Percona Live 2017

Santa Clara, California | April 24-27, 2017

MySQL INDEX Cookbook

How to Build the Best INDEX for a Given SELECT

Rick James

Agenda

Limitations: InnoDB indexes only, not **`FULLTEXT`** or **`SPATIAL`**

Agenda:

- Definitions
- Examples
- Algorithm
- What Works; What Doesn't
- PRIMARY KEY
- Other Issues
- Table Patterns

Definitions

Terminology -- (will be repeated as we go)

Syntax Keywords

- **`PRIMARY KEY`** is a **`UNIQUE KEY`**
  - plus "clustered"
- **`UNIQUE KEY`** is an **`INDEX`**
  - plus a uniqueness constraint
- Synonymous: "**`INDEX`**" "**`KEY`**"
- **`FULLTEXT`**, **`SPATIAL`**, **`HASH`** not being discussed

More Types of indexes

- "Secondary index"
  - not **`PRIMARY`**, hence not "clustered"
- "Clustered"
  - **`PRIMARY KEY`** lives with the data
- "Covering"
  - All the columns of the **`SELECT`** are in the index
  - Don't have more than, say, 5 columns
- "Composite" (aka "compound")
  - Multiple columns: **`INDEX(a,b)`**

Filtering - "Equal"

```
WHERE x = 123
```

```
WHERE str = 'foo'
```

Filtering - "IN"

```
WHERE b IN (1, 2)
```

- If single item, works like `=`
- If multiple items, maybe like `=`, maybe like range

```
WHERE x IN ( SELECT … )
```

- Optimizes *poorly*
- Turn into `JOIN` – Example…

IN to JOIN Example

```
SELECT  …
    FROM t
    WHERE some_test
      AND x IN (
            SELECT x FROM … );
```

⇒

```
SELECT  …
    FROM t
    JOIN ( SELECT x FROM … ) b
        USING(x)
    WHERE some_test;
```

Filtering - "Range"

A sequence of consecutive values

- `x < 123`
- `x BETWEEN 100 and 199`
- `str LIKE 'foo%'`
- *No:* `str LIKE '%foo'`

"Index Merge"

- A single **`SELECT`** will use at most *one* **`INDEX`**.
  - A few (very few) exceptions.
    - Called "index merge"

mysql.rjweb.org/doc.php/index1

Definitions - Q&A

1 question (hold rest until end)

Examples

Some Simple Examples --
Develop an Algorithm later

Single Filter

```
SELECT … WHERE a = 11
SELECT … WHERE a >= 11

INDEX(a)      -- perfect
INDEX(a, b)   -- good

SELECT … WHERE name = 'Rick'
SELECT … WHERE name LIKE 'R%'

INDEX(name)      -- perfect
INDEX(name, b)   -- good
```

Multiple '=' Filters

```
WHERE a  = 12  AND bb = 345
WHERE bb = 345 AND a  = 12

INDEX(a, bb)   -- perfect
INDEX(a)       -- somewhat good
INDEX(bb)      -- somewhat good
```

- Order in **`WHERE`** does *not* matter
  - assuming **`AND'd`**

Equal and Range

```
WHERE a  = 12  AND bb > 345
WHERE bb > 345 AND a  = 12
```

- `INDEX(a, bb)` – perfect
- `INDEX(a)` – somewhat good
- `INDEX(bb)` – somewhat good
- `INDEX(bb, a)` – *no better than (bb)!*

Two Ranges

Punt!

```
WHERE a > 12 AND bb > 345
```

No index with both `a` and `bb` is *fully* useful

Ditto for "=" plus multiple "ranges":

```
WHERE c = 9 AND a > 12 AND bb > 345
```

Either might be useful:

```
INDEX(c, a)
INDEX(c, bb)
```

Covering

Examples above have an exception…

IF all columns in the **`SELECT`** are in the index, then the index is "covering", hence at least a little better

```
SELECT x FROM t WHERE y = 5;
INDEX(y,x)
```

- The algorithm says just INDEX(y)

```
SELECT x FROM t WHERE y > 5 AND q > 7;
INDEX(y,q,x)
```

- y or q first (that's as far as the Algorithm goes); then other two

BTree - 1

Technically it is a B+Tree.

This is the structure of the indexes being discussed.

- Very efficient at
  - Locating a single row, given the key
  - Scanning a range of rows

en.wikipedia.org/wiki/B+_tree

BTree - 2

The data (clustered with **`PRIMARY KEY`**) is also a BTree.

- Leaf nodes of the Data BTree
  - contains entire rows
- Leaf nodes of the Secondary index BTree
  - contains secondary key and **`PRIMARY KEY`**

Rule of Thumb: Fanout ~100x

Examples - Q&A

1 question (hold rest until end)

Algorithm

Build the best INDEX

First, some Caveats

- No **`OR`**
- No **`IN`**
- Just a bunch of filters **`ANDd`** together in the **`WHERE`** clause

We'll fold those in later

Step 1 - Equals

- Find all filters of the form `col = constant`
  - Put those column names in the `INDEX` *first*
    - In any order
    - "Cardinality" does *not* matter

Step 2

- You can add one more column
  - Range, or
  - **`GROUP BY`**, or
  - **`ORDER BY`**

Step 2a - Range

If you have a "range" filter, add its column.

Then *stop*; no further columns will help.

Step 2b - GROUP BY

- If
  - No range, and
  - *All* of the **`WHERE`** is handled
- Then
  - Add all the columns of the **`GROUP BY`** to the index
    - In the same order
  - And stop

Step 2c - ORDER BY

- If *all* are true:
  - No range,
  - *All* of the **`WHERE`** is handled,
  - No **`GROUP BY`**,
  - Have **`ORDER BY`** with all **`ASC`** or all **`DESC`** (Ver 8.0 relaxes this)
- Then
  - Add all the columns of the **`ORDER BY`** to the index
    - In the same order

GROUP BY + ORDER BY + LIMIT

If you consumed *all*

- consumed all of **`WHERE`**, and
- consumed all of **`GROUP BY`**, and
- **`ORDER BY`** is
  - missing, or
  - identical to **`GROUP BY`** (or **`DESC`**)

Then, you can consume the **`LIMIT`**…

Consume the LIMIT

- Avoid "temporary" and "filesort"
- Looks only at **`LIMIT`** rows, not all the rows

- It does not make much sense to have a **`LIMIT`** without an **`ORDER BY`**.

- **`OFFSET`** rows must be stepped over

ORDER BY

Sometimes the Optimizer decides to

- Ignore **`WHERE`**
- Use index suitable for **`ORDER BY`**

Sometimes good, sometimes not.

Perhaps add an **`INDEX`** aimed just at **`ORDER BY`**

Algorithm - Q&A

1 question (hold rest until end)

What Works; What Doesn't

Issues that help/hurt indexing

Index killers - functions

These don't let you use an index:

- Implicit or explicit functions

```
DATE(dt) = '…',
LOWER(s) = '…'
CAST(s …) = '…',
x = '…' COLLATE…
```

en.wikipedia.org/wiki/Sargable

Index killers - others

- Leading wildcard

```
s LIKE '%…'
```

- Different tables

```
t1.x = 8  AND  t2.y = 11
```

  - `INDEX(x)` or `INDEX(y)` may be useful

- Negatives
  - `NOT IN`, `NOT EXISTS`, and `LEFT JOIN..IS NULL`
    - new versions of MySQL/MariaDB may work better

Flags - bad

TRUE/FALSE or other low carndinality columns are not worth indexing:

```
WHERE flag = TRUE
```

- won't use `INDEX(flag)`

OK in combo:

```
WHERE flag = TRUE
  AND dt > '…'
```

- will use `INDEX(flag, dt)`

UNION for OR

Sometimes it is useful to turn **`OR`** into **`UNION`**.

```
  WHERE a = 1  OR  x = 4
```

This shows adding a LIMIT:

```
  ( SELECT … WHERE a = 1 ORDER BY … LIMIT 5 )
  UNION ALL
  ( SELECT … WHERE x = 4 ORDER BY … LIMIT 5 )
  ORDER BY … LIMIT 5;
```

Switch to **`UNION DISTINCT`** if you need dedup.

UNION with OFFSET

To get the 10th 'page':

```
  ( SELECT … ORDER BY … LIMIT 50 )
  UNION ALL
  ( SELECT … ORDER BY … LIMIT 50 )
  ORDER BY … LIMIT 45, 5;
```

Pagination:
mysql.rjweb.org/doc.php/pagination

ASC / DESC

```
ORDER BY a ASC,  b ASC
ORDER BY a DESC, b DESC
```

- Both work with `INDEX(a,b)`; the second is slightly less efficient

```
ORDER BY a ASC,  b DESC
INDEX(   a ASC,  b DESC )
```

- (pre-8.0): `ASC` and `DESC` are ignored in index, so index can't be used

Prefix - INDEX(foo(5)) - poor

- Use for **`TEXT`** or **`BLOB`**
- Do not use otherwise
- Often the Optimizer will eschew the index

- **`UNIQUE(foo(5))`** is "wrong"
  - uniqueness check on only 5 chars
- **`INDEX(last(3), first)`**
  - won't get past **`last`**

Using temporary, Using filesort

This is often *necessary*.
It is not the villain by itself.

- **`GROUP BY team  ORDER BY score`**
  - Leads to *second* temp+sort

DATEs - bad cases

Tempting, but cannot use index because the column is hiding in an explicit or implicit function:

```
  date LIKE '2016-12%'
```

```
  LEFT(date, 7) = '2016-12'
```

```
  YEAR(date) = 2016
```

Instead…

DATEs - good

Range, so index possible:

```
        date >= '2016-12-01'
  AND date  < '2016-12-01'
                + INTERVAL 3 MONTH
```

Avoids problems with

- Month/year boundaries & Leap days
- Last second (**`BETWEEN`** is "inclusive")
- Works for **`DATE`**, **`DATETIME(6)`**, **`TIMESTAMP`**

What Works/Doesn't - Q&A

1 question (hold rest until end)

PRIMARY KEY

PRIMARY KEY issues

What [not] to use for PK

Choices for **`PRIMARY KEY`**

- (usually best) "Natural" column(s)
- (decent fallback) **`AUTO_INCREMENT`**
  - Make it **`UNSIGNED`** and **`NOT NULL`**
  - **`BIGINT`** (8 bytes) is usually overkill
- (terrible for huge table) UUID/GUID/MD5
  - Randomness ⇒ I/O ⇒ Slow
- (usually bad) No PK
  - Some maintenance operations must have PK

Natural benefits

- Avoids need for **`AUTO_INCREMENT`**
- Faster access by that column
- Works fine in most cases
- Might lead to "covering"

AUTO_INCREMENT benefits

- Less 'bulky'
    - Shrinks secondary keys
        - A copy of PK is in every Secondary key

Burning IDs (gaps)

Some operations waste **`AUTO_INCREMENT`** ids because they allocate the id before seeing if they need it

- **`INSERT IGNORE …`**
- **`INSERT … ON DUPLICATE KEY UPDATE …`**
- **`REPLACE …`** (mostly replaced by IODKU)

Beware of hitting the max value for the id!

PRIMARY KEY - Q&A

1 question (hold rest until end)

Other Issues

Miscellany

More than one INDEX

- A **`SELECT`** will (usually) use only one **`INDEX`**.
  - Each subquery or **`UNION`** counts separately
    - So, they may use different indexes

Tweaks

- Avoid **`USE/FORCE/IGNORE INDEX`**, **`STRAIGHT_JOIN`**
  - except in desperation
- **`LIMIT 9999999999`**
  - tricks Optimizer into doing an otherwise unnecessary **`ORDER BY`**

767 Limitation

Err: "max key length is 767" usually happens with **`VARCHAR(255) CHARACTER SET utf8mb4`**.

- Workaround: do one of
  - Upgrade to 5.7.7 for 3072 byte limit
  - Change 255 to 191 on the **`VARCHAR`**
  - **`ALTER .. CONVERT TO utf8`**
    - but disallows Emoji and some Chinese
  - Use a "prefix" index (ill-advised)
  - Reconfigure (for 5.6.3 - 5.7.6)
    - Barracuda + innodb_file_per_table + innodb_large_prefix + dynamic/compressed

Redundant indexes - waste

```
PRIMARY KEY(id)
UNIQUE (id) -- Drop

INDEX(a, b)
INDEX(a)    -- Drop

INDEX(a, b)
INDEX(b, a) -- May be redundant
```

Signs of a Newbie

- No PRIMARY KEY
- No composite indexes
- "But I indexed everything"
- Redundant indexes
  - eg, `PRIMARY KEY(id)`, `KEY(id)`
- "Commajoin"
  - `FROM a , b WHERE a.x=b.x AND c=1` ⇒
  - `FROM a JOIN b ON a.x=b.x WHERE c=1`

PARTITION

Partitioning has a lot of limitations on indexes.

Try to avoid partitioning by building better indexes.

mariadb.com/kb/en/mariadb/partition-maintenance/

JOINs

- Designing **`INDEX`**es for a **`JOIN`**
  - Design index for first table
  - Design index for next table
  - Etc
- Which is "first"?
  - Not necessarily the order specified
  - **`LEFT JOIN`** *may* force left table before right
  - Optimizer prefers table with **`WHERE`**s

JOIN example

```
SELECT …
    FROM a
    JOIN  b  ON where a.x = b.y
    WHERE b.z = 123
```

- First `b` with `INDEX(z)`
- Then `a` with `INDEX(x)`

Other Issues - Q&A

1 question (hold rest until end)

Table Patterns

Some Patterns

Many:Many Mapping

```
CREATE TABLE student_class (
    id_student … NOT NULL,
    id_class   … NOT NULL,
    … optional atributes …,
    PRIMARY KEY(id_student, id_class),
    INDEX      (id_class, id_student)
) ENGINE=InnoDB;
```

Notes ⇒

Many:Many notes

- No **`AUTO_INCREMENT id`**
- Small ids (**`MEDIUMINT`**, etc)
- **`UNSIGNED`** & **`NOT NULL`**
- InnoDB – to get clustered PK
- **`INDEX`** provides opposite path
- Conditionally insert:
  - **`INSERT IGNORE …`**, or
  - **`INSERT … ON DUPLICATE KEY UPDATE …`**

Normalization

```
CREATE TABLE Hosts (
  id MEDIUMINT UNSIGNED  -- 3 byte
      NOT NULL  AUTO_INCREMENT,
  name  VARCHAR(…)  NOT NULL,
  PRIMARY KEY(id),
  UNIQUE(name)          -- uniq; lookup
) ENGINE=InnoDB;        -- clustering
```

wp_postmeta

```
CREATE TABLE wp_postmeta (
    post_id …,
    meta_key …,
    meta_value …,
    PRIMARY KEY(post_id, meta_key),
    INDEX(meta_key)
    ) ENGINE=InnoDB;
```

- **`AUTO_INCREMENT`** was a waste
- Much better 'natural' PK; InnoDB to get clustering
- Use 191 if necessary; not "prefix" index

Table Patterns - Q&A

1 question (hold rest until end)

Closing

Let the questions flow!
Rate My Session

These Slides / longer version
mysql.rjweb.org/slides/cook.pdf
mysql.rjweb.org/doc.php/index_cookbook_mysql

Rick: mysql@rjweb.org
mysql.rjweb.org/
stackoverflow.com - tag **[mysql] [indexing]**