the text form – the index will be smaller and thus faster to traverse, faster to backup…

Finally, there is the question of how to smartly re-arrange bytes in the binary form.

To understand that, we need to learn more about UUIDs. They exist in several versions, and different sources can generate different versions. MySQL's UUID() uses version 1, which implies, as explained in paragraph 4.1.2 of the RFC, that the three leftmost dash-separated groups are a 8-byte timestamp: leftmost group is the low four bytes of the timestamp; second group is the middle two bytes, third group is the high (most significant) two bytes of the timestamp. Thus the leftmost group varies the fastest (10 times per microsecond). We can verify that:

```
1  mysql> select uuid(); do sleep(2); select uuid();
2  +--------------------------------------+
3  | uuid()                               |
4  +--------------------------------------+
5  | 3b96402f-70c5-11e5-a4fb-b026b977eb28 |
6  +--------------------------------------+
7  +--------------------------------------+
8  | uuid()                               |
9  +--------------------------------------+
10 | 3cc7f7dc-70c5-11e5-a4fb-b026b977eb28 |
11 +--------------------------------------+
```

You can see how the 8 leftmost characters changed while the others did not.

So, in a sequence of UUIDs continuously generated by a single machine, all UUIDs have different first bytes. Inserting this sequence into an indexed column (in binary or text form) will thus modify a different index page each time, preventing in-memory caching. So it makes sense to re-arrange the UUID, making the rapidly-changing parts go last, before we store into id_bin. Again, note that this idea applies only to UUIDs of version 1.

This idea isn't mine; I saw it first in this blog and that one.

Below, the binary form is re-arranged, by changing time-low/time-mid/time-high to time-high/time-mid/time-low.

```
1  create table users(id_bin binary(16), name varchar(200));
2
3  set @u = unhex(replace(uuid(),'-',''));
4
5  insert into users (id_bin,name)
6  values
7  (
8   concat(substr(@u, 7, 2), substr(@u, 5, 2),
9          substr(@u, 1, 4), substr(@u, 9, 8)),
10  'Andromeda'
```

```
11 );
```

I used a user variable above (@u), because each SUBSTR() invokation needs to reference the UUID value, but I cannot afford to write UUID() four times: it would generate a new UUID each time! So I call UUID() once, remove dashes, convert it to binary, store it in a variable, and do the four SUBSTR on it.

However, I still want the text form to be in "non-rearranged" order, because… perhaps this text form will be used for some error logging, debugging? If humans are to read it, I don't want to confuse them by using a rearranged order.
Adding id_text can be done in CREATE TABLE, or as a follow-up ALTER TABLE:

```
1  alter table users add
2  id_text varchar(36) generated always as
3  (
4   insert(
5     insert(
6       insert(
7         insert(
8           hex(
9             concat(substr(id_bin,5,4),substr(id_bin,3,2),
10                  substr(id_bin,1,2),substr(id_bin,9,8))
11          ),
12          9,0,'-'),
13      14,0,'-'),
14    19,0,'-'),
15   24,0,'-')
16 ) virtual;
```

which takes the parts out of the binary form (with SUBSTR), puts them in the "normal" position (CONCAT), converts bytes to hexadecimal digits (HEX) and inserts dashes. Right, it's a complex expression, but you type it only once, when creating the generated column.

Now look at the data:

```
d_text from users;
------------------+-----------+--------------------------------------+
                  | name      | id_text                              |
------------------+-----------+--------------------------------------+
\3A059CCBA4FBB026B977EB28 | Andromeda | 3A059CCB-70EA-11E5-A4FB-B026B977EB28 |
------------------+-----------+--------------------------------------+
```

Columns are:

1. the cryptic characters (rearranged binary UUID)
2. the corresponding hex codes for the first column
3. the name
4. the non-rearranged text UUID. See how the leftmost 3A059CCB of this text form, the rapidly-changing part, is in the middle of the binary form (column 2), so the binary form will give more efficient indexing.

Conclusion

This mere blog post has recapitulated some common good practices for storage of UUIDs, on top of making a use case for indexed virtual columns. And, it also reminded us of the power of simple string functions 🙂

Creative thinking

If we had bit-wise operations on the BINARY type, as envisioned in Morgan's blog, then they could be an alternative to CONCAT/SUBSTR. For example, this expression taken from above:

```
1 CONCAT(SUBSTR(@u, 7, 2), SUBSTR(@u, 5, 2), SUBSTR(@u, 1, 4), SUBSTR(@u, 9,
```

would become – @u being VARBINARY(16) :

```
1 # Grab bytes 7-8, move them first at 1-2
2 (@u & x'000000000000FFFF0000000000000000') << 48 |
3 # Grab bytes 5-6, move second at 3-4
4 (@u & x'00000000FFFF0000000000000000000') << 16 |
5 # Grab bytes 1-4, move third at 5-8
6 (@u & x'FFFFFFFF00000000000000000000000') >> 32 |
7 # Leave bytes 9-16 in place
8 (@u & x'0000000000000000FFFFFFFFFFFFFFFF')
9 # And all parts are united with OR.
```

The bit masks could also be generated with functions; for example x'000000000000FFFF0000000000000000' is the same thing as RPAD(LPAD(x'FFFF',8,x'00'),16,x'00')).

Between CONCAT/SUBSTR and &|<<, which one would be best… is probably a matter of taste.

Will we have bit-wise operations on BINARY one day? I am not a decider, but I hope we will have them or some equivalent. They would allow operations on "bigger than 64 bits"… stay tuned!

That's it for now. **THANK YOU** for using MySQL!

20 thoughts on "Storing UUID Values in MySQL Tables"

Pingback: Storing UUID Values in MySQL Tables | Dinesh Ram Kali.

Pingback: Storing UUID Values in MySQL Tables | Dinesh Ram Kali.

October 16, 2015 at 2:27 pm

Ike Walker

If you are using statement-based replication and want to store UUIDs in your database, you need to use a user-defined variable to store the UUID() value to assure it is replicated properly. See more details in my blog post here: http://mechanics.flite.com/blog/2014/12/04/how-to-safely-replicate-uuid-values-in-mysql/

In your examples you use a user-defined variable when re-arranging the UUID parts, but with statement-based replication you always need to use a user-defined variable even if you don't re-arrange the UUID parts.

★Guilhem Bichot

October 19, 2015 at 10:07 am

Hi Ike. Indeed. Storing in a user variable works because user variables' values are written to the statement-based binary log. Fortunately, as you show in your blog, a warning is printed when one directly uses UUID() in INSERT with statement-based replication.

Pingback: MySQL 5.7 GA Release 釋出 - Tsung's Blog

Pingback: MySQL en vrac (3) | L'Endormitoire

Sam Pinizzotto

May 12, 2016 at 1:46 pm

If I want to use the id as a querystring in a hyperlink to pass to the query, would I have to have it in the table or can i still use the virtualized field?

Do I have to keep converting the value if I need to pass it as a querystring?

Thanks,
Sam

★Guilhem Bichot

May 20, 2016 at 8:29 am

It doesn't matter if you have it in a stored column or a virtual one, MySQL can read both.
Conversion from text to binary(16) reduces storage and improves indexing, but for exchanging with clients it isn't so important.

Pingback: UUID con MySql InnoDB | Clevercloud

Pingback: mysql create table - Mihan وب هاست

Pingback: Mysql 8.0: UUID support | MySQL Server Blog

Pingback: Mysql 8.0: UUID support – Cloud Data Architect

Pingback: mysql insert - Mihan وب هاست

Pingback: Mysql 8.0: UUID support | Database Administrator

Pingback: UUID or GUID as Primary Keys? Be Careful! – Tom Harrison Jr | Artificia Intelligence

Pingback: Error Fetching Database Uuid For Backup Dictionary

Pingback: UUID storage in MySQL – Triple-networks

Pingback: Optimised UUIDs in mysql - Abdallah Samy

Pingback: Cómo optimizar el uso de UUIDs en MySQL – Neoguias

Pingback: Can I use a function for a default value in MySql? - IZZIDB