

Query Optimization: From 0 to 10 (and up to 5.7)

Jaime Crespo

Percona Live Europe 2015

-Amsterdam, 21 Sep 2015-

dbahire.com/pleu15

Agenda - First 3 hours

1. Introduction

2. Access Types and Basic Indexing Techniques

3. Break

4. Multi-Column Indexing

5. FULLTEXT Search

6. Joins

7. Subqueries

8. Query Profiling

Agenda - Last 3 hours

1. General Optimizer Improvements	6. SQL Mode Changes
2. Computed/Virtual Columns	7. GIS Improvements and JSON Type
3. Query Rewrite Plugins	8. Results and Conclusions
4. Break	9. Q&A
5. Optimizer Hints	

Query Optimization: From 0 to 10 (and up to 5.7)

INTRODUCTION

This is me fighting bad query performance



- Sr. Database Administrator at Wikimedia Foundation
- Used to work as a trainer for Oracle (MySQL), as a Consultant (Percona) and as a Freelance administrator (DBAHire.com)

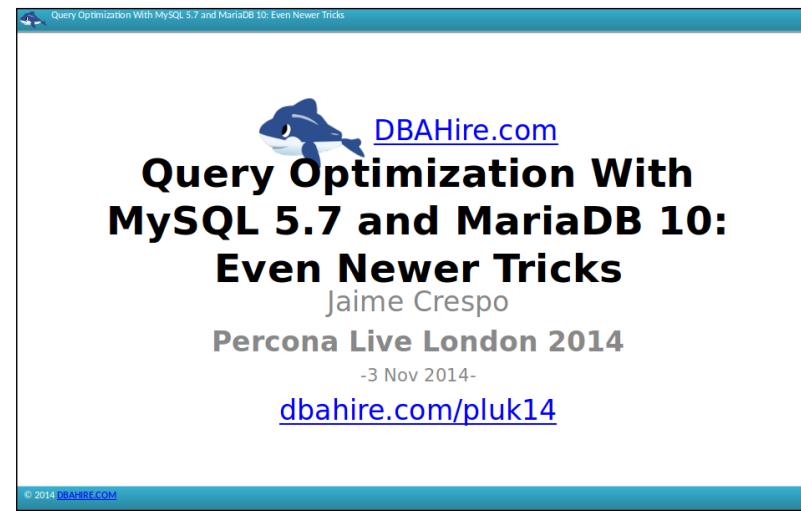
MySQL Versions

- 5.1 no longer has official support
- I will be showing you the results on mysql versions 5.5-5.7/10.1
- MySQL 5.7 and MariaDB 10.1 in RC with great new features

Recently Added Features Related to Query Optimization

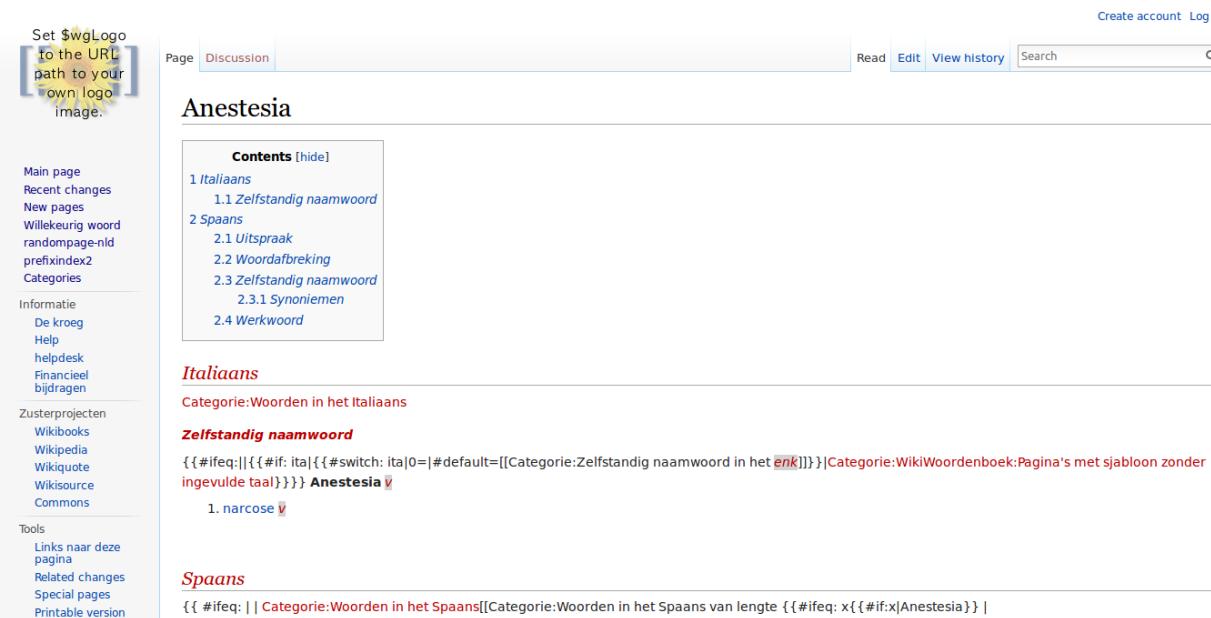
- Index Condition Pushdown
- Subquery Optimizations (materialization and semijoin)
- IN-to-EXISTS/EXISTS-to-IN
- JOIN-to-WHERE
- Multi-Range Read
- Batched Key Access
- Persistent InnoDB Statistics
- UNION ALL optimization
- Improved GIS support
- EXPLAIN FORMAT=JSON
- EXPLAIN INSERT/UPDATE/DELETE
- Hash Join
- New optimizer hints
- New cost-based optimizer
- Optimizer Trace
- Filesort optimizations
- Virtual/computed columns and “functional indexes”
- New JSON type

I Already Mentioned Some of Those Topics during the Last Years



- Check my presentations here:
<http://www.slideshare.net/jynus/>

Example Application (I)



The screenshot shows a Wikipedia-style page titled "Anestesia". The page has a sidebar on the left containing links like "Main page", "Recent changes", and "Categories". The main content area includes a "Contents" section with links to "Italiaans" and "Spaans" sections. The "Italiaans" section contains a single link to "Anestesia". The "Spaans" section also contains a single link to "Anestesia". The page footer includes a "Create account" and "Log in" link, and a search bar.

Anestesia

Contents [hide]

1 Italiaans

1.1 [Zelfstandig naamwoord](#)

2 Spaans

2.1 [Uitspraak](#)

2.2 [Woordafbreking](#)

2.3 [Zelfstandig naamwoord](#)

2.3.1 [Synoniemen](#)

2.4 [Werkwoord](#)

Italiaans

Categorie:Woorden in het Italiaans

Zelfstandig naamwoord

{}{{#ifeq:||{{#if: ita}}{{#switch: ita|0|#default=[[Categorie:Zelfstandig naamwoord in het [en](#)]]}}|Categorie:WikiWoordenboek:Pagina's met sjabloon zonder ingevulde taal}}}} Anestesia

1. narcose

Spaans

{}{{#ifeq: | | Categorie:Woorden in het Spaans[[Categorie:Woorden in het Spaans van lengte {{#ifeq: x{{#if:x|Anestesia}}}} |

- Wiktionary (and all Wikimedia project's data) is licensed under the Creative Commons BY-SA-2.5 License and is Copyright its Contributors

Example Application (II)



- OSM Database is licensed under the Open DataBase License and is Copyright OpenStreetMap Contributors

Install the example databases

- Downloads and instructions at:
<http://dbahire.com/pleu15>
 - Requirements: a MySQL or MariaDB installation
(MySQL Sandbox is suggested)
 - The wiktionary and OSM extracts
- Import them by doing:
\$ bzcat <file> | mysql <database>

Query Optimization: From 0 to 10 (and up to 5.7)

ACCESS TYPES AND BASIC INDEXING TECHNIQUES

EXPLAIN

- Essential to understand the execution plan of our queries
 - Works on SELECTs, INSERTs, UPDATEs, REPLACEs, DELETEs and connections
 - Fully documented on:
<http://dev.mysql.com/doc/refman/5.6/en/explain-output.html>

EXPLAIN Example

```
MariaDB [nlwiktio...]> EXPLAIN SELECT * FROM page WHERE page_title = 'Dutch';
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	page	ALL	NULL			NULL	90956	Using where

1 row in set (0.00 sec)

Difficult to see something

EXPLAIN Example (vertical format)

```
MariaDB [nlwictionary]> EXPLAIN SELECT * FROM page WHERE
page_title = 'Dutch'\G
***** 1. row *****
      id: 1
select_type: SIMPLE
      table: page
        type: ALL
possible_keys: NULL
          key: NULL
     key_len: NULL
       ref: NULL
      rows: 90956
    Extra: Using where
1 row in set (0.00 sec)
```

Use \G for
vertical
formatting

EXPLAIN Example (id)

```
MariaDB [nlwictionary]> EXPLAIN SELECT * FROM page WHERE
page_title = 'Dutch'\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: page
         type: ALL
possible_keys: NULL
           key: NULL
      key_len: NULL
         ref: NULL
        rows: 90956
       Extra: Using where
1 row in set (0.00 sec)
```

***** 1. row *****

id: 1

select_type: SIMPLE

table: page

type: ALL

possible_keys: NULL

key: NULL

key_len: NULL

ref: NULL

rows: 90956

Extra: Using where

1 row in set (0.00 sec)



Indicates hierarchy level, not execution order

EXPLAIN Example (select_type)

```
MariaDB [nlwictionary]> EXPLAIN SELECT * FROM page WHERE
page_title = 'Dutch'\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: page
         type: ALL
possible_keys: NULL
           key: NULL
      key_len: NULL
         ref: NULL
        rows: 90956
       Extra: Using where
1 row in set (0.00 sec)
```



Not a subquery or a UNION

EXPLAIN Example (table)

```
MariaDB [nlwictionary]> EXPLAIN SELECT * FROM page WHERE
page_title = 'Dutch'\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: page
         type: ALL
possible_keys: NULL
           key: NULL
      key_len: NULL
         ref: NULL
        rows: 90956
    Extra: Using where
1 row in set (0.00 sec)
```



Table scanned for
this step

EXPLAIN Example (type)

```
MariaDB [nlwictionary]> EXPLAIN SELECT * FROM page WHERE
page_title = 'Dutch'\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: page
         type: ALL
possible_keys: NULL
           key: NULL
      key_len: NULL
         ref: NULL
        rows: 90956
    Extra: Using where
1 row in set (0.00 sec)
```



All rows are read for
this table (FULL
TABLE SCAN)

EXPLAIN Example (rows)

```
MariaDB [nlwictionary]> EXPLAIN SELECT * FROM page WHERE
page_title = 'Dutch'\G
***** 1. row *****
      id: 1
select_type: SIMPLE
      table: page
      type: ALL
possible_keys: NULL
          key: NULL
key_len: NULL
        ref: NULL
      rows: 90956
    Extra: Using where
1 row in set (0.00 sec)
```

Estimated number
of rows to be read
(all table rows)

How to improve performance?

```
MariaDB [nlwictionary]> SELECT * FROM page  
WHERE page_title = 'Dutch';  
1 row in set (0.11 sec)
```

- Let's add an index on page.page_title:

```
MariaDB [nlwictionary]> ALTER TABLE page ADD INDEX  
page_title (page_title);  
Query OK, 0 rows affected (0.19 sec)  
Records: 0  Duplicates: 0  Warnings: 0
```

Index creation results (type)

```
MariaDB [nlwictionary]> EXPLAIN SELECT * FROM page WHERE
page_title = 'Dutch'\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: page
        type: ref
possible_keys: page_
               key: page_title
      key_len: 257
        ref: const
       rows: 1
     Extra: Using index condition
1 row in set (0.00 sec)
```



type: ref means that an equality comparison will be checked against an index and several results could be returned

Index creation results (possible_keys and key)

```
MariaDB [nlwictionary]> EXPLAIN SELECT * FROM page WHERE
page_title = 'Dutch'\G
***** 1. row *****
      id: 1
select_type: SIMPLE
      table: page
      type: ref
possible_keys: page_title
      key: page_title
    key_len: 257
      ref: const
     rows: 1
    Extra: Using index condition
1 row in set (0.00 sec)
```

index(es) that the optimizer considered potentially useful, and final index chosen

Index creation results (ref)

```
MariaDB [nlwictionary]> EXPLAIN SELECT * FROM page WHERE
page_title = 'Dutch'\G
***** 1. row *****
      id: 1
select_type: SIMPLE
      table: page
      type: ref
possible_keys: page_title
            key: page_title
      key_len: 257
        ref: const
       rows: 1
     Extra: Using index
1 row in set (0.00 sec)
```

Index is compared
with a constant, not
with another table

Index creation results (rows)

```
MariaDB [nlwictionary]> EXPLAIN SELECT * FROM page WHERE
page_title = 'Dutch'\G
***** 1. row *****
      id: 1
select_type: SIMPLE
      table: page
      type: ref
possible_keys: page_title
            key: page_title
      key_len: 257
      ref: const
      rows: 1
    Extra: Using index
1 row in set (0.00 sec)
```

Only 1 row read. In this case, estimation is exact (thanks to index dive)

Index creation results (query time)

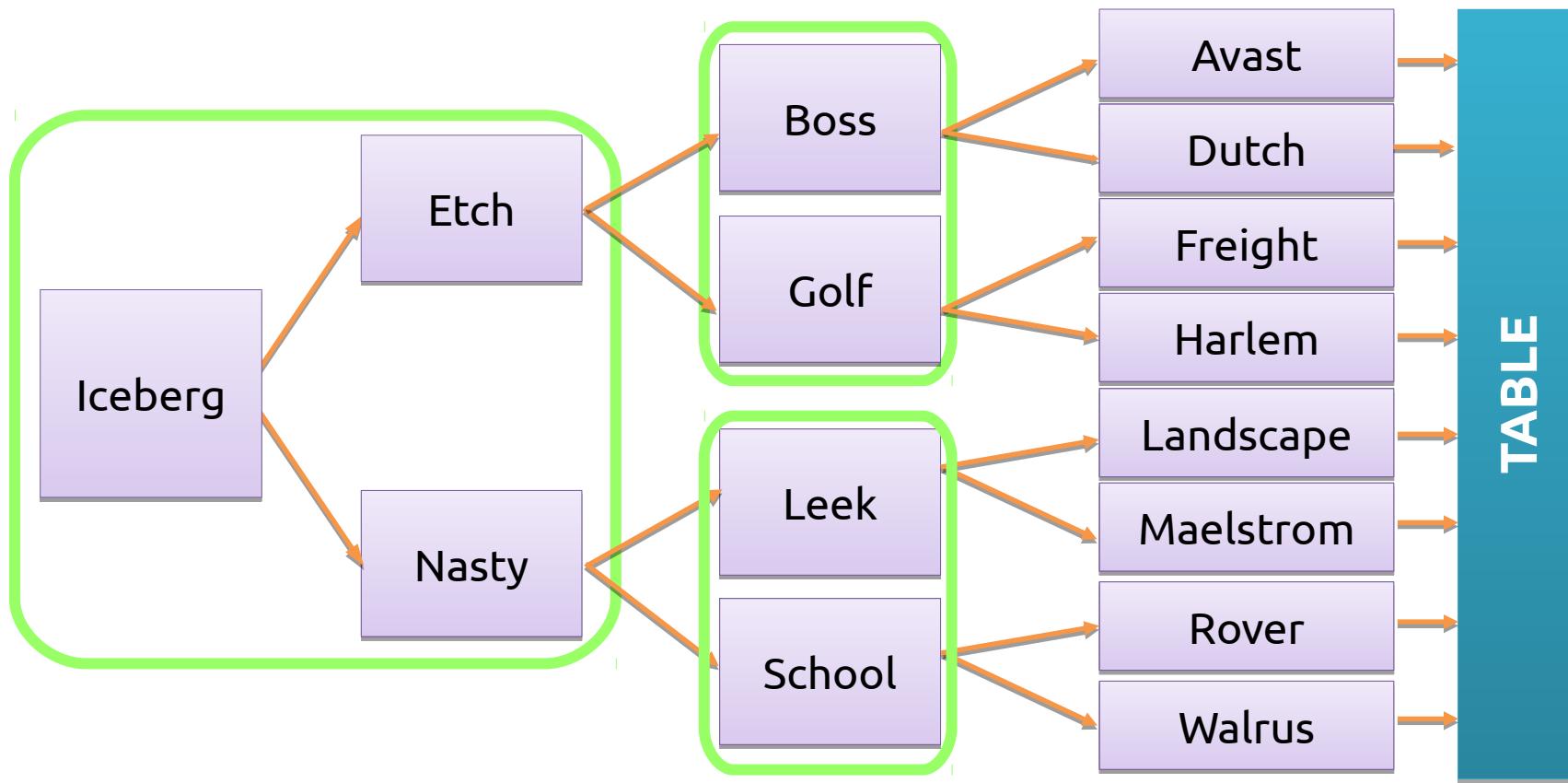
```
MariaDB [nlwictionary]> SELECT * FROM page  
WHERE page_title = 'Dutch';  
1 row in set (0.00 sec)
```

Query time has been reduced substantially

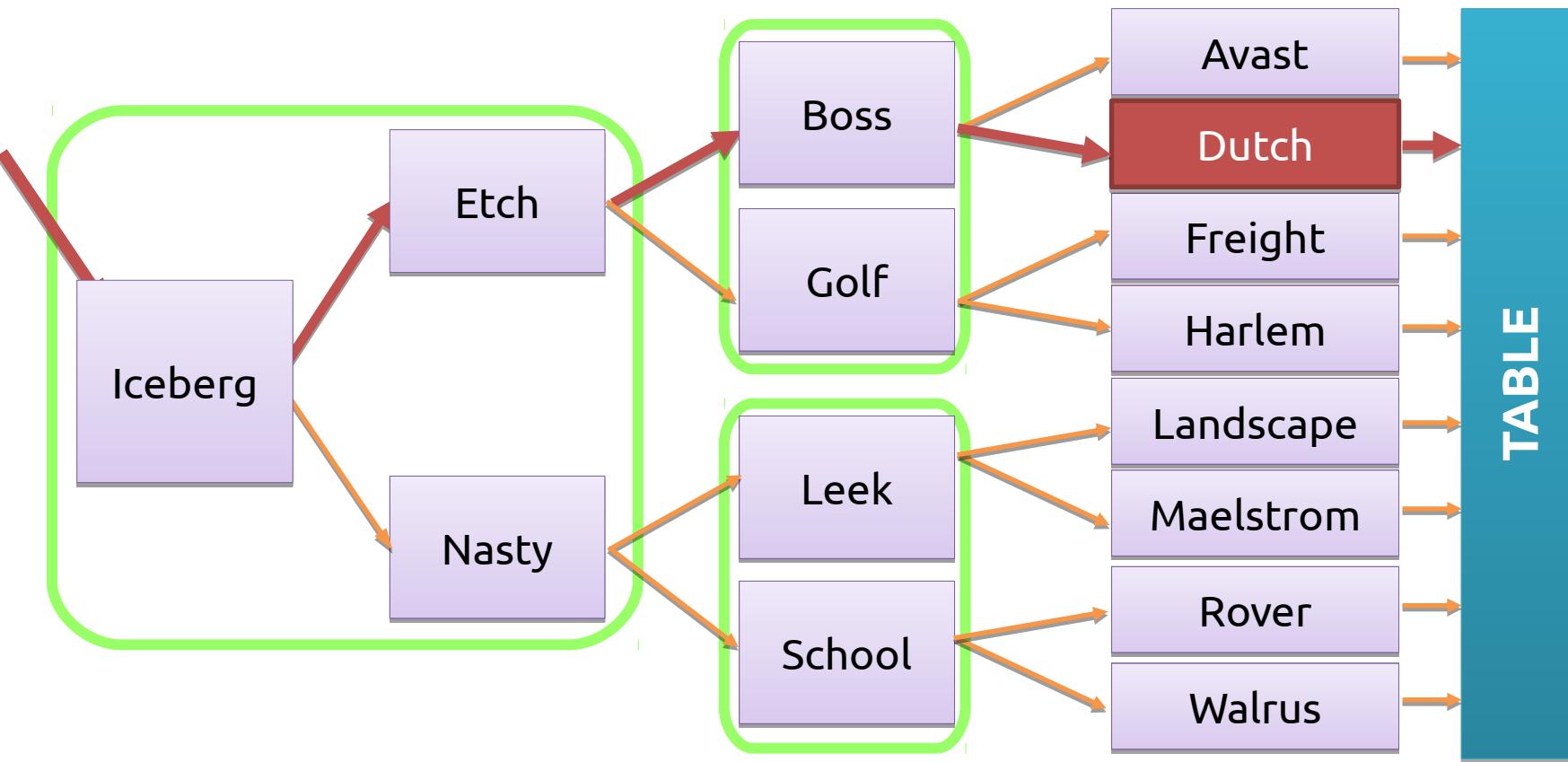
Types of indexes

- BTREE
 - B-TREE in MyISAM, B+TREE in InnoDB
- HASH
 - Only available for MEMORY and NDB
- FULLTEXT
 - Inverted indexes in MyISAM and InnoDB
- SPATIAL
 - RTREEs in MyISAM and InnoDB

Finding “Dutch” with a BTREE



Finding “Dutch” with a BTREE



Do indexes always work? (1/2)

- Can we use an index to make this query faster?

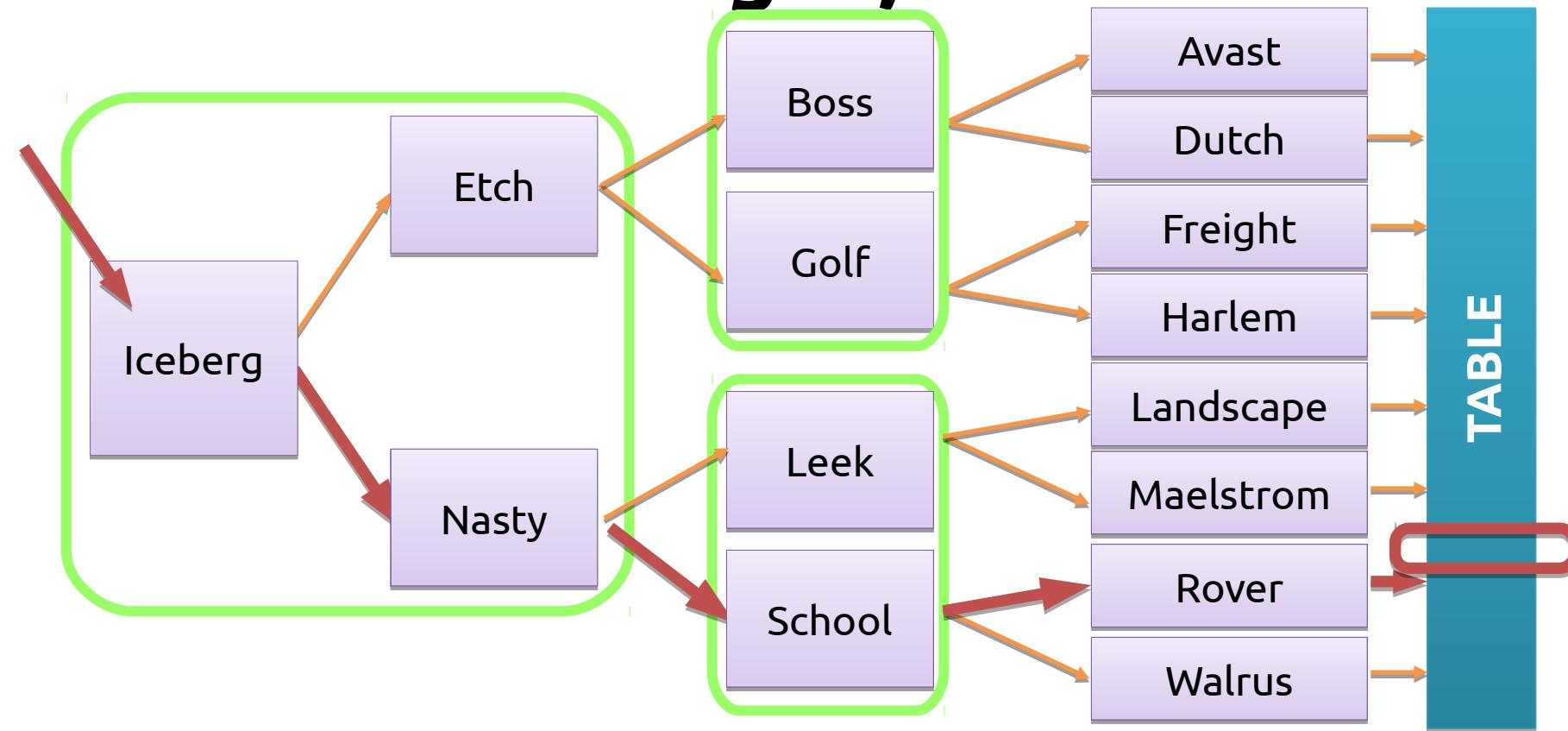
```
SELECT * FROM page WHERE page_title like 'Spa%';
```

It is a range

```
MariaDB [nlwictionary]> EXPLAIN SELECT * FROM page WHERE
page_title like 'Spa%'\G
***** 1. row *****
      id: 1
select_type: SIMPLE
      table: page
      type: range
possible_keys: page_title
            key: page_title
key_len: 257
      ref: NULL
     rows: 94
    Extra: Using index condition
1 row in set (0.00 sec)
```

Despite not being an equality, we can use the index to find the values quickly

BTREE Indexes can be used for ranges, too



Do indexes always work? (2/2)

- What about this other query?

```
SELECT * FROM page WHERE page_title like '%utch';
```

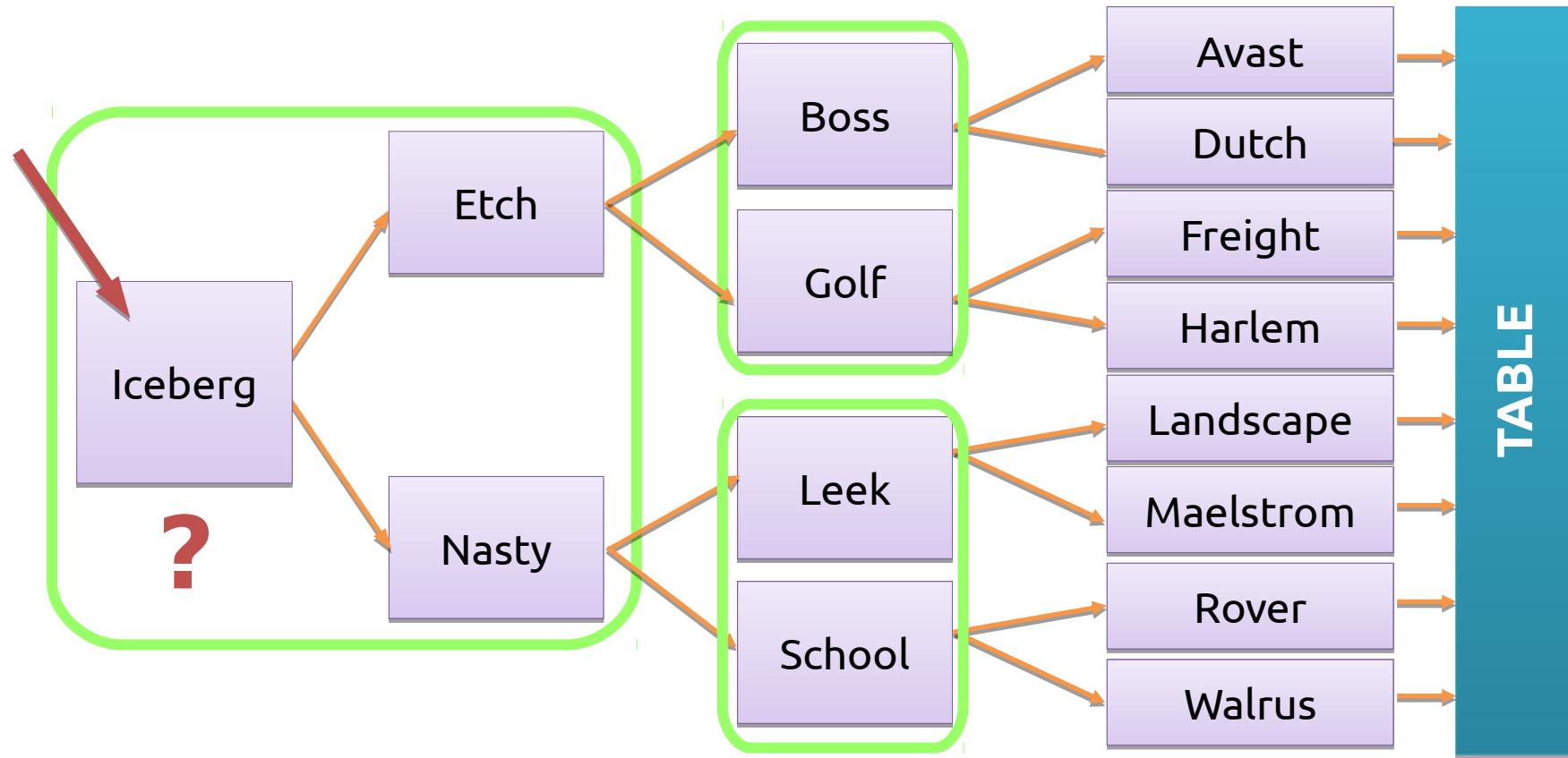
Let's check with EXPLAIN

```
MariaDB [nlwictionary]> EXPLAIN SELECT * FROM page WHERE
page_title like '%utch'\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: page
         type: ALL
possible_keys: NULL
           key: NULL
      key_len: NULL
         ref: NULL
        rows: 93189
    Extra: Using where
1 row in set (0.00 sec)
```



No index can be used for filtering. A full table scan is performed.

BTREE Index



Btree indexes usage

- Filtering
 - Equality (operator '=')
 - Ranges (BETWEEN ... AND, >, <, >=, <=, like 'prefix%')
 - “EXISTS” operators: IN, OR on the same column
- Ordering
 - ORDER BY (indexed columns)
 - GROUP BY (indexed columns)
- Returning values directly from the index
 - Covering index
 - Functions like max(), min(), etc.

type: const

```
MariaDB [nlwictionary]> EXPLAIN SELECT * FROM revision WHERE
rev_id = 2\G
*****
1. row ****
      id: 1
  select_type: SIMPLE
        table: revision
        type: const
possible_keys: PRIMARY
          key: PRIMARY
    key_len: 4
        ref: const
       rows: 1
      Extra:
1 row in set (0.00 sec)
```

'const' is a special case of 'ref', when the index can assure that only 1 results can be returned (equality + primary key or unique key). It is faster.

type: NULL

```
MariaDB [nlwiktioNary]> EXPLAIN SELECT * FROM revision WHERE
rev_id = -1\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: NULL
        type: NULL
possible_keys: NULL
           key: NULL
      key_len: NULL
         ref: NULL
        rows: NULL
    Extra: Impossible WHERE noticed after reading const
tables
1 row in set (0.00 sec)
```

'NULL' is not really a plan,
just an optimization that
allow discarding
immediately impossible
conditions

type: ref_or_null

```
MariaDB [nlwictionary]> EXPLAIN SELECT * FROM user WHERE
user_email_token = '0' OR user_email_token IS NULL\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: user
        type: ref_or_null
possible_keys: user_email_token
          key: user_email_token
    key_len: 33
      ref: const
     rows: 2
    Extra: Using index condition; Using where
1 row in set (0.00 sec)
```

Equivalent to 'ref', but
also has into account
NULL values

type: range (using IN / OR)

```
MariaDB [nlwictionary]> EXPLAIN SELECT * FROM page WHERE
page_title IN ('Dutch', 'English', 'Spanish')\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: page
        type: range
possible_keys: page_title
          key: page_title
     key_len: 257
       ref: NULL
      rows: 4
    Extra: Using index condition
1 row in set (0.00 sec)
```



Despite being a range, its execution is very different from ranges using like, between or inequality operators

Is this a bug? (1/2)

```
MariaDB [nlwictionary]> EXPLAIN SELECT * FROM page WHERE
page_namespace = 2\G
***** 1. row *****
      id: 1
select_type: SIMPLE
      table: page
      type: ref
possible_keys: name_title
            key: name_title
      key_len: 4
      ref: const
     rows: 45
    Extra:
1 row in set (0.00 sec)
```

An index is used to
return pages with ns=2

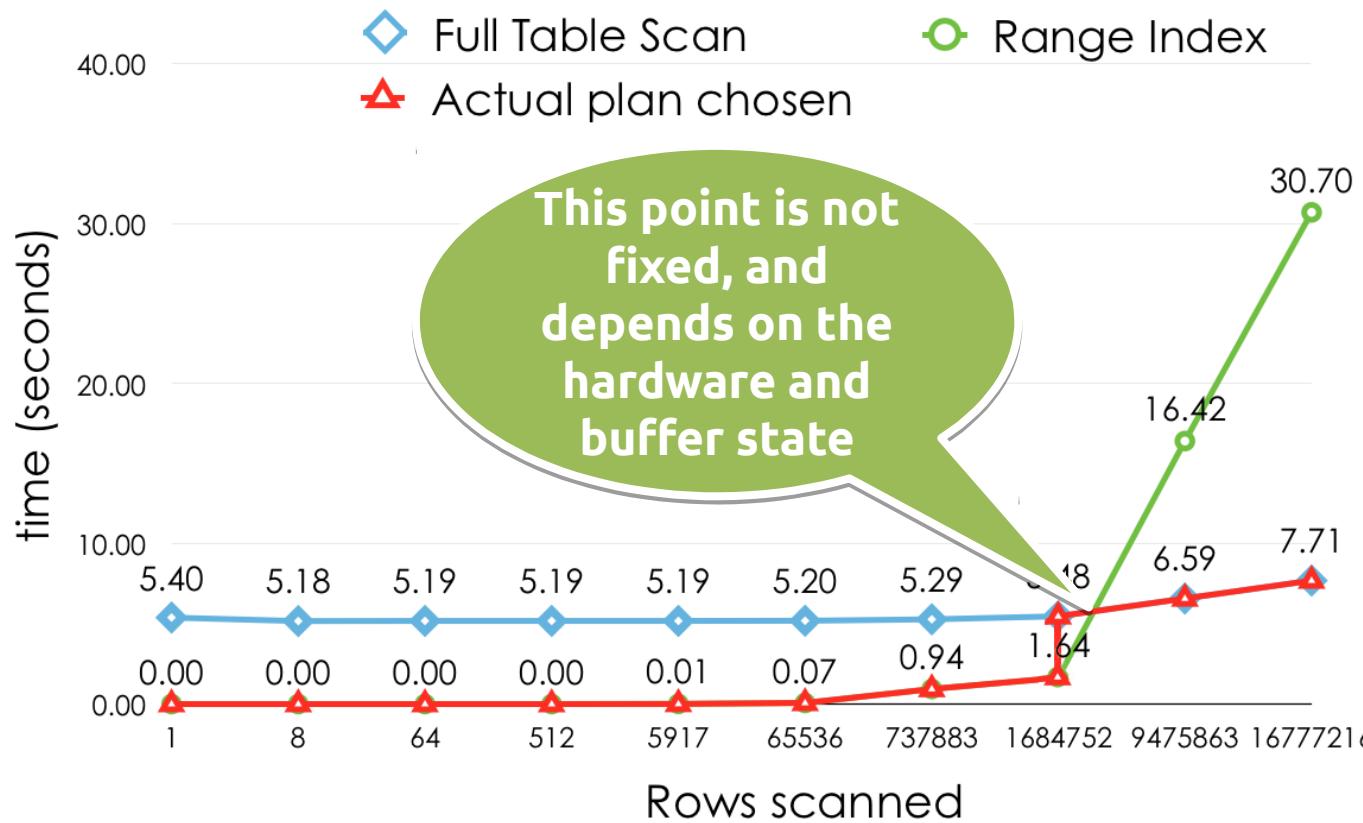
Is this a bug? (2/2)

```
MariaDB [nlwiktio...]> EXPLAIN SELECT * FROM page WHERE
page_namespace = 0\G
***** 1. row *****
      id: 1
select_type: SIMPLE
      table: page
      type: ALL
possible_keys: name_title
      key: NULL
    key_len: NULL
      ref: NULL
     rows: 7493
    Extra: Using where
1 row in set (0.00 sec)
```



The index is not used with ns=0

Using an index is sometimes suboptimal



What index should we add to make this query faster?

```
MariaDB [nlwictionary]> EXPLAIN SELECT * FROM revision WHERE
left(rev_timestamp, 6) = '201509'\G
***** 1. row *****
    id: 1
select_type: SIMPLE
    table: revision
        type: ALL
possible_keys: NULL
        key: NULL
key_len: NULL
        ref: NULL
    rows: 163253
    Extra: Using where
1 row in set (0.00 sec)
```

The table has already an index on rev_timestamp

```
MariaDB [nlwictionary]> SHOW CREATE TABLE revision\G
```

```
***** 1. row *****
```

```
Table: revision
```

```
Create Table: CREATE TABLE `revision` (
```

```
...
```

```
KEY `rev_timestamp`(`rev_timestamp`),
```

We need to rewrite the query

```
MariaDB [nlwictionary]> EXPLAIN SELECT * FROM revision WHERE
rev_timestamp >= '201509' and rev_timestamp < '201510'\G
***** 1. row *****
      id: 1
select_type: SIMPLE
      table: revision
        type: range
possible_keys: rev_timestamp
          key: rev_timestamp
      key_len: 14
        ref: NULL
      rows: 7
    Extra: Using index condition
1 row in set (0.00 sec)
```

This transformation is not trivial or even possible in all cases

```
MariaDB [nlwictionary]> EXPLAIN SELECT * FROM revision WHERE  
substr(rev_timestamp, 5, 2) = '09'\G  
***** 1. row *****
```

```
      id: 1  
select_type: SIMPLE  
      table: revision  
      type: ALL  
possible_keys: NULL  
          key: NULL  
    key_len: NULL  
        ref: NULL  
      Rows: 173154  
    Extra: Using where  
1 row in set (0.00 sec)
```

Can you think a way to improve this query?

Indexes for Ordering

```
MariaDB [nlwiktio...]> EXPLAIN SELECT * FROM page ORDER BY
page_touched DESC LIMIT 10\G
***** 1. row *****
      id: 1
select_type: SIMPLE
      table: page
      type: ALL
possible_keys: NULL
          key: NULL
key_len: NULL
      ref: NULL
     rows: 8720
    Extra: Using filesort
1 row in set (0.00 sec)
```

"Using filesort" indicates that an ordering is needed before returning the results

If that is frequent, we can create an index on page_touched...

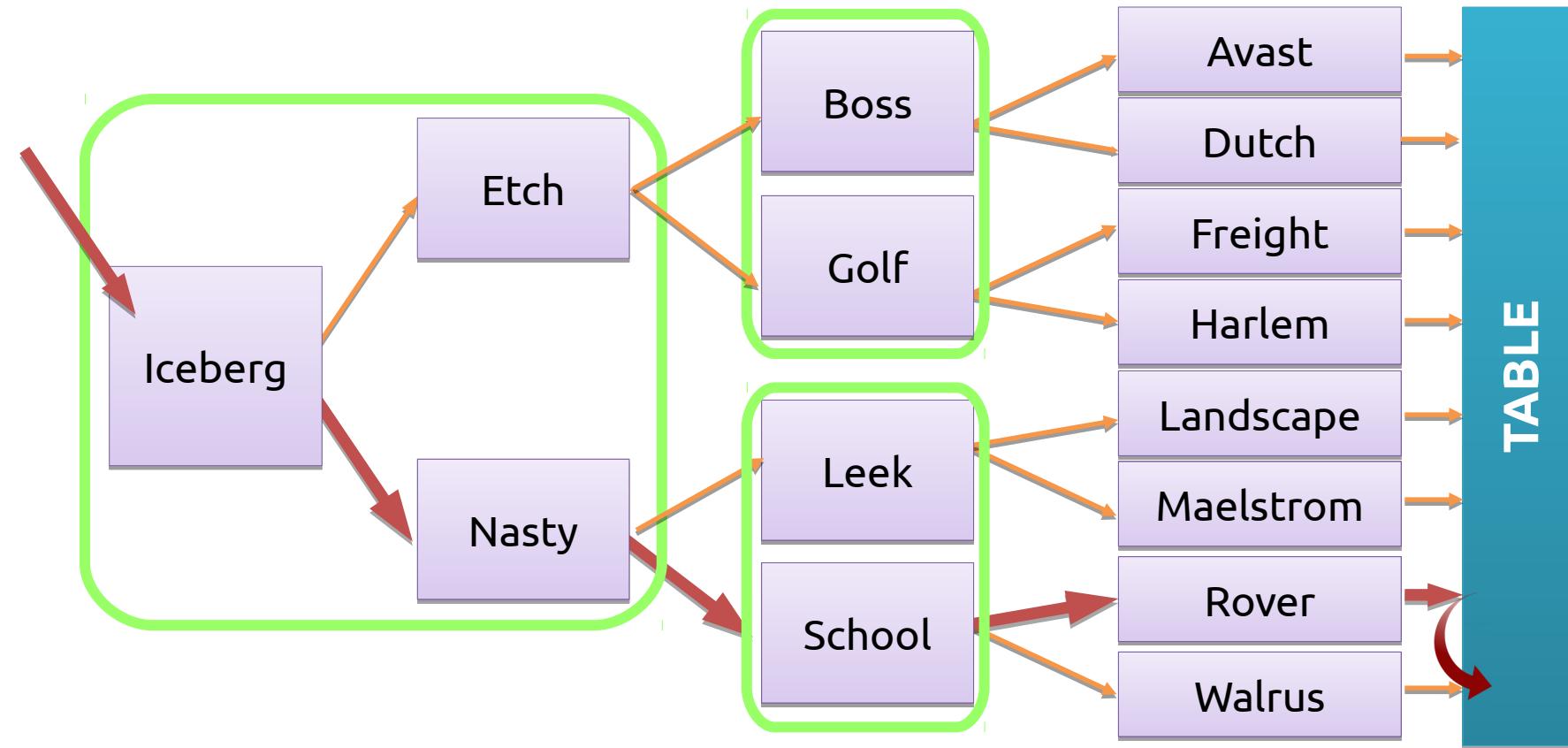
```
[nlwiktictionary]> ALTER TABLE page ADD INDEX page_page_touched(page_touched);  
Query OK, 0 rows affected (0.30 sec)  
Records: 0  Duplicates: 0  Warnings: 0
```

```
[nlwiktictionary]> EXPLAIN SELECT * FROM page ORDER BY page_touched DESC LIMIT 10\G  
*****  
1. row *****  
      id: 1  
select_type: SIMPLE  
      table: page  
      type: index  
possible_keys: NULL  
      key: page_page_touched  
key_len: 14  
      ref: NULL  
      rows: 10  
     Extra:  
1 row in set (0.00 sec)
```

The index does not produce any advantage for filtering

However, it is very effective by helping avoiding the sort phase

It can return data in index order faster



Indexes and GROUP BY (no indexes)

```
MariaDB [nlwictionary]> EXPLAIN SELECT rev_page, count(*) FROM revision IGNORE INDEX(rev_page_id, page_timestamp, page_user_timestamp) GROUP BY rev_page\G
```

```
***** 1. row *****
```

```
      id: 1
select_type: SIMPLE
      table: revision
        type: ALL
possible_keys: NULL
        key: NULL
    key_len: NULL
        ref: NULL
       rows: 201094
     Extra: Using temporary; Using filesort
1 row in set (0.00 sec)
```

Without indexes, a temporary table is created to order results

Trick: ORDER BY NULL avoids filesort

```
MariaDB [nlwiktio...]> EXPLAIN SELECT * FROM revision GROUP BY substr(rev_timestamp, 5, 2) = '09'\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: revision
       type: ALL
possible_keys: NULL
         key: NULL
    key_len: NULL
       ref: NULL
      rows: 196824
     Extra: Using temporary; Using filesort
1 row in set (0.00 sec)
```

There is no good index in this case

```
MariaDB [nlwiktio...]> EXPLAIN SELECT * FROM revision GROUP BY substr(rev_timestamp, 5, 2) = '09' ORDER BY NULL\G
***** 1. row *****
...
      rows: 196871
     Extra: Using temporary
1 row in set (0.00 sec)
```

The advantage is not too big, but it avoids the filesort

Indexes and GROUP BY (rev_page_id)

```
MariaDB [nlwiktio...]> EXPLAIN SELECT rev_page, count(*) FROM revision GROUP BY rev_page\G
```

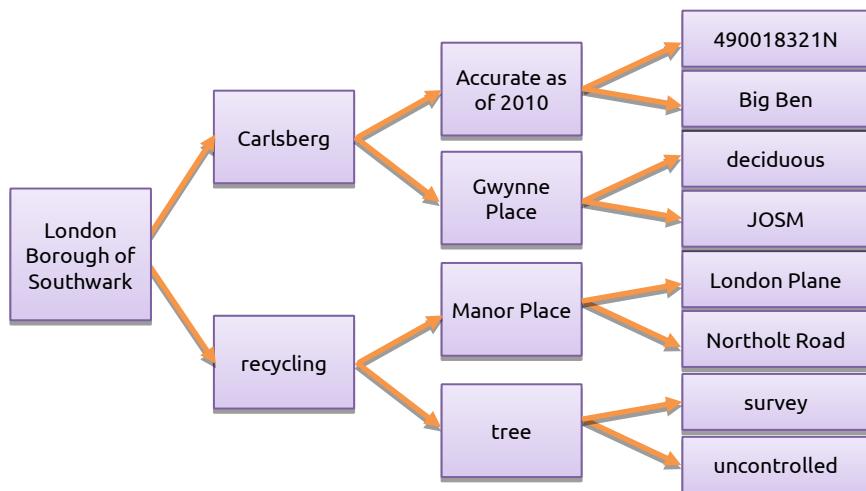
```
***** 1. row *****
```

```
    id: 1
select_type: SIMPLE
    table: revision
      type: index
possible_keys: NULL
      key: rev_page_id
    key_len: 8
      ref: NULL
     rows: 192388
    Extra: Using index
1 row in set (0.00 sec)
```

The index does not produce any advantage for filtering (there is no WHERE clause)

However, thanks to it we avoid a sort and a temporary table

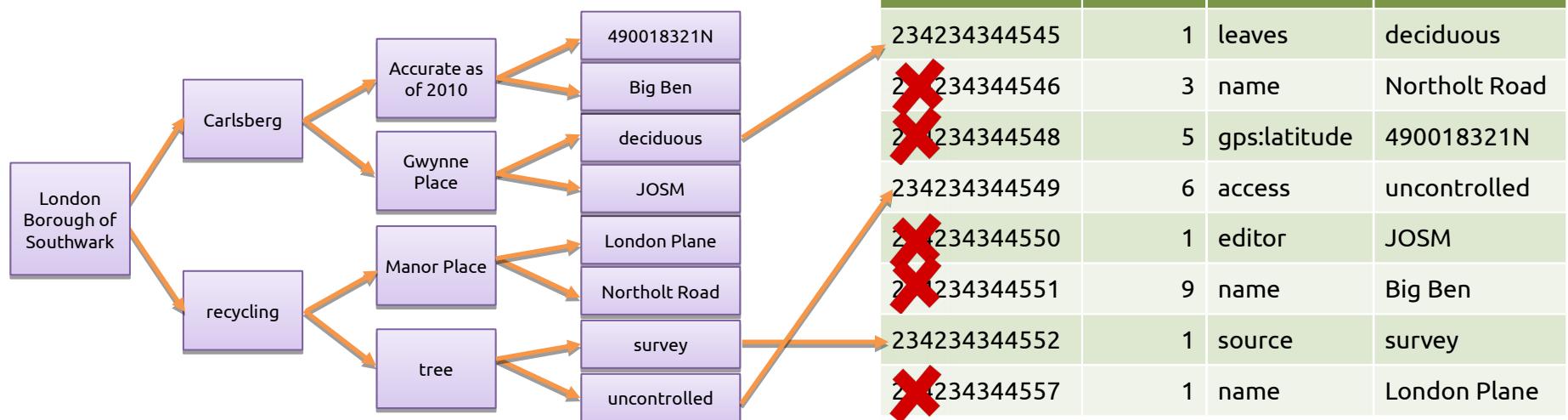
type: index, loose index scan and covering index (1/3)



node_id	version	k	v
234234344545	1	leaves	deciduous
234234344546	3	name	Northolt Road
234234344548	5	gps:latitude	490018321N
234234344549	6	access	uncontrolled
234234344550	1	editor	JOSM
234234344551	9	name	Big Ben
234234344552	1	source	survey
234234344557	1	name	London Plane

With 'type:index', all rows are read in index order (full index scan)

type: index, loose index scan and covering index (2/3)

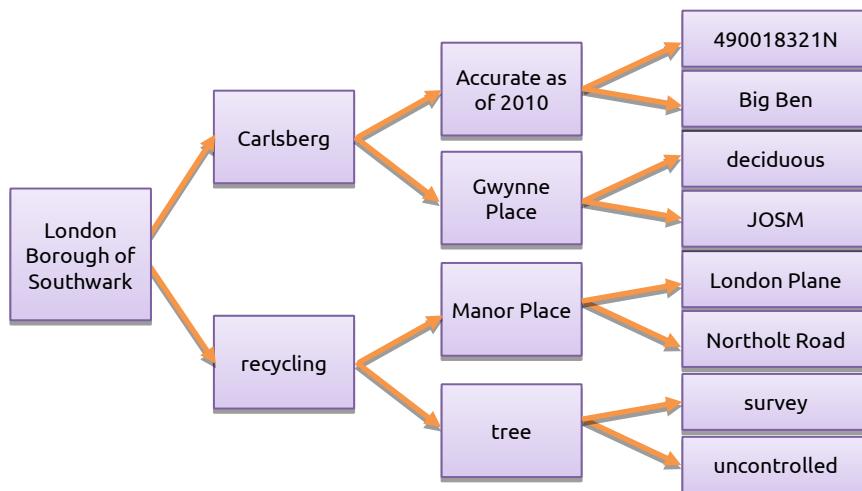


If we have in addition 'Using index for group-by' we have the loose index scan optimization

Loose Index Scan Example

```
MariaDB [nlwictionary]> EXPLAIN SELECT rev_page,
max(rev_timestamp) FROM revision GROUP BY rev_page\G
***** 1. row *****
      id: 1
select_type: SIMPLE
      table: revision
        type: range
possible_keys: NULL
          key: page_timestamp
    key_len: 4
      ref: NULL
     rows: 9769
    Extra: Using index for group-by
1 row in set (0.00 sec)
```

type: index, loose index scan and covering index (3/3)



node	version	v
234234344548	1	deciduous
234234344549	1	Northolt Road
234234344549	1	clude
234234344549	1	490018321N
234234344549	1	uncontrolled
234234344549	1	JOSM
234234344549	1	Big Ben
234234344549	1	survey
234234344549	1	London Plane

If we have in addition 'Using index' we have the covering index optimization

Covering Index Example (1/3)

```
MariaDB [nlwiktictionary]> ALTER TABLE revision DROP INDEX rev_page_id, drop index page_timestamp,  
drop index page_user_timestamp;  
Query OK, 0 rows affected (0.05 sec)  
Records: 0  Duplicates: 0  Warnings: 0
```

Let's start with no indexes

```
MariaDB [nlwiktictionary]> EXPLAIN SELECT count(DISTINCT rev_user) FROM revision WHERE rev_page =  
790\G  
*****  
      1. row *****  
      id: 1  
 select_type: SIMPLE  
       table: revision  
        type: ALL  
possible_keys: NULL  
         key: NULL  
    key_len: NULL  
       ref: NULL  
      rows: 218384  
     Extra: Using where  
1 row in set (0.00 sec)
```

```
MariaDB [nlwiktictionary]> SELECT count(DISTINCT rev_user) FROM revision WHERE rev_page = 790\G  
*****  
      1. row *****  
count(DISTINCT rev_user): 1  
1 row in set (0.06 sec)
```

Covering Index Example (2/3)

```
MariaDB [nlwiktictionary]> ALTER TABLE revision ADD INDEX revision_rev_page(rev_page);
Query OK, 0 rows affected (0.44 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
MariaDB [nlwiktictionary]> EXPLAIN SELECT count(DISTINCT rev_user) FROM revision WHERE rev_page =
790\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: revision
       type: ref
possible_keys: revision_rev_page
         key: revision_rev_page
      key_len: 4
        ref: const
       rows: 4863
      Extra:
1 row in set (0.00 sec)
```

Adding an index on rev_page
increases the speed due to
improved filtering

```
MariaDB [nlwiktictionary]> SELECT count(DISTINCT rev_user) FROM revision WHERE rev_page = 790\G
***** 1. row *****
count(DISTINCT rev_user): 1
1 row in set (0.01 sec)
```

Covering Index Example (3/3)

```
MariaDB [nlwiktictionary]> ALTER TABLE revision ADD INDEX revision_rev_page_rev_user(rev_page,
rev_user);
```

```
Query OK, 0 rows affected (1.48 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
MariaDB [nlwiktictionary]> EXPLAIN SELECT count(DISTINCT rev_user) FROM revision WHERE rev_page =
790\G
```

```
***** 1. row *****
    id: 1
  select_type: SIMPLE
        table: revision
       type: ref
possible_keys: revision_rev_page,revision_rev_page_rev_
               key: revision_rev_page_rev_user
      key_len: 4
        ref: const
       rows: 4863
     Extra: Using index
1 row in set (0.00 sec)
```

rev_page, rev_user does not increase the index selectiveness, but allow to return results directly from the index

```
MariaDB [nlwiktictionary]> SELECT count(DISTINCT rev_user) FROM revision WHERE rev_page = 790\G
***** 1. row *****
count(DISTINCT rev_user): 1
1 row in set (0.00 sec)
```

The speed difference can be huge

Query Optimization: From 0 to 10 (and up to 5.7)

MULTI-COLUMN INDEXES

In many cases, conditions are applied on more than one column

```
MariaDB [nlwictionary]> EXPLAIN SELECT * FROM revision WHERE  
rev_page = 790 and rev_timestamp < '2008'\G  
*****  
1. row *****
```

```
      id: 1  
select_type: SIMPLE  
      table: revision  
      type: ALL  
possible_keys: NULL  
          key: NULL  
key_len: NULL  
          ref: NULL  
      rows: 686822  
Extra: Using where  
1 row in set (0.00 sec)
```

Assuming there were no previously created indexes, which would the optimal one be?

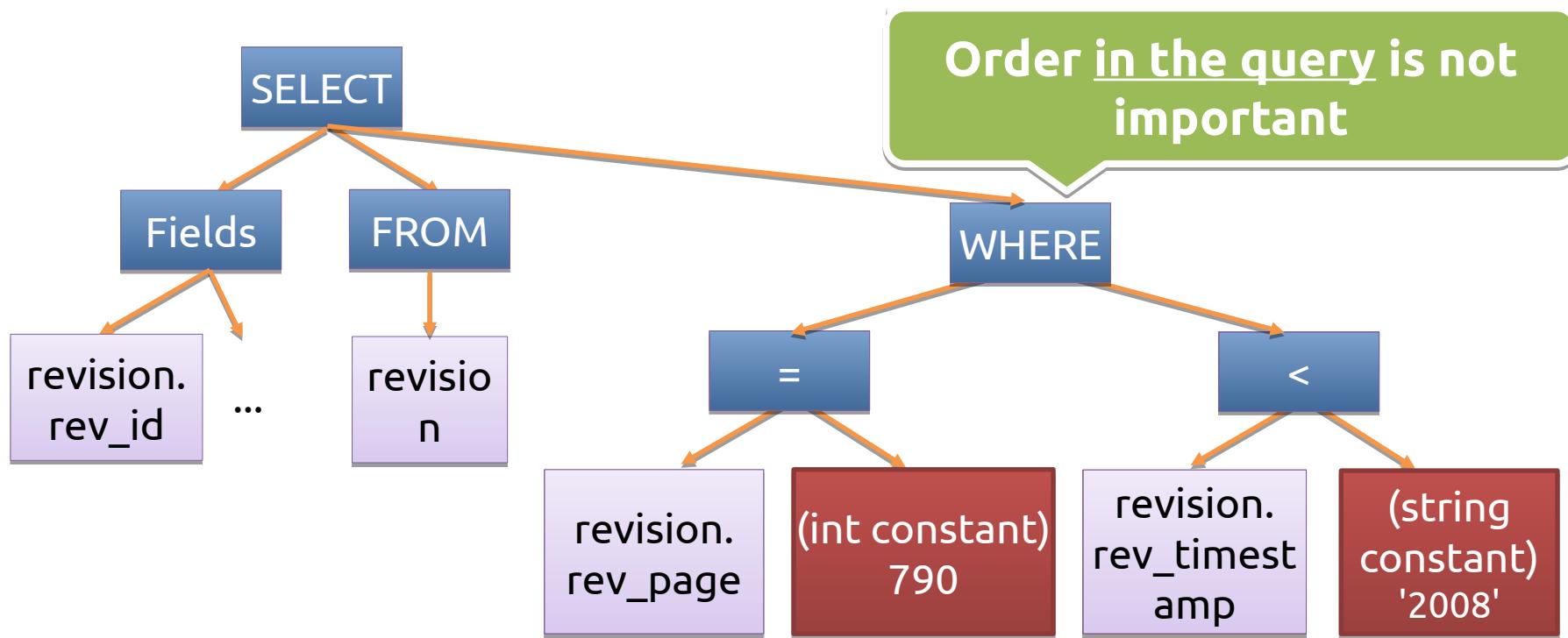
Options for indexes

- 1 index on column (`rev_page`)
- 1 index on column (`rev_timestamp`)
- 2 indexes, 1 on (`rev_page`) and another on (`rev_timestamp`)
- 1 multi-column index on (`rev_page, rev_timestamp`)
- 1 multi-column index on (`rev_timestamp, rev_page`)

Are these last 2 different from each other?
Would it depend on the query order?

A brief reminder about query parsing

```
SELECT * FROM revision WHERE rev_page = 790 and rev_timestamp < '2008'
```



Index on (rev_page)

```
MariaDB [nlwiktictionary]> ALTER TABLE revision ADD INDEX rev_page (rev_page);  
Query OK, 0 rows affected (2.31 sec)  
Records: 0  Duplicates: 0  Warnings: 0
```

```
MariaDB [nlwiktictionary]> EXPLAIN SELECT * FROM revision WHERE rev_page = 790 and  
rev_timestamp < '2008'\G  
*****  
      1. row *****  
      id: 1  
  select_type: SIMPLE  
        table: revision  
        type: ref  
possible_keys: rev_page  
        key: rev_page  
    key_len: 4  
        ref: const  
      rows: 4863  
    Extra: Using where  
1 row in set (0.00 sec)
```

Query time improves significantly with this index

Less rows are scanned

Adding (rev_timestamp)

```
MariaDB [nlwictionary]> ALTER TABLE revision ADD INDEX rev_timestamp  
(rev_timestamp);  
Query OK, 0 rows affected (1.77 sec)  
Records: 0 Duplicates: 0 Warnings: 0
```

```
MariaDB [nlwictionary]> EXPLAIN SELECT * FROM revision WHERE rev_page = 790  
and rev_timestamp < '2008'\G  
***** 1. row *****
```

```
    id: 1  
select_type: SIMPLE  
    table: revision  
        type: ref  
possible_keys: rev_page,rev_timestamp  
        key: rev_page  
key_len: 4  
        ref: const  
        rows: 4863  
        Extra: Using where  
1 row in set (0.01 sec)
```

In general, only one index can
be used per table access

rev_page is preferred
over rev_timestamp

Forcing the use of (rev_timestamp)

```
MariaDB [nlwictionary]> EXPLAIN SELECT * FROM revision FORCE INDEX(rev_timestamp) WHERE rev_page = 790 and rev_timestamp < '2008'\G
```

Forcing the index is worse than type:ALL!

```
id: 1
select_type: SIMPLE
table: revision
type: range
possible_keys: rev_timestamp
key: rev_timestamp
key_len: 14
ref: NULL
rows: 343411
Extra: Using index condition; Using where
1 row in set (0.00 sec)
```

It is a range access

A lot more accessed rows

Adding (rev_page, rev_timestamp)

```
MariaDB [nlwictionary]> ALTER TABLE revision ADD INDEX  
rev_page_rev_timestamp(rev_page, rev_timestamp);  
Query OK, 0 rows affected (1.59 sec)  
Records: 0 Duplicates: 0 Warnings: 0
```

```
MariaDB [nlwictionary]> EXPLAIN SELECT * FROM revision WHERE rev_page = 790  
and rev_timestamp < '2008'\G  
***** 1. row *****  
      id: 1  
select_type: SIMPLE  
      table: revision  
      type: range  
possible_keys: rev_page,rev_timestamp,rev_page_rev_timestamp  
      key: rev_page_rev_timestamp  
key_len: 18  
      ref: NULL  
      rows: 1048  
      Extra: Using index condition  
1 row in set (0.00 sec)
```

Reduced number of
rows scanned

Is (rev_timestamp, rev_page) a better option?

```
MariaDB [nlwiktio...]> ALTER TABLE revision ADD INDEX rev_timestamp_rev_page  
(rev_timestamp, rev_page);  
Query OK, 0 rows affected (1.76 sec)  
Records: 0 Duplicates: 0 Warnings: 0
```

```
MariaDB [nlwiktio...]> EXPLAIN SELECT * FROM revision WHERE rev_page = 790 and  
rev_timestamp < '2008'\G  
*****  
 1. row *****  
      id: 1  
 select_type: SIMPLE  
       table: revision  
        type: range  
possible_keys: rev_page,rev_timestamp,rev_page_rev_timestamp,rev_timestamp_rev_page  
         key: rev_page_rev_timestamp  
    key_len: 18  
       ref: NULL  
      rows: 1048  
     Extra: Using index condition  
1 row in set (0.00 sec)
```

Previous index is still preferred, why?

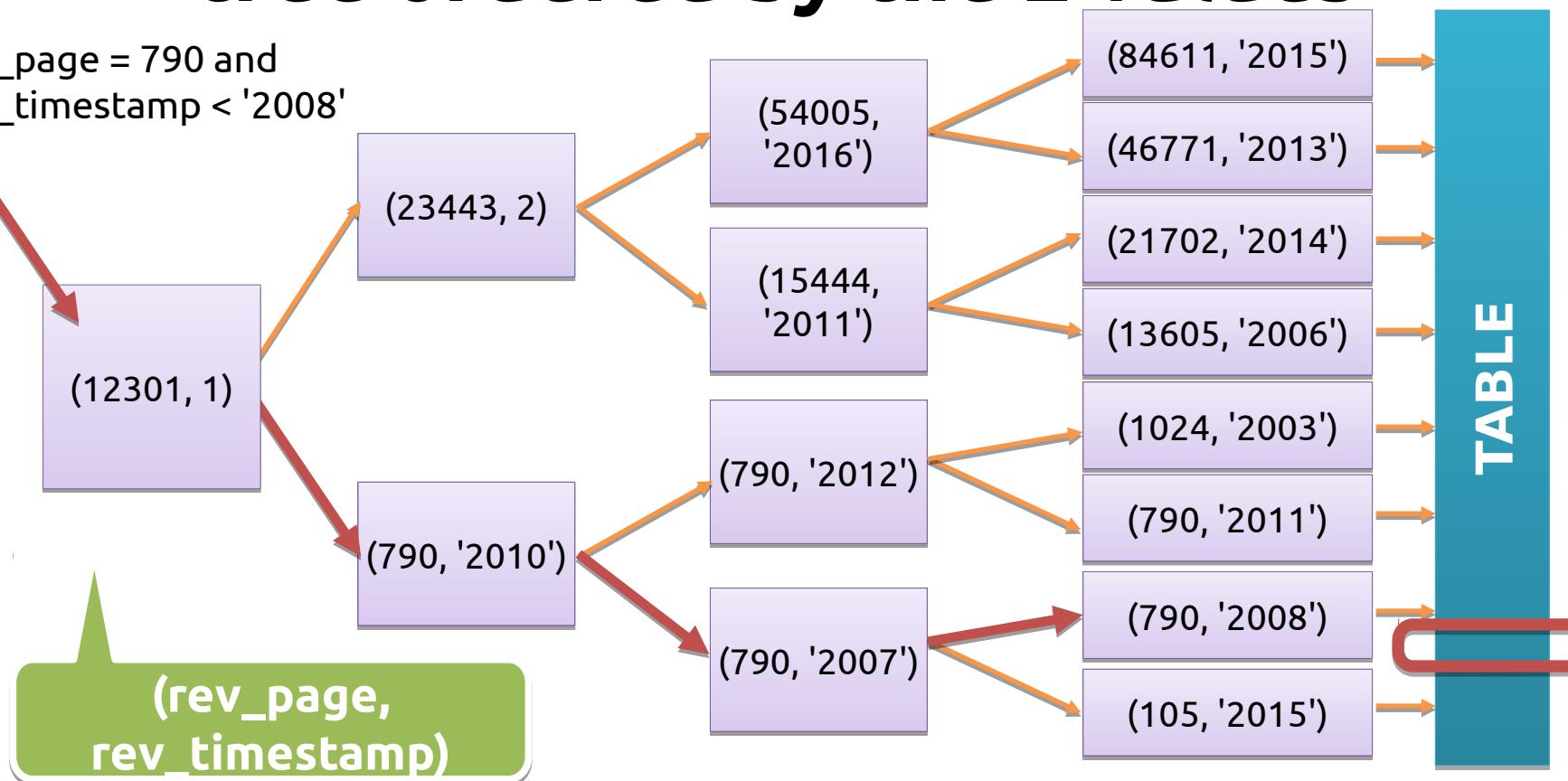
Forcing (rev_timestamp, rev_page)

```
MariaDB [nlwictionary]> EXPLAIN SELECT * FROM revision FORCE
INDEX(rev_timestamp_rev_page) WHERE rev_page = 790 and
rev_timestamp < '2008'\G
***** 1. row *****
      id: 1
select_type: SIMPLE
      table: revision
        type: range
possible_keys: rev_timestamp_rev_page
          key: rev_timestamp_rev_page
key_len: 18
      ref: NULL
     rows: 343411
    Extra: Using index condition
1 row in set (0.00 sec)
```

Only the first column is being
used effectively for filtering

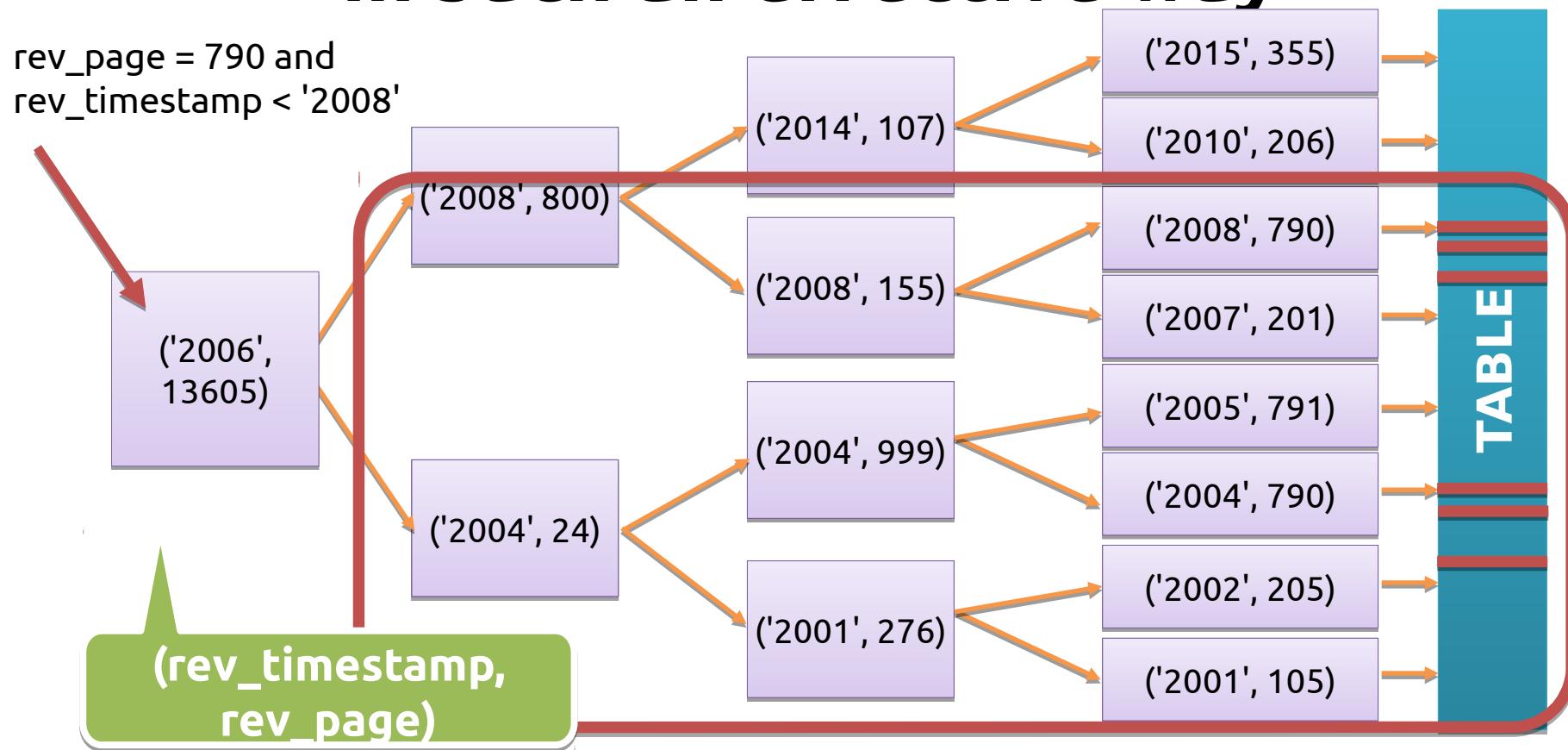
A compound index produces a single tree ordered by the 2 values

`rev_page = 790 and
rev_timestamp < '2008'`



The alternative index cannot be used in such an effective way

rev_page = 790 and
rev_timestamp < '2008'



Order and column selection

- Range access using $>$, $<$, $>=$, $<=$, BETWEEN can only be filtered once effectively, at the end of an index
- When selecting indexes, prefer columns with high cardinality (very selective)
 - The optimal index can depend on the constants used

Use of "Handler_*" statistics

- They are post-execution statistics at row level
 - Unlike EXPLAIN's "rows" column, they are exact, not a guess
 - They allow to compare query execution performance in a deterministic way, independently of the execution time

"Handler" Stats (indexed)

```
mysql> SHOW SESSION STATUS like 'Hand%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| Handler_commit | 80    |
| Handler_delete | 0     |
| Handler_discover | 0     |
| Handler_external_lock | 166   |
| Handler_mrr_init | 0     |
| Handler_prepare | 0     |
| Handler_read_first | 23    |
| Handler_read_key | 736212 |
| Handler_read_last | 0     |
| Handler_read_next | 22208001 |
| Handler_read_prev | 0     |
| Handler_read_rnd | 665215  |
| Handler_read_rnd_next | 14223297 |
| Handler_rollback | 0     |
| Handler_savepoint | 0     |
| Handler_savepoint rollback | 0     |
| Handler_update | 66970   |
| Handler_write | 2869409 |
+-----+-----+
18 rows in set (0.00 sec)
```

Number of times that the first entry of an index was read. It may indicate the number of full index scans

Number of time a row has been retrieved using an index

Next row has been requested in index order (typical for index scans or ranges)

"Handler" Stats (unindexed)

```
mysql> SHOW SESSION STATUS like 'Hand%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| Handler_commit | 80    |
| Handler_delete | 0     |
| Handler_discover | 0     |
| Handler_external_lock | 166   |
| Handler_mrr_init | 0     |
| Handler_prepare | 0     |
| Handler_read_first | 23    |
| Handler_read_key | 736212 |
| Handler_read_last | 0     |
| Handler_read_next | 22208001 |
| Handler_read_prev | 0     |
| Handler_read_rnd | 665215  |
| Handler_read_rnd_next | 14223297 |
| Handler_rollback | 0     |
| Handler_savepoint | 0     |
| Handler_savepoint rollback | 0     |
| Handler_update | 66970  |
| Handler_write | 2869409 |
+-----+-----+
18 rows in set (0.00 sec)
```

A row has been requested in a specific position (typical for joins or order by without indexes)

Request tp read the next row in “table order” (typical for full table scans)

Insertions in SELECTS may indicate temporary tables

Comparing statistics of the previous indexes (no indexes)

```
MariaDB [nlwiktionsary]> FLUSH STATUS;  
Query OK, 0 rows affected (0.00 sec)
```

```
MariaDB [nlwiktionsary]> SELECT * FROM revision IGNORE INDEX(rev_page, rev_timestamp, rev_page_rev_timestamp,  
rev_timestamp_rev_page) WHERE rev_page = 790 and rev_timestamp < '2008';  
1049 rows in set (0.58 sec)
```

```
MariaDB [nlwiktionsary]> SHOW STATUS like 'Handler%';
```

Variable_name	Value
Handler_commit	1
Handler_delete	0
Handler_read_first	0
Handler_read_key	0
Handler_read_last	0
Handler_read_next	0
Handler_read_prev	0
Handler_read_rnd	0
Handler_read_rnd_deleted	0
Handler_read_rnd_next	822222
Handler_rollback	0
Handler_update	0
Handler_write	0

Typical result for a full table scan

Index on (rev_page)

```
MariaDB [nlwiktio...]> SHOW STATUS like 'Handler%';
```

Variable_name	Value
Handler_commit	1
Handler_delete	0
...	
Handler_read_first	0
Handler_read_key	1
Handler_read_last	0
Handler_read_next	4864
Handler_read_prev	0
Handler_read_rnd	0
Handler_read_rnd_deleted	0
Handler_read_rnd_next	0
Handler_rollback	0
...	
Handler_update	0
Handler_write	0

25 rows in set (0.01 sec)

Using the index, request the first row with rev_page=790

Then, scan them one by one in index order

Index on (rev_timestamp)

```
MariaDB [nlwiktio...]> SHOW STATUS like 'Hand%';
```

Variable_name	Value
Handler_commit	1
Handler_delete	0
...	
Handler_read_first	0
Handler_read_key	1
Handler_read_last	0
Handler_read_next	199155
Handler_read_prev	0
Handler_read_rnd	0
Handler_read_rnd_deleted	0
Handler_read_rnd_next	0
Handler_rollback	0
...	
Handler_update	0
Handler_write	0

Let's ignore ICP for now

Using the index, request the first row where rev_timestamp<2008

Then, scan them one by one in index order (more are matched)

25 rows in set (0.00 sec)

Index on (rev_page, rev_timestamp)

```
MariaDB [nlwiktio...]> SHOW STATUS like 'Hand%';
```

Variable_name	Value
Handler_commit	1
Handler_delete	0
...	
Handler_read_first	0
Handler_read_key	1
Handler_read_last	0
Handler_read_next	1049
Handler_read_prev	0
Handler_read_rnd	0
Handler_read_rnd_deleted	0
Handler_read_rnd_next	0
Handler_rollback	0
...	
Handler_update	0
Handler_write	0

25 rows in set (0.00 sec)

With both conditions covered, we can find the actual first row that matches the condition using the index

Rows scanned == Rows returned

Index on (rev_timestamp, rev_page), no ICP

```
MariaDB [nlwiktio...]> SHOW STATUS like 'Handler%';
```

Variable_name	Value
Handler_commit	1
Handler_delete	0
...	
Handler_read_first	0
Handler_read_key	1
Handler_read_last	0
Handler_read_next	199155
Handler_read_prev	0
Handler_read_rnd	0
Handler_read_rnd_deleted	0
Handler_read_rnd_next	0
Handler_rollback	0
...	
Handler_update	0
Handler_write	0

25 rows in set (0.00 sec)

Assuming no ICP, exact same results than with (rev_timestamp). The extra column does not help. Also, EXPLAIN's row count was very off.

Redundant Indexes

- Creating all 4 previous indexes in production is not a great idea
 - "Left-most index prefix" allows, for example (rev_page, rev_timestamp) doing everything you can do with (rev_page)
 - If two indexes have equal selectivity, MySQL chooses the shortest one

"Left-most index" Example

```
MariaDB [nlwictionary]> EXPLAIN SELECT * FROM revision FORCE
INDEX(rev_page_rev_timestamp) WHERE rev_page = 790\G
***** 1. row *****
      id: 1
select_type: SIMPLE
      table: revision
        type: ref
possible_keys: rev_page_rev_timestamp
          key: rev_page_rev_time
      key_len: 4
        ref: const
       rows: 4863
      Extra:
1 row in set (0.00 sec)
```

Only the first column is used

Duplicate Indexes

- It is very easy to create indexes with the same exact definition (same columns and ordering)
 - Set a convention for index naming (e.g tablename_column1_column2_idx) – MySQL does not allow 2 indexes with the same identifier
 - Since MySQL 5.6, a warning is thrown if a duplicate index is created

pt-duplicate-index-checker

```
$ pt-duplicate-key-checker h=127.0.0.1,P=5621,u=msandbox,p=msandbox  
[...]  
# rev_timestamp is a left-prefix of rev_timestamp_rev_page  
# Key definitions:  
#   KEY `rev_timestamp` (`rev_timestamp`),  
#   KEY `rev_timestamp_rev_page` (`rev_timestamp`, `rev_page`)  
# Column types:  
#   `rev_timestamp` binary(14) not null default '\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0'  
#   `rev_page` int(10) unsigned not null  
# To remove this duplicate index, execute:  
ALTER TABLE `nlwiktionary`.`revision` DROP INDEX `rev_timestamp`;  
  
# rev_page is a left-prefix of rev_page_rev_timestamp  
# Key definitions:  
#   KEY `rev_page` (`rev_page`),  
#   KEY `rev_page_rev_timestamp` (`rev_page`, `rev_timestamp`),  
# Column types:  
#   `rev_page` int(10) unsigned not null  
#   `rev_timestamp` binary(14) not null default '\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0'  
# To remove this duplicate index, execute:  
ALTER TABLE `nlwiktionary`.`revision` DROP INDEX `rev_page`;  
  
# ##### Summary of indexes  
#####  
# Size Duplicate Indexes 15478317  
# Total Duplicate Indexes 4  
# Total Indexes 285
```

Simple tool to check redundant and duplicate indexes

"OR"-style conditions over the same column

```
MariaDB [nlwiktio...]> EXPLAIN SELECT * FROM revision WHERE  
rev_page = 790 OR rev_page = 795 OR rev_page = 1024\\G  
***** 1 row *****
```

```
      id: 1  
select_type: SIMPLE  
      table: revision  
        type: range  
possible_keys: rev_page,  
          key: rev_page  
key_len: 4  
      ref: NULL  
    rows: 4890  
  Extra: Using index condition; Using where  
1 row in set (0.01 sec)
```

Equivalent to:
**SELECT * FROM revision WHERE
rev_page IN (790, 795, 1024)**

Handlers on "IN" / "OR" conditions over the same column

Variable_name	Value
Handler_commit	1
Handler_delete	0
...	
Handler_prepare	0
Handler_read_first	0
Handler_read_key	3
Handler_read_last	0
Handler_read_next	4891
Handler_read_prev	0
Handler_read_rnd	0
Handler_read_rnd_deleted	0
Handler_read_rnd_next	0
Handler_rollback	0
...	
Handler_update	0
Handler_write	0

25 rows in set (0.00 sec)

Despite identifying themselves as “range”s, the execution is slightly different, one index dive (similar to a ref) is done per value. This can be an issue in conditions with thousands of items.

"OR"-style conditions over different columns

- We cannot use a single index efficiently for both conditions
 - We can scan both conditions separately and mix the results, discarding duplicates
 - Or use an index for one condition and not for the other
 - Index merge allows to use two indexes for a single table access simultaneously

Index Merge Example

```
MariaDB [nlwiktio...]> EXPLAIN SELECT * FROM revision WHERE
rev_page = 790 OR rev_timestamp < '2004'\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: revision
         type: index_merge
possible_keys: rev_page,rev_timestamp,rev_page_rev_timestamp,
rev_timestamp_rev_page
              key: rev_page,rev_timestamp
            key_len: 4,14
              ref: NULL
             rows: 4871
       Extra: Using sort_union(rev_page,rev_timestamp); Using
where
1 row in set (0.00 sec)
```

Both indexes are used, then
combined using the "union"
operation



INDEX_MERGE Issues

- Sometimes it is faster to execute the sentence using UNION:
 - This is specially true with (UNION ALL) since MySQL 5.7, if you do not care about duplicates
- There are also intersection merges, but multi-column indexes are preferred

Disabling optimizer features (I)

- The `optimizer_switch` variable allows enabling and disabling globally or per session many query optimizer features:

```
MariaDB [nlwiktioary]> SHOW VARIABLES like 'optimizer_switch'\G
***** 1. row *****
Variable_name: optimizer_switch
      Value:
index_merge=on,index_merge_union=on,index_merge_sort_union=on,index_merge_intersection=on,
index_merge_sort_intersection=off,engine_condition_pushdown=off,index_condition_pushdown=on,
derived_merge=on,derived_with_keys=on,firstmatch=on,loosescan=on,materialization=on,in_to_exists=on,
semijoin=on,partial_match_rowid_merge=on,partial_match_table_scan=on,subquery_cache=on,mrr=off,mrr_cost_based=off,mrr_sort_keys=off,
outer_join_with_cache=on,semijoin_with_cache=on,join_cache_incremental=on,join_cache_hashed=on,
join_cache_bka=on,optimize_join_buffer_size=off,table_elimination=on,extended_keys=on,exists_to_in=on
1 row in set (0.00 sec)
```

Deshabilitar características del optimizador (II)

```
MariaDB [nlwictionary]> SET optimizer_switch='index_merge_sort_union=off';  
Query OK, 0 rows affected (0.00 sec)
```

```
MariaDB [nlwictionary]> EXPLAIN SELECT * F  
790 or rev_timestamp < '2004'\G  
***** 1. row *****
```

```
      id: 1  
select_type: SIMPLE  
      table: revision  
      type: ALL  
possible_keys:  
rev_page,rev_timestamp,rev_page_rev_timestamp,rev_timestamp_rev_page  
      key: NULL  
key_len: NULL  
      ref: NULL  
      rows: 686822  
      Extra: Using where  
1 row in set (0.00 sec)
```

This will only have effect for
the current session.

What happens if we have two ranges?

- As seen previously, we cannot use efficiently two range types on the same table access. Alternatives:
 - Use only one index for the most selective column
 - Use index condition pushdown to get an advantage
 - Change one of the two ranges into a discrete "IN" comparison/bucketize with a new column
 - Use quadtrees or R-TREEs (spatial indexing)

Example of Bucketizing (I)

```
MariaDB [nlwiktio...]> EXPLAIN SELECT count(*) FROM revision  
WHERE rev_timestamp < '2008' AND rev_len > 5500\G  
***** 1. row *****  
      id: 1  
select_type: SIMPLE  
      table: revision  
      type: ALL  
possible_keys: rev_timestamp,rev_timestamp_rev_page  
            key: NULL  
key_len: NULL  
      ref: NULL  
     rows: 686822  
    Extra: Using where  
1 row in set (0.00 sec)
```

Looks like only an index on (rev_timestamp) or (rev_len) would be useful as we have 2 ranges.

Example of Bucketizing (II)

```
MariaDB [nlwiktictionary]> ALTER TABLE revision ADD COLUMN  
rev_len_cat int;
```

```
Query OK, 0 rows affected (38.28 sec)
```

```
Records: 0 Duplicates: 0 Warnings: 0
```

```
MariaDB [nlwiktictionary]> UPDATE revision set rev_len_cat =  
IF(rev_len < 10000, rev_len div 1000, 10);
```

```
Query OK, 820308 rows affected (15.19 sec)
```

```
Rows matched: 820308 Changed: 820308 Warnings: 0
```

```
MariaDB [nlwiktictionary]> ALTER TABLE revision ADD INDEX  
rev_len_cat_rev_timestamp (rev_len_cat, rev_timestamp);
```

```
Query OK, 0 rows affected (2.11 sec)
```

```
Records: 0 Duplicates: 0 Warnings: 0
```

Example of Bucketizing (III)

```
MariaDB [nlwiktio...]> EXPLAIN SELECT count(*) FROM revision WHERE
rev_timestamp < '2008' AND rev_len > 5500 AND rev_len_cat IN (5, 6,
7, 8, 9, 10)\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: revision
         type: range
possible_keys:
rev_timestamp,rev_timestamp_rev_page,rev_len_cat_rev_timestamp
              key: rev_len_cat_rev_timestamp
      key_len: 19
        ref: NULL
      rows: 4442
     Extra: Using where
1 row in set (0.00 sec)
```

We did some transformations to both the structure and the query.

Example of Index Condition Pushdown

```
MariaDB [nlwiktio...]> ALTER TABLE revision ADD INDEX  
rev_len_rev_timestamp(rev_len, rev_timestamp);  
Query OK, 0 rows affected (1.77 sec)  
Records: 0 Duplicates: 0 Warnings: 0
```

```
MariaDB [nlwiktio...]> SET optimizer_switch='index_condition_pushdown=on'; EXPLAIN  
SELECT * FROM revision WHERE rev_timestamp < '2008' AND rev_len > 5500\G  
Query OK, 0 rows affected (0.00 sec)
```

```
***** 1. row *****  
      id: 1  
select_type: SIMPLE  
      table: revision  
      type: range  
possible_keys: rev_timestamp,rev_timestamp_  
      key: rev_len_rev_timestamp  
    key_len: 5  
      ref: NULL  
      rows: 38744  
    Extra: Using index condition  
1 row in set (0.00 sec)
```

Index condition pushdown (ICP) enables the engines to use extra parts of the index while avoiding costly row movements to and from the SQL layer

ICP Issues

- Differences in execution time is more significative when the extra column condition is very selective (getting 5x the original performance)
- ICP is ignored when using covering Index, making the performance worse

Does LIMIT improve the performance? (I)

```
MariaDB [nlwiktictionary]> EXPLAIN SELECT * FROM page ORDER BY page_touched\G  
*****  
 1. row *****
```

```
    type: ALL  
possible_keys: NULL  
        key: NULL  
key_len: NULL  
      ref: NULL  
     rows: 90956  
    Extra: Using filesort
```

```
MariaDB [nlwiktictionary]> EXPLAIN SELECT * FROM page ORDER BY page_touched LIMIT 10\G  
*****  
 1. row *****
```

```
    type: index  
possible_keys: NULL  
        key: page_page_touched  
key_len: 14  
      ref: NULL  
     rows: 10  
    Extra:  
1 row in set (0.00 sec)
```

In some cases it can be essential to allow effective usage of the indexes

Does LIMIT improve the performance? (II)

```
MariaDB [nlwiktionary]> EXPLAIN SELECT * FROM revision ORDER BY rev_comment\G
***** 1. row *****
[...]    type: ALL
possible_keys: NULL
          key: NULL
key_len: NULL
          ref: NULL
      rows: 817636
     Extra: Using filesort
1 row in set (0.00 sec)
```

```
MariaDB [nlwiktionary]> EXPLAIN SELECT * FROM revision ORDER BY rev_comment LIMIT 10\G
***** 1. row *****
[...]    table: revision
          type: ALL
possible_keys: NULL
          key: NULL
key_len: NULL
          ref: NULL
      rows: 817636
     Extra: Using filesort
1 row in set (0.00 sec)
```

In other cases, it has no effect on the scanned rows (just on the returned ones)

Does LIMIT improve the performance? (I)

```
MariaDB [nlwiktictionary]> EXPLAIN SELECT * FROM page ORDER BY page_title LIMIT 100\G
***** 1. row *****
    type: index
possible_keys: NULL
      key: page_title
    key_len: 257
      ref: NULL
     rows: 100
    Extra:
1 row in set (0.00 sec)
```

```
MariaDB [nlwiktictionary]> EXPLAIN SELECT * FROM page ORDER BY page_title LIMIT 10000,
100\G
***** 1. row *****
    type: ALL
possible_keys: NULL
      key: NULL
    key_len: NULL
      ref: NULL
     rows: 90956
    Extra: Using filesort
1 row in set (0.00 sec)
```

In this case,
performance will vary
depending on the
offset (not ideal)

Can we filter and sort at the same time using indexes?

```
MariaDB [nlwiktio...]> EXPLAIN SELECT * FROM revision WHERE  
rev_comment=' ' ORDER BY rev_timestamp ASC\G  
***** 1. row *****
```

```
      id: 1  
select_type: SIMPLE  
      table: revision  
      type: ALL  
possible_keys: NULL  
          key: NULL  
     key_len: NULL  
        ref: NULL  
       rows: 817636  
    Extra: Using where; Using filesort  
1 row in set (0.00 sec)
```

This query is slow because a) the full table scan

b) Required sort after filtering

Adding an index on (rev_comment, rev_timestamp)

```
MariaDB [nlwiktio...]> ALTER TABLE revision ADD INDEX
rev_comment_rev_timestamp (rev_comment, rev_timestamp);
Query OK, 0 rows affected (3.19 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
MariaDB [nlwiktio...]> EXPLAIN SELECT * FROM revision WHERE rev_comment=''
ORDER BY rev_timestamp ASC\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: revision
        type: ref
possible_keys: rev_comment_rev_timestamp
              key: rev_comment_rev_timestamp
            key_len: 769
            ref: const
          rows: 266462
        Extra: Using index condition; Using where
1 row in set (0.00 sec)
```

Both type: ALL and
filesort have
disappeared

This is not always possible

```
MariaDB [nlwiktio...]> EXPLAIN SELECT * FROM revision WHERE
rev_len > 5500 ORDER BY rev_timestamp ASC\G
***** 1. row *****
      id: 1
select_type: SIMPLE
      table: revision
        type: range
possible_keys: rev_len_rev_timest
              key: rev_len_rev_timest
      key_len: 5
        ref: NULL
      rows: 38744
    Extra: Using index condition; Using filesort
1 row in set (0.00 sec)
```

The range makes impossible to use the index optimally for the the ORDER BY: either we filter (rev_len) or sort (rev_timestamp)

A Strange Game. The Only Winning Move is Not to Play

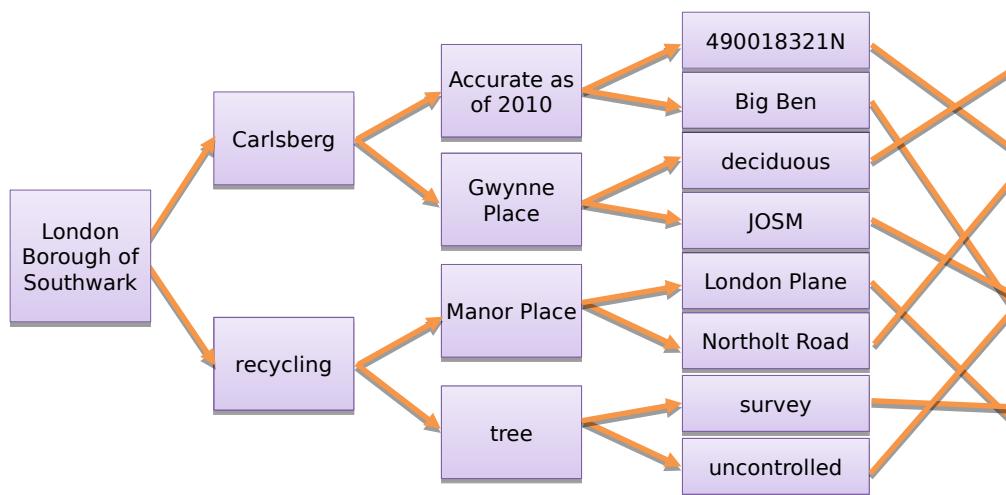
```
mysql-5.6.21 (osm) > SELECT * FROM nodes FORCE  
INDEX(version_idx) WHERE version < 15 ORDER BY changeset_id;  
/* type: range, Using filesort */  
2859673 rows in set (30.58 sec)
```

```
mysql-5.6.21 (osm) > SELECT * FROM nodes FORCE  
INDEX(changeset_id_idx) WHERE version < 15 ORDER BY  
changeset_id;  
/* type: index */  
2859673 rows in set (30.92 sec)
```

```
mysql-5.6.21 (osm) > SELECT * FROM nodes WHERE version < 15  
ORDER BY changeset_id;  
/* type: ALL, Using filesort */  
2859673 rows in set (16.54 sec)
```

MyISAM Internals

Index (part of revision.MYI)



Data (revision.MYD)

node_id	versio	k	v
234234344545	1	leaves	deciduous
(empty row)			
234234344548	5	gps:latitude	490018321N
234234344549	6	access	uncontrolled
(empty row)			
234234344551	9	name	Big Ben
234234344552	1	source	survey
234234344557	1	name	London Plane
234234344552	2	source	survey
234234344557	2	name	London Plane

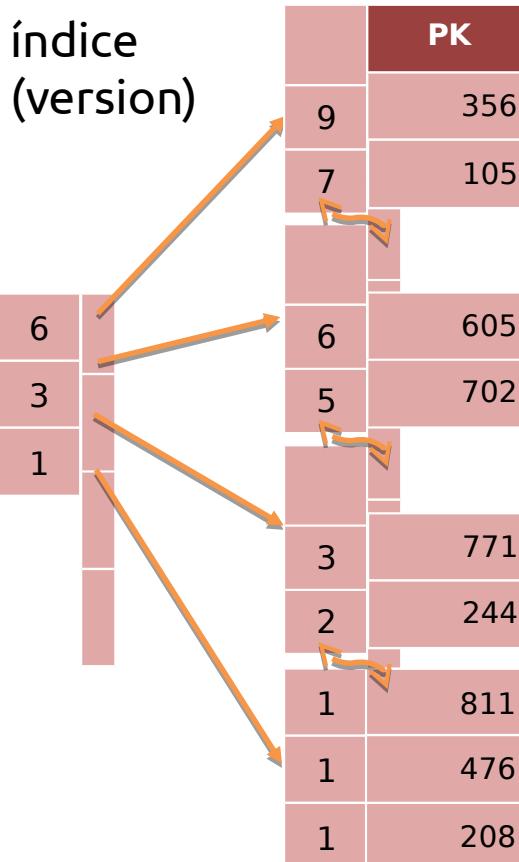
InnoDB Internals (PRIMARY)

	version	k	v
811	1	leaves	deciduous
771	3	name	Northolt Road
702	5	gps:latitu	490018321N
702	6	access	uncontrolled
605	1	editor	JOSM
476	9	name	Big Ben
476	2	source	survey
356	1	name	London Plane
244	7	amenity	pub
208			
105			

Data
clustered
always
using the
primary
key

InnoDB Internals (Secondary)

Secondary indexes contain the primary key value



PK / Datos

	versio	k	v
811	1	leaves	deciduous
702	3	name	Northolt Road
702	5	gps:latitu	490018321N
476	6	access	uncontrolled
605	1	editor	JOSM
476	9	name	Big Ben
244	2	source	survey
208	1	name	London Plane
105	7	amenity	pub

Consequences of using InnoDB (I)

- Every table should have a primary key
 - If one is not defined, MySQL will choose an available NOT NULL unique key
 - If that is not possible, an internal 6-byte row identifier will be generated (not user-accessible)

Consequences of using InnoDB (II)

- Inserting in primary key order is much faster
 - Less fragmentation/page-split
 - Usage of "batch" mode, improving insert speed
- Using auto-increment keys as primary keys can be a good idea for InnoDB

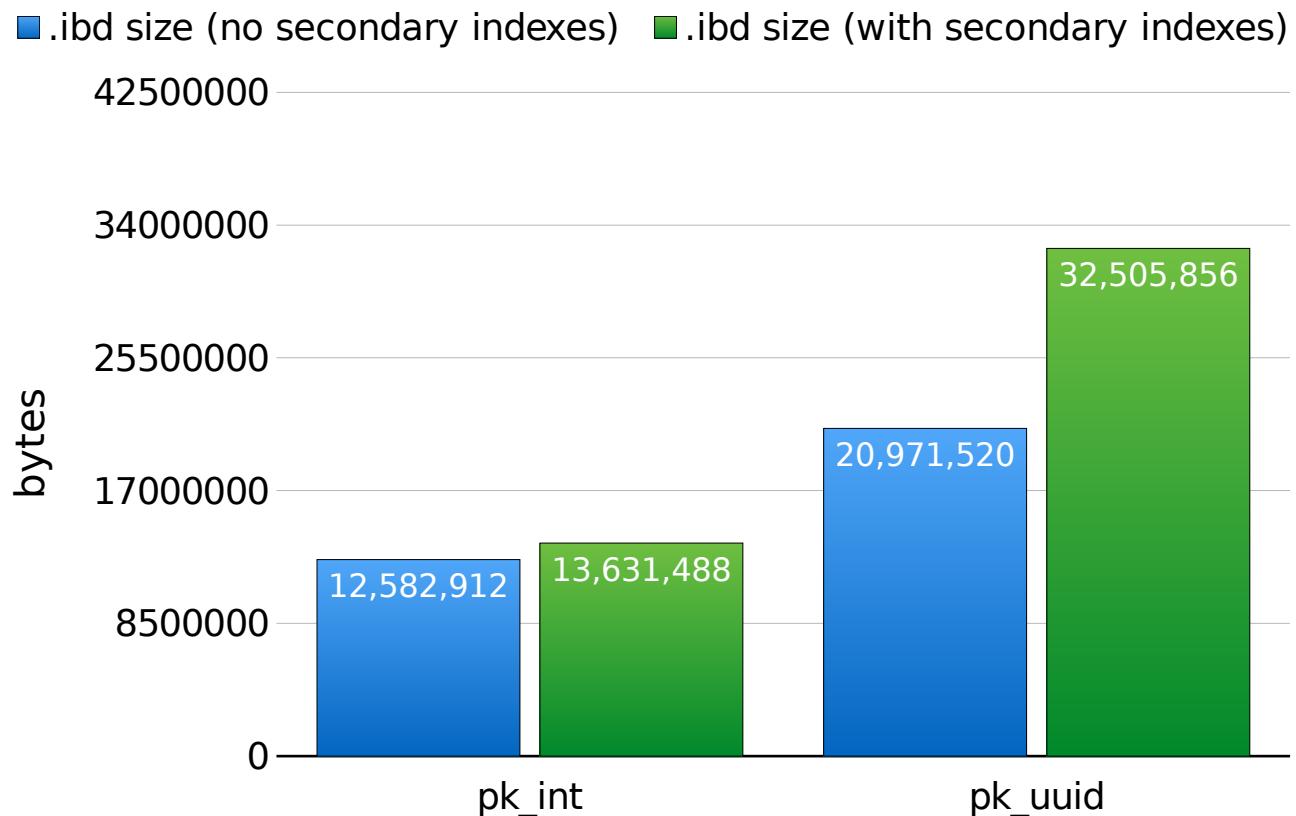
Consequences of using InnoDB (III)

- A very long primary key may increment substantially the size of secondary keys
 - Int or bigint types are recommended instead of UUIDs or other long strings

Differences in size

```
mysql-5.6.21 (osm) > CREATE  
TABLE pk_int (id int  
PRIMARY KEY auto_increment,  
a int,  
b int,  
c int,  
d int);  
Query OK, 0 rows affected  
(0.16 sec)
```

```
mysql-5.6.21 (osm) > CREATE  
TABLE pk_uuid (id char(36)  
PRIMARY KEY,  
a int,  
b int,  
c int,  
d int);  
Query OK, 0 rows affected  
(0.04 sec)
```



Extended primary key optimization

- As the primary key is part of all secondary keys, this can be used “for free”:
 - For row filtering (since MySQL 5.6)
 - To return results in primary key order
 - To avoid reading data from the table (covering index)

Extended Primary Key Example

```
mysql-5.6.21 (osm) > EXPLAIN SELECT node_id FROM nodes WHERE
changeset_id = 24284 and node_id <> 146472942\G
***** 1. row *****
      id: 1
select_type: SIMPLE
      table: nodes
      type: range
possible_keys: PRIMARY,changeset_id_idx
      key: changeset_id_idx
    key_len: 16
      ref: NULL
      rows: 50
    Extra: Using where; Using index
1 row in set (0.07 sec)
```

Query Optimization: From 0 to 10 (and up to 5.7)

FULLTEXT SEARCH

Fuzzy Search of “gloucester/Gloucester’s/etc”

- “Typical” way to solve this:

```
mysql-5.7.5 (osm) >
SELECT way_id as id, v
FROM way_tags
WHERE v like '%gloucester%';
425 rows in set (0.46 sec)
```



Too slow

Let's Add an Index

```
mysql-5.7.5 (osm) > ALTER TABLE way_tags ADD INDEX(v);
Query OK, 0 rows affected (6.44 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Still slow, why?

```
mysql-5.7.5 (osm) > SELECT ....;
425 rows in set (0.38 sec)
```

```
mysql-5.7.5 (osm) > EXPLAIN SELECT way as type, way_id as
id, v FROM way_tags WHERE v like '%gloucester%';
```

id	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	index	NULL	v_2	767	NULL	1333338	11.11	Using where; Using index

1 row in set, 1 warning (0.01 sec)

Fulltext Index

```
mysql-5.7.5 (osm) > ALTER TABLE way_tags add FULLTEXT index(v);
Query OK, 0 rows affected (3.20 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
mysql-5.7.5 (osm) > SELECT ...;
425 rows in set (0.00 sec)
```

```
mysql-5.7.5 (osm) > EXPLAIN SELECT 'way' as type, way_id as id, v
FROM way_tags WHERE MATCH(v) AGAINST ('+gloucester*' IN BOOLEAN
MODE);
```

id	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	fulltext	v	v	0	const	1	100.00	Using where; Ft_hints: no_ranking

1 row in set, 1 warning (0.00 sec)

Newer Fulltext Optimizations

```
mysql-5.5.40 (osm) > EXPLAIN SELECT count(*) FROM way_tags_myisam WHERE MATCH(v) AGAINST('gloucester');
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table      | type      | possible_keys | key     | key_len | ref    | rows | Extra
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1  | SIMPLE      | way_tags_myisam | fulltext | v           | v       | 0       |        | 1    | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

```
mysql-5.5.40 (osm) > SHOW STATUS like 'Handler%';
+-----+-----+
| Variable_name      | Value |
+-----+-----+
| Handler_commit     | 0     |
| Handler_delete     | 0     |
| ...
| Handler_read_first | 0     |
| Handler_read_key   | 0     |
| Handler_read_last  | 0     |
| Handler_read_next  | 425   |
| Handler_read_prev  | 0     |
| Handler_read_rnd   | 0     |
| Handler_read_rnd_next | 0    |
| ...
| Handler_update     | 0     |
| Handler_write      | 0     |
+-----+-----+
```

16 rows in set (0.00 sec)

Newer Fulltext Optimizations (cont.)

```
mysql-5.7.5 (osm) > EXPLAIN SELECT count(*) FROM way_tags WHERE MATCH(v) AGAINST('gloucester');
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	Select tables optimized away

1 row in set, 1 warning (0.00 sec)

```
mysql-5.7.5 (osm) > SHOW STATUS like 'Handler%';
```

Variable_name	Value
Handler_commit	1
Handler_delete	0
...	
Handler_read_first	0
Handler_read_key	0
Handler_read_last	0
Handler_read_next	0
Handler_read_prev	0
Handler_read_rnd	0
Handler_read_rnd_next	0
...	
Handler_update	0
Handler_write	0

18 rows in set (0.00 sec)



It's
counting
directly
from the
FULLTEXT
index

Open Issues and Limitations

- No postfix support (wildcards)
- Simple Ranking (and different from MyISAM)
- No stemming support
- Some multi-language limitations

More on FULLTEXT InnoDB support:

<http://www.drdobbs.com/database/full-text-search-with-innodb/231902587>

Alternatives

- Apache Lucene
 - Solr
 - Elasticsearch
- Sphinx
 - SphinxSE

Query Optimization: From 0 to 10 (and up to 5.7)

JOINS

(Block) Nested Loop Join

- Until MySQL 5.5 there was only one algorithm to execute a JOIN:

node_id	version	lat	lon
1	1	52	0.5
1	2	52	0.5
2	1	51	1
3	1	53	1.5



node_id	version	k	v
1	1	name	Big Benn
1	1	tourism	attraction
1	2	name	Big Ben
1	2	tourism	attraction
3	1	name	London Eye

Extra Access type: eq_ref

```
mysql-5.6.21 (osm) > EXPLAIN SELECT * FROM nodes JOIN node_tags  
    USING(node_id, version) WHERE node_tags.v= 'Big Ben'\G  
*****  
      1. row *****  
      id: 1  
 select_type: SIMPLE  
       table: node_tags  
        type: ref  
possible_keys: PRIMARY,v_idx  
         key: v_idx  
      key_len: 767  
        ref: const  
       rows: 1  
     Extra: Using where; Using index  
*****  
      2. row *****  
      id: 1  
 select_type: SIMPLE  
       table: nodes  
        type: eq_ref  
possible_keys: PRIMARY,version_idx  
         key: PRIMARY  
      key_len: 16  
        ref: osm.node_tags.node_id,osm.node_tags.version  
       rows: 1  
     Extra: NULL  
2 rows in set (0.00 sec)
```

eq_ref is similar to ref, but allows faster JOINS because, by using a unique key, it only has to search one row for each previous result

JOIN Optimization

- Two main goals:
 - Perform an effective filtering on each table access, if possible using indexes
 - Perform the access in the most efficient table order
- When joining 3 or more tables in a star schema, the "covering index" strategy can have a huge impact

Example: optimize this JOIN (I)

```
SELECT n.node_id, n.latitude, n.longitude  
FROM way_nodes w_n  
JOIN way_tags w_t  
  ON w_n.way_id = w_t.way_id and  
      w_n.version = w_t.version  
JOIN nodes n  
  ON w_n.node_id = n.node_id  
JOIN node_tags n_t  
  ON n.node_id = n_t.node_id and  
      n.version = n_t.version  
WHERE w_t.k = 'building' and  
      n_t.k = 'entrance' and  
      n_t.v = 'main';
```



We start without secondary indexes

Example: optimize this JOIN (II)

```
***** 1. row ***** 3. row *****
    id: 1
  select_type: SIMPLE
    table: w_t
    type: index
possible_keys: PRIMARY
    key: PRIMARY
key_len: 783
  ref: NULL
  rows: 1335702
  Extra: Using where; Using index
***** 2. row ***** 4. row *****
    id: 1
  select_type: SIMPLE
    table: w_n
    type: ref
possible_keys: PRIMARY
    key: PRIMARY
key_len: 16
  ref: osm.w_t.way_id,osm.w_t.version
  rows: 3
  Extra: NULL

    id: 1
  select_type: SIMPLE
    table: n_t
    type: ref
possible_keys: PRIMARY
    key: PRIMARY
key_len: 8
  ref: osm.w_n.node_id
  rows: 1
  Extra: Using where
    id: 1
  select_type: SIMPLE
    table: n
    type: eq_ref
possible_keys: PRIMARY
    key: PRIMARY
key_len: 16
  ref: osm.w_n.node_id,osm.n_t.version
  rows: 1
  Extra: Using index
4 rows in set (0.01 sec)

mysql-5.6.21 (osm) > SELECT ...
858 rows in set (9.00 sec)
```

Example: optimize this JOIN (III)

```
mysql-5.6.21 (osm) > ALTER TABLE way_tags ADD  
INDEX k_idx(k);  
Query OK, 0 rows affected (4.80 sec)  
Records: 0  Duplicates: 0  Warnings: 0
```



Creating an index
on way_tags.k

Example: optimize this JOIN (IV)

```
***** 1. row *****
    id: 1
  select_type: SIMPLE
      table: w_t
      type: ref
possible_keys: PRIMARY,k_idx
      key: k_idx
    key_len: 767
      ref: const
     rows: 452274
    Extra: Using where; Using index
***** 2. row *****
    id: 1
  select_type: SIMPLE
      table: w_n
      type: ref
possible_keys: PRIMARY
      key: PRIMARY
    key_len: 16
      ref: osm.w_t.way_id,osm.w_t.version
     rows: 3
    Extra: NULL

mysql-5.6.21 (osm) > SELECT ...
858 rows in set (8.58 sec)
```

```
***** 3. row *****
    id: 1
  select_type: SIMPLE
      table: n_t
      type: ref
possible_keys: PRIMARY
      key: PRIMARY
    key_len: 8
      ref: osm.w_n.node_id
     rows: 1
    Extra: Using where
***** 4. row *****
    id: 1
  select_type: SIMPLE
      table: n
      type: eq_ref
possible_keys: PRIMARY
      key: PRIMARY
    key_len: 16
      ref: osm.w_n.node_id,osm.n_t.version
     rows: 1
    Extra: NULL
4 rows in set (0.00 sec)
```

**It seems like the index
is not very useful**

Example: optimize this JOIN (V)

```
mysql-5.6.21 (osm) > ALTER TABLE node_tags ADD  
INDEX k_idx(k);  
Query OK, 0 rows affected (2.82 sec)  
Records: 0  Duplicates: 0  Warnings: 0
```

The order does not seem
to be adequate, let's try
adding an index to start
by accessing node_tags

Example: optimize this JOIN (VI)

```
***** 1. row ***** 3. row *****
    id: 1
  select_type: SIMPLE
    table: w_t
    type: ref
possible_keys: PRIMARY,k_idx
    key: k_idx
  key_len: 767
    ref: const
  rows: 452274
  Extra: Using where; Using index
***** 2. row *****
    id: 1
  select_type: SIMPLE
    table: w_n
    type: ref
possible_keys: PRIMARY
    key: PRIMARY
  key_len: 16
    ref: osm.w_t.way_id,osm.w_t.version
  rows: 3
  Extra: NULL

mysql-5.6.21 (osm) > SELECT ...
858 rows in set (7.33 sec)
```

```
    id: 1
  select_type: SIMPLE
    table: n_t
    type: ref
possible_keys: PRIMARY,k_idx
    key: PRIMARY
  key_len: 8
    ref: osm.w_n.node_id
  rows: 1
  Extra: Using where
***** 4. row *****
    id: 1
  select_type: SIMPLE
    table: n
    type: eq_ref
possible_keys: PRIMARY
    key: PRIMARY
  key_len: 16
    ref: osm.w_n.node_id,osm.n_t.version
  rows: 1
  Extra: NULL
4 rows in set (0.00 sec)
```

It keeps using
the wrong
order, even if
we delete the
`w_t.k_idx`
index

Example: optimize this JOIN (VII)

```
SELECT STRAIGHT_JOIN n.node_id, n.latitude, n.longitude
  FROM node_tags n_t
  JOIN nodes n
    ON n.node_id = n_t.node_id and
       n.version = n_t.version
  JOIN way_nodes w_n
    ON w_n.node_id = n.node_id
  JOIN way_tags w_t
    ON w_n.way_id = w_t.way_id and
       w_n.version = w_t.version
 WHERE w_t.k = 'building' and
       n_t.k = 'entrance' and
       n_t.v = 'main';
```



Let's see why rewriting it
into this query

Example: optimize this JOIN (VIII)

```
***** 1. row *****
    id: 1
  select_type: SIMPLE
      table: n_t
      type: ref
possible_keys: PRIMARY,k_idx
      key: k_idx
    key_len: 767
      ref: const
     rows: 2390
  Extra: Using index condition; Using where
***** 2. row *****
    id: 1
  select_type: SIMPLE
      table: n
      type: eq_ref
possible_keys: PRIMARY
      key: PRIMARY
    key_len: 16
      ref: osm.n_t.node_id,osm.n_t.version
     rows: 1
  Extra: NULL
***** 3. row *****
    id: 1
  select_type: SIMPLE
      table: w_n
      type: ALL
possible_keys: PRIMARY
      key: NULL
    key_len: NULL
      ref: NULL
     rows: 3597858
  Extra: Using where; Using join buffer (Block Nested Loop)
***** 4. row *****
    id: 1
  select_type: SIMPLE
      table: w_t
      type: eq_ref
possible_keys: PRIMARY
      key: PRIMARY
    key_len: 783
      ref: osm.w_n.way_id,osm.w_n.version,const
     rows: 1
  Extra: Using where; Using index
4 rows in set (0.00 sec)
```

There is no index on w_n that would allow efficient access

Example: optimize this JOIN (IX)

```
mysql-5.6.21 (osm) > ALTER TABLE way_nodes ADD  
INDEX node_id_idx(node_id);  
Query OK, 0 rows affected (17.77 sec)  
Records: 0  Duplicates: 0  Warnings: 0
```

Example: optimize this JOIN (X)

```
***** 1. row *****
    id: 1
select_type: SIMPLE
    table: n_t
    type: ref
possible_keys: PRIMARY,k_idx
    key: k_idx
key_len: 767
    ref: const
rows: 2390
Extra: Using index condition; Using where
***** 2. row *****
    id: 1
select_type: SIMPLE
    table: n
    type: eq_ref
possible_keys: PRIMARY
    key: PRIMARY
key_len: 16
    ref: osm.n_t.node_id,osm.n_t.version
rows: 1
Extra: NULL
```

Now it starts by
the right table
(without
STRAIGHT_JOIN)

```
***** 3. row *****
    id: 1
select_type: SIMPLE
    table: w_n
    type: ref
possible_keys: PRIMARY,node_id_idx
    key: node_id_idx
key_len: 8
    ref: osm.n_t.node_id
rows: 1
Extra: Using index
***** 4. row *****
    id: 1
select_type: SIMPLE
    table: w_t
    type: eq_ref
possible_keys: PRIMARY
    key: PRIMARY
key_len: 783
    ref: osm.w_n.way_id,osm.w_n.version,const
rows: 1
Extra: Using where; Using index
4 rows in set (0.04 sec)
```

```
mysql-5.6.21 (osm) > SELECT ...
858 rows in set (0.73 sec)
```

Example: optimize this JOIN (XI)

```
***** 1. row *****
    id: 1
  select_type: SIMPLE
      table: n_t
      type: ref
possible_keys: PRIMARY,k_idx,k_v_i
      key: k_v_idx
    key_len: 1534
      ref: const,const
     rows: 900
    Extra: Using where; Using index
***** 2. row *****
    id: 1
  select_type: SIMPLE
      table: n
      type: eq_ref
possible_keys: PRIMARY
      key: PRIMARY
    key_len: 16
      ref: osm.n_t.node_id,osm.n_t.version
     rows: 1
    Extra: NULL

mysql-5.6.21 (osm) > SELECT ...
858 rows in set (0.02 sec)
```

An index on (k,v) is even better

```
***** 3. row *****
    id: 1
  select_type: SIMPLE
      table: w_n
      type: ref
possible_keys: PRIMARY,node_id_idx
      key: node_id_idx
    key_len: 8
      ref: osm.n_t.node_id
     rows: 1
    Extra: Using index
***** 4. row *****
    id: 1
  select_type: SIMPLE
      table: w_t
      type: eq_ref
possible_keys: PRIMARY
      key: PRIMARY
    key_len: 783
      ref: osm.w_n.way_id,osm.w_n.version,const
     rows: 1
    Extra: Using where; Using index
4 rows in set (0.00 sec)
```

“New” JOIN methods

- MySQL 5.6 added:
 - Batch Key Access
- MariaDB has since 5.3:
 - Batch Key Access
 - Hash Joins
 - Slightly modified versions of the above ones (with “incremental” buffers to join 3 or more tables)

Multi-range read

- This optimization orders results obtained from a secondary key in primary key/physical order before accessing the rows
 - It may help execution time of queries when disk-bound
 - It requires tuning of the `read_rnd_buffer_size` (size of the buffer used for ordering the results)
- BKA JOINs are based on the mrr optimization

MRR Example (I)

```
mysql-5.6.21 (osm) > EXPLAIN SELECT * FROM nodes WHERE timestamp >=
'2013-07-01 00:00:00' AND timestamp < '2014-01-01 00:00:00'\G
*****1. row *****
      id: 1
select_type: SIMPLE
      table: nodes
        type: range
possible_keys: nodes_timestamp_idx
          key: nodes_timestamp_idx
    key_len: 5
      ref: NULL
     rows: 429684
    Extra: Using index condition; Using MRR
1 row in set (0.02 sec)
```

MRR example (II)

[restart]

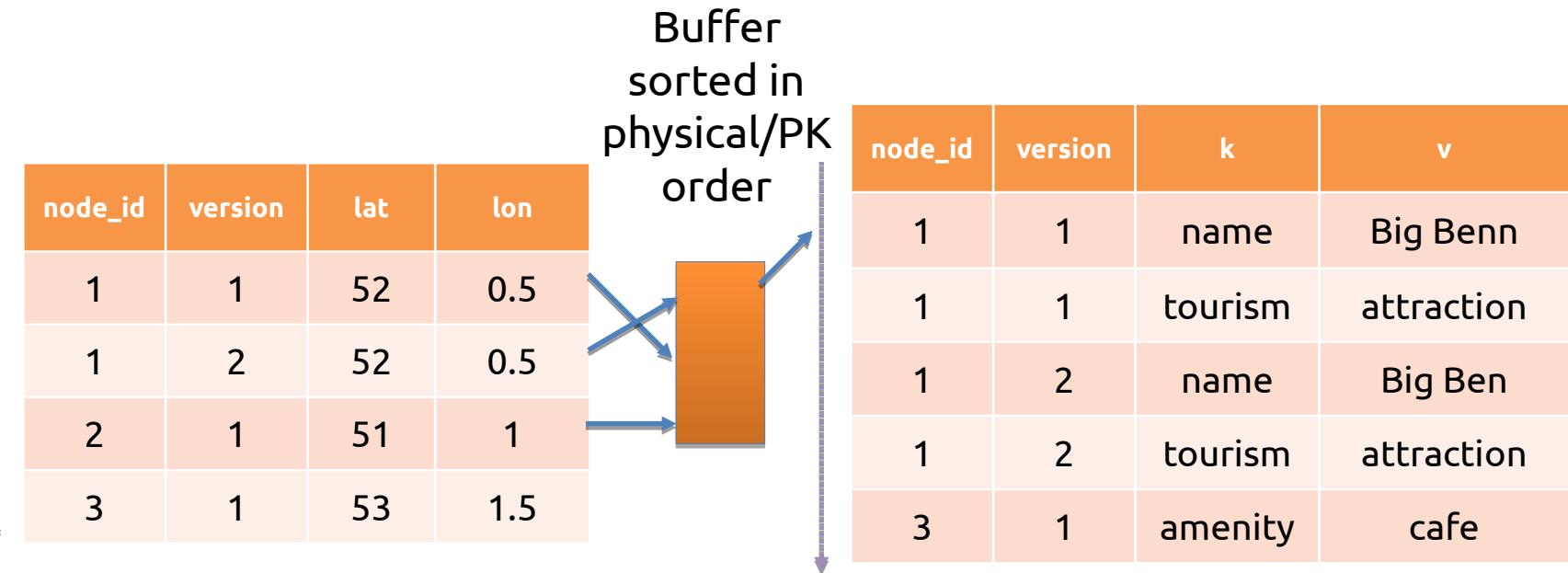
```
mysql> SET optimizer_switch='mrr=off';
mysql> SELECT * FROM nodes WHERE timestamp >= '2013-07-01 00:00:00' AND timestamp <
'2014-01-01 00:00:00';
205617 rows in set (5.16 sec)
mysql> SELECT * FROM nodes WHERE timestamp >= '2013-07-01 00:00:00' AND timestamp <
'2014-01-01 00:00:00';
205617 rows in set (0.60 sec)
```

[restart]

```
mysql> SET read_rnd_buffer_size=50 * 1024 * 1024;
mysql> SELECT * FROM nodes WHERE timestamp >= '2013-07-01 00:00:00' AND timestamp <
'2014-01-01 00:00:00';
205617 rows in set (2.39 sec)
mysql> SELECT * FROM nodes WHERE timestamp >= '2013-07-01 00:00:00' AND timestamp <
'2014-01-01 00:00:00';
205617 rows in set (0.73 sec)
```

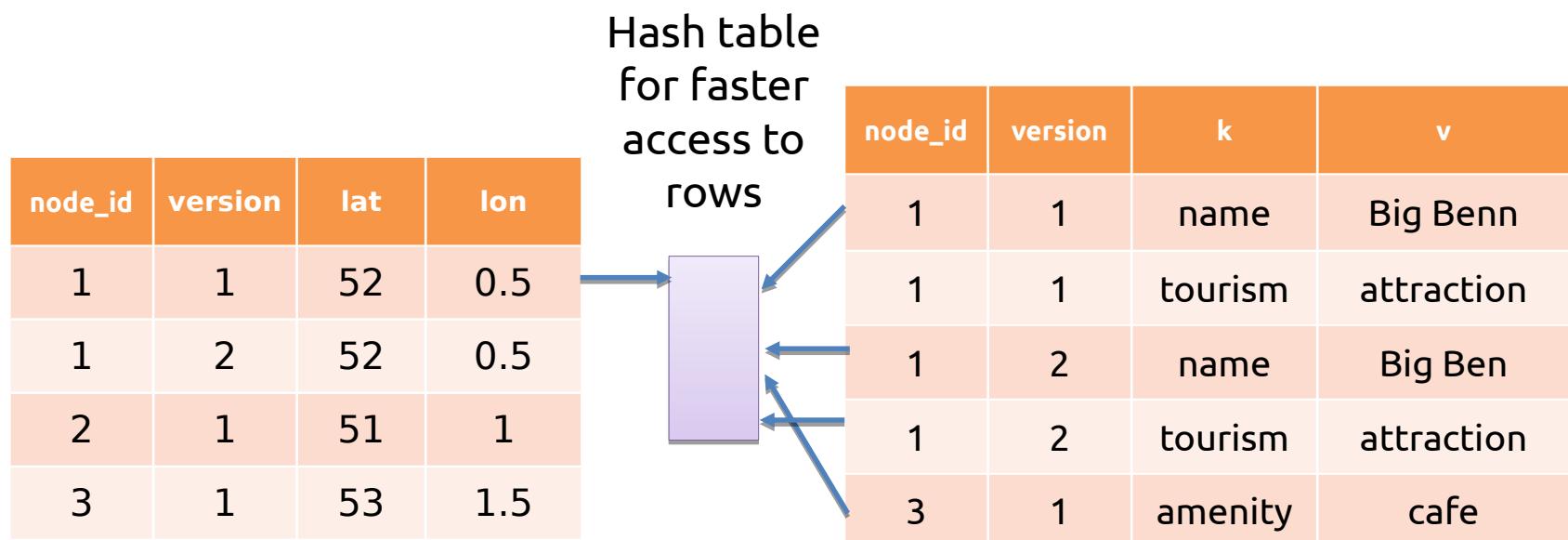
“Cold” results are significantly better with mrr (but it can impact negatively, too)

Batch Key Access



Hash Joins

- Only work for equi-joins



MySQL Configuration

- BKA requires changes of default optimizer configuration:

```
mysql> SET optimizer_switch= 'mrrr=on';
```

```
mysql> SET optimizer_switch= 'mrrr_cost_based=off';
```

```
mysql> SET optimizer_switch= 'batch_key_access=on';
```

- Additionally, configuring the join_buffer_size adequately

MariaDB configuration

```
mariadb-10.0.14 (osm) > SET optimizer_switch = 'join_cache_incremental=on';  
mariadb-10.0.14 (osm) > SET optimizer_switch = 'join_cache_hashed=on';  
mariadb-10.0.14 (osm) > SET optimizer_switch = 'join_cache_bka=on';
```

- Enabled by default

```
mariadb-10.0.14 (osm) > SET join_cache_level = 3 (for hash joins)  
mariadb-10.0.14 (osm) > SET join_cache_level = 5 (for BKA)
```

- Also, configure join_buffer_size appropriately.
- Hash joins, like BKA, are highly dependent on disk-bound DBs to be effective due to the extra overhead

Nested Loop Join (cold buffers buffer_pool=100MB, join_buffer=4M)

```
mariadb-10.0.14 (osm) > EXPLAIN SELECT
changeset_id, count(*) FROM changesets JOIN nodes
on changesets.id = nodes.changeset_id GROUP BY
visible\G
*****
1. row *****
    id: 1
  select_type: SIMPLE
      table: changesets
      type: index
possible_keys: PRIMARY
      key: PRIMARY
    key_len: 8
      ref: NULL
      rows: 69115
  Extra: Using index; Using temporary; Using
        filesort
```

```
***** 2. row *****
    id: 1
  select_type: SIMPLE
      table: nodes
      type: ref
possible_keys: changeset_id
      key: changeset_id
    key_len: 8
      ref: osm.changesets.id
      rows: 19
  Extra:
2 rows in set (0.00 sec)
```

```
mariadb-10.0.14 (osm) > SELECT visible, count(*)
FROM changesets JOIN nodes on changesets.id =
nodes.changeset_id GROUP BY visible;
+-----+-----+
| visible | count(*) |
+-----+-----+
|      1 |  2865312 |
+-----+-----+
1 row in set (32.86 sec)
```

Hash Join (cold buffers, buffer_pool=100M, join_buffer=4M)

```
mariadb-10.0.14 (osm) > EXPLAIN SELECT
changeset_id, count(*) FROM changesets JOIN
nodes on changesets.id = nodes.changeset_id
GROUP BY visible\G
*****
1. row *****
      id: 1
      select_type: SIMPLE
          table: changesets
          type: index
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 8
          ref: NULL
          rows: 69115
      Extra: Using index; Using temporary;
Using filesort
```

```
***** 2. row *****
      id: 1
      select_type: SIMPLE
          table: nodes
          type: hash_ALL
possible_keys: changeset_id
          key: #hash#changeset_id
      key_len: 8
          ref: osm.changesets.id
          rows: 2781732
      Extra: Using join buffer (flat, BNLH join)
2 rows in set (0.00 sec)

mariadb-10.0.14 (osm) > SELECT visible, count(*)
FROM changesets JOIN nodes on changesets.id =
nodes.changeset_id GROUP BY visible;
+-----+-----+
| visible | count(*) |
+-----+-----+
|       1 |    2865312 |
+-----+-----+
1 row in set (6.66 sec)
```

Query Optimization: From 0 to 10 (and up to 5.7)

SUBQUERIES

Access types: unique_subquery/index_subquery

```
mysql-5.6.21 (osm) > EXPLAIN SELECT **** 2. row ****  
* FROM node_tags WHERE v = 'Big Ben'           id: 2  
and node_id NOT IN (SELECT node_id          select_type: DEPENDENT SUBQUERY  
FROM nodes WHERE tile < 100000000)\G        table: nodes  
***** 1. row *****                         type: index_subquery  
                                              possible_keys: PRIMARY,nodes_tile_idx  
                                              key: PRIMARY  
                                              key_len: 8  
                                              ref: func  
                                              rows: 1  
                                              Extra: Using where  
2 rows in set (0.00 sec)  
  
id: 1  
select_type: PRIMARY  
table: node_tags  
type: ref  
possible_keys: v_idx  
key: v_idx  
key_len: 767  
ref: const  
rows: 1  
Extra: Using where; Using  
index
```

Unique
subquery
is similar,
but using a
unique or
primary
key

Subqueries in MySQL

- MySQL versions traditionally had very bad press regarding subqueries
 - It was common to recommend rewriting them (when possible) into JOINS
- Since MySQL 5.6, its query execution plans have improved significantly

Lazy Materialization of derived tables

- Option available since MySQL 5.6
 - Improves the execution time of EXPLAIN (it no longer needs to execute subqueries)
 - Derived tables can be indexed automatically at execution time to improve its performance

Derived Table Example

```
mysql-5.5.40 (osm) > EXPLAIN SELECT count(*) FROM (SELECT * FROM nodes WHERE VISIBLE = 1) n JOIN changesets ON n.changeset_id = changesets.id;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	<derived2>	ALL	NULL	NULL	NULL	NULL	2865312	
1	PRIMARY	changesets	eq_ref	PRIMARY	PRIMARY	8	n.changeset_id	1	Using index
2	DERIVED	nodes	ALL	NULL	NULL	NULL	NULL	2865521	Using where

3 rows in set (1.42 sec)

```
mysql-5.6.21 (osm) > EXPLAIN SELECT count(*) FROM (SELECT * FROM nodes WHERE VISIBLE = 1) n JOIN changesets ON n.changeset_id = changesets.id;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	changesets	index	PRIMARY	PRIMARY	1	.id	70917	Using
1	PRIMARY	<derived>	derived	<auto_key0>	<auto_key0>	1	.id	40	NULL
2	DERIVED	nodes	where					2853846	Using

3 rows in set (0.00 sec)

Subquery is not executed

Auto-generated index

A Common 5.5 Performance Problem

```
mysql-5.5.40 (osm) > EXPLAIN SELECT * FROM nodes
    WHERE nodes.changeset_id IN (
        SELECT changesets.id
        FROM changesets
        JOIN users
            ON changesets.user_id = users.id and users.display_name = 'Steve');
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	nodes	ALL	NULL	NULL	NULL	NULL	2865521	Using where
2	DEPENDENT SUBQUERY	users	const	PRIMARY,users_display_name_idx	users_display_name_idx	767	const	1	Using index
2	DEPENDENT SUBQUERY	changesets	eq_ref	..	PRIMARY	8	func	1	Using where

3 rows in set (0.00 sec)

```
mysql-5.5.40 (osm) > SELECT ...;
+-----+-----+-----+-----+
| node_id | latitude | longitude | changeset_id | visiti
+-----+-----+-----+-----+
| 99890 | 515276425 | -1497621 | 552 |
| 109174 | 515364532 | -1457329 | 1875 |
| 276538 | 515324296 | -2094688 | 810 |
| 442987 | 515449207 | -1275650 | 1941 |
| 442988 | 515449741 | -1272860 | 1941 |
| 498803 | 515438432 | -1269436 | 2171 |
| 138212838 | 513010180 | -1699929 | 7757299 |
+-----+-----+-----+-----+
7 rows in set (2.60 sec)
```

This means that
the subquery is
executed almost
3 million times

Semijoin Optimization

- The only way to execute certain IN subqueries was to execute them with poor strategy
 - This forced to rewrite certain queries into JOINS or scalar subqueries, when possible
- There are now several additional automatic options:
 - Convert to a JOIN
 - Materialization (including index creation)
 - FirstMatch
 - LooseScan
 - Duplicate Weedout

The Previous Query is Not a Problem in 5.6/5.7/MariaDB 5.3+

```
mysql-5.6.21 (osm) > EXPLAIN SELECT * FROM nodes
    WHERE nodes.changeset_id IN (
        SELECT changesets.id
        FROM change_sets
        JOIN users
            ON changesets.user_id = users.id and users.display_name = 'Steve');
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	users	const	PRIMARY,users_display_name_idx	users_display_name_idx	767	const	1	Using index
1	SIMPLE	changesets	ALL	PRIMARY	NULL	NULL	NULL	70917	Using where
1	SIMPLE	nodes	ref	changeset_id	changeset_id	8	osm.changesets.id	21	NULL

3 rows in set (0.00 sec)

```
mysql-5.6.21 (osm) > SELECT ...;
```

node_id	latitude	longitude	changeset_id	visible	timestamp	tile	version
99890	515276425	-1497621	552	1	2005-10-25 00:35:24	2062268512	1
138212838	513010180	-1699929	7757299	1	2011-04-03 18:14:14	2062220563	6

7 rows in set (0.02 sec)



Executed as
a regular
JOIN

First Match Strategy

```
mysql-5.7.5 (osm) > EXPLAIN SELECT * FROM changesets WHERE id IN (SELECT changeset_id FROM nodes)\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: changesets
    partitions: NULL
        type: ALL
possible_keys: PRIMARY
          key: NULL
      key_len: NULL
        ref: NULL
       rows: 70917
  filtered: 100.00
     Extra: NULL
***** 2. row *****
      id: 1
  select_type: SIMPLE
        table: nodes
    partitions: NULL
        type: ref
possible_keys: changeset_id
          key: changeset_id
      key_len: 8
        ref: osm.changesets.id
       rows: 33
  filtered: 100.00
     Extra: Using index; FirstMatch(changesets)
2 rows in set, 1 warning (0.00 sec)
```

It is converting
the ref into an
eq_ref, short-
circuiting the
execution

Enabling and disabling materialization, semijoin, etc

```
mysql-5.7.8 (osm) > SHOW VARIABLES like 'optimizer_switch'\G
***** 1. row *****
Variable_name: optimizer_switch
      Value:
index_merge=on,index_merge_union=on,index_merge_sort_union=on,index_merge_intersection=on,engine_condition_pushdown=on,index_condition_pushdown=on,mrr=on,mrr_cost_based=on,block_nested_loop=on,batched_key_access=off,materialization=on,semijoin=on,loosescan=on,firstmatch=on,duplicateweedout=on,subquery_materialization_cost_based=on,use_index_extensions=on,condition_fanout_filter=on,derived_merge=on
1 row in set (0.00 sec)
```

Query Optimization: From 0 to 10 (and up to 5.7)

QUERY PROFILING

Which Queries Should I Optimize First?

- My two favorite methods:
 - pt-query-digest
 - PERFORMANCE_SCHEMA
- I prefer pt-query-digest for long-term reports, PERFORMANCE_SCHEMA for more real-time evaluation and fine-tuning
 - Also, PERFORMANCE_SCHEMA was not “ready” until MySQL 5.6

pt-query-digest

- It is a 3rd party tool written in Perl, originally created by Baron Schwartz
- It requires activation of the slow log:
 - `SET GLOBAL slow_query_log = 1;`
 - `SET long_query_time = 0;`
- In Percona Server and MariaDB it can provide extra information:
 - `SHOW GLOBAL VARIABLES like 'log_slow_verbosity';`



Be careful with extra
IO and latency!

pt-query-digest Execution (I)

```

# 1094.7s user time, 9.4s system time, 141.22M rss, 205.21M vsz
# Current date: Wed Jul 1 07:32:28 2015
# Hostname: db1018
# Files: STDIN
# Overall: 4.66M total, 640 unique, 53.47 QPS, 0.02x concurrency _____
# Time range: 2015-06-30 07:00:10 to 2015-07-01 07:11:37
# Attribute          total      min      max      avg      95%    stddev   median
# ======          ======      ===      ===      ===      ===      ======    ======   ======
# Exec time        1320s     1us      3s    283us    332us     3ms    152us
# Lock time       238s      0      13ms    51us    93us    39us    52us
# Rows sent       5.02M      0      4.16k    1.13    1.96    8.69    0.99
# Rows examine    9.50M      0    232.93k    2.14    3.89  261.15    0.99
# Merge passes      0      0      0      0      0      0      0
# Query size      1.06G     17    67.89k   243.89   511.45   368.99   192.76
# Boolean:
# Filesort        8% yes,  91% no
# Full scan       94% yes,   5% no
# Priority que    3% yes,  96% no
# Tmp table       29% yes,  70% no
# Tmp table on    1% yes,  98% no

```

Actual execution
on Wikipedia
production
servers

pt-query-digest Execution (II)

```

# Profile
# Rank Query ID          Response time   Calls   R/Call V/M    Item
# ===== ======          ====== ======   =====  =====  =====
#   1 0xSANITIZED        242.2765 18.4%  691005 0.0004  0.00  SELECT revision page user
#   2 0xSANITIZED        204.7052 15.5%  80863  0.0025  0.01  SELECT revision page user
#   3 0xSANITIZED        162.8476 12.3%  1025179 0.0002  0.00  SELECT page
#   4 0xSANITIZED        68.1164  5.2%   93928  0.0007  0.01  SELECT revision page user
#   5 0xSANITIZED        66.8302  5.1%   354562 0.0002  0.00  SELECT page revision
#   6 0xSANITIZED        57.0374  4.3%   211631 0.0003  0.00  SELECT page revision
#   7 0xSANITIZED        44.0751  3.3%    6925  0.0064  0.07  SELECT page categorylinks
category
#   8 0xSANITIZED        35.0655  2.7%   9689  0.0036  0.00  SELECT text
#   9 0xSANITIZED        29.4363  2.2%  152259 0.0002  0.00  SELECT page
#  10 0xSANITIZED        24.1864  1.8%  176927 0.0001  0.00  SELECT msg_resource
#  11 0xSANITIZED        23.7016  1.8%  144807 0.0002  0.00  SELECT page_restrictions
#  12 0xSANITIZED        16.6547  1.3%   10135 0.0016  0.03  SELECT revision
#  13 0xSANITIZED        15.0564  1.1%  263809 0.0001  0.00  SET

```

pt-query-digest Execution (III)

```

# Query 1: 7.93 QPS, 0.00x concurrency, ID 0xSANITIZED at byte 1553864032
# This item is included in the report because it matches -limit.
# Scores: V/M = 0.00
# Time range: 2015-06-30 07:00:10 to 2015-07-01 07:11:37
# Attribute   pct   total      min      max      avg     95% stddev median
# ======  ==  =====  ======  ======  ======  ======  ======  ======
# Count       14  691005
# Exec time    18   242s   163us    91ms   350us   348us   563us   301us
# Lock time    26   63s    47us     7ms    91us   103us   14us    84us
# Rows sent    12  657.18k      0      1    0.97    0.99    0.16    0.99
# Rows examine   6  657.18k      0      1    0.97    0.99    0.16    0.99
# Query size   31  345.42M    501    749  524.16   537.02   9.22   511.45
# String:
# Databases    itwiki (225976/32%), enwiktiona... (219461/31%)... 15 more
# Hosts
# Users        wikiuser
# Query_time distribution
#   1us
#  10us
# 100us #####
#  1ms #
# 10ms #
# 100ms
#   1s
# 10s+
# Tables
#   SHOW TABLE STATUS FROM `enwiktionary` LIKE 'revision'\G
#   SHOW CREATE TABLE `enwiktionary`.`revision`\G
#   SHOW TABLE STATUS FROM `enwiktionary` LIKE 'page'\G
#   SHOW CREATE TABLE `enwiktionary`.`page`\G
#   SHOW TABLE STATUS FROM `enwiktionary` LIKE 'user'\G
#   SHOW CREATE TABLE `enwiktionary`.`user`\G
# EXPLAIN /*!50100 PARTITIONS*/
SELECT /* Revision::fetchFromConds SANITIZED */ * FROM `revision`
INNER JOIN `page` ON ((page_id = rev_page)) LEFT JOIN `user` ON
((rev_user != 0) AND (user_id = rev_user)) WHERE page_namespace = '0' AND
page_title = 'SANITIZED' AND (rev_id=page_latest) LIMIT 1\G

```

PERFORMANCE_SCHEMA

- Monitoring schema (engine) enabled by default since MySQL 5.6
 - `performance_schema = 1` (it is not dynamic)
- Deprecates the old query profiling
- It is way more user-friendly when combined with the [SYS schema/ps_helper](#) (a set of views and stored procedures created by Mark Leith)
 - Included by default since 5.7.7

Installation of the SYS Schema for 5.6/MariaDB

```
$ git clone https://github.com/MarkLeith/mysql-sys.git
Cloning into 'mysql-sys'...
remote: Counting objects: 926, done.
remote: Compressing objects: 100% (73/73), done.
remote: Total 926 (delta 35), reused 6 (delta 2)
Receiving objects: 100% (926/926), 452.19 KiB | 225.00
KiB/s, done.
Resolving deltas: 100% (584/584), done.
$ cd mysql-sys/
$ ~/sandboxes/msb_5_6_24/use < sys_56.sql
```

Example Usage: Discovering Unused Indexes

```
mysql-5.7.8 (osm) > SELECT * FROM sys.schema_unused_indexes LIMIT 5;
```

object_schema	object_name	index_name
osm	acls	acls_k_idx
osm	changeset_tags	changeset_tags_id_idx
osm	current_nodes	current_nodes_timestamp_idx
osm	current_nodes	current_nodes_tile_idx
osm	current_relations	current_relations_timestamp_idx

5 rows in set (0.04 sec)

```
mysql-5.7.8 (osm) > SELECT * FROM current_nodes WHERE tile = 100;
```

...

```
mysql-5.7.8 (osm) > SELECT * FROM sys.schema_unused_indexes LIMIT 5;
```

object_schema	object_name	index_name
osm	acls	acls_k_idx
osm	changeset_tags	changeset_tags_id_idx
osm	current_nodes	current_nodes_timestamp_idx
osm	current_relations	current_relations_timestamp_idx
osm	current_relations	changeset_id

5 rows in set (0.03 sec)



With enough activity, it can help us clean up our schema

Example Usage: Slow Queries (ordered by server time)

```
mysql-5.7.8 (osm) > SELECT * FROM sys.statement_analysis LIMIT 10\G
***** 1. row *****
query: SELECT `way_id` AS id , `v` FROM `way_tags` WHERE `v` LIKE ? rows_examined: 20152155
db: osm rows_examined_avg: 1343477
full_scan: * rows_affected: 0
exec_count: 15 rows_affected_avg: 0
err_count: 0 tmp_tables: 0
warn_count: 0 tmp_disk_tables: 0
total_latency: 7.83 s rows_sorted: 0
max_latency: 1.33 s sort_merge_passes: 0
avg_latency: 521.84 ms digest:
lock_latency: 17.94 ms 21f90695b1ebf20a5f4d4c1e5e860f58
rows_sent: 6779 first_seen: 2014-11-01 17:04:51
rows_sent_avg: 452 last_seen: 2014-11-01 17:05:22
```

Example Usage: Top Queries Creating Temporary Tables

```
mysql-5.7.8 (osm) > SELECT * FROM
sys.statements_with_temp_tables WHERE db = 'osm' LIMIT 10\G
*****
 1. row ****
      query: SELECT ? AS TYPE , `node_id` A ... gs` WHERE `k` = ? AND `v` = ?
      db: osm
  exec_count: 11
 total_latency: 7.57 s
memory_tmp_tables: 11
 disk_tmp_tables: 0
avg_tmp_tables_per_query: 1
tmp_tables_to_disk_pct: 0
      first_seen: 2014-11-01 17:33:55
      last_seen: 2014-11-01 17:34:45
      digest: 5e6e82799b7c7c0e5c57cf63eb98d5d
```

Example Usage: Top Queries

Creating Temporary Tables (cont.)

```
mysql-5.7.8 (osm) > SELECT DIGEST_TEXT FROM performance_schema.events_statements_summary_by_digest
WHERE digest = '5e6e82799b7c7c0e5c57cfe63eb98d5d'\G
*****
1. row ****
DIGEST_TEXT: SELECT ? AS TYPE , `node_id` AS `id` FROM `node_tags` WHERE `k` = ? AND `v` = ? UNION
SELECT ? AS TYPE , `way_id` AS `id` FROM `way_tags` WHERE `k` = ? AND `v` = ? UNION SELECT ? AS
TYPE , `relation_id` AS `id` FROM `relation_tags` WHERE `k` = ? AND `v` = ?
1 row in set (0.00 sec)
```

```
mysql-5.7.8 (osm) > EXPLAIN SELECT 'node' as type, node_id as id FROM node_tags WHERE k='amenity'
and v='cafe' UNION SELECT 'way' as type, way_id as id FROM way_tags WHERE k='amenity' and v='cafe'
UNION SELECT 'relation' as type, relation_id as id FROM relation_tags WHERE k='amenity' and
v='cafe';
```

id select_type table partitions type possible_keys key key_len ref rows filtered Extra
1 PRIMARY node_tags NULL ALL NULL NULL NULL NULL 851339 0.00 Using where
2 UNION way_tags NULL ALL NULL NULL NULL NULL 1331016 0.00 Using where
3 UNION relation_tags NULL ALL NULL NULL NULL NULL 63201 0.00 Using where
NULL UNION RESULT <union1,2,3> NULL ALL NULL NULL NULL NULL NULL NULL Using temporary

4 rows in set, 1 warning (0.01 sec)

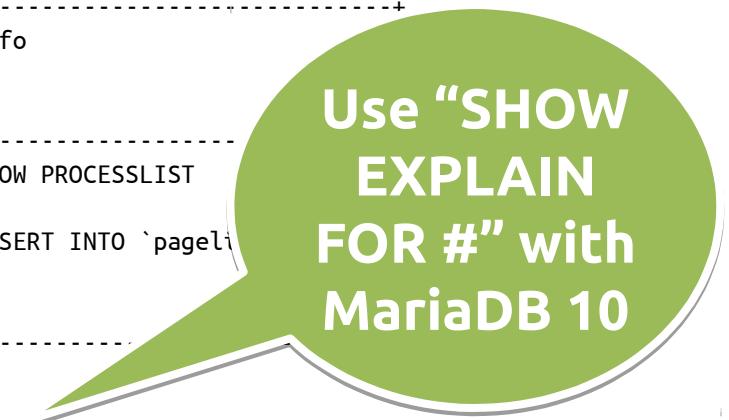
Query Optimization: From 0 to 10 (and up to 5.7)

GENERAL OPTIMIZER IMPROVEMENTS

EXPLAIN FOR CONNECTION (SHOW EXPLAIN FOR)

```
mysql 5.7.8 > SHOW PROCESSLIST;
+----+-----+-----+-----+-----+-----+
| Id | User      | Host      | db           | Command | Time   | State       | Info
+----+-----+-----+-----+-----+-----+
| 4  | msandbox  | localhost | NULL        | Query    | 1      | starting    | SHOW PROCESSLIST
| 8  | msandbox  | localhost | nlwiktionary | Query    | 6      | update      | INSERT INTO `pageli
(74239,0,0,'WikiWoordenboek:Genus'),(74240,0,0,'WikiWoordenboek:Genus'
+----+-----+-----+-----+-----+-----+
+-----+
2 rows in set (0.14 sec)
```

```
mysql 5.7.8 > EXPLAIN FOR CONNECTION 8;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table      | partitions | type   | possible_keys | key    | key_len | ref    | rows   | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1  | INSERT      | pagelinks  | NULL       | ALL    | NULL         | NULL   | NULL    | NULL   | NULL   | NULL    | NULL |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.03 sec)
```



Use “SHOW EXPLAIN FOR #” with MariaDB 10

MySQL 5.7 Optimizer Tweaks

```
mysql-5.5.40 (osm) > SELECT *  
    FROM nodes  
    JOIN node_tags  
    ON node_tags.node_id = nodes.node_id  
    WHERE nodes.latitude  
        BETWEEN 517000000 and 520000000\G  
59 rows in set (1.37 sec)
```

```
mysql-5.5.40 (osm) > SELECT STRAIGHT_JOIN *  
    FROM nodes  
    JOIN node_tags  
    ON node_tags.node_id =  
    WHERE nodes.latitude  
        BETWEEN 517000000 and 520000000\G  
59 rows in set (0.86 sec)
```

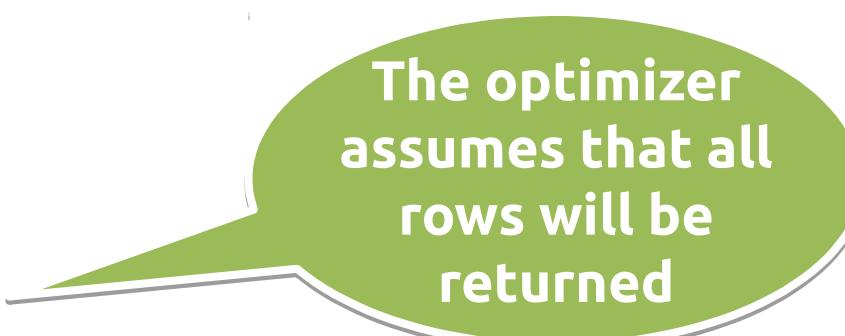
This condition is very selective, but there is no index available

Why does it take the wrong table order?

```
mysql-5.5.40 (osm) > EXPLAIN EXTENDED SELECT * FROM nodes JOIN node_tags ON node_tags.node_id = nodes.node_id WHERE nodes.latitude BETWEEN 517000000 and 520000000\G
***** 1. row *****
    id: 1
  select_type: SIMPLE
        table: node_tags
        type: ALL
possible_keys: PRIMARY
        key: NULL
      key_len: NULL
        ref: NULL
      rows: 839031
  filtered: 100.00
    Extra:
***** 2. row *****
    id: 1
  select_type: SIMPLE
        table: nodes
        type: ref
possible_keys: PRIMARY
        key: PRIMARY
      key_len: 8
        ref: osm.node_tags.node_id
      rows: 1
  filtered: 100.00
    Extra: Using where
2 rows in set, 1 warning (0.00 sec)
```

Unindexed Columns Are not Accounted

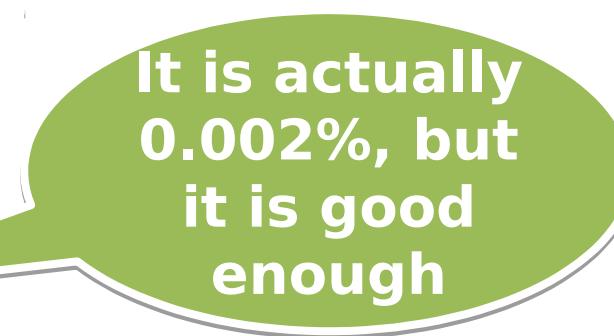
```
mysql-5.5.40 (osm) > EXPLAIN EXTENDED SELECT STRAIGHT_JOIN * FROM nodes JOIN node_tags ON node_tags.node_id = nodes.node_id WHERE nodes.latitude BETWEEN 517000000 and 520000000\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: nodes
         type: ALL
possible_keys: PRIMARY
           key: NULL
      key_len: NULL
         ref: NULL
        rows: 2865521
  filtered: 100.00
    Extra: Using where
***** 2. row *****
      id: 1
  select_type: SIMPLE
        table: node_tags
         type: ref
possible_keys: PRIMARY
           key: PRIMARY
      key_len: 8
         ref: osm.nodes.node_id
        rows: 1
  filtered: 100.00
    Extra:
2 rows in set, 1 warning (0.00 sec)
```



The optimizer assumes that all rows will be returned

What's new in 5.7?

```
mysql-5.7.5 (osm) > EXPLAIN SELECT * FROM nodes JOIN node_tags ON node_tags.node_id = nodes.node_id WHERE nodes.latitude BETWEEN 517000000 and 520000000\G
***** 1. row *****
    id: 1
  select_type: SIMPLE
        table: nodes
    partitions: NULL
       type: ALL
possible_keys: PRIMARY
         key: NULL
      key_len: NULL
        ref: NULL
       rows: 2773853
  filtered: 11.11
    Extra: Using where
***** 2. row *****
    id: 1
  select_type: SIMPLE
        table: node_tags
    partitions: NULL
       type: ref
possible_keys: PRIMARY
         key: PRIMARY
      key_len: 8
        ref: osm.nodes.node_id
       rows: 3
  filtered: 100.00
    Extra: NULL
2 rows in set, 1 warning (0.00 sec)
```



It is actually
0.002%, but
it is good
enough

If things go wrong -our Friend optimizer_switch

```
mysql-5.7.8 (osm) > SET optimizer_switch='condition_fanout_filter=off';  
Query OK, 0 rows affected (0.00 sec)
```

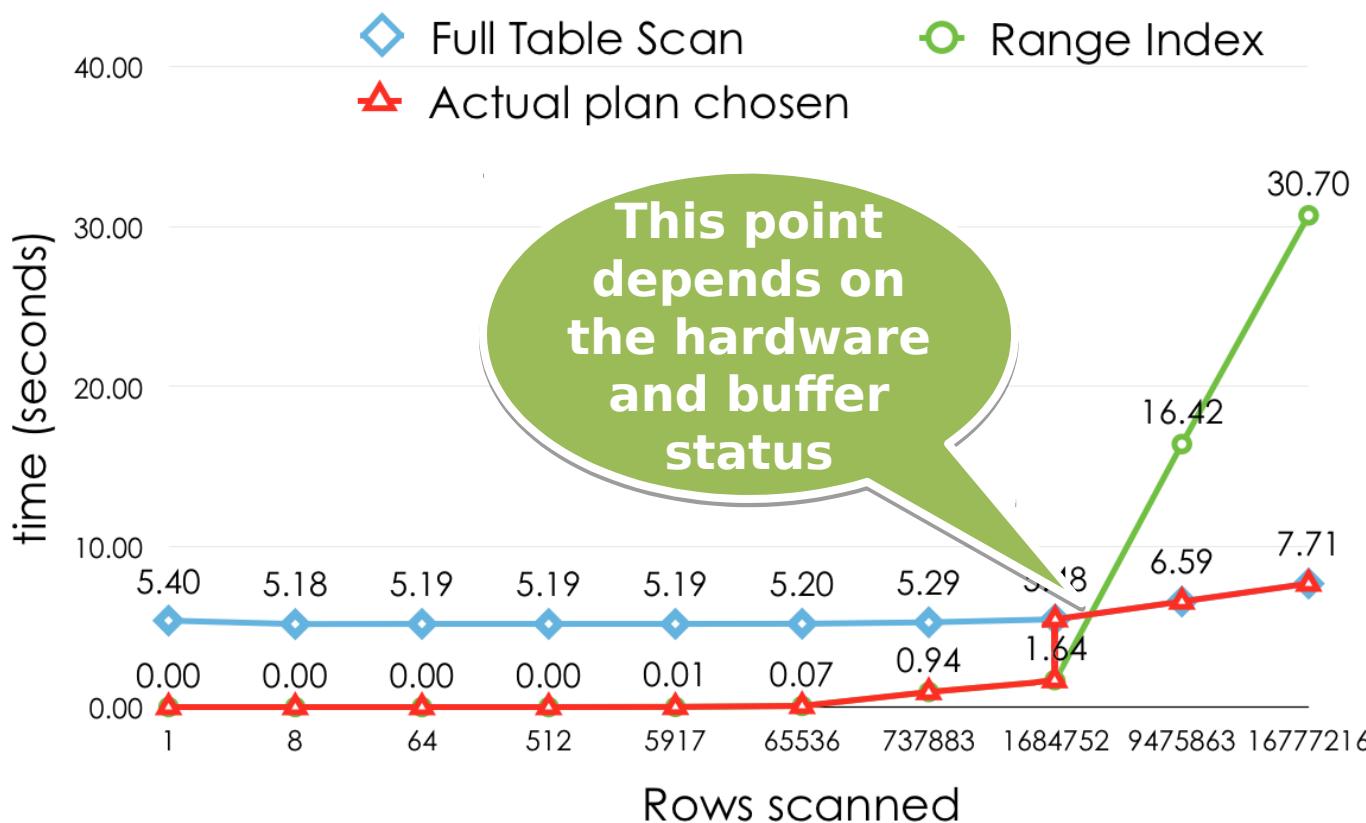
Improved EXPLAIN FORMAT=JSON

```
"nested_loop": [  
  {  
    "table": {  
      "table_name": "nodes",  
      "access_type": "ALL",  
      "possible_keys": [  
        "PRIMARY"  
      ],  
      "rows_examined_per_scan": 2773853,  
      "rows_produced_per_join": 308175,  
      "filtered": 11.11,  
      "cost_info": {  
        "read_cost": "512783.58",  
        "eval_cost": "61635.01",  
        "prefix_cost": "574418.60",  
        "data_read_per_join": "21M"  
      },  
    }]
```

Cost information is now included

Engine statistics are now floats for improved precision

Optimizer Cost Tuning



Configurable Costs

```
mysql 5.7.8> SELECT * FROM mysql.server_cost;
```

cost_name	cost_value	last_update	comment
disktemptable_create_cost	NULL	2015-09-20 20:48:10	NULL
disktemptable_row_cost	NULL	2015-09-20 20:48:10	NULL
key_compare_cost	NULL	2015-09-20 20:48:10	NULL
memorytemptable_create_cost	NULL	2015-09-20 20:48:10	NULL
memorytemptable_row_cost	NULL	2015-09-20 20:48:10	NULL
row_evaluate_cost	NULL	2015-09-20 20:48:10	NULL

6 rows in set (0.00 sec)

The unit is
“read of a
random
data page”

```
mysql 5.7.8> SELECT * FROM mysql.engine_cost;
```

engine_name	device_type	cost_name	cost_value	last_update	comment
default	0	io_block_read_cost	NULL	2015-09-20 20:48:10	NULL
default	0	memory_block_read_cost	NULL	2015-09-20 20:48:10	NULL

2 rows in set (0.00 sec)

Changing Costs Example

```
mysql-5.7.5 (osm) > EXPLAIN FORMAT=JSON SELECT 'node' as type, node_id as id FROM
node_tags WHERE k='amenity' and v='cafe' UNION SELECT 'way' as type, way_id as id FROM
way_tags WHERE k='amenity' and v='cafe' UNION SELECT 'relation' as type, relation_id as
id FROM relation_tags WHERE k='amenity' and v='cafe'\G
    "cost_info": {
        "query_cost": "22567.80"
1 row in set, 1 warning (0.00 sec)

mysql-5.7.5 (osm) > UPDATE mysql.server_cost SET cost_value = 10;
mysql-5.7.5 (osm) > FLUSH OPTIMIZER_COSTS;
<session restart>
mysql-5.7.5 (osm) > EXPLAIN ... \G
    "cost_info": {
        "query_cost": "661371.00"
1 row in set, 1 warning (0.00 sec)
```

More info on usage in the manual: <http://dev.mysql.com/doc/refman/5.7/en/cost-model.html>

Still waiting for...

- Utility to analyze the underlying technology (HD, SSD, memory) and filling up the tables automatically
- Buffer “hotness”-aware statistics
- Statistics for non-indexed columns/histograms
 - MariaDB has histograms since 10

Engine-Independent Statistics/Histograms in MariaDB 10

```
mariadb-10.0.14 (osm) > SET histogram_size = 255;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
mariadb-10.0.14 (osm) > SET use_stat_tables = 2;
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
mariadb-10.0.14 (osm) > ANALYZE TABLE node_tags;
```

Table	Op	Msg_type	Msg_text
osm.node_tags	analyze	status	Engine-independent statistics collected
osm.node_tags	analyze	status	OK

```
2 rows in set (3.01 sec)
```

```
mariadb-10.0.14 (osm) > ANALYZE TABLE nodes;
```

Table	Op	Msg_type	Msg_text
osm.nodes	analyze	status	Engine-independent statistics collected
osm.nodes	analyze	status	OK

```
2 rows in set (32.52 sec)
```

```
mariadb-10.0.14 (osm) > SET optimizer_use_condition_selectivity = 3; (or 4 for histograms)
```

```
Query OK, 0 rows affected (0.00 sec)
```

Better Stats!

```
mariadb-10.0.14 (osm) > EXPLAIN EXTENDED SELECT * FROM nodes JOIN node_tags ON node_tags.node_id = nodes.node_id WHERE nodes.latitude BETWEEN 517000000 and 520000000\G
***** 1. row *****
    id: 1
  select_type: SIMPLE
      table: nodes
      type: ALL
possible_keys: PRIMARY
      key: NULL
     key_len: NULL
       ref: NULL
      rows: 2865312
  filtered: 0.39
     Extra: Using where
***** 2. row *****
    id: 1
  select_type: SIMPLE
      table: node_tags
      type: ref
possible_keys: PRIMARY
      key: PRIMARY
     key_len: 8
       ref: osm.nodes.node_id
      rows: 3
  filtered: 100.00
     Extra:
2 rows in set, 1 warning (0.01 sec)
```



Much better estimation

Other Changes/Bugs Fixed

- UNION ALL does not create temporary tables, returns tables faster
- (a, b) IN ((1, 2), (2, 3)) can use index ranges
- EXPLAIN EXTENDED is now the default behavior
- I.C. Pushdown support for partitioned tables
- IGNORE clause meaning has been standardized between sentence types
- Increased default for optimizer_search_depth

Query Optimization: From 0 to 10 (and up to 5.7)

COMPUTED/VIRTUAL COLUMNS

Syntax

ALTER TABLE nodes

```
ADD COLUMN lon DECIMAL (10, 7)
    as (longitude/1000000) VIRTUAL,
ADD COLUMN lat DECIMAL (9, 7)
    as (latitude/1000000) VIRTUAL;
```

VIRTUAL is
optional
(default)

- They can be used to simplify SELECTS, calculating values on the fly
- Non accessed rows are not calculated

Functional Indexes

- Before 5.7.8:

```
mysql-5.7.5 (osm) > ALTER TABLE nodes add index(lon);
ERROR 1951 (HY000): Key/Index cannot be defined on a non-
stored virtual column.
```

- Now:

```
mysql-5.7.8 (osm) > ALTER TABLE nodes add index(lon);
Query OK, 0 rows affected (16.54 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

- This effectively is an implementation of functional indexes, allowing to solve previous query optimization issues without the overhead of an extra column

Do you remember this query?

```
MariaDB [nlwictionary]> EXPLAIN SELECT * FROM revision WHERE
substr(rev_timestamp, 5, 2) = '09'\G
***** 1. row *****
      id: 1
select_type: SIMPLE
      table: revision
        type: ALL
possible_keys: NULL
          key: NULL
    key_len: NULL
      ref: NULL
     Rows: 173154
    Extra: Using where
1 row in set (0.00 sec)
```

Can you think a way to
improve this query?

Now we can solve it like this

```
mysql-5.7.8> ALTER TABLE revision ADD COLUMN rev_month tinyint AS  
(substr(rev_timestamp, 5, 2)) VIRTUAL, ADD INDEX rev_month (rev_month);  
Query OK, 820308 rows affected (3 min 29.48 sec)  
Records: 820308 Duplicates: 0 Warnings: 0
```

```
mysql-5.7.8> EXPLAIN SELECT * FROM revision WHERE rev_month = 9\G  
*****  
      1. row *****  
      id: 1  
 select_type: SIMPLE  
       table: revision  
    partitions: NULL  
        type: ref  
possible_keys: rev_month  
          key: rev_month  
     key_len: 2  
       ref: const  
      rows: 104112  
 filtered: 100.00  
    Extra: NULL  
1 row in set, 1 warning (0.01 sec)
```

The column does not take space, only the index

Stored Columns

```
mysql-5.7.8 (osm) > ALTER TABLE nodes CHANGE lat lat DECIMAL (9, 7) as  
(latitude/10000000) STORED;  
ERROR 1954 (HY000): 'Changing the STORED status' is not supported for virtual  
columns.
```

They dropped the 'yet' :-)

```
mysql-5.7.8 (osm) > ALTER TABLE nodes DROP COLUMN lat, ADD COLUMN lat DECIMAL  
(9, 7) as (latitude/10000000) STORED;  
Query OK, 2865312 rows affected (4 min 51.05 sec)  
Records: 2865312 Duplicates: 0 Warnings: 0
```

MariaDB uses
the
'PERMANENT'
keyword

Features and Limitations

- ~~Virtual “non-stored” columns cannot be indexed~~
- “Stored columns” can be PRIMARY, UNIQUE, FOREIGN and MULTI keys
- It cannot contain subqueries or other tables or rows references
- It cannot contain user-defined stored functions
- It can contain server and user-defined variables
- It can be computed based on other virtual columns

Query Optimization: From 0 to 10 (and up to 5.7)

QUERY REWRITE PLUGINS

New APIs for Query Rewriting

- One for pre-parsing rewriting
- Another for post-parsing rewriting

Example Plugin Installation

```
$ ~/sandboxes/msb_5_7_8/my sql <  
./share/install_rewriter.sql
```

More on this: <http://mysqlserverteam.com/the-query-rewrite-plugins/>

Query Rewriting Setup

```
mysql-5.7.8 > INSERT INTO query_rewrite.rewrite_rules (pattern, replacement) VALUES ('SELECT ?', 'SELECT ? + 1');
Query OK, 1 row affected (0.01 sec)
```

```
mysql-5.7.8 > CALL query_rewrite.flush_rewrite_rules();
Query OK, 0 rows affected (0.01 sec)
```

```
mysql-5.7.8 > SELECT 1;
+-----+
| 1 + 1 |
+-----+
|      2 |
+-----+
1 row in set, 1 warning (0.00 sec)
```

```
mysql-5.7.8 > SHOW WARNINGS;
+-----+-----+-----+
| Level | Code | Message           |
+-----+-----+-----+
| Note  | 1105 | Query 'SELECT 1' rewritten to 'SELECT 1 + 1' by a query rewrite plugin |
+-----+-----+-----+
1 row in set (0.00 sec)
```

Considerations

- It cannot correct malformed queries- pattern and replacement must be syntactically correct
- Useful for query optimizations for 3rd party applications
- User is notified that a rewrite happened with a note-level message
- Low overhead (5%); specially for untouched queries
- You can do stupid things like:

```
mysql-5.7.5 ((none)) > INSERT INTO
query_rewrite.rewrite_rules( pattern, replacement ) VALUES
( 'SELECT 1', 'DROP TABLE test.test' );
Query OK, 1 row affected (0.01 sec)
```

Query Optimization: From 0 to 10 (and up to 5.7)

OPTIMIZER HINTS

New functionality since 5.7.7

- MySQL accepts hints for query execution with the syntax /*+ ... */:

```
mysql> EXPLAIN SELECT /*+ NO_ICP(revision)*/ * FROM revision WHERE rev_comment like 'jaime%' AND rev_timestamp > '2008'\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: revision
    partitions: NULL
        type: range
possible_keys:
rev_timestamp,rev_timestamp_rev_page,rev_timestamp_2,rev_comment_rev_timestamp
          key: rev_comment_rev_timestamp
      key_len: 783
        ref: NULL
       rows: 1
  filtered: 50.00
     Extra: Using where
1 row in set, 1 warning (0.00 sec)
```

More info on this feature:

<https://www.percona.com/blog/2015/04/30/optimizer-hints-mysql-5-7-7-missed-manual/>

Syntax

- The syntax is identical to the one Oracle Database uses, but does not deprecate yet the old hint syntax (USE INDEX, STRAIGHT_JOIN, ...)
 - “planned”
- It has some overlap with optimizer_switch
 - Although it has lower granularity (table instead of statement)

Max Query Execution

- Indicate it in milliseconds:

```
mysql> SELECT /*+ MAX_EXECUTION_TIME(1784) */ SLEEP(10)\G  
*****  
1. row *****  
SLEEP(2000): 1  
1 row in set (1.78 sec)
```

- `max_statement_time` variable was renamed to `max_execution_time` since 5.7.8 to match the hint

Other hints

- BKA / NO_BKA
- BNL / NO_BNL
- MRR / NO_MRR
- NO_ICP
- NO_RANGE_OPTIMIZATION
- QB_NAME (controls the query block where to apply the hint to)

Query Optimization: From 0 to 10 (and up to 5.7)

SQL MODE CHANGES

Default SQL Mode Changes

- MySQL 5.5 and earlier
 - ''
- MySQL 5.6 (from 5.6.6):
 - '**NO_ENGINE_SUBSTITUTION**' is the default
 - NO_ENGINE_SUBSTITUTION and STRICT_TRANS_TABLES were suggested in upstream default my.cnf
- MySQL 5.7 (from 5.7.5):
 - '**ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ENGINE_SUBSTITUTION**' is the new default

More on this: http://www.tocker.ca/2014/01/14/making-strict-sql_mode-the-default.html

Stricter Defaults

```
mysql> CREATE TABLE `test` (
    `id` int(11) NOT NULL,
    `type` enum('movie','album','videogame') NOT NULL,
    `date` datetime NOT NULL,
    PRIMARY KEY (`id`)
) ENGINE=InnoDB;
```

```
mysql> INSERT INTO test (type, date) VALUES ('tv show', -1);
```



What
happens?

MySQL 5.5/5.6 with default settings

```
mysql> INSERT INTO test (type, date) VALUES ('tv show', -1);
Query OK, 1 row affected, 3 warnings (0.00 sec)
```

```
mysql> SHOW WARNINGS;
+-----+-----+
| Level | Code | Message
+-----+-----+
| Warning | 1364 | Field 'id' doesn't have a default value
| Warning | 1265 | Data truncated for column 'type' at row 1
| Warning | 1264 | Out of range value for column 'date' at row 1
+-----+-----+
3 rows in set (0.00 sec)
```

```
mysql> SELECT * FROM test;
+-----+-----+
| id | type | date
+-----+-----+
| 0 |      | 0000-00-00 00:00:00 |
+-----+-----+
1 row in set (0.00 sec)
```

MySQL 5.7

```
mysql> INSERT INTO test (type, date) VALUES ('tv show', -1);
ERROR 1265 (01000): Data truncated for column 'type' at row 1
```

```
mysql> INSERT INTO test (type, date) VALUES ('videogame', -1);
ERROR 1292 (22007): Incorrect datetime value: '-1' for column 'date' at row
1
```

```
mysql> INSERT INTO test (type, date) VALUES ('movie', now());
ERROR 1364 (HY000): Field 'id' doesn't have a default value
```

```
mysql> INSERT INTO test (id, type, date) VALUES (1, 'videogame', now());
Query OK, 1 row affected (0.01 sec)
```

GROUP BY Behavior Changes

```
mysql-5.6.21 (osm) > SELECT way_id, count(*), node_id  
    FROM way_nodes  
   GROUP BY way_id  
  ORDER BY count(*) DESC  
 LIMIT 10;
```

way_id	count(*)	node_id
155339744	1187	1558095871
243986064	1156	2604713337
87136668	1128	1013304944
148812873	852	1618837453
149200774	835	34921158
183618216	826	1940223096
273858696	824	1267549776
261584374	770	2669122104
227880171	704	2240011804
193564006	684	1808872763



Non-deterministic behavior

With ONLY_FULL_GROUP_BY (default in 5.7)

```
mysql-5.7.5 (osm) > SELECT way_id, count(*), node_id  
    FROM way_nodes  
    GROUP BY way_id  
    ORDER BY count(*) DESC  
    LIMIT 10;
```

ERROR 1055 (42000): Expression #3 of SELECT list is not in GROUP BY clause and contains nonaggregated column 'osm.way_nodes.node_id' which is not functionally dependent on columns in GROUP BY clause; this is incompatible with sql_mode=only_full_group_by

Problem with ONLY_FULL_GROUP_BY in MySQL <= 5.6

```
mysql-5.6.21 (osm) > SET SQL_mode='ONLY_FULL_GROUP_BY';
Query OK, 0 rows affected (0.00 sec)
```

```
mysql-5.6.21 (osm) > SELECT
        u.id as `user id`,
        u.display_name as `user name`,
        count(*) as `# changesets`
    FROM users u
    IN changesets c
    ON u.id = c.user_id
    GROUP BY u.id
    ORDER BY count(*) DESC
    LIMIT 10;
```

Functional
dependenc
y

ERROR 1055 (42000): 'osm.u.display_name' isn't in GROUP BY

More on this: <http://rpbouman.blogspot.com/2014/09/mysql-575-group-by-respects-functional.html>

5.7 Aims for SQL99 Compliance

```
mysql-5.7.5 (osm) > SELECT u.id as `user id`, u.display_name as `user name`, count(*) as `# changesets` FROM users u JOIN changesets c ON u.id = c.user_id GROUP BY u.id ORDER BY count(*) DESC LIMIT 10;
```

user id	user name	# changesets
31257	Ed Avis	4333
508	Welshie	2696
1016290	Amaroussi	2351
352985	ecatmur	1450
142807	SDavies	1342
736	Steve Chilton	1182
346	Tom Chance	1175
38784	Tom Morris	1165
88718	UrbanRambler	1151
1611	Harry Wood	1091

```
10 rows in set (0.09 sec)
```

Backward Compatibility

- Some ORMs and frameworks change the default SQL Mode:
 - Ruby on Rails 4+ sets STRICT_ALL_TABLES
 - Drupal 7+ sets TRADITIONAL
 - Mediawiki will set TRADITIONAL, ONLY_FULL_GROUP_BY
- Other applications do not work in standard-compliance modes:
 - Wordpress used to not work in strict mode (fixed):
<http://www.xaprb.com/blog/2013/03/15/wordpress-and-mysqls-strict-mode/>
 - Cacti does not work with ONLY_FULL_GROUP_BY,NO_ZERO_DATE

Deprecated Modes

- `ERROR_FOR_DIVISION_BY_ZERO`, `NO_ZERO_DATE`, and `NO_ZERO_IN_DATE` were considered deprecated
 - To avoid compatibility problems on upgrade (e.g. 5.6 → 5.7 replication) they still maintain their meaning **bug #75439**
 - The default (and recommended) value is `ONLY_FULL_GROUP_BY`, `STRICT_TRANS_TABLES`, `NO_ZERO_IN_DATE`, `NO_ZERO_DATE`, `ERROR_FOR_DIVISION_BY_ZERO`, `NO_AUTO_CREATE_USER`, and `NO_ENGINE_SUBSTITUTION`

Query Optimization: From 0 to 10 (and up to 5.7)

GIS IMPROVEMENTS & JSON TYPES

Find the Closest Starbucks

```
mysql-5.7.5 (osm) > SET @lat:=51.49353; SET @lon:=-0.18340;
mysql-5.7.5 (osm) > SELECT n.node_id,
   n.longitude/10000000 as longitude,
   n.latitude/10000000 as latitude,
   sqrt(pow((latitude/10000000 - @lat) * 111257.67,
            pow((longitude/10000000 - @lon) * 69450.32, 2,
            as `distance in metres`
   FROM nodes n
   JOIN node_tags n_t1
     ON n.node_id = n_t1.node_id
   JOIN node_tags n_t2
     ON n.node_id = n_t2.node_id
   WHERE
     n_t1.k = 'amenity' and
     n_t1.v = 'cafe' and
     n_t2.k = 'name' and
     n_t2.v = 'Starbucks'
   ORDER BY `distance in metres` ASC
   LIMIT 1;
+-----+-----+-----+
| node_id | longitude | latitude | distance in metres |
+-----+-----+-----+
| 699693936 |    -0.1823 |   51.4945 | 130.9792513096838 |
+-----+-----+-----+
1 row in set (0.20 sec)
```



You are here

This Query is Slow

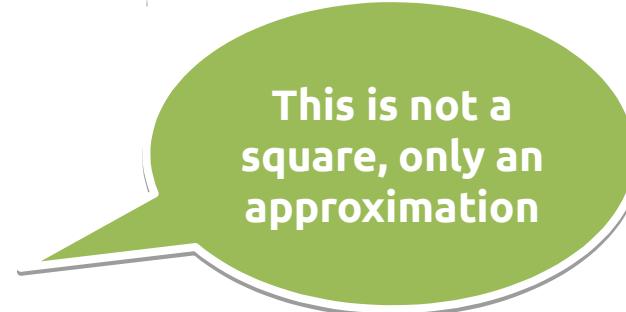
```
mysql-5.7.5 (osm) > EXPLAIN SELECT n.node_id,
n.longitude/10000000 as longitude,
n.latitude/10000000 as latitude,
sqrt(pow((latitude/10000000-@lat)*111257.67, 2) +
pow((longitude/10000000-@lon)*69450.32, 2)) as
`distance in metres` FROM nodes n JOIN node_tags n_t1
ON n.node_id = n_t1.node_id JOIN node_tags n_t2 ON
n.node_id = n_t2.node_id WHERE n_t1.k = 'amenity'
and n_t1.v = 'cafe' and n_t2.k = 'name' and n_t2.v =
'Starbucks' ORDER BY `distance in metres` ASC LIMIT
1\G
***** 1. row *****
    id: 1
  select_type: SIMPLE
      table: n_t1
     partitions: NULL
       type: ALL
possible_keys: PRIMARY
      key: NULL
    key_len: NULL
      ref: NULL
     rows: 832040
  filtered: 0.00
    Extra: Using where; Using temporary; Using
file sort
```

```
***** 2. row *****
    id: 1
  select_type: SIMPLE
      table: n_t2
     partitions: NULL
       type: ref
possible_keys: PRIMARY
      key: PRIMARY
    key_len: 8
      ref: osm.n_t1.node_id
     rows: 3
  filtered: 1.41
    Extra: Using where
***** 3. row *****
    id: 1
  select_type: SIMPLE
      table: n
     partitions: NULL
       type: ref
possible_keys: PRIMARY
      key: PRIMARY
    key_len: 8
      ref: osm.n_t1.node_id
     rows: 1
  filtered: 100.00
    Extra: NULL
3 rows in set, 1 warning (0.00 sec)
```

Can We Optimize it?

- We could add a bounding box:

```
mysql-5.7.5 (osm) > ALTER TABLE nodes add index(latitude, longitude);  
mysql-5.7.5 (osm) >  
    SELECT n.node_id,  
          n.longitude/10000000 as longitude,  
          n.latitude/10000000 as latitude,  
          sqrt(pow((latitude/10000000 - @lat) * 111257.67, 2) +  
                pow((longitude/10000000 - @lon) * 69450.32, 2))  
                as `distance in metres`  
    FROM nodes n  
    JOIN node_tags n_t1  
        ON n.node_id = n_t1.node_id  
    JOIN node_tags n_t2  
        ON n.node_id = n_t2.node_id  
    WHERE  
        n_t1.k = 'amenity' and  
        n_t1.v = 'cafe' and  
        n_t2.k = 'name' and  
        n_t2.v = 'Starbucks' and  
        n.latitude BETWEEN ((@lat - 1000/111257.67) * 10000000)  
                      AND ((@lat + 1000/111257.67) * 10000000) and  
        n.longitude BETWEEN ((@lon - 1000/69450.32) * 10000000)  
                      AND ((@lon + 1000/69450.32) * 10000000)  
    ORDER BY `distance in metres` ASC  
    LIMIT 1;
```



This is not a square, only an approximation

We Create an Index... and Force It

```
mysql-5.7.5 (osm) > EXPLAIN SELECT ...;
```

id	select_type	table	partitions	type	possible_keys	Extra					
						key	key_len	ref	rows	filtered	Extra
1	SIMPLE	n_t1	NULL	ALL	PRIMARY	NULL	0		1	1	Using where;
1	SIMPLE	n	NULL	ref	PRIMARY, latitude	PRIMARY	8	osm.n_t1.node_id	1	1	Using temporary;
1	SIMPLE	n_t2	NULL	ref	PRIMARY	PRIMARY	8	osm.n_t1.node_id	3	3	Using filesort
											5.00 Using where
											1.41 Using where

3 rows in set, 1 warning (0.00 sec)

MySQL ignores the
newly created index,
why?

```
mysql-5.7.5 (osm) > EXPLAIN SELECT ... FROM nodes n FORCE INDEX(latitude) ...;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra	
											len	
1	SIMPLE	n	NULL	range	latitude	latitude	8	NULL	493666	11.11	Using where;	Using index;
1	SIMPLE	n_t1	NULL	ref	PRIMARY	PRIMARY	8	osm.n.node_id	3	3	Using filesort	1.41 Using where
1	SIMPLE	n_t2	NULL	ref	PRIMARY	PRIMARY	8	osm.n.node_id	3	3	1.41 Using where	1.41 Using where

3 rows in set, 1 warning (0.00 sec)

Performance improvement is not great

Still many rows are examined

Most of the gain comes from the covering index, not the filtering

node_id	longitude	latitude
699693936	-0.1823	51.4082

1 row in set (0.09 sec)

Multiple Range Scans Cannot Be Optimized with BTREE Indexes

- We need quadtrees or R-TREE Indexes for indexing in multiple dimensions
 - The later are implemented in MySQL with the name “SPATIAL indexes”, as they only apply to GIS types
- Spatial indexing is available for the first time for InnoDB tables on MySQL 5.7.5

Creating a Spatial Index

```
mysql-5.7.5 (osm) > ALTER TABLE nodes  
                      ADD COLUMN coord GEOMETRY NOT NULL;
```

Query OK, 0 rows affected (21.80 sec)

Records: 0 Duplicates: 0 Warnings: 0

```
mysql-5.7.5 (osm) > UPDATE nodes  
                      SET coord = point(longitude/10000000,  
                               latitude/10000000);
```

Query OK, 2865312 rows affected (34.66 sec)

Rows matched: 2865312 Changed: 2865312 Warnings: 0

```
mysql-5.7.5 (osm) > ALTER TABLE nodes add SPATIAL  
index(coord);
```

Query OK, 0 rows affected (1 min 50.00 sec)

Records: 0 Duplicates: 0 Warnings: 0



This is new
in 5.7

New Query

```
mysql> SET @area := envelope(linestring(POINT(@lon - 500/69450.32, @lat - 500/111257.67), POINT(@lon +  
500/69450.32, @lat + 500/111257.67)));  
  
mysql> SELECT n.node_id,  
        x(n.coord) as longitude,  
        y(n.coord) as latitude,  
        st_distance(POINT(@lon, @lat), coord) as distance  
    FROM nodes n  
    JOIN node_tags n_t1  
        ON n.node_id = n_t1.node_id  
    JOIN node_tags n_t2  
        ON n.node_id = n_t2.node_id  
    WHERE  
        n_t1.k = 'amenity' and  
        n_t1.v = 'cafe' and  
        n_t2.k = 'name' and  
        n_t2.v = 'Starbucks' and  
        st_within(coord, @area)  
    ORDER BY st_distance(POINT(@lon, @lat), coord) ASC  
    LIMIT 1;
```



We can use any shape we want thanks to 5.6 improvements

Better Performance

```
mysql-5.7.5 (osm) > SELECT ...;
```

```
+-----+-----+-----+
| node_id | longitude | latitude | distance
+-----+-----+-----+
| 699693936 | -0.1822879 | 51.4944808 | 0.0014631428672541478 |
+-----+-----+-----+
1 row in set (0.02 sec)
```

This field is
almost
useless

```
mysql-5.7.5 (osm) > EXPLAIN SELECT ...;
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select | table | parti | type | possible | key      | key    | ref   | rows | filtered | Extra
|   | _type  |       | tions |       | _keys    | coord    | _len   |       |       |       |       |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | n     | NULL  | range  | PRIMARY  | coord    |        | NULL  | 2    | 100.00  | Using where;
|   |         |       |       |         |          | ,coord   |        |       |       |       |          |
| 1 | SIMPLE | n_t1  | NULL  | ref    | PRIMARY  | PRIMARY   | osm.n.  | node_id| 3    | 1.41    | Using where
|   |         |       |       |         |          |          |        |       |       |       |          |
| 1 | SIMPLE | n_t2  | NULL  | ref    | PRIMARY  | PRIMARY   | osm.n.  | node_id| 3    | 1.41    | Using where
|   |         |       |       |         |          |          |        |       |       |       |          |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
3 rows in set, 1 warning (0.00 sec)
```

Better Filtering

mysql-5.7.5 (osm) > SHOW STATUS LIKE 'Hand%';

Not using the index: Using the BTREE index: Using the SPATIAL index:

Variable_name	Value
Handler_commit	1
Handler_delete	0
Handler_discover	0
Handler_external_lock	6
Handler_mrr_init	0
Handler_prepare	0
Handler_read_first	1
Handler_read_key	1914
Handler_read_last	0
Handler_read_next	1954
Handler_read_prev	0
Handler_read_rnd	1
Handler_read_rnd_next	833426
Handler_rollback	0
Handler_savepoint	0
Handler_savepoint rollback	0
Handler_update	0
Handler_write	1

Variable_name	Value
Handler_commit	1
Handler_delete	0
Handler_discover	0
Handler_external_lock	6
Handler_mrr_init	0
Handler_prepare	0
Handler_read_first	0
Handler_read_key	274
Handler_read_last	0
Handler_read_next	246540
Handler_read_prev	0
Handler_read_rnd	0
Handler_read_rnd_next	0
Handler_rollback	0
Handler_savepoint	0
Handler_savepoint_rollback	0
Handler_update	0
Handler_write	0

18 rows in set (0.00 sec)

Variable_name	Value
Handler_commit	1
Handler_delete	0
Handler_discover	0
Handler_external_lock	6
Handler_mrr_init	0
Handler_prepare	0
Handler_read_first	0
Handler_read_key	522
Handler_read_last	0
Handler_read_next	5254
Handler_read_prev	0
Handler_read_rnd	259
Handler_read_rnd_next	0
Handler_rollback	0
Handler_savepoint	0
Handler_savepoint_rollback	0
Handler_update	0
Handler_write	0

18 rows in set (0.00 sec)

Geohash Functions

```
mysql-5.7.5 (osm) > SELECT ST_GeoHash(@lon, @lat, 10);
```

```
+-----+
| ST_GeoHash(@lon, @lat, 10) |
+-----+
| gcpugy47w3                 |
+-----+
1 row in set (0.00 sec)
```

- Useful to index coordinates with a BTREE
 - It could be specially useful combined with indexed STORED columns (emulating quadtrees)

More on Geohashing: <http://mysqlserverteam.com/geohash-functions/>

GeoJSON Functions

```
mysql-5.7.5 (osm) > SELECT nm.v, ST_AsGeoJson(n.coord)
    FROM node_tags n_t
    JOIN nodes n USING (node_id, version)
    JOIN node_tags nm USING (node_id, version)
   WHERE n_t.k='tourism' AND
         n_t.v='attraction' AND
         nm.k='name';
```

v	ST_AsGeoJson(n.coord)
BedZED	{"type": "Point", "coordinates": [-0.1560632, 51.3821745]}
Blewcoat School	{"type": "Point", "coordinates": [-0.1360484, 51.4983179]}
Camden / Buck Street Market	{"type": "Point", "coordinates": [-0.143193, 51.5400398]}
Camden Lock Village	{"type": "Point", "coordinates": [-0.1447181, 51.5416552]}
Wimbledon Windmill	{"type": "Point", "coordinates": [-0.2315468, 51.4376583]}
.	
.	
.	

GeoJSON Functions (cont.)

```
$ ./sandboxes/msb_5_7_5/use osm -B -e "SET SESSION group_concat_max_len = 10000; SELECT
CONCAT('{"type":"FeatureCollection", "features":[',
GROUP_CONCAT(CONCAT('{"type":"Feature", "geometry":', ST_AsGeoJson(n.coord),
', "properties":{"name":"' ,nm.v,'"}})), ' ]}') FROM node_tags n_t JOIN nodes n USING
(node_id, version) JOIN node_tags nm USING (node_id, version) WHERE n_t.k='tourism' and
n_t.v='attraction' AND nm.k='name'"
```

<http://geojsonlint.com/>

The code editor displays the following JSON:

```
{"type": "FeatureCollection", "features": [{"type": "Feature", "geometry": {"type": "Point", "coordinates": [-0.0794636, 51.5225685]}, "properties": {"name": "clueQuest"}}, {"type": "Feature", "geometry": {"type": "Point", "coordinates": [-0.1214769, 51.5069283]}}, {"type": "Feature", "geometry": {"type": "Point", "coordinates": [-0.0772253, 51.4220538]}}, {"type": "Feature", "geometry": {"type": "Point", "coordinates": [-0.0863928, 51.5044739]}}, {"type": "Feature", "geometry": {"type": "Point", "coordinates": [-0.0096037, 51.4828754]}}]}
```

Clear Current Features

Test GeoJSON **Clear**

Open Issues

- The SRID can be set and retrieved, but all operations are done in squared euclidean coordinates:

```
mysql-5.7.5 (osm) > SET @p1 := GeomFromText('POINT(-1 51)', 4326);
Query OK, 0 rows affected (0.00 sec)
```

```
mysql-5.7.5 (osm) > SET @p2 := GeomFromText('POINT(0 51)', 4326);
Query OK, 0 rows affected (0.00 sec)
```

```
mysql-5.7.5 (osm) > SET @p3 := GeomFromText('POINT(-1 52)', 4326);
Query OK, 0 rows affected (0.00 sec)
```

```
mysql-5.7.5 (osm) > SELECT srid(@p1);
+-----+
| srid(@p1) |
+-----+
|      4326 |
+-----+
1 row in set (0.00 sec)
```

```
mysql-5.7.5 (osm) > SELECT st_distance(@p1, @p2);
+-----+
| st_distance(@p1, @p2) |
+-----+
|          1           |
+-----+
1 row in set (0.00 sec)
```

```
mysql-5.7.5 (osm) > SELECT st_distance(@p1, @p3);
+-----+
| st_distance(@p1, @p3) |
+-----+
|          1           |
+-----+
1 row in set (0.00 sec)
```

New JSON Native Data Type

- Since 5.7.8, MySQL allows columns defined with the JSON data type:

```
mysql> CREATE TABLE json_test(id int PRIMARY KEY  
auto_increment, content JSON);  
Query OK, 0 rows affected (0.03 sec)
```

```
mysql> INSERT INTO json_test (content) VALUES ('{"type":  
"correct_json"}');  
Query OK, 1 row affected (0.00 sec)
```

They get
validated on
insert

```
mysql> INSERT INTO json_test (content) VALUES ('{"type":  
"incorrect_json"}');  
ERROR 3140 (22032): Invalid JSON text: "Missing a closing quotation  
mark in string." at position 24 in value (or column) '{"type":  
"incorrect_json"}'.
```

JSON functions

- MySQL includes almost all functions to manipulate JSON that you may think of:
 - Validation test: `JSON_TYPE`
 - Object creation: `JSON_ARRAY`, `JSON_MERGE`, ...
 - Searching: `JSON_EXTRACT`
 - Modifying: `JSON_SET`, `JSON_INSERT`, ...

Indexing JSON

- JSON Columns cannot be indexed:

```
mysql [localhost] {msandbox} (test) > ALTER TABLE  
json_test ADD INDEX(content);  
ERROR 3152 (42000): JSON column 'content' cannot be  
used in key specification.
```

- However, they can be compared with regular fields and use indexes thanks to virtual columns

Query Optimization: From 0 to 10 (and up to 5.7)

CONCLUSIONS

5.7/10.1 About to be released

- Both are currently in Release Candidate
- Unless you are desperate for a feature, skip the first releases (or backport it to your current version)

MySQL 5.7 New Features

- MySQL 5.6 seemed Percona Server-inspired
- MySQL 5.7 seems MariaDB/Galera-inspired
 - Competition is always good for consumer



Many Optimizer Advantages Have to Be Manually Enabled

- Modifying on a per-query basis:

```
SET optimizer_switch='batched_key_access=on';  
SET join_cache_level=8; # for MariaDB
```

in order to take advantage of them make the features useless unless you are fine-tuning

- I expect that to change in the future

I Herby Declare MyISAM as Dead

- All major MyISAM-only features are now on MySQL 5.7
 - FULLTEXT
 - GIS
 - Transportable tables
- There are still reasons to use MyISAM
 - MyISAM is still required for the mysql schema and non-durable temporary tables (WIP)

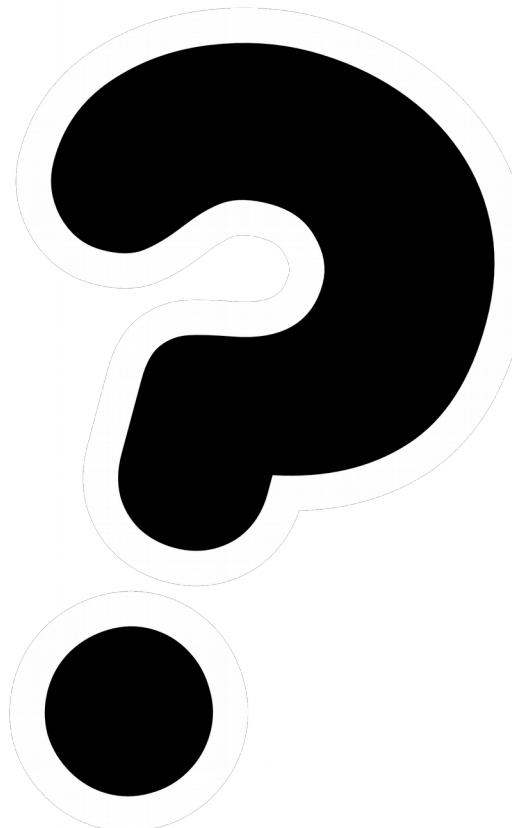


Benchmarks

- Do not trust first party benchmarks
 - In fact, do not trust 3rd party benchmarks either
- Only care about the performance of your application running on your hardware



Q&A



Not to Miss

- Official track:
MySQL at Wikipedia: How we do relational data at the Wikimedia Foundation
- Do you want to do query optimization for a website with 20 Billion views per month?
<http://grnh.se/0y4pxm>



Thank You for Attending!

- Do not forget, after the session finishes, to please login with your Percona Live account and “Rate This Session”
- Special thanks to in order by rand() to: Morgan Tocker, Sean Pringle, David Hildebrandt, Bill Karwin, Domas Mituzas, Mark Callaghan, Shlomi Noach, Mark Bergsma, Valerii Kravchuk, Miguel Ángel Nieto, Dimitri Kravtchuk, Olav Sandst  and the whole Wikimedia Team, and all people at the MariaDB, Percona and MySQL/Oracle teams, and the Percona Live Organization and Sponsors