All you need to know about MySQL InnoDB Primary Keys

Frédéric Descamps
Community Manager
Oracle MySQL
preFOSDEM MySQL Belgian Days - January 2025

Who am I ?

about.me/lefred

Frédéric Descamps

- 𝕏 @lefred
- @lefredbe.bsky.social
- @lefred@fosstodon.org
- MySQL Evangelist
- using MySQL since version 3.20
- devops believer
- living in 🇧🇪
- https://lefred.be

MySQL InnoDB Clustered Index

Concept

Clustered Index

Each InnoDB table has a special index called the clustered index that stores row data. Typically, the clustered index is synonymous with the primary key.

Clustered Index

Each InnoDB table has a special index called the clustered index that stores row data. Typically, the clustered index is synonymous with the primary key.

Let's check now an example on how we usually mentally represent a table and an index.

Mental representation of a table and an index

Table 1

Index

Indexed Column	column 1	column 2	column x	column n

Mental representation of a table and an index

Table 1

Index
5001

Indexed Column	column 1	column 2	column x	column n
5001	a	b	2022-03-14	NULL

Mental representation of a table and an index

Table 1

Index
3
5001

Indexed Column	column 1	column 2	column x	column n
5001	a	b	2022-03-14	NULL
3	c	a	2022-03-15	NULL

Mental representation of a table and an index

Table 1

Index
3
6
5001

Indexed Column	column 1	column 2	column x	column n
5001	a	b	2022-03-14	NULL
3	c	a	2022-03-15	NULL
6	f	be	2022-03-15	1

Mental representation of a table and an index

Table 1

Index
3
6
27
5001

Indexed Column	column 1	column 2	column x	column n
5001	a	b	2022-03-14	NULL
3	c	a	2022-03-15	NULL
6	f	be	2022-03-15	1
27	d	it	2022-03-16	101

Mental representation of a table and an index

Table 1

Index
3
6
12
27
5001

Indexed Column	column 1	column 2	column x	column n
5001	a	b	2022-03-14	NULL
3	c	a	2022-03-15	NULL
6	f	be	2022-03-15	1
27	d	it	2022-03-16	101
12	e	uk	2022-03-22	NULL

Mental representation of a table and an index

Table 1

Index
3
6
12
27
5001

Indexed Column	column 1	column 2	column x	column n
5001	a	b	2022-03-14	NULL
3	c	a	2022-03-15	NULL
6	f	be	2022-03-15	1
27	d	it	2022-03-16	101
12	e	uk	2022-03-22	NULL

InnoDB representation of a table and PK

Table 1

```
insert into table1 values
(5001,'a','b','2022-03-14', NULL);
```

PRIMARY KEY	column 1	column 2	column x	column n
5001	a	b	2022-03-14	NULL

InnoDB representation of a table and PK

Table 1

```
insert into table1 values
(3,'c','a','2022-03-15', NULL);
```

PRIMARY KEY	column 1	column 2	column x	column n
3	c	a	2022-03-15	NULL
5001	a	b	2022-03-14	NULL

InnoDB representation of a table and PK

Table 1

```
insert into table1 values
(6,'f','be','2022-03-15', 1);
```

PRIMARY KEY	column 1	column 2	column x	column n
3	c	a	2022-03-15	NULL
6	f	be	2022-03-15	1
5001	a	b	2022-03-14	NULL

InnoDB representation of a table and PK

Table 1

```
insert into table1 values
(27,'d','it','2022-03-16', 101);
```

PRIMARY KEY	column 1	column 2	column x	column n
3	c	a	2022-03-15	NULL
6	f	be	2022-03-15	1
27	d	it	2022-03-16	101
5001	a	b	2022-03-14	NULL

InnoDB representation of a table and PK

Table 1

```
insert into table1 values
(12,'e','uk','2022-03-22', NULL);
```

PRIMARY KEY	column 1	column 2	column x	column n
3	c	a	2022-03-15	NULL
6	f	be	2022-03-15	1
12	e	uk	2022-03-22	NULL
27	d	it	2022-03-16	101
5001	a	b	2022-03-14	NULL

InnoDB representation of a table and PK

Table 1

PRIMARY KEY	column 1	column 2	column x	column n
3	c	a	2022-03-15	NULL
6	f	be	2022-03-15	1
12	e	uk	2022-03-22	NULL
27	d	it	2022-03-16	101
5001	a	b	2022-03-14	NULL

InnoDB representation of a table and PK

Table 1

PRIMARY KEY	column 1	column 2	column x	column n
3	c	a	2022-03-15	NULL
6	f	be	2022-03-15	1
12	e	uk	2022-03-22	NULL
27	d	it	2022-03-16	101
5001	a	b	2022-03-14	NULL

This is the clustered index representation: stored by order of Primary Key

MySQL InnoDB Primary Keys

They are mandatory !

InnoDB Primary Key

InnoDB stores data in table spaces.

And so far, we know that records are stored and sorted using the clustered index.

- *The Primary Key is a key for the index that uniquely defined for a row, should be immutable.*
- *InnoDB needs a PRIMARY KEY*
- *No* `NULL` *values are allowed*
- *Monotonically increasing*
  - *use* `UUID_TO_BIN()` *if you must use UUIDs, otherwise avoid them*

InnoDB Primary Key (2)

What we don't know is that all secondary indexes also contain the primary key as the right-most column in the index (even if this is not exposed). That means when a secondary index is used to retrieve a record, two indexes are used: first the secondary one pointing to the primary key that will be used to finally retrieve the record.

InnoDB Primary Key (2)

What we don't know is that all secondary indexes also contain the primary key as the right-most column in the index (even if this is not exposed). That means when a secondary index is used to retrieve a record, two indexes are used: first the secondary one pointing to the primary key that will be used to finally retrieve the record.

When no primary key is defined, the first unique not null key is used. And if none is available, InnoDB will do something we will cover in the next chapter.

InnoDB Primary Key (2)

What we don't know is that all secondary indexes also contain the primary key as the right-most column in the index (even if this is not exposed). That means when a secondary index is used to retrieve a record, two indexes are used: first the secondary one pointing to the primary key that will be used to finally retrieve the record.

When no primary key is defined, the first unique not null key is used. And if none is available, InnoDB will do something we will cover in the next chapter.

If you use HA solutions, Primary Keys are mandatory !

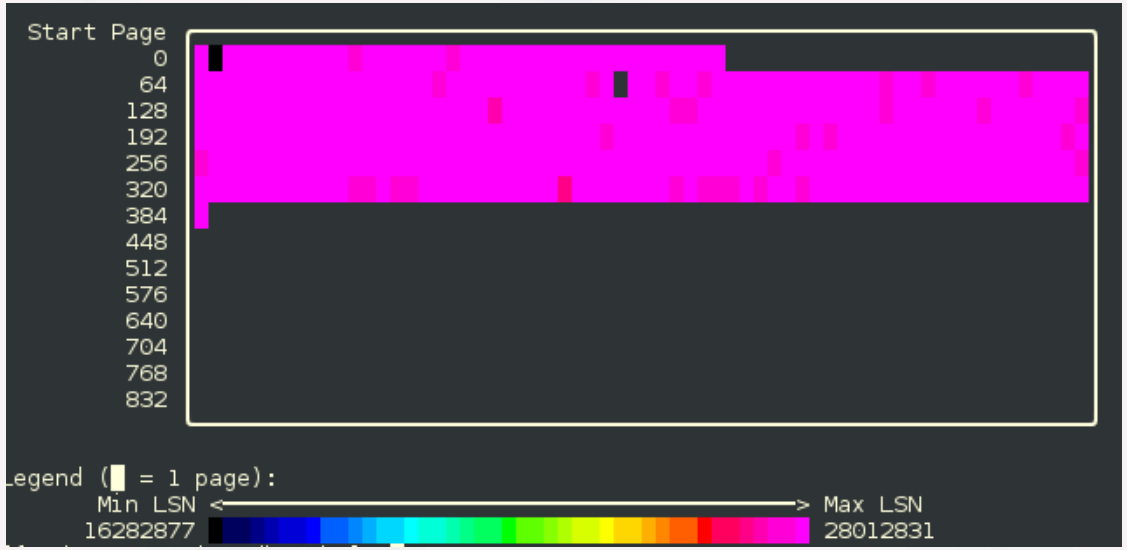
InnoDB Primary Key (3)

Primary Keys impact how the values are inserted and the size of the secondary indexes. A non sequential PK can lead to many random IOPS.

InnoDB Primary Key (3)

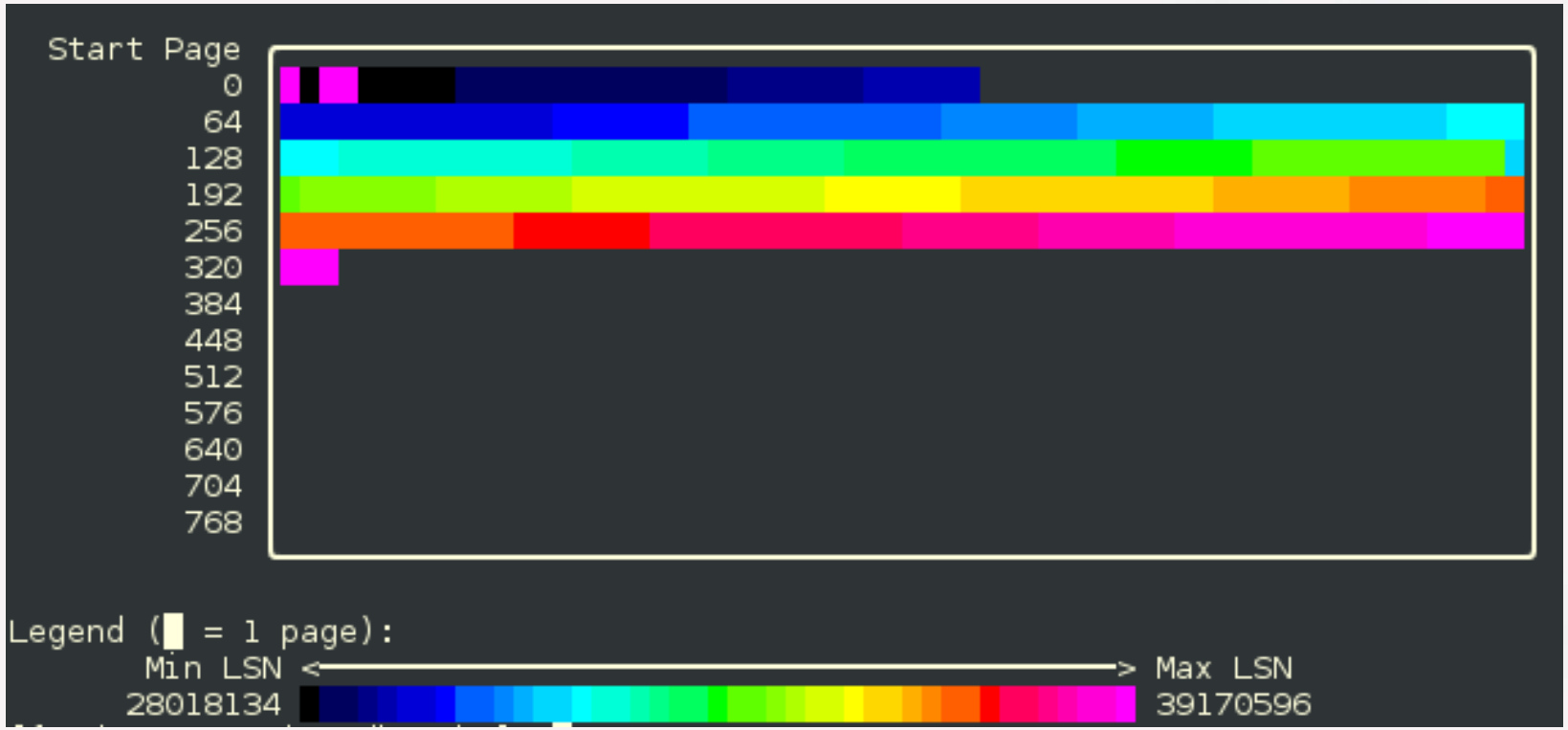
Primary Keys impact how the values are inserted and the size of the secondary indexes. A non sequential PK can lead to many random IOPS.

Also, it's more and more common to use applications that generate complete random primary keys...that means if the Primary Key is not sequential, InnoDB will have to heavily re-balance all the pages on inserts.



InnoDB Primary Key (4)

If we compare the same load (inserts) when using an auto_increment integer as Primary Key, we can see that only the latest pages are recently touched:



Generated with https://github.com/jeremycole/innodb_ruby from @jeremycole

InnoDB Primary Key (5)

It's possible to return an error if a table is created without defining any Primary Key: `sql_require_primary_key`.

```
SQL> SET sql_require_primary_key=1 ;

SQL> CREATE TABLE nopk2 (i int not null, name varchar(20));
ERROR: 3750 (HY000): Unable to create or change a table without a primary key,
when the system variable 'sql_require_primary_key' is set.
Add a primary key to the table or unset this variable to avoid this message.
Note that tables without a primary key can cause performance problems in row-based
replication, so please consult your DBA before changing this setting.
```

GEN_CLUST_INDEX

The Ennemy

GEN_CLUST_INDEX

As we saw, when no primary key is defined, the first unique not null key is used. And if none is available, InnoDB creates an hidden primary key (6 bytes).

GEN_CLUST_INDEX

As we saw, when no primary key is defined, the first unique not null key is used. And if none is available, InnoDB creates an hidden primary key (6 bytes).

This hidden generated key is called `GEN_CLUST_INDEX`.

GEN_CLUST_INDEX

As we saw, when no primary key is defined, the first unique not null key is used. And if none is available, InnoDB creates an hidden primary key (6 bytes).

This hidden generated key is called `GEN_CLUST_INDEX`.

The problem with such key is that you don't have any control on it and worse, this value is global to all tables without primary keys and can be a contention problem if you perform multiple simultaneous writes on such tables (`dict_sys->mutex`*).*

GEN_CLUST_INDEX - things you don't know

- *stored on 6 bytes in memory as unsigned:*
  - `0 to 281,474,976,710,655`
- *in memory this is stored as* `dict_sys->row_id`
- *on disk it's stored on* `ibdata1` *at offset* `7 * page_size + 38` *as 8 bytes*
- *InnoDB writes the value on disk only once every* ***256*** *updates*
- *at restart InnoDB knows that the value should be in the range of* `[current disk value + 256]` *so the new value becomes* `current disk value + 256`.

GEN_CLUST_INDEX

The `GEN_CLUST_INDEX` *set a* `ROW_ID` *to each records in a table without Primary Key (or unique not null key).*

The value is global and we can access it from memory.

Let's create a table without Primary Key:

```
SQL> create table nopk (name varchar(20), age int unsigned);
```

```
./gen_clust_id.py /home/fred/mysql-sandboxes/3310/sandboxdata/ibdata1
b'\x00\x00\x00\x00\x00\x00\x01\x00'
GEN_CLUST_INDEX value: 256
```

GEN_CLUST_INDEX

Let's add a record to the table now:

```
SQL> insert into nopk values ('aaaa', 99);
```

GEN_CLUST_INDEX

Let's add a record to the table now:

```
SQL> insert into nopk values ('aaaa', 99);
```

Now let's have a look what is the `ROW_ID` *value in memory:*

```
$ gdb -p 2315547 -ex "set pagination off" -ex 'thread 1' -ex 'p dict_sys->row_id' -batch
[Thread debugging using libthread_db enabled]
[...]
$1 = 513
```

GEN_CLUST_INDEX

The value on disk was `256`*, as nothing was on memory, when we added the new record, the* `ROW_ID` *moved to* `512 (256+256)`*.*

So the next record will have `513` *as* `ROW_ID`*.*

GEN_CLUST_INDEX

The value on disk was `256`*, as nothing was on memory, when we added the new record, the* `ROW_ID` *moved to* `512 (256+256)`*.*

So the next record will have `513` *as* `ROW_ID`*.*

```
SQL> insert into nopk values ('bbbb', 11);
```

```
$ gdb -p 2315547 -ex "set pagination off" -ex 'thread 1' -ex 'p dict_sys->row_id' -batch
[Thread debugging using libthread_db enabled]
[...]
$1 = 514
```

GEN_CLUST_INDEX

What is the value on of `GEN_CLUST_INDEX` *on disk now ?*

A. 256

B. 512

C. 514

GEN_CLUST_INDEX

What is the value on of `GEN_CLUST_INDEX` *on disk now ?*

A. 256

B. 512

C. 514

GEN_CLUST_INDEX

What is the value on of `GEN_CLUST_INDEX` *on disk now ?*

A. 256

B. 512

C. 514

```
./gen_clust_id.py /home/fred/mysql-sandboxes/3310/sandboxdata/ibdata1
b'\x00\x00\x00\x00\x00\x00\x02\x00'
GEN_CLUST_INDEX value: 512
```

GEN_CLUST_INDEX

Now let's restart MySQL:

```
SQL> restart;
```

And what is the value now of `GEN_CLUST_INDEX` *on disk ater the restart?*

A. 512

B. 514

C. 768

GEN_CLUST_INDEX

Now let's restart MySQL:

```
SQL> restart;
```

And what is the value now of `GEN_CLUST_INDEX` *on disk ater the restart?*

A. 512

B. 514

C. 768

GEN_CLUST_INDEX

```
./gen_clust_id.py /home/fred/mysql-sandboxes/3310/sandboxdata/ibdata1
b'\x00\x00\x00\x00\x00\x00\x03\x00'
GEN_CLUST_INDEX value: 768
```

GEN_CLUST_INDEX

```
./gen_clust_id.py /home/fred/mysql-sandboxes/3310/sandboxdata/ibdata1
b'\x00\x00\x00\x00\x00\x00\x03\x00'
GEN_CLUST_INDEX value: 768
```

Now let's insert another record:

```
SQL> insert into nopk values ('cccc', 22);
```

GEN_CLUST_INDEX

So we have 3 records in our table:

```
SQL> select * from nopk;
+------+-----+
| name | age |  ROW_ID:      HEX:
+------+-----+
| aaaa |  99 |       512     02 00
| bbbb |  11 |       513     02 01
| cccc |  22 |       768     03 00
+------+-----+
```

GEN_CLUST_INDEX

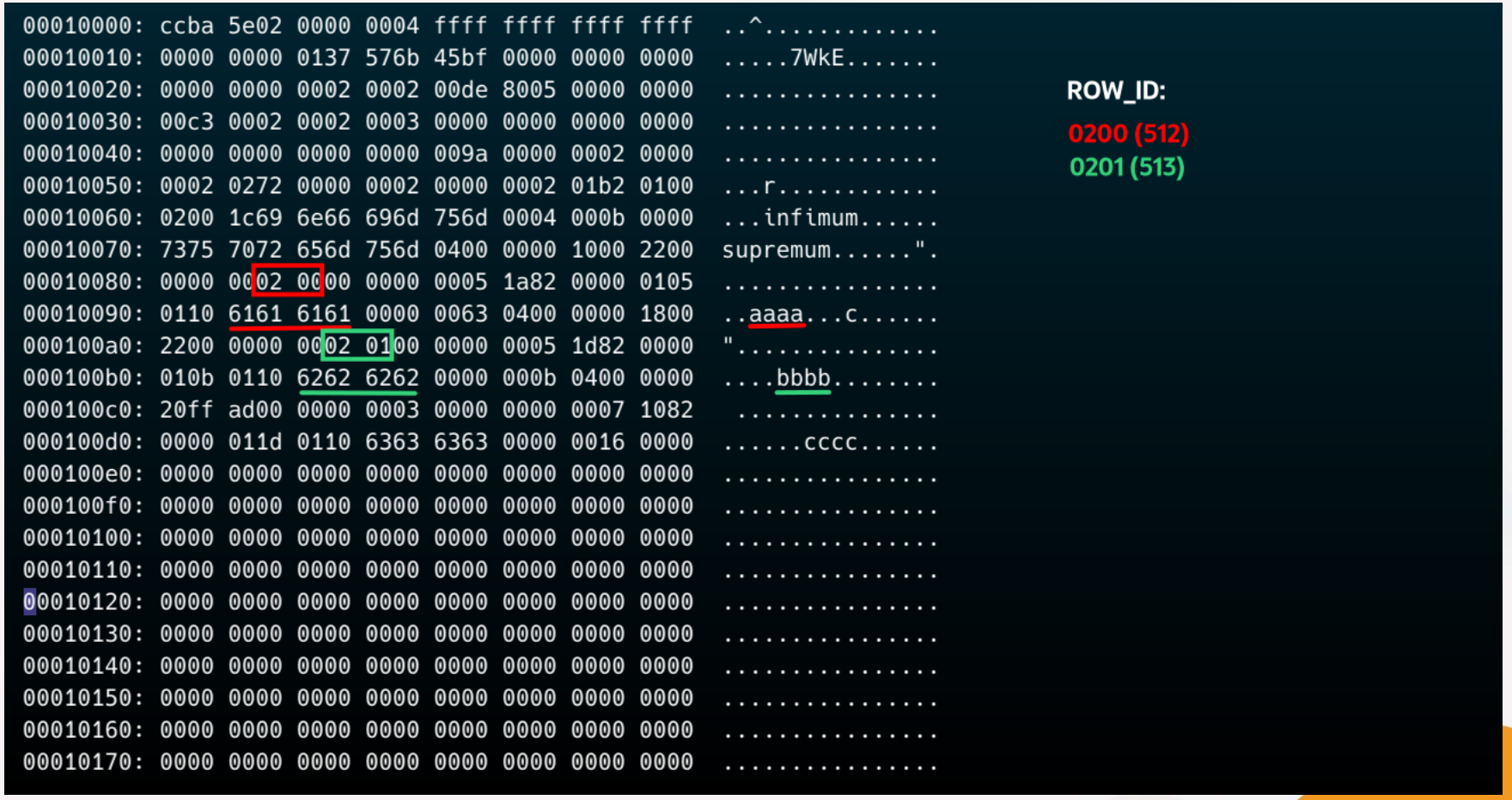
Let's have a look at the InnoDB page:

```
00010000: ccba 5e02 0000 0004 ffff ffff ffff ffff  ..^.............
00010010: 0000 0000 0137 576b 45bf 0000 0000 0000  .....7WkE.......
00010020: 0000 0000 0002 0002 00de 8005 0000 0000  ................
00010030: 00c3 0002 0002 0003 0000 0000 0000 0000  ................
00010040: 0000 0000 0000 0000 009a 0000 0002 0000  ................
00010050: 0002 0272 0000 0002 0000 0002 01b2 0100  ...r............
00010060: 0200 1c69 6e66 696d 756d 0004 000b 0000  ...infimum......
00010070: 7375 7072 656d 756d 0400 0000 1000 2200  supremum......".
00010080: 0000 0002 0000 0000 0005 1a82 0000 0105  ................
00010090: 0110 6161 6161 0000 0063 0400 0000 1800  ..aaaa...c......
000100a0: 2200 0000 0002 0100 0000 0005 1d82 0000  "...............
000100b0: 010b 0110 6262 6262 0000 000b 0400 0000  ....bbbb........
000100c0: 20ff ad00 0000 0003 0000 0000 0007 1082   ...............
000100d0: 0000 011d 0110 6363 6363 0000 0016 0000  ......cccc......
000100e0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000100f0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00010100: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00010110: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00010120: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00010130: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00010140: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00010150: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00010160: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00010170: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

ROW_ID:
0200 (512)

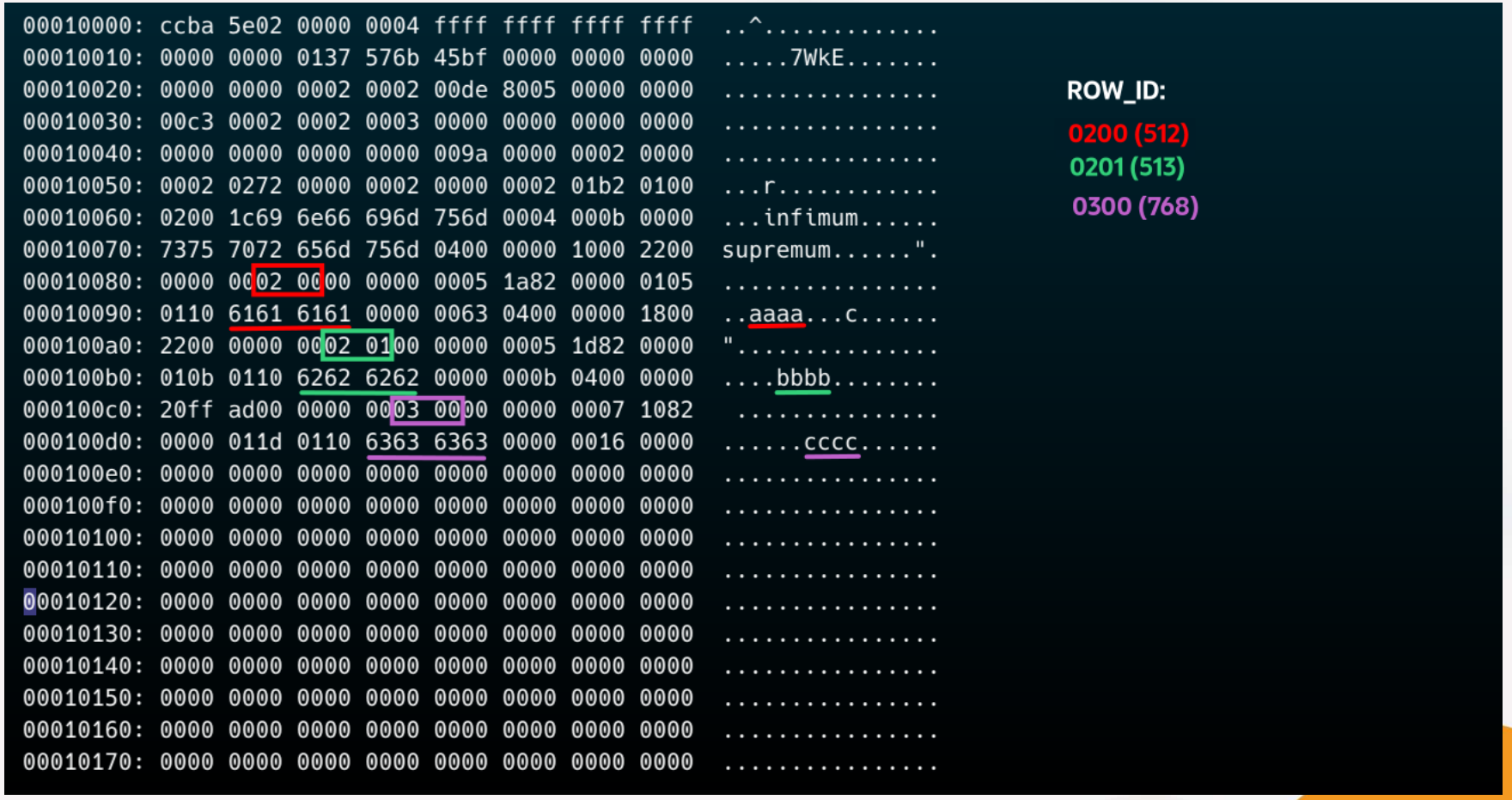
GEN_CLUST_INDEX

Let's have a look at the InnoDB page:



GEN_CLUST_INDEX

Let's have a look at the InnoDB page:



GEN_CLUST_INDEX - concurrency

I said that `GEN_CLUST_INDEX` *(*`dict_sys->row_id`*) was global to all tables without a Primary Key.*

So if we create new table without a Primary Key and we insert a record, it should have the value of `dict_sys->row_id`*.*

GEN_CLUST_INDEX - concurrency

I said that `GEN_CLUST_INDEX` *(*`dict_sys->row_id`*) was global to all tables without a Primary Key.*

So if we create new table without a Primary Key and we insert a record, it should have the value of `dict_sys->row_id`*.*

```
SQL> create table nopk2 (name varchar(20), age int unsigned);

SQL> insert into nopk2 values ('dddd', 1);
```

GEN_CLUST_INDEX - concurrency

Let's verify:

```
$ gdb -p 2315547 -ex "set pagination off" -ex 'thread 1' -ex 'p dict_sys->row_id' -batch
[Thread debugging using libthread_db enabled]
[...]
$1 = 770
```

GEN_CLUST_INDEX - concurrency

Let's verify:

```
$ gdb -p 2315547 -ex "set pagination off" -ex 'thread 1' -ex 'p dict_sys->row_id' -batch
[Thread debugging using libthread_db enabled]
[...]
$1 = 770
```

Let's get the hexadecimal representation of the previous value:

```
$ python3 -c "print(format(769, '04x'))"
0301
```

GEN_CLUST_INDEX - concurrency

Let's have a look at the `ROW_ID` *in the table:*

```
0000ffe0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000fff0: 0000 0000 0070 0063 763e 3547 0138 cd84  .....p.cv>5G.8..
00010000: fa3e 2144 0000 0004 ffff ffff ffff ffff  .>!D............
00010010: 0000 0000 0138 d314 45bf 0000 0000 0000  .....8..E.......
00010020: 0000 0000 0003 0002 009a 8003 0000 0000  ................
00010030: 007f 0005 0000 0001 0000 0000 0000 0000  ................
00010040: 0000 0000 0000 0000 009d 0000 0003 0000  ................
00010050: 0002 0272 0000 0003 0000 0002 01b2 0100  ...r............
00010060: 0200 1c69 6e66 696d 756d 0002 000b 0000  ...infimum......
00010070: 7375 7072 656d 756d 0400 0000 10ff f100  supremum........
00010080: 0000 0003 0100 0000 0007 2981 0000 010f  ..........).....
00010090: 0110 6464 6464 0000 0001 0000 0000 0000  ..dddd..........
000100a0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000100b0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000100c0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000100d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000100e0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000100f0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00010100: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00010110: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00010120: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00010130: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00010140: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00010150: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

ROW_ID:
0301 (769)

GEN_CLUST_INDEX

This is how to find all the tables using the `GEN_CLUST_INDEX`*:*

```
SQL> select i.table_id, t.name
     from information_schema.innodb_indexes i
     join information_schema.innodb_tables t on (i.table_id = t.table_id)
     where i.name='GEN_CLUST_INDEX';
```

MySQL InnoDB Primary Keys

Keep them short

Size matters

*So the data is actually stored in an index: the **clustered index**.*

Size matters

*So the data is actually stored in an index: the **clustered index**.*

*Secondary indexes are a copy of some parts of the data, sorted... **but with a pointer to the record on the table.***

Size matters

So the data is actually stored in an index: the ***clustered index****.*

Secondary indexes are a copy of some parts of the data, sorted… ***but with a pointer to the record on the table.***

That pointer is the ***Primary Key*** *!*

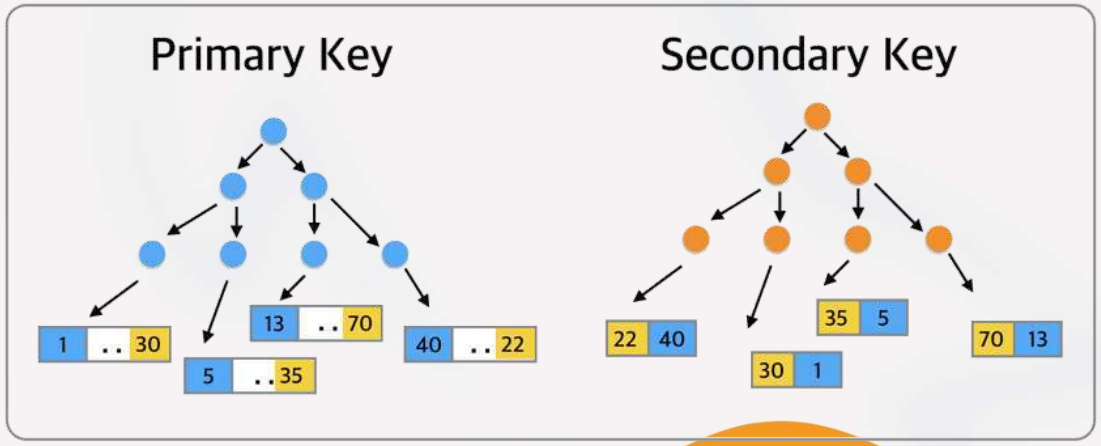
Size matters

*So the data is actually stored in an index: the **clustered index**.*

*Secondary indexes are a copy of some parts of the data, sorted... **but with a pointer to the record on the table.***

*That pointer is the **Primary Key** !*

*This is why we always say that the **right most column** of a secondary index is the Primary Key and it's infact really included in it.*



Size matters - example

Let's take a look at these two similar tables, both having one Primary Key and one Secondary Key:

```
CREATE TABLE `sbtest1` (
  `id` char(200) NOT NULL,
  `k` int NOT NULL DEFAULT '0',
  `c` char(120) NOT NULL DEFAULT '',
  `pad` char(60) NOT NULL DEFAULT '',
  PRIMARY KEY (`id`),
  KEY `k_1` (`k`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
  COLLATE=utf8mb4_0900_ai_ci
```

```
CREATE TABLE `sbtest2` (
  `id` int NOT NULL AUTO_INCREMENT,
  `k` int NOT NULL DEFAULT '0',
  `c` char(120) NOT NULL DEFAULT '',
  `pad` char(60) NOT NULL DEFAULT '',
  PRIMARY KEY (`id`),
  KEY `k_2` (`k`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
  COLLATE=utf8mb4_0900_ai_ci
```

Size matters - example (2)

Let's have a look at the table space's size on disk:

```
[root@dynabook sbtest]# ls -lh
total 1017M
-rw-r----- 1 mysql mysql 744M Feb  7 09:36 sbtest1.ibd
-rw-r----- 1 mysql mysql 272M Feb  7 09:38 sbtest2.ibd
```

Size matters - example (2)

Let's have a look at the table space's size on disk:

```
[root@dynabook sbtest]# ls -lh
total 1017M
-rw-r----- 1 mysql mysql 744M Feb  7 09:36 sbtest1.ibd
-rw-r----- 1 mysql mysql 272M Feb  7 09:38 sbtest2.ibd
```

Do you think the size difference can be just related to the Primary Key difference and not impacting the secondary index ?

Size matters - example (3)

Let's have a look:

```
SELECT NAME, TABLE_ROWS, format_bytes(data_length) DATA_SIZE,
       format_bytes(index_length) INDEX_SIZE,
       format_bytes(data_length+index_length) TOTAL_SIZE,
       format_bytes(data_free) DATA_FREE, format_bytes(FILE_SIZE) FILE_SIZE,
       format_bytes((FILE_SIZE/10 - (data_length/10 + index_length/10))*10) WASTED_SIZE
FROM information_schema.TABLES as t
JOIN information_schema.INNODB_TABLESPACES as it
  ON it.name = concat(table_schema,"/",table_name)
WHERE name like 'sbtest/%' ORDER BY (data_length + index_length) desc limit 5;
+---------------+------------+------------+------------+------------+-----------+------------+-------------+
| NAME          | TABLE_ROWS | DATA_SIZE  | INDEX_SIZE | TOTAL_SIZE | DATA_FREE | FILE_SIZE  | WASTED_SIZE |
+---------------+------------+------------+------------+------------+-----------+------------+-------------+
| sbtest/sbtest1 |     973080 | 491.00 MiB | 239.00 MiB | 730.00 MiB | 3.00 MiB  | 744.00 MiB | 14.00 MiB   |
| sbtest/sbtest2 |     986328 | 245.72 MiB | 15.52 MiB  | 261.23 MiB | 5.00 MiB  | 272.00 MiB | 10.77 MiB   |
+---------------+------------+------------+------------+------------+-----------+------------+-------------+
```

MySQL InnoDB Primary Keys

More on Secondary Index & Primary Key

Secondary Index & Primary Key

We saw that every Secondary Index includes the primary index at the right-most column (hidden).

But this is not always the case... Let's check this example:

Secondary Index & Primary Key

We saw that every Secondary Index includes the primary index at the right-most column (hidden).

But this is not always the case... Let's check this example:

```
CREATE TABLE `t1` (
`a` int NOT NULL,
`b` int NOT NULL,
`c` int NOT NULL,
`d` int NOT NULL,
`e` int NOT NULL,
`f` varchar(10) DEFAULT 'aaa',
`inserted` datetime DEFAULT CURRENT_TIMESTAMP,
PRIMARY KEY(`a`,`b`,`c`) ) ENGINE=InnoDB;
```

Secondary Index & Primary Key (2)

The table contains only the following two records:

```
SELECT * FROM t1;
+---+---+---+----+----+-----+---------------------+
| a | b | c | d  | e  | f   | inserted            |
+---+---+---+----+----+-----+---------------------+
| 1 | 2 | 3 |  4 |  5 | abc | 2024-02-11 17:37:16 |
| 7 | 8 | 9 | 10 | 11 | def | 2024-02-11 17:37:26 |
+---+---+---+----+----+-----+---------------------+
```

Secondary Index & Primary Key (3)

*This is an illutration of the table, mind the **bold columns** that are part of the Primary Key:*

t1 `PRIMARY KEY (`a`, `b`, `c`)`

a	**b**	**c**	d	e	f	inserted
1	**2**	**3**	4	5	abc	2024-02-11 17:37:16
7	**8**	**9**	10	11	def	2024-02-11 17:37:26

Secondary Index & Primary Key (4)

Now let's create a secondary index for the column `f`*:*

```
ALTER TABLE t1 ADD INDEX f_idx(f);
```

This key will then include the Primary Key as the right-most column(s) on the secondary index.

Secondary Index & Primary Key (5)

This is the representation of that secondary index's entries:

t1 `PRIMARY KEY (`a`, `b`, `c`)`

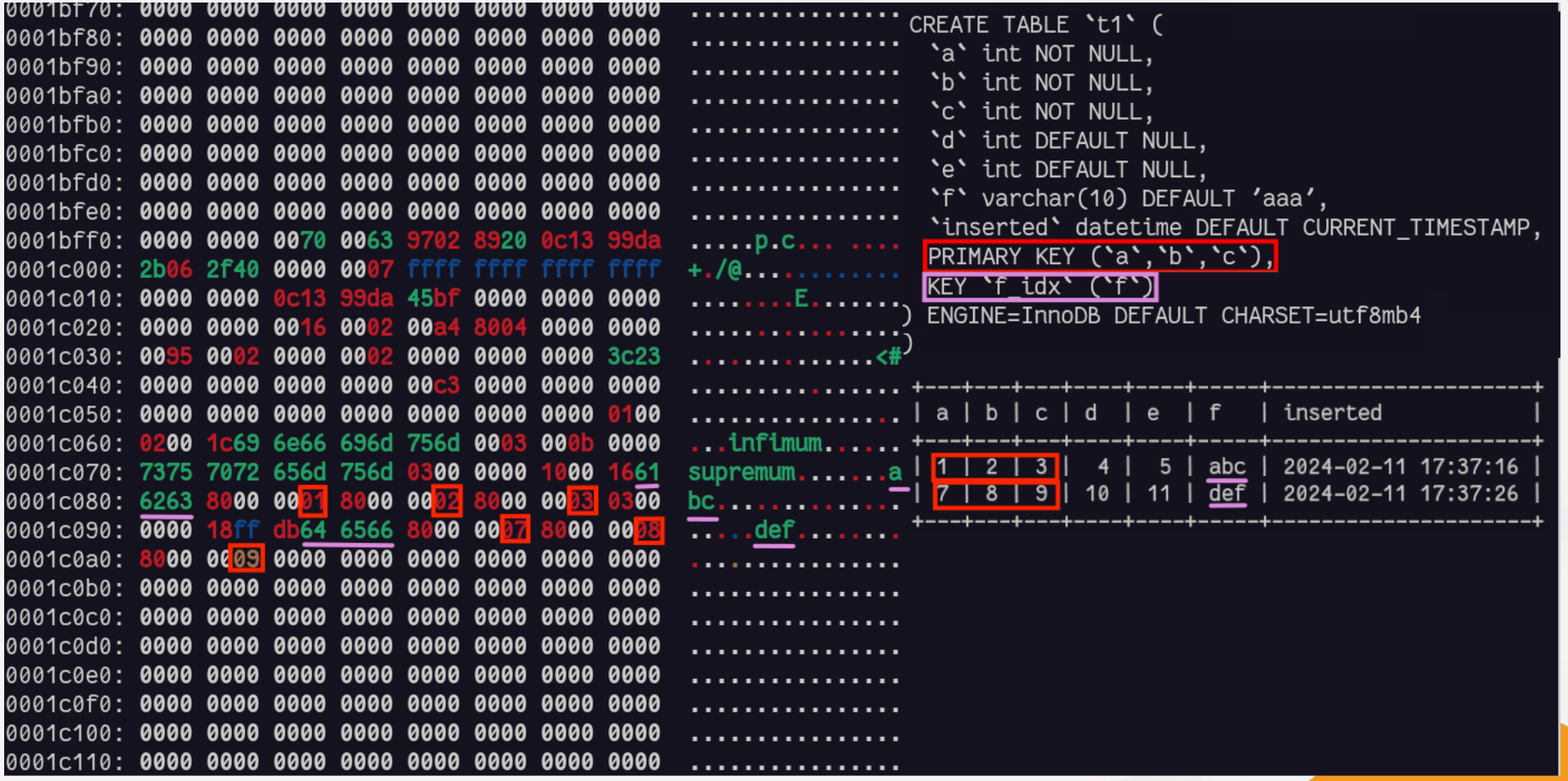
a	b	c	d	e	f	inserted
1	2	3	4	5	abc	2024-02-11 17:37:16
7	8	9	10	11	def	2024-02-11 17:37:26

`KEY `f_idx` (`f`)`

abc	1	2	3
def	7	8	9

Secondary Index & Primary Key (6)

Let's verify this on the InnoDB page for that index:



Secondary Index & Primary Key (7)

This is exaclty what we already knew about Secondary Indexes and the Primary Key.... but when we have a Primary Key or Part of a Primary Key ***included*** *in a Secondary Index, only the eventual missing columns of the Primary Key will be added as right-most and hidden entries to the secondary index.*

Let's create a secondary index where the column **b** *will be missing:*

```
ALTER TABLE t1 ADD INDEX sec_idx (`d`,`c`,`e`,`a`);
```

Secondary Index & Primary Key (8)

This is the illustration of such secondary index:

t1 `PRIMARY KEY (`a`, `b`, `c`)`

a	b	c	d	e	f	inserted
1	2	3	4	5	abc	2024-02-11 17:37:16
7	8	9	10	11	def	2024-02-11 17:37:26

`KEY `sec_idx` (`d`, `c`, `e`, `a`)`

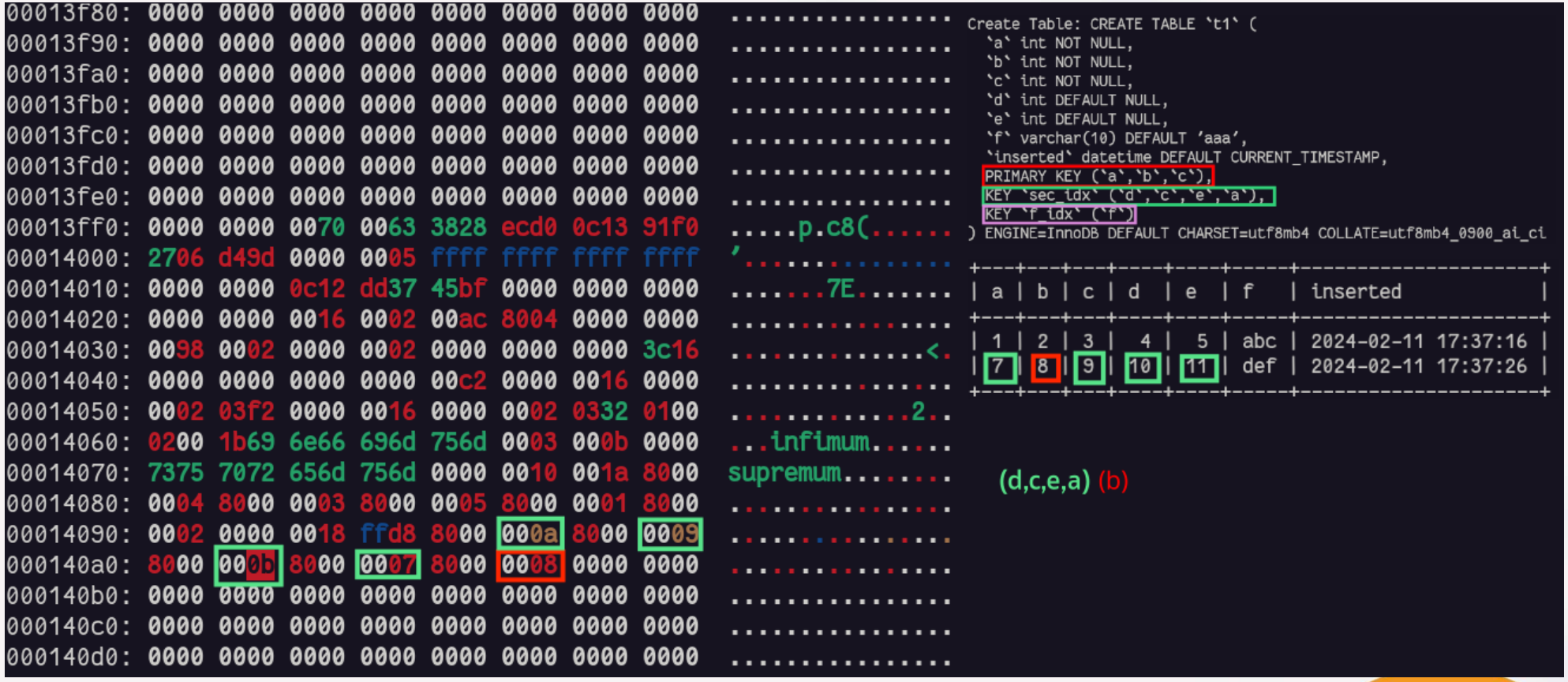
4	3	5	1	2
10	9	11	7	8

Secondary Index & Primary Key (9)

We saw that only the missing column `b` *is added as right-most hidden column of the secondary index:*

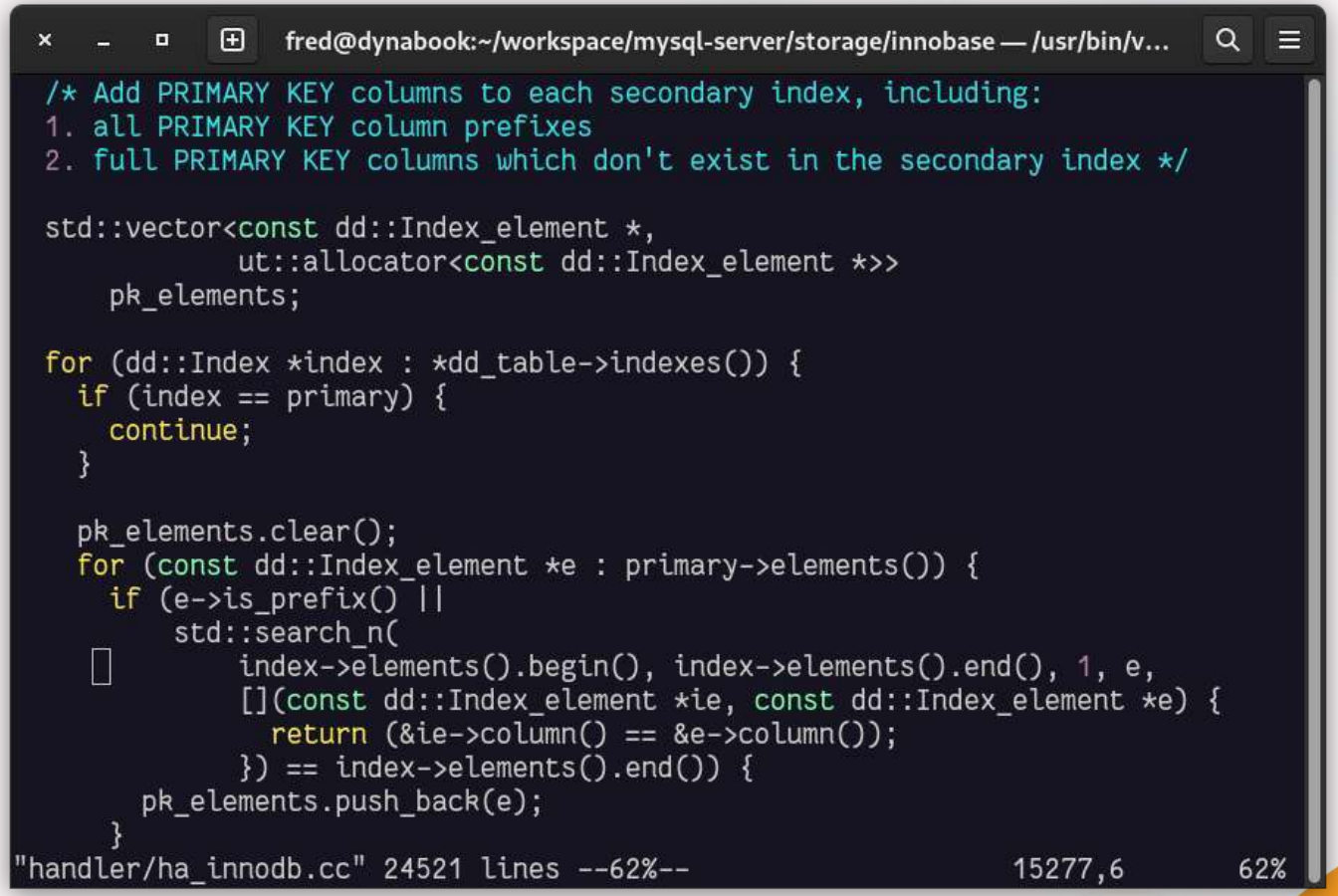
Secondary Index & Primary Key (10)

Same for the second entry of the index:



Secondary Index & Primary Key (11)

This is documented in InnoDB's' source code:



Even Further with Secondary Index & Primary Key

But what will happen, if we just use a prefix part of the Primary Key in the secondary index?

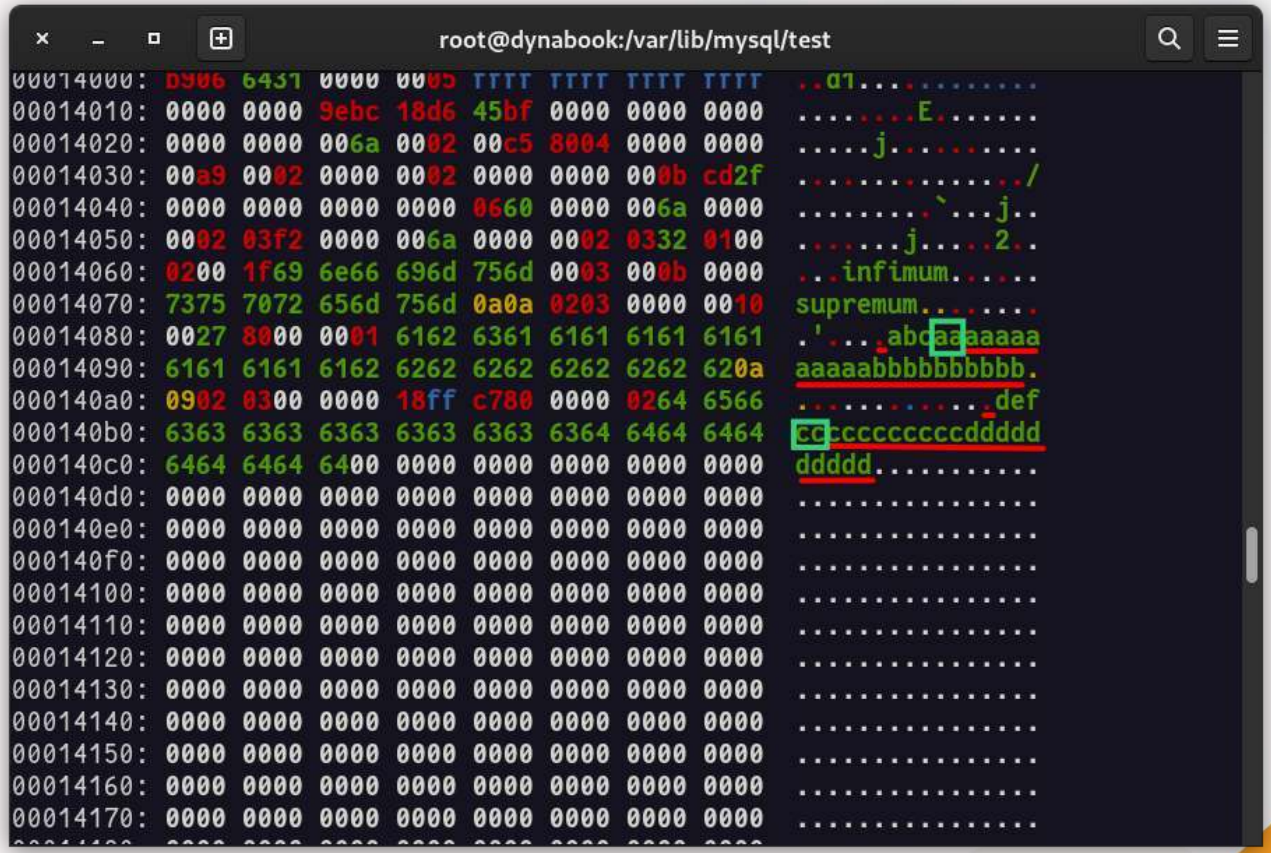
```
CREATE TABLE `t1` (
  `a` varchar(10) NOT NULL DEFAULT 'aaaaaaaaaa',
  `b` varchar(10) NOT NULL DEFAULT 'bbbbbbbbbb',
  `c` int NOT NULL DEFAULT '1',
  `f` varchar(10) DEFAULT NULL,
  PRIMARY KEY (`a`,`b`,`c`),
  KEY `sec_idx` (`c`,`f`,`a`(2))
) ENGINE=InnoDB

SELECT * FROM t1;
+------------+------------+---+-----+
| a          | b          | c | f   |
+------------+------------+---+-----+
| aaaaaaaaaa | bbbbbbbbbb | 1 | abc |
| ccccccccc  | ddddddddd  | 2 | def |
+------------+------------+---+-----+
```

*We can see that **only 2 characters** of the column* **a** *are used in the secondary index.*

Even Further with Secondary Index & Primary Key

If we check in the InnoDB page, we can notice that in-fact, the full column will also be added as the right-most hidden part of the secondary index:



MySQL InnoDB Primary Keys

But...

InnoDB Primary Key (5)

My legacy application didn't define any Primary Key, adding one (auto_increment) breaks the application ! What can I do ?

InnoDB Primary Key (5)

My legacy application didn't define any Primary Key, adding one (auto_increment) breaks the application ! What can I do ?

Easy, just create a new invisible column and define it as Primary Key !

Invisible column as Primary Key

```
select * from actors;
+-----------------+-----+
| name            | age |
+-----------------+-----+
| Al Pacino       |  80 |
| Robert De Niro  |  77 |
| Joe Pesci       |  78 |
| Sharon Stone    |  63 |
| Diane Keaton    |  75 |
| Talia Shire     |  74 |
+-----------------+-----+
```

Invisible column as Primary Key

```
select * from actors;
+-----------------+-----+
| name            | age |
+-----------------+-----+
| Al Pacino       |  80 |
| Robert De Niro |  77 |
| Joe Pesci       |  78 |
| Sharon Stone    |  63 |
| Diane Keaton    |  75 |
| Talia Shire     |  74 |
+-----------------+-----+
```

Do we have a Primary Key ?

Invisible column as Primary Key (2)

Let's find out by listing all tables where the clustered index was generated (internal hidden key):

```
select i.table_id, t.name
from information_schema.innodb_indexes i
join information_schema.innodb_tables t on (i.table_id = t.table_id)
where i.name='GEN_CLUST_INDEX';
+----------+------------------+
| table_id | name             |
+----------+------------------+
|     1293 | hollywood/actors |
+----------+------------------+
1 row in set (0.0211 sec)
```

Invisible column as Primary Key (3)

We can verify with the table's definition:

```
show create table actors\G
 *************************** 1. row ***************************
       Table: actors
Create Table: CREATE TABLE `actors` (
  `name` varchar(20) DEFAULT NULL,
  `age` int unsigned DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

Invisible column as Primary Key (3)

We can verify with the table's definition:

```
show create table actors\G
 *************************** 1. row ***************************
       Table: actors
Create Table: CREATE TABLE `actors` (
  `name` varchar(20) DEFAULT NULL,
  `age` int unsigned DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

Now let's add a hidden column as primary key:

```
alter table actors add id int unsigned auto_increment primary key invisible first;
```

Invisible column as Primary Key (4)

We can now test again our application's queries:

```
select * from actors;
+-----------------+-----+
| name            | age |
+-----------------+-----+
| Al Pacino       |  80 |
| Robert De Niro  |  77 |
| Joe Pesci       |  78 |
| Sharon Stone    |  63 |
| Diane Keaton    |  75 |
| Talia Shire     |  74 |
+-----------------+-----+
```

Invisible column as Primary Key (4)

We can now test again our application's queries:

```
select * from actors;
+-----------------+-----+
| name            | age |
+-----------------+-----+
| Al Pacino       |  80 |
| Robert De Niro  |  77 |
| Joe Pesci       |  78 |
| Sharon Stone    |  63 |
| Diane Keaton    |  75 |
| Talia Shire     |  74 |
+-----------------+-----+
```

```
show create table actors\G
 ************* 1. row ******************
       Table: actors
Create Table: CREATE TABLE `actors` (
  `id` int unsigned NOT NULL
    AUTO_INCREMENT /*!80023 INVISIBLE */,
  `name` varchar(20) DEFAULT NULL,
  `age` int unsigned DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=7 DEFAULT
  CHARSET=utf8mb4
  COLLATE=utf8mb4_0900_ai_ci
```

Invisible column as Primary Key (5)

Great ! But what about inserts without specifying the columns ?

Invisible column as Primary Key (5)

Great ! But what about inserts without specifying the columns ?

```
insert into actors values ('James Caan', 81);
Query OK, 1 row affected (0.0248 sec)
```

Invisible column as Primary Key (6)

But if needed we have access to that PK `id`*:*

```
select id, a.* from actors a;
+----+-----------------+-----+
| id | name            | age |
+----+-----------------+-----+
|  1 | Al Pacino       |  80 |
|  2 | Robert De Niro  |  77 |
|  3 | Joe Pesci       |  78 |
|  4 | Sharon Stone    |  63 |
|  5 | Diane Keaton    |  75 |
|  6 | Talia Shire     |  74 |
|  7 | James Caan      |  81 |
+----+-----------------+-----+
```

Invisible column as Primary Key (6)

But if needed we have access to that PK `id`*:*

```
select id, a.* from actors a;
+----+----------------+-----+
| id | name           | age |
+----+----------------+-----+
|  1 | Al Pacino      |  80 |
|  2 | Robert De Niro |  77 |
|  3 | Joe Pesci      |  78 |
|  4 | Sharon Stone   |  63 |
|  5 | Diane Keaton   |  75 |
|  6 | Talia Shire    |  74 |
|  7 | James Caan     |  81 |
+----+----------------+-----+
```

And this `id` *is sequential, used as clustered index to store the data and externalized for replication !*

MySQL InnoDB Primary Keys

GIPK Mode

Invisible column as Primary Key - automatic

*Since MySQL 8.0.30 you can also enable **GIPK mode** !*

Invisible column as Primary Key - automatic

*Since MySQL 8.0.30 you can also enable **GIPK mode** !*

Generated Invisible Primary Key

Invisible column as Primary Key - automatic

*Since MySQL 8.0.30 you can also enable **GIPK mode** !*

Generated Invisible Primary Key

GIPK mode is controlled by the `sql_generate_invisible_primary_key` *server system variable.*

When MySQL is running in GIPK mode, a primary key is added to a table by the server, the column and key name is always `my_row_id`.

GIPK Mode - example

```
SQL > SELECT @@sql_generate_invisible_primary_key;
+-----------------------------------------+
| @@sql_generate_invisible_primary_key |
+-----------------------------------------+
|                                       1 |
+-----------------------------------------+

SQL > CREATE TABLE sweden_mug (name varchar(20), beers int unsigned);

SQL > INSERT INTO sweden_mug VALUES ('Ted', 5), ('lefred',1);

SQL > SELECT * FROM sweden_mug;
+--------+-------+
| name   | beers |
+--------+-------+
| Ted    |     5 |
| lefred |     1 |
+--------+-------+
2 rows in set (0.0002 sec)
```

GIPK Mode - example (2)

```
SQL > SHOW CREATE TABLE sweden_mug\G
 *************************** 1. row ***************************
       Table: sweden_mug
Create Table: CREATE TABLE `sweden_mug` (
  `my_row_id` bigint unsigned NOT NULL AUTO_INCREMENT /*!80023 INVISIBLE */,
  `name` varchar(20) DEFAULT NULL,
  `beers` int unsigned DEFAULT NULL,
  PRIMARY KEY (`my_row_id`)
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

GIPK Mode - example (2)

```
SQL > SHOW CREATE TABLE sweden_mug\G
 *************************** 1. row ***************************
       Table: sweden_mug
Create Table: CREATE TABLE `sweden_mug` (
  `my_row_id` bigint unsigned NOT NULL AUTO_INCREMENT /*!80023 INVISIBLE */,
  `name` varchar(20) DEFAULT NULL,
  `beers` int unsigned DEFAULT NULL,
  PRIMARY KEY (`my_row_id`)
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

```
SQL > SELECT *, my_row_id FROM sweden_mug;
+--------+-------+-----------+
| name   | beers | my_row_id |
+--------+-------+-----------+
| ted    |     5 |         1 |
| lefred |     1 |         2 |
+--------+-------+-----------+
```

GIPK Mode - example (3)

The information is also part of `Information_Schema`*:*

```
SQL > SELECT COLUMN_NAME, ORDINAL_POSITION, DATA_TYPE, COLUMN_KEY
        FROM INFORMATION_SCHEMA.COLUMNS
        WHERE TABLE_NAME = "sweden_mug";
+-------------+------------------+-----------+-------------+
| COLUMN_NAME | ORDINAL_POSITION | DATA_TYPE | COLUMN_KEY |
+-------------+------------------+-----------+-------------+
| beers       |                3 | int       |             |
| my_row_id   |                1 | bigint    | PRI         |
| name        |                2 | varchar   |             |
+-------------+------------------+-----------+-------------+
```

GIPK Mode - example (4)

Oh nice ! But I have this legacy application that breaks if there is an extra column part of information_schema. It seems to use it to generate forms!

GIPK Mode - example (4)

Oh nice ! But I have this legacy application that breaks if there is an extra column part of information_schema. It seems to use it to generate forms!

No worry, you can use one the variable with the longest name (a part of some pfs related ones):

`show_gipk_in_create_table_and_information_schema`

GIPK Mode - example (5)

```
SQL > SET show_gipk_in_create_table_and_information_schema = 0;

SQL > SELECT COLUMN_NAME, ORDINAL_POSITION, DATA_TYPE, COLUMN_KEY
        FROM INFORMATION_SCHEMA.COLUMNS
        WHERE TABLE_NAME = "sweden_mug";
+-------------+------------------+-----------+-------------+
| COLUMN_NAME | ORDINAL_POSITION | DATA_TYPE | COLUMN_KEY |
+-------------+------------------+-----------+-------------+
| beers       |                3 | int       |             |
| name        |                2 | varchar   |             |
+-------------+------------------+-----------+-------------+
```

GIPK Mode - example (6)

```
SQL> select @@show_gipk_in_create_table_and_information_schema;
+-----------------------------------------------------+
| @@show_gipk_in_create_table_and_information_schema |
+-----------------------------------------------------+
|                                                   0 |
+-----------------------------------------------------+

SQL > SHOW CREATE TABLE sweden_mug\G
 *************************** 1. row ***************************
       Table: sweden_mug
Create Table: CREATE TABLE `sweden_mug` (
  `name` varchar(20) DEFAULT NULL,
  `beers` int unsigned DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

MySQL InnoDB Primary Keys

Keep an eye to your Auto-Increment values

Auto-Increment Value Exhaustion

When using Auto-Increment columns for Primary Key, usually the type is an integer. But MySQL supports many different type of integers.

They way the auto-increment column is defined also defines the maximum value that can be inserted, meaning the maximum records allowed in the table.

Auto-Increment Value Exhaustion

When using Auto-Increment columns for Primary Key, usually the type is an integer. But MySQL supports many different type of integers.

They way the auto-increment column is defined also defines the maximum value that can be inserted, meaning the maximum records allowed in the table.

If the max is reached, it won't be possible to insert any new row into the table:

```
SQL> INSERT INTO t4 (id) VALUES (0);
ERROR: 1062 (23000): Duplicate entry '127' for key 't4.PRIMARY'
```

Auto-Increment Value Exhaustion (2)

It's the MySQL DBA's responsability to prevent this to happen.

Using the following query, you can have an overview of the auto-increment columns:

```
SELECT TABLE_SCHEMA, TABLE_NAME, COLUMN_NAME, DATA_TYPE, COLUMN_TYPE, IF( LOCATE('unsigned', COLUMN_TYPE) > 0,
   1, 0) AS IS_UNSIGNED, ( CASE DATA_TYPE
    WHEN 'tinyint' THEN 255 WHEN 'smallint' THEN 65535 WHEN 'mediumint' THEN 16777215 WHEN 'int' THEN 4294967295
    WHEN 'bigint' THEN 18446744073709551615
    END >> IF(LOCATE('unsigned', COLUMN_TYPE) > 0, 0, 1)) AS MAX_VALUE, AUTO_INCREMENT, CONCAT(ROUND( AUTO_INCREMENT / (
   CASE DATA_TYPE
   WHEN 'tinyint' THEN 255 WHEN 'smallint' THEN 65535 WHEN 'mediumint' THEN 16777215 WHEN 'int' THEN 4294967295
   WHEN 'bigint' THEN 18446744073709551615
   END >> IF(LOCATE('unsigned', COLUMN_TYPE) > 0, 0, 1))*100), '%') AS AUTO_INCREMENT_RATIO
   FROM INFORMATION_SCHEMA.COLUMNS INNER JOIN INFORMATION_SCHEMA.TABLES USING (TABLE_SCHEMA, TABLE_NAME)
   WHERE TABLE_SCHEMA NOT IN ('mysql', 'INFORMATION_SCHEMA', 'performance_schema') AND EXTRA='auto_increment'
   ORDER BY CAST(AUTO_INCREMENT_RATIO AS SIGNED INTEGER);
```

Auto-Increment Value Exhaustion - Sample

```
*************************** 1. row ***************************
       TABLE_SCHEMA: test
         TABLE_NAME: t4
        COLUMN_NAME: id
          DATA_TYPE: tinyint
        COLUMN_TYPE: tinyint
        IS_UNSIGNED: 0
          MAX_VALUE: 127
     AUTO_INCREMENT: 124
AUTO_INCREMENT_RATIO: 98%
```

Auto-Increment Value Exhaustion (3)

The best is to define it as `BIGINT UNSIGNED` *if you expect a lot of record:*

```
CREATE TABLE scott (
    id BIGINT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    inserted DATETIME DEFAULT current_timestamp);
```

Auto-Increment Value Exhaustion (3)

The best is to define it as `BIGINT UNSIGNED` *if you expect a lot of record:*

```
CREATE TABLE scott (
    id BIGINT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    inserted DATETIME DEFAULT current_timestamp);
```

```
SQL> SHOW CREATE TABLE scott\G
 *************************** 1. row ***************************
       Table: scott
Create Table: CREATE TABLE `scott` (
  `id` bigint unsigned NOT NULL AUTO_INCREMENT,
  `inserted` datetime DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

What about SERIAL ?

You can also define your auto-increment column using the keyword `SERIAL`*:*

```
CREATE TABLE scott (
    id SERIAL PRIMARY KEY,
    inserted DATETIME DEFAULT current_timestamp);
```

What about SERIAL ?

You can also define your auto-increment column using the keyword `SERIAL`*:*

```
CREATE TABLE scott (
    id SERIAL PRIMARY KEY,
    inserted DATETIME DEFAULT current_timestamp);
```

And we can see that MySQL made the right transformation:

```
SQL> SHOW CREATE TABLE scott\G
[...] Create Table: CREATE TABLE `scott` (
  `id` bigint unsigned NOT NULL AUTO_INCREMENT,
  `inserted` datetime DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (`id`),
  UNIQUE KEY `id` (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

What about SERIAL ? (2)

`SERIAL` *is an alias of* `BIGINT UNSIGNED NOT NULL AUTO_INCREMENT UNIQUE` *but as InnoDB requires that the* `AUTO_INCREMENT` *column* ***MUST*** *be part of the Primary Key.*

This is valid:

```
CREATE TABLE scott (id serial, name varchar(10),
                    inserted DATETIME default current_timestamp,
                    primary key(id, name));
```

What about SERIAL ? (3)

```
SQL> SHOW CREATE TABLE scott\G
 *************************** 1. row ***************************
       Table: scott
Create Table: CREATE TABLE `scott` (
  `id` bigint unsigned NOT NULL AUTO_INCREMENT,
  `name` varchar(10) NOT NULL,
  `inserted` datetime DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (`id`,`name`),
  UNIQUE KEY `id` (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
1 row in set (0.0017 sec)
```

MySQL InnoDB Primary Keys

Replication & HA

Why is it bad to replicate without PK ?

*Missing Primary Keys when using Row Based Replication (RBR) is the main cause of **replication lag**.*

If one runs DML on a table that has no indexes, a full table scan is done.

*With RBR, the replica will need to scan the full table for **each** row changed !*

Why is it bad to replicate without PK ?

Example

```
SQL> CREATE TABLE `nopk` (
  `i` int NOT NULL,
  `name` varchar(20) DEFAULT NULL,
  `inserted` timestamp NULL DEFAULT CURRENT_TIMESTAMP,
  `g` int NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

As you can see, we don't have any Primary Key defined.

Why is it bad to replicate without PK ?

Example (2)

```
SQL> select * from nopk;
+-----+----------+---------------------+---+
| i   | name     | inserted            | g |
+-----+----------+---------------------+---+
| 100 | 9A2B4503 | 2024-01-22 11:25:45 | 0 |
|  20 | 9B004592 | 2024-01-22 11:25:46 | 8 |
|  89 | 9C37EA4E | 2024-01-22 11:25:48 | 6 |
|  85 | 9CA3C011 | 2024-01-22 11:25:49 | 0 |
|  98 | 9D084BB1 | 2024-01-22 11:25:50 | 5 |
|  73 | 9D739EDB | 2024-01-22 11:25:50 | 0 |
|  90 | 9DD30035 | 2024-01-22 11:25:51 | 3 |
|  31 | 9E39D1E3 | 2024-01-22 11:25:52 | 3 |
|  75 | 9EA8DE8E | 2024-01-22 11:25:52 | 6 |
|  19 | A0C11A3A | 2024-01-22 11:25:56 | 8 |
+-----+----------+---------------------+---+
```

Why is it bad to replicate without PK ?

Example (3)

We will now delete some records and check what's happening.

First we check the Query Execution Plan:

```
SQL> explain delete from nopk where g < 5\G
*************************** 1. row ***************************
           id: 1
  select_type: DELETE
        table: nopk
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 10
     filtered: 100
        Extra: Using where
```

Why is it bad to replicate without PK ?

Example (4)

```
SQL> flush status;

SQL> delete from nopk where g < 5\G
Query OK, 5 rows affected (0.0080 sec)

SQL> show status like 'Handler_%';
+----------------------------+-------+
| Variable_name              | Value |
+----------------------------+-------+
| Handler_commit             | 2     |
| Handler_delete             | 5     |
...
| Handler_read_rnd_next      | 11    |
...
```

```
#240122 11:34:46 server id 1  end_log_pos 11663 CRC32 0x844fe80a     Delete_rows: table id 652 flags: STMT_END_F
### DELETE FROM `smug`.`nopk`
### WHERE
###   @1=100 /* INT meta=0 nullable=0 is_null=0 */
###   @2='9A2B4503' /* VARSTRING(80) meta=80 nullable=1 is_null=0 */
###   @3=1705919145 /* TIMESTAMP(0) meta=0 nullable=1 is_null=0 */
###   @4=0 /* INT meta=0 nullable=0 is_null=0 */
### DELETE FROM `smug`.`nopk`
### WHERE
###   @1=85 /* INT meta=0 nullable=0 is_null=0 */
###   @2='9CA3C011' /* VARSTRING(80) meta=80 nullable=1 is_null=0 */
###   @3=1705919149 /* TIMESTAMP(0) meta=0 nullable=1 is_null=0 */
###   @4=0 /* INT meta=0 nullable=0 is_null=0 */
### DELETE FROM `smug`.`nopk`
### WHERE
###   @1=73 /* INT meta=0 nullable=0 is_null=0 */
###   @2='9D739EDB' /* VARSTRING(80) meta=80 nullable=1 is_null=0 */
###   @3=1705919150 /* TIMESTAMP(0) meta=0 nullable=1 is_null=0 */
###   @4=0 /* INT meta=0 nullable=0 is_null=0 */
### DELETE FROM `smug`.`nopk`
### WHERE
###   @1=90 /* INT meta=0 nullable=0 is_null=0 */
###   @2='9DD30035' /* VARSTRING(80) meta=80 nullable=1 is_null=0 */
###   @3=1705919151 /* TIMESTAMP(0) meta=0 nullable=1 is_null=0 */
###   @4=3 /* INT meta=0 nullable=0 is_null=0 */
### DELETE FROM `smug`.`nopk`
### WHERE
###   @1=31 /* INT meta=0 nullable=0 is_null=0 */
###   @2='9E39D1E3' /* VARSTRING(80) meta=80 nullable=1 is_null=0 */
###   @3=1705919152 /* TIMESTAMP(0) meta=0 nullable=1 is_null=0 */
###   @4=3 /* INT meta=0 nullable=0 is_null=0 */
# at 11663
#240122 11:34:46 server id 1  end_log_pos 11694 CRC32 0xaf0c107f     Xid = 139
COMMIT/*!*/;
```

Replication and GIPK Mode

GIPK mode supports row-based replication of `CREATE TABLE ... SELECT`*; the information written to the binary log for this statement in such cases includes the GIPK definition, and thus is replicated correctly.*

Statement-based replication of `CREATE TABLE ... SELECT` *is not supported in GIPK mode.*

It's also possible to define a table Primary Key Policy per replication channels using `REQUIRE_TABLE_PRIMARY_KEY_CHECK`*.*

This can be useful for live migrations.

See https://dev.mysql.com/doc/refman/8.3/en/change-replication-source-to.html

Primary Keys and High Availability

*When using Group Replication (MySQL InnoDB Cluster) every table that is to be replicated by the group **must have** a defined primary key, or primary key equivalent where the equivalent is a non-null unique key.*

*Such keys are required as a unique identifier for every row within a table, enabling the system to determine which transactions conflict by identifying exactly which rows each transaction has modified: **the "certification" phase**.*

MySQL InnoDB Primary Keys

MySQL Shell Dump & Load

mysqldump & GIPK

When creating or importing dumps of MySQL instances in which GIPK mode is in use, it is possible to exclude generated invisible PK columns and values.

The `--skip-generated-invisible-primary-key` *option for* `mysqldump` *causes GIPK information to be excluded in the program's output.*

If you are importing a dump file that contains GIPK keys and values, you can also use `--skip-generated-invisible-primary-key`*.*

mysqldump & GIPK

When creating or importing dumps of MySQL instances in which GIPK mode is in use, it is possible to exclude generated invisible PK columns and values.

The `--skip-generated-invisible-primary-key` *option for* `mysqldump` *causes GIPK information to be excluded in the program's output.*

If you are importing a dump file that contains GIPK keys and values, you can also use `--skip-generated-invisible-primary-key`*.*

Avoid the use of `mysqldump`, and use MySQL Shell !

MySQL Shell Dump Utility & GIPK

`create_invisible_pks` *compatibility option of MySQL Shell Dump Utility adds a flag in the dump metadata to notify MySQL Shell Load Utility to add primary keys in invisible columns, for each table that does not contain a primary key.*

The dump data is unchanged by this modification, as the tables do not contain the invisible columns until they have been processed by the dump loading utility.

```
JS> util.dumpSchemas(["smug"], "/tmp/smug", {compatibility: ['create_invisible_pks']})
....
NOTE: Table `smug`.`nopk` does not have a Primary Key, this will be
fixed when the dump is loaded
...
```

MySQL Shell Load Utility & GIPK

MySQL Shell's load utility option `createInvisiblePKs` *uses the server's GIPK mode to generate invisible primary keys for tables which do not have primary keys.*

Requires server 8.0.24 or newer.

```
JS> util.loadDump("/tmp/smug", {createInvisiblePKs: true})
```

MySQL Shell Copy Utility & GIPK

All the `copy` *utilities from MySQL Shell support the compatibilty option to create invisible Primary Keys:*

- `util.copyInstance()`
- `util.copySchemas()`
- `util.copyTables()`

```
JS> util.copySchemas(["smug"],"mysql://smug@smug-host2/",
    {compatibility: ['create_invisible_pks']})
```

MySQL InnoDB Primary Keys

What about UUIDs ?

InnoDB Primary Key - What about UUID ?

*There are 2 major problems with **UUID**'s as Primary Key:*

1. *generally they are completely random and cause clustered index re-banlancing*
2. *they are included in each secondary indexes (consuming disk and memory)*

InnoDB Primary Key - What about UUID ? (2)

Example:

```
SQL> CREATE TABLE smug (
       uuid VARCHAR(36) DEFAULT (UUID()) PRIMARY KEY,
       name VARCHAR(20), beers int unsigned);

SQL> SELECT * FROM smug;
+--------------------------------------+---------+-------+
| uuid                                 | name    | beers |
+--------------------------------------+---------+-------+
| c15d29cb-b924-11ee-9aaa-f4a475a3749c | Ted     |     0 |
| c15d2dbd-b924-11ee-9aaa-f4a475a3749c | lefred  |     1 |
| c15d2ed4-b924-11ee-9aaa-f4a475a3749c | Staffan |     0 |
| c15d2fe6-b924-11ee-9aaa-f4a475a3749c | Lenka   |     1 |
+--------------------------------------+---------+-------+
```

InnoDB Primary Key - What about UUID ? (3)

Let's insert 2 new records:

```
SQL> INSERT INTO smug (name, beers) VALUES ("Carsten",1), ("Scott",5);
Query OK, 2 rows affected (0.0069 sec)
```

InnoDB Primary Key - What about UUID ? (3)

Let's insert 2 new records:

```
SQL> INSERT INTO smug (name, beers) VALUES ("Carsten",1), ("Scott",5);
Query OK, 2 rows affected (0.0069 sec)
```

```
SQL> SELECT * FROM smug;
+--------------------------------------+---------+-------+
| uuid                                 | name    | beers |
+--------------------------------------+---------+-------+
| 4d16b6ef-b925-11ee-9aaa-f4a475a3749c | Carsten |     1 |
| 4d16bd15-b925-11ee-9aaa-f4a475a3749c | Scott   |     5 |
| c15d29cb-b924-11ee-9aaa-f4a475a3749c | Ted     |     0 |
| c15d2dbd-b924-11ee-9aaa-f4a475a3749c | lefred  |     1 |
| c15d2ed4-b924-11ee-9aaa-f4a475a3749c | Staffan |     0 |
| c15d2fe6-b924-11ee-9aaa-f4a475a3749c | Lenka   |     1 |
+--------------------------------------+---------+-------+
```

InnoDB Primary Key - What about UUID ? (4)

OUPS ! We have rebalanced the clustered index !

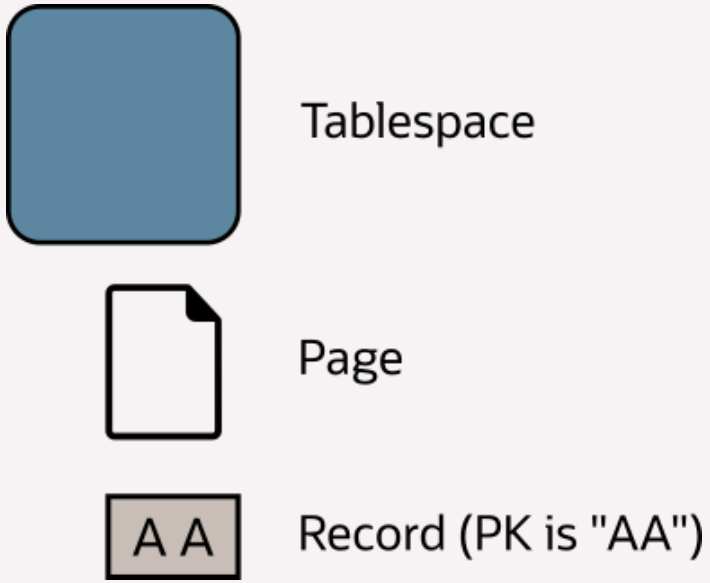
What does that mean again ??

InnoDB Primary Key - What about UUID ? (4)

OUPS ! We have rebalanced the clustered index !

What does that mean again ??

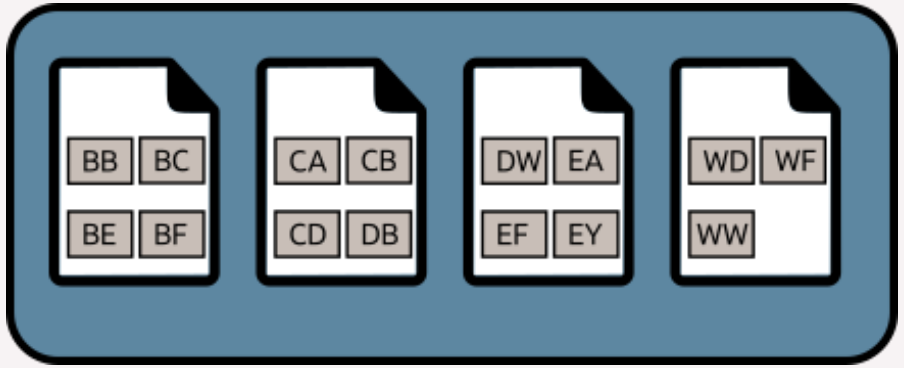
Let me try to explain this with this high level and simplified example:



InnoDB Primary Key - What about UUID ? (5)

OUPS ! We have rebalanced the clustered index !

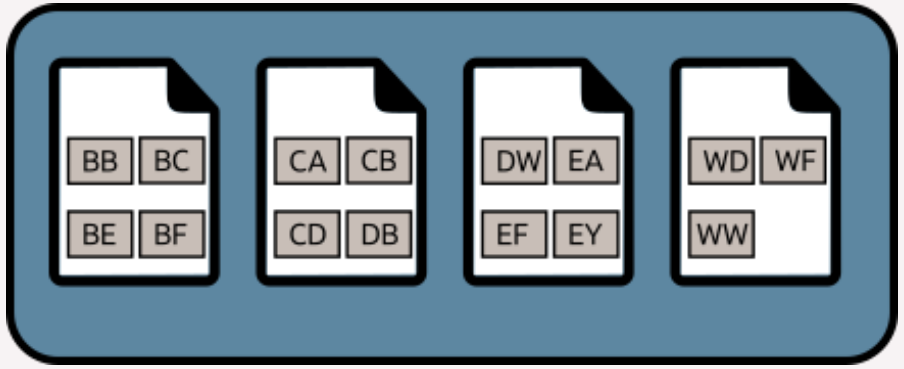
Let's imagine one InnoDB Page can store 4 records (this is just a fiction), and we have inserted some records using a random Primary Key:



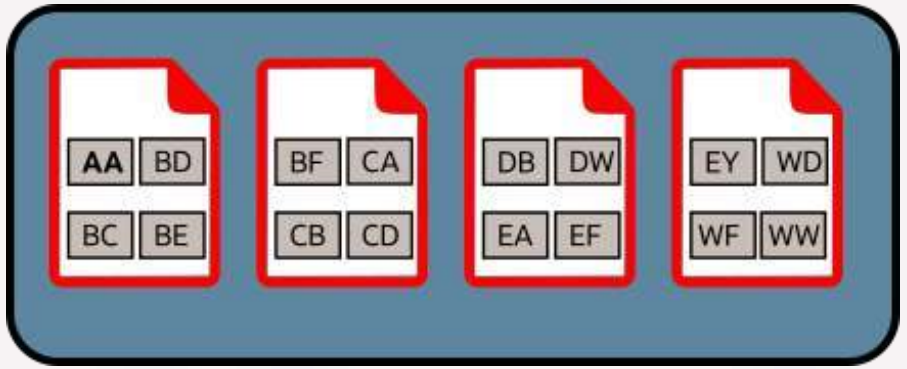
InnoDB Primary Key - What about UUID ? (5)

OUPS ! We have rebalanced the clustered index !

Let's imagine one InnoDB Page can store 4 records (this is just a fiction), and we have inserted some records using a random Primary Key:



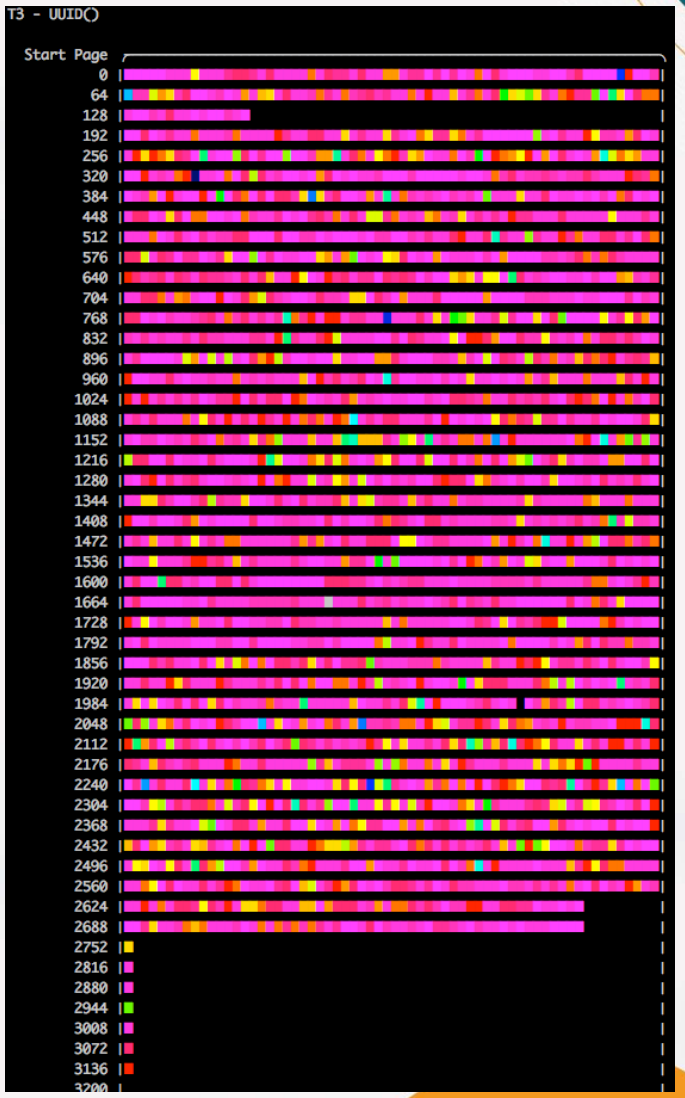
And now we insert a new record and the Primary Key is AA:



All pages were modified to "rebalance" the clustered index ! Imagine if this was a 4TB table !!

InnoDB Primary Key - What about UUID ? (6)

This an overview of inserts into a table using random UUIDs as Primary Key:



InnoDB Primary Key - What about UUID ? (7)

*And just for info, each entry in the Primary Key Index could take up to **146 bytes**(*):*

```
SQL > EXPLAIN SELECT * FROM smug WHERE
        uuid='c15d2dbd-b924-11ee-9aaa-f4a475a3749c'\G
 *************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: smug
   partitions: NULL
         type: const
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 146
          ref: const
         rows: 1
     filtered: 100
        Extra: NULL
```

(*) worse case when using characters using 4 bytes each (uft8mb4)

InnoDB Primary Key - What about UUID ? (8)

Recommended solution

1. *use a smaller datatype:* `BINARY(16)`

2. *store the UUID sequentially:* `UUID_TO_BIN(..., swap_flag)`

    - *The time-low and time-high parts (the first and third groups of hexadecimal digits, respectively) are swapped.*

InnoDB Primary Key - What about UUID ? (9)

Recommended solution - example

```
SQL> CREATE TABLE smug2 (
          uuid BINARY(16) DEFAULT (UUID_TO_BIN(UUID(), 1)) PRIMARY KEY,
          name VARCHAR(20), beers int unsigned);

SQL> SELECT * FROM smug2;
+------------------------------------+--------+-------+
| uuid                               | name   | beers |
+------------------------------------+--------+-------+
| 0x11EEB92771438C999AAAF4A475A3749C | Ted    |     0 |
| 0x11EEB9277143902F9AAAF4A475A3749C | lefred |     1 |
+------------------------------------+--------+-------+
```

InnoDB Primary Key - What about UUID ? (9)

Recommended solution - example

```
SQL> CREATE TABLE smug2 (
          uuid BINARY(16) DEFAULT (UUID_TO_BIN(UUID(), 1)) PRIMARY KEY,
          name VARCHAR(20), beers int unsigned);

SQL> SELECT * FROM smug2;
+--------------------------------------+--------+-------+
| uuid                                 | name   | beers |
+--------------------------------------+--------+-------+
| 0x11EEB92771438C999AAAF4A475A3749C | Ted    |     0 |
| 0x11EEB9277143902F9AAAF4A475A3749C | lefred |     1 |
+--------------------------------------+--------+-------+
```

```
SQL > SELECT BIN_TO_UUID(uuid,1), name, beers FROM smug2;
+--------------------------------------+--------+-------+
| bin_to_uuid(uuid,1)                  | name   | beers |
+--------------------------------------+--------+-------+
| 71438c99-b927-11ee-9aaa-f4a475a3749c | Ted    |     0 |
| 7143902f-b927-11ee-9aaa-f4a475a3749c | lefred |     1 |
+--------------------------------------+--------+-------+
```

InnoDB Primary Key - What about UUID ? (10)

Recommended solution - example

```
SQL> INSERT INTO smug2 (name, beers) VALUES ("Carsten",1), ("Scott",5);

SQL> SELECT * FROM smug2;
+--------------------------------------+---------+-------+
| uuid                                 | name    | beers |
+--------------------------------------+---------+-------+
| 0x11EEB92771438C999AAAF4A475A3749C | Ted     |     0 |
| 0x11EEB9277143902F9AAAF4A475A3749C | lefred  |     1 |
| 0x11EEB9284D66AA559AAAF4A475A3749C | Carsten |     1 |
| 0x11EEB9284D66B0439AAAF4A475A3749C | Scott   |     5 |
+--------------------------------------+---------+-------+
```

InnoDB Primary Key - What about UUID ? (10)

Recommended solution - example

```
SQL> INSERT INTO smug2 (name, beers) VALUES ("Carsten",1), ("Scott",5);

SQL> SELECT * FROM smug2;
+--------------------------------------+---------+-------+
| uuid                                 | name    | beers |
+--------------------------------------+---------+-------+
| 0x11EEB92771438C999AAAF4A475A3749C | Ted     |     0 |
| 0x11EEB9277143902F9AAAF4A475A3749C | lefred  |     1 |
| 0x11EEB9284D66AA559AAAF4A475A3749C | Carsten |     1 |
| 0x11EEB9284D66B0439AAAF4A475A3749C | Scott   |     5 |
+--------------------------------------+---------+-------+
```

```
SQL> SELECT BIN_TO_UUID(uuid,1), name, beers FROM smug2;
+--------------------------------------+---------+-------+
| bin_to_uuid(uuid,1)                  | name    | beers |
+--------------------------------------+---------+-------+
| 71438c99-b927-11ee-9aaa-f4a475a3749c | Ted     |     0 |
| 7143902f-b927-11ee-9aaa-f4a475a3749c | lefred  |     1 |
| 4d66aa55-b928-11ee-9aaa-f4a475a3749c | Carsten |     1 |
| 4d66b043-b928-11ee-9aaa-f4a475a3749c | Scott   |     5 |
+--------------------------------------+---------+-------+
```

InnoDB Primary Key - What about UUID ? (11)

Recommended solution - example

Take a look at the size of each entry in the INDEX (and same amount added to each secondary index)

```
SQL > EXPLAIN SELECT * FROM smug2
        WHERE uuid=UUID_TO_BIN("7143902f-b927-11ee-9aaa-f4a475a3749c",1)\G
 *************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: smug2
   partitions: NULL
         type: const
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 16
          ref: const
         rows: 1
     filtered: 100
        Extra: NULL
```

MySQL UUID

*The UUIDs generated by MySQL are **UUID v1** as described in RFC4122.*

- *UUID v1 : is a universally unique identifier that is generated using a timestamp and the MAC address of the computer on which it was generated.*
- *UUID v4 : is a universally unique identifier that is generated using random numbers.*
- *UUID v7 : is a universally unique identifier that is generated using a timestamp and random numbers.*

MySQL UUID

*The UUIDs generated by MySQL are **UUID v1** as described in RFC4122.*

- *UUID v1 : is a universally unique identifier that is generated using a timestamp and the MAC address of the computer on which it was generated.*
- *UUID v4 : is a universally unique identifier that is generated using random numbers.*
- *UUID v7 : is a universally unique identifier that is generated using a timestamp and random numbers.*

With UUID v4, it's not possible to generate any sequential output.

MySQL - More with UUID v1

Do you know you can now the exact time when your MySQL UUIDs (v1) were created ??

MySQL - More with UUID v1

Do you know you can now the exact time when your MySQL UUIDs (v1) were created ??

Take a look at https://github.com/lefred/mysql-component-uuid_v1:

```
SQL> select uuid_to_timestamp('2f1d6a6f-b9ee-11ee-8b9b-c8cb9e32df8e') timestamp;
+-------------------------+
| timestamp               |
+-------------------------+
| 2024-01-23 13:51:53.887 |
+-------------------------+

SQL> select uuid_to_timestamp_long('2f1d6a6f-b9ee-11ee-8b9b-c8cb9e32df8e') timestamp;
+-----------------------------+
| timestamp                   |
+-----------------------------+
| Tue Jan 23 13:51:53 2024 CET |
+-----------------------------+
```

MySQL - extra

If you would like to play with UUIDv4 and UUIDv7, take a look at these MySQL components:

- *https://github.com/lefred/mysql-component-uuid_v4*
- *https://github.com/lefred/mysql-component-uuid_v7*

Percona Server also includes a UUID component:

- *https://github.com/percona/percona-server/tree/8.0/components/uuid_vx_udf*

MySQL - extra (2)

Example

```
SQL> select now(), uuid(), uuid_v4(), uuid_v7();
+---------------------+--------------------------------------+--------------------------------------+--------------------------------------+
| now()               | uuid()                               | uuid_v4()                            | uuid_v7()                            |
+---------------------+--------------------------------------+--------------------------------------+--------------------------------------+
| 2024-01-23 13:46:22 | 6990aafd-b9ed-11ee-8b9b-c8cb9e32df8e | c16d507f-d62e-4879-956d-d188c062184a | 018d365b-7ef6-7d99-aaf0-673e8d739b3e |
+---------------------+--------------------------------------+--------------------------------------+--------------------------------------+
```

MySQL - extra (2)

Example

```
SQL> select now(), uuid(), uuid_v4(), uuid_v7();
+---------------------+--------------------------------------+--------------------------------------+--------------------------------------+
| now()               | uuid()                               | uuid_v4()                            | uuid_v7()                            |
+---------------------+--------------------------------------+--------------------------------------+--------------------------------------+
| 2024-01-23 13:46:22 | 6990aafd-b9ed-11ee-8b9b-c8cb9e32df8e | c16d507f-d62e-4879-956d-d188c062184a | 018d365b-7ef6-7d99-aaf0-673e8d739b3e |
+---------------------+--------------------------------------+--------------------------------------+--------------------------------------+

SQL> select now(), uuid(), uuid_v4(), uuid_v7();
+---------------------+--------------------------------------+--------------------------------------+--------------------------------------+
| now()               | uuid()                               | uuid_v4()                            | uuid_v7()                            |
+---------------------+--------------------------------------+--------------------------------------+--------------------------------------+
| 2024-01-23 13:49:00 | c7dfe39c-b9ed-11ee-8b9b-c8cb9e32df8e | 1bc1d087-de40-496a-aa67-aefc87aa05d0 | 018d365d-e907-70d1-a565-ed4017772ed3 |
+---------------------+--------------------------------------+--------------------------------------+--------------------------------------+
```

MySQL - extra (2)

Example

```
SQL> select now(), uuid(), uuid_v4(), uuid_v7();
+---------------------+--------------------------------------+--------------------------------------+--------------------------------------+
| now()               | uuid()                               | uuid_v4()                            | uuid_v7()                            |
+---------------------+--------------------------------------+--------------------------------------+--------------------------------------+
| 2024-01-23 13:46:22 | 6990aafd-b9ed-11ee-8b9b-c8cb9e32df8e | c16d507f-d62e-4879-956d-d188c062184a | 018d365b-7ef6-7d99-aaf0-673e8d739b3e |
+---------------------+--------------------------------------+--------------------------------------+--------------------------------------+

SQL> select now(), uuid(), uuid_v4(), uuid_v7();
+---------------------+--------------------------------------+--------------------------------------+--------------------------------------+
| now()               | uuid()                               | uuid_v4()                            | uuid_v7()                            |
+---------------------+--------------------------------------+--------------------------------------+--------------------------------------+
| 2024-01-23 13:49:00 | c7dfe39c-b9ed-11ee-8b9b-c8cb9e32df8e | 1bc1d087-de40-496a-aa67-aefc87aa05d0 | 018d365d-e907-70d1-a565-ed4017772ed3 |
+---------------------+--------------------------------------+--------------------------------------+--------------------------------------+

SQL> select now(), uuid(), uuid_v4(), uuid_v7();
+---------------------+--------------------------------------+--------------------------------------+--------------------------------------+
| now()               | uuid()                               | uuid_v4()                            | uuid_v7()                            |
+---------------------+--------------------------------------+--------------------------------------+--------------------------------------+
| 2024-01-23 13:51:53 | 2f1d6a6f-b9ee-11ee-8b9b-c8cb9e32df8e | a41ce1b4-d28a-4288-a5b8-cd7683664631 | 018d3660-8d9f-79eb-a683-8d4365b6f087 |
+---------------------+--------------------------------------+--------------------------------------+--------------------------------------+
```

MySQL Document Store

JSON & Primary Key: _id

MySQL Document Store

With MySQL, it's also possible to store JSON documents and use CRUD operations to deal with them.

You can use MySQL without a single line of SQL!

MySQL Document Store

With MySQL, it's also possible to store JSON documents and use CRUD operations to deal with them.

You can use MySQL without a single line of SQL!

```
$session = mysql_xdevapi\getSession("mysqlx://fred:MyP@ssw0rd%@localhost");
$schema = $session->getSchema("docstore");
$collection = $schema->getCollection("restaurants");
$results = $collection->find("cuisine='italian'")->execute()->fetchAll();
```

MySQL Document Store

Collections are special tables that look like this in SQL:

```
       Table: restaurants
Create Table: CREATE TABLE `restaurants` (
  `doc` json DEFAULT NULL,
  `_id` varbinary(32) GENERATED ALWAYS
         AS (json_unquote(json_extract(`doc`,_utf8mb4'$._id'))) STORED NOT NULL,
  `_json_schema` json GENERATED ALWAYS AS (_utf8mb4'{"type":"object"}') VIRTUAL,
  PRIMARY KEY (`_id`),
  CONSTRAINT `$val_strict_B38964F8D1504551AB639A4D860DFA5460340231`
  CHECK (json_schema_valid(`_json_schema`,`doc`)) /*!80016 NOT ENFORCED */
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

MySQL Document Store: _id

We can see that **_id** *is the Primary Key and generated automatically:*

```
MySQL  localhost:33060+  docstore  2024-04-30 11:30:56
JS  var pizza={
        "name": "Pizzeria Di Stefano",
        "grades": [
            {
                "date": {
                    "$date": "2024-04-30T00:00:00.000+0000"
                },
                "grade": "A",
                "score": 95
            }
        ],
        "address": {
            "coord": [
                53.214803,
                6.561811
            ],
            "street": "Gedempte Zuiderdiep",
            "zipcode": "9711HN",
            "building": "154"
        },
        "borough": "Groningen",
        "cuisine": "Italian"
    }

MySQL  localhost:33060+  docstore  2024-04-30 11:31:44
JS  db.restaurants.add(pizza)
Query OK, 1 item affected (0.0040 sec)
```

MySQL Document Store: _id

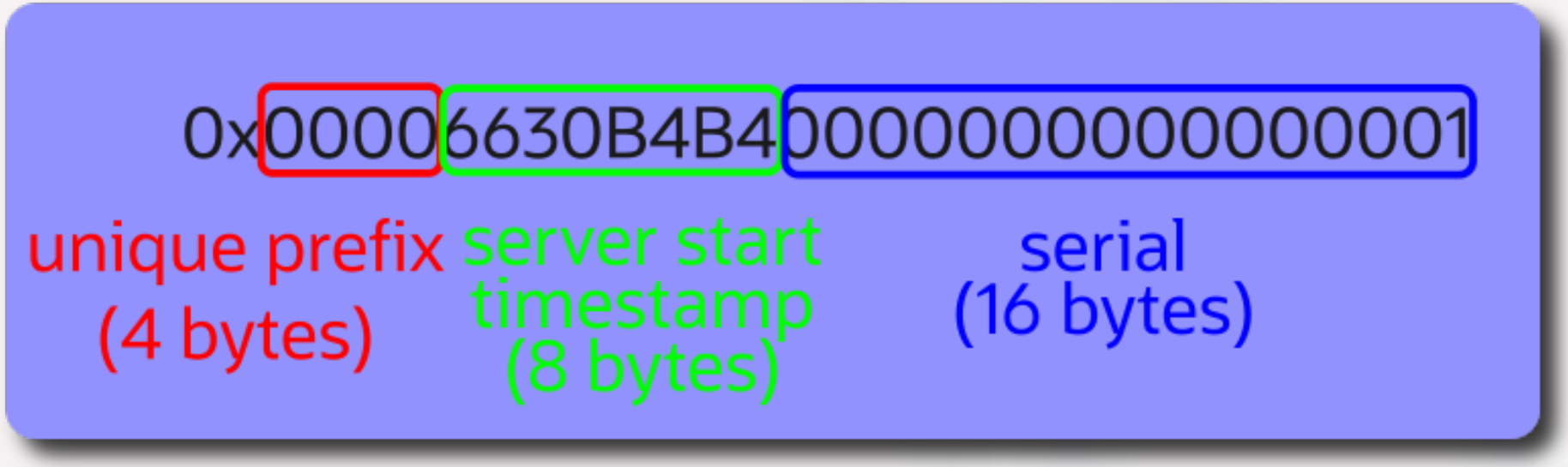
```
MySQL localhost:33060+ docstore 2024-04-30 11:33:34
JS db.restaurants.find("borough='Groningen'")
{
    "_id": "00006630b4b40000000000000001",
    "name": "Pizzeria Di Stefano",
    "grades": [
        {
            "date": {
                "$date": "2024-04-30T00:00:00.000+0000"
            },
            "grade": "A",
            "score": 95
        }
    ],
    "address": {
        "coord": [
            53.214803,
            6.561811
        ],
        "street": "Gedempte Zuiderdiep",
        "zipcode": "9711HN",
        "building": "154"
    },
    "borough": "Groningen",
    "cuisine": "Italian"
}
1 document in set (0.0220 sec)
```

MySQL Document Store: _id in SQL

```
SQL > select _id, json_pretty(doc) from restaurants where doc->>"$.borough" = 'Groningen'\G
*************************** 1. row ***************************
            _id: 0x3030303036363330623462343030303030303030303030303030303031
json_pretty(doc): {
  "_id": "00006630b4b40000000000000001",
  "name": "Pizzeria Di Stefano",
  "grades": [
    {
      "date": {
        "$date": "2024-04-30T00:00:00.000+0000"
      },
      "grade": "A",
      "score": 95
    }
  ],
  "address": {
    "coord": [
      53.214803,
      6.561811
    ],
    "street": "Gedempte Zuiderdiep",
    "zipcode": "9711HN",
    "building": "154"
  },
  "borough": "Groningen",
  "cuisine": "Italian"
}
1 row in set (0.0194 sec)
```

MySQL Document Store: `_id`: details

The generation of Document's `_id` *is defined in WL#10955:*



The unique prefix is defined with the variable `mysqlx_document_id_unique_prefix`.

https://dev.mysql.com/worklog/task/?id=10955

MySQL Document Store: fun with _id

You can decode the timestamp to find out when the server that generated the identifier was started:

```
select FROM_UNIXTIME(conv(hex(substring(UNHEX(_id),3,4)),16,10)) server_start_time
from restaurants where doc->>"$.borough" = 'Groningen';
+---------------------+
| server_start_time   |
+---------------------+
| 2024-04-30 11:07:00 |
+---------------------+
1 row in set (0.0170 sec)
```

Share your ❤️ to MySQL

#mysql #MySQLCommunity

Join our slack channel!

bit.ly/mysql-slack

MySQL

Questions ?