MySQL Indexing; Best Practices

10 November 2016 | Peter Zaitsev | Percona

Percona University

Ульяновск, россия

You've Made a Great Choice !

Understanding indexing is crucial both for Developers and DBAs

Poor index choices are responsible for large portion of production problems

Indexing is not a rocket science

MySQL Indexing: Agenda

- Understanding Indexing
- Setting up best indexes for your applications
- Working around common MySQL limitations

Indexing in the Nutshell

What are indexes for ?

- Speed up access in the database
- Help to enforce constraints (**UNIQUE**, **FOREIGN KEY**)
- Queries can be ran without any indexes
  - But it can take a really long time

Types of Indexes you might heard about

BTREE Indexes	• Majority of indexes you deal in MySQL is this type
RTREE Indexes	• MyISAM only, for GIS
HASH Indexes	• MEMORY, NDB
FULLTEXT Indexes	• MyISAM, Innodb starting 5.6

Family of BTREE like Indexes

A lot of different implementations

- Share same properties in what operations they can speed up
- Memory vs Disk is life changer

B+ Trees are typically used for Disk storage

- Data stored in leaf nodes

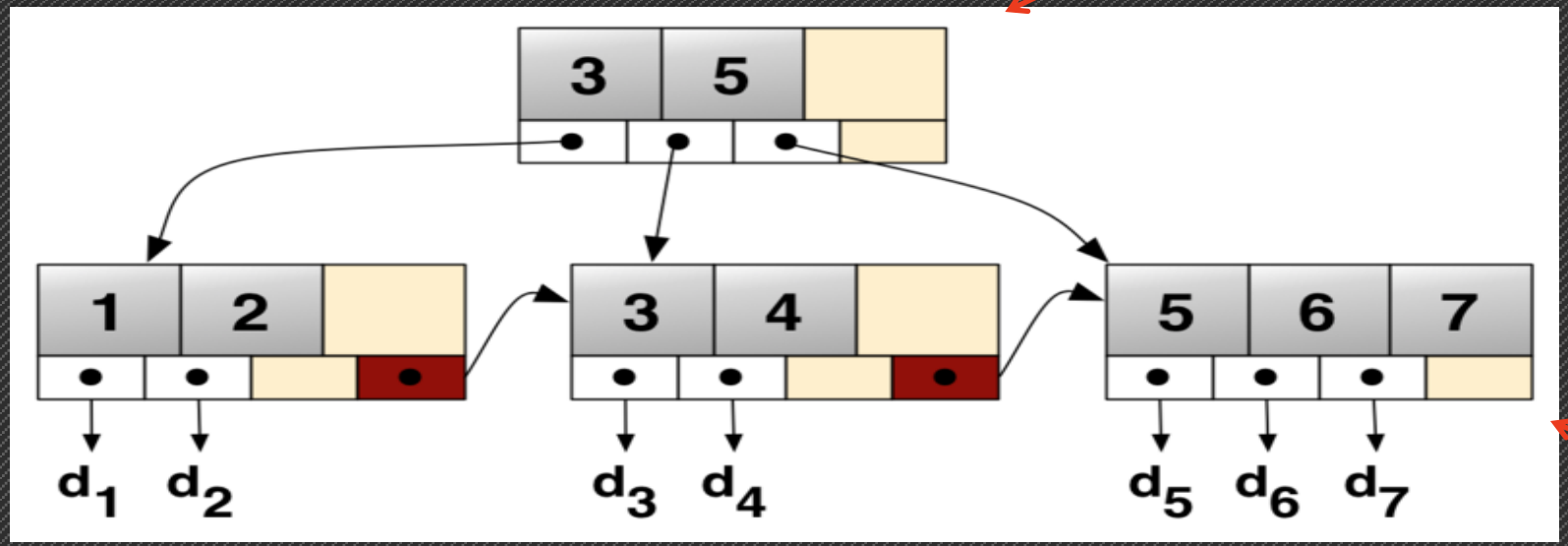
TokuDB Fractal Trees are logically similar

- But physically a lot different

B+Tree Example

Branch/Root Node

Less than 3



Data Pointers

Leaf Node

Indexes in MyISAM vs Innodb

In MyISAM data pointers point to physical offset in the data file

- All indexes are essentially equivalent

In Innodb

- **PRIMARY KEY** (Explicit or Implicit) - stores data in the leaf pages of the index, not pointer
- Secondary Indexes – store primary key as data pointer

What Operations can BTREE Index do ?

Find all rows with **KEY=5** (point lookup)

Find all rows with **KEY>5** (open range)

Find all rows with **5<KEY<10** (closed range)

NOT find all rows with last digit of the KEY is Zero

- This can't be defined as a "range" operation

String Indexes

There is no difference… really

- Sort order is defined for strings (collation)
  - “AAAA” < “AAAB”

Prefix LIKE is a special type of Range

- **LIKE “ABC%”** means
  - “ABC[LOWEST]”<KEY<“ABC[HIGHEST]”
- **LIKE “%ABC”** can’t be optimized by use of the index

Multiple Column Indexes

Sort Order is defined, comparing leading column, then second etc

- **KEY(col1,col2,col3)**
- (1,2,3) < (1,3,1)

It is still one BTREE Index

- not a separate BTREE index for each level

Overhead of The Indexing

Indexes are costly; Do not add more than you need

- In most cases extending index is better than adding new one

Writes - Updating indexes is often major cost of database writes

Reads - Wasted space on disk and in memory; additional overhead during query optimization

Indexing Innodb Tables

Data is clustered by Primary Key

- Pick **PRIMARY KEY** what suites you best
- For comments – **(POST_ID,COMMENT_ID)** can be good **PRIMARY KEY** storing all comments for single post close together
  - Alternatively “pack” to single **BIGINT**

PRIMARY KEY is implicitly appended to all indexes

- **KEY (A)** is really **KEY (A,ID)** internally
- Useful for sorting, Covering Index.

How MySQL Uses Indexes

- Data Lookups
- Sorting
- Avoiding reading “data”
- Special Optimizations

Using Indexes for Data Lookups

SELECT * FROM EMPLOYEES WHERE LAST_NAME="Smith"

- The classical use of index on (LAST_NAME)

Can use Multiple column indexes

- **SELECT * FROM EMPLOYEES WHERE LAST_NAME="Smith" AND DEPT="Accounting"**
- Will use index on (DEPT,LAST_NAME)

It Gets Tricky With Multiple Columns

Index (A,B,C)	• - order of columns matters
Will use Index for lookup (all listed keyparts)	• **A>5** • **A=5 AND B>6** • **A=5 AND B=6 AND C=7** • **A=5 AND B IN (2,3) AND C>5**
Will NOT use Index	• **B>5** – Leading column is not referenced • **B=6 AND C=7** - Leading column is not referenced
Will use Part of the index	• **A>5 AND B=2** - range on first column; only use this key part • **A=5 AND B>6 AND C=2** - range on second column, use 2 parts

The First Rule of MySQL Optimizer

MySQL will stop using key parts in multi part index as soon as it met the real range (<,>, BETWEEN), it however is able to continue using key parts further to the right if IN(…) range is used

Using Index for Sorting

SELECT * FROM PLAYERS ORDER BY SCORE DESC LIMIT 10

- Will use index on SCORE column
- Without index MySQL will do “filesort” (external sort) which is very expensive

Often Combined with using Index for lookup

- **SELECT * FROM PLAYERS WHERE COUNTRY=“US” ORDER BY SCORE DESC LIMIT 10**
  - Best served by Index on **(COUNTRY,SCORE)**

Multi Column indexes for efficient sorting

It becomes even more restricted!

KEY(A,B)

Will use Index for Sorting

- **ORDER BY A** - sorting by leading column
- **A=5 ORDER BY B** - EQ filtering by 1st and sorting by 2nd
- **ORDER BY A DESC, B DESC** - Sorting by 2 columns in same order
- **A>5 ORDER BY A** - Range on the column, sorting on the same

Will NOT use Index for Sorting

- **ORDER BY B** - Sorting by second column in the index
- **A>5 ORDER BY B** – Range on first column, sorting by second
- **A IN(1,2) ORDER BY B** - In-Range on first column
- **ORDER BY A ASC, B DESC** - Sorting in the different order

MySQL Using Index for Sorting Rules

You can’t sort in different order by 2 columns

You can only have Equality comparison (=) for columns which are not part of ORDER BY

- Not even IN() works in this case

Avoiding Reading The data

"Covering Index"	• Applies to index use for specific query, not type of index.
Reading Index ONLY and not accessing the "data"	• Index is typically smaller than data
SELECT STATUS FROM ORDERS WHERE CUSTOMER_ID=123	• **KEY(CUSTOMER_ID,STATUS)**
Access is a lot more sequential	• Access through data pointers is often quite "random"

Min/Max Optimizations

Index help **MIN()/MAX()** aggregate functions

- But only these

SELECT MAX(ID) FROM TBL;

SELECT MAX(SALARY) FROM EMPLOYEE GROUP BY DEPT_ID

- Will benefit from **(DEPT_ID,SALARY)** index
- "Using index for group-by"

Indexes and Joins

MySQL Performs Joins as "Nested Loops"

- **SELECT * FROM POSTS,COMMENTS WHERE AUTHOR="Peter" AND COMMENTS.POST_ID=POSTS.ID**
  - Scan table **POSTS** finding all posts which have Peter as an Author
  - For every such post go to **COMMENTS** table to fetch all comments

Index is only needed on table which is being looked up

- The index on **POSTS.ID** is not needed for this query performance

Very important to have all JOINs Indexed

- Re-Design JOIN queries which can't be well indexed

Using Multiple Indexes for the table

MySQL Can use More than one index

- "Index Merge"

SELECT * FROM TBL WHERE A=5 AND B=6

- Can often use Indexes on **(A)** and **(B)** separately
- Index on **(A,B)** is much better

SELECT * FROM TBL WHERE A=5 OR B=6

- 2 separate indexes is as good as it gets
- Index **(A,B)** can't be used for this query

Prefix Indexes

You can build Index on the leftmost prefix of the column

- ALTER TABLE TITLE ADD KEY(TITLE(20));
- Needed to index BLOB/TEXT columns
- Can be significantly smaller
- Can’t be used as covering index
- Choosing prefix length becomes the question

What is new with MySQL 5.6 ?

Many Optimizer improvements

- Most of them will make your queries better automatically
- **join_buffer_size** variable has whole new meaning
  - Values if 32MB+ can make sense

Focus on Index Design Practices for this presentation

- Most important one: ICP (Index Condition Pushdown)

Understanding ICP

Push where clause "Conditions" for Storage engine to filter	• Think **NAME LIKE "%ill%"** (will not convert to range)
"Much more flexible covering Index"	• Plus filtering done on the engine level – efficient
Before MySQL 5.5	• All or none. All is resolved through the index or "row" is read if within range

ICP Examples

SELECT A … WHERE B=2 AND C LIKE "%ill%'

- MySQL 5.5 and below
  - ***KEY (B)*** *– Traditional. Using index for range only*
  - ***KEY (B,C,A)*** *- Covering. All involved columns included*
- MySQL 5.6
  - ***KEY(B,C)***
    - Range access by B; Filter clause on C only read full row if match

More cases

- **SELECT * …WHERE A=5 and C=6** ; KEY (A,B,C)
  - *Will scan all* ***index entries*** *with A=5 not all* ***rows***

How MySQL Picks which Index to Use ?

Performs dynamic picking for every query execution	• The constants in query texts matter a lot
Estimates number of rows it needs to access for given index	• by doing “dive” in the table
Uses “Cardinality” statistics if impossible	• This is what **ANALYZE TABLE** updates

Use EXPLAIN

EXPLAIN is a great tool to see how MySQL plans to execute the query

- http://dev.mysql.com/doc/refman/5.6/en/using-explain.html
- Remember real execution might be different

```
mysql> explain select max(season_nr) from title group by production_year;
+----+-------------+-------+-------+---------------+-----------------+---------+------+------+----------------------------+
| id | select_type | table | type  | possible_keys | key             | key_len | ref  | rows | Extra                      |
+----+-------------+-------+-------+---------------+-----------------+---------+------+------+----------------------------+
|  1 | SIMPLE      | title | range | NULL          | production_year | 5       | NULL |  201 | Using index for group-by |
+----+-------------+-------+-------+---------------+-----------------+---------+------+------+----------------------------+
1 row in set (0.01 sec)
```

MySQL 5.6 JSON EXPLAIN FORMAT

```
mysql> EXPLAIN FORMAT=JSON SELECT * FROM t1 JOIN t2 ON t1.i = t2.i WHERE t1.j > 1 AND t2.j < 3;
| {
  "query_block": {
    "select_id": 1,
    "nested_loop": [
      {
        "table": {
          "table_name": "t1",
          "access_type": "ALL",
          "rows": 3,
          "filtered": 100,
          "attached_condition": "((`test`.`t1`.`j` > 1) and (`test`.`t1`.`i` is not null))"
        } /* table */
      },
....
```

Indexing Strategy

Build indexes for set of your performance critical queries	• Look at them together not just one by one
Best if all **WHERE** clause and **JOIN** clauses are using indexes for lookups	• At least most selective parts are
Generally extend index if you can, instead of creating new indexes	• There are some exceptions
Validate performance impact as you're doing changes	• Revisit Performance of All Queries!

Indexing Strategy Example

Build Index order which benefits more queries

- **SELECT * FROM TBL WHERE A=5 AND B=6**
- **SELECT * FROM TBL WHERE A>5 AND B=6**
- **KEY (B,A)** Is better for such query mix

All being equal put more selective key part first

- Do not start with this!

Do not add indexes for non performance critical queries

- Many indexes slow system down

Trick #1: Enumerating Ranges

KEY (A,B)

- Assume we need to stick to this order

SELECT * FROM TBL WHERE A BETWEEN 2 AND 4 AND B=5

- Will only use first key part of the index

SELECT * FROM TBL WHERE A IN (2,3,4) AND B=5

- Will use both key parts

Trick #2: Adding Fake Filter

KEY (GENDER,CITY)	• Want to use one index only
SELECT * FROM PEOPLE WHERE CITY="NEW YORK"	• Will not be able to use the index at all
SELECT * FROM PEOPLE WHERE GENDER IN ("M","F") AND CITY="NEW YORK"	• Will be able to use the index
The trick works best with low selectivity columns.	• Gender, Status, Boolean Types etc

Trick #3: Unionizing Filesort

KEY(A,B)	• No Other key would work ether
SELECT * FROM TBL WHERE A IN (1,2) ORDER BY B LIMIT 5;	• Will not be able to use index for SORTING
(SELECT * FROM TBL WHERE A=1 ORDER BY B LIMIT 5) UNION ALL (SELECT * FROM TBL WHERE A=2 ORDER BY B LIMIT 5) ORDER BY B LIMIT 5;	• Will use the index for Sorting. "filesort" will be needed only to sort over 10 rows.

Thank You !

pz@percona.com
https://www.linkedin.com/in/peterzaitsev
https://twitter.com/peterzaitsev