

Practical MySQL indexing guidelines

Percona Live

October 24th-25th, 2011

London, UK

Stéphane Combaudon

stephane.combaudon@dailymotion.com

Agenda

- Introduction
- Bad indexes & performance drops
- Guidelines for efficient indexing
- Tools and methods to improve index usage

Introduction

Goals

- Having fun with indexes!!!
- Getting rid of trial-and-error approach
- Knowing performance penalty of bad indexes
- Being productive
 - Knowing simple rules to design indexes
 - Knowing tools that can help

Indexing basics

- Index: data structure to speed up SELECTs
 - Think of an index in a book
 - In MySQL, key = index
 - We'll consider that indexes are trees
- InnoDB's clustered index
 - Data is stored with the PK: PK lookups are fast
 - Secondary keys hold the PK values
 - Designing InnoDB's PKs with care is critical for perf.

Strengths

- An index can filter and/or sort values
- An index can contain all the fields needed for a query
 - No need to access data anymore
- A leftmost prefix can be used
 - Indexes on several columns are useful
 - Order of columns in composite keys is important

Limitations

- MySQL only uses 1 index per table per query
 - Ok, that's not 100% true (OR clauses...)
 - Think of composite indexes when you can!!
- Can't index full TEXT fields
 - You must use a prefix
 - Same for BLOBS and long VARCHARs
- Maintaining an index has a cost
 - Read speed vs write speed

Sample table

```
CREATE TABLE t (  
  id INT NOT NULL AUTO_INCREMENT,  
  a INT NOT NULL DEFAULT 0,  
  b INT NOT NULL DEFAULT 0,  
  [more columns here]  
  PRIMARY KEY(id)  
)ENGINE=InnoDB;
```

- Populated with "many" rows
 - Means that queries against table are "slow"
- Replace "many" and "slow" with your own values

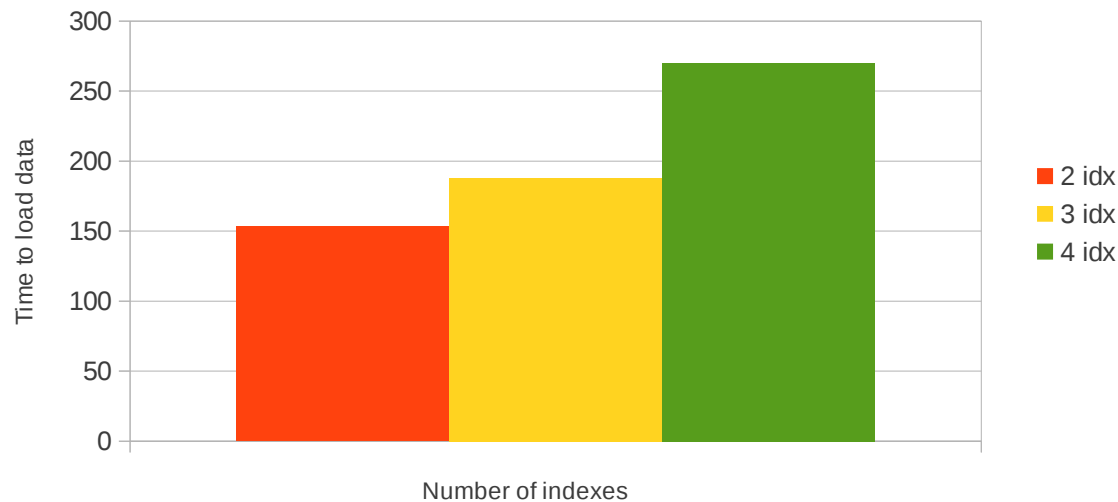
Bad indexes & performance drops

Adding an index

- 3 main consequences:
 - Can speed up queries (good)
 - Increases the size of your dataset (bad)
 - Slows down writes (bad)
- How big is the write slow-down?
 - Let's have simple tests

Write slow-downs, pictured

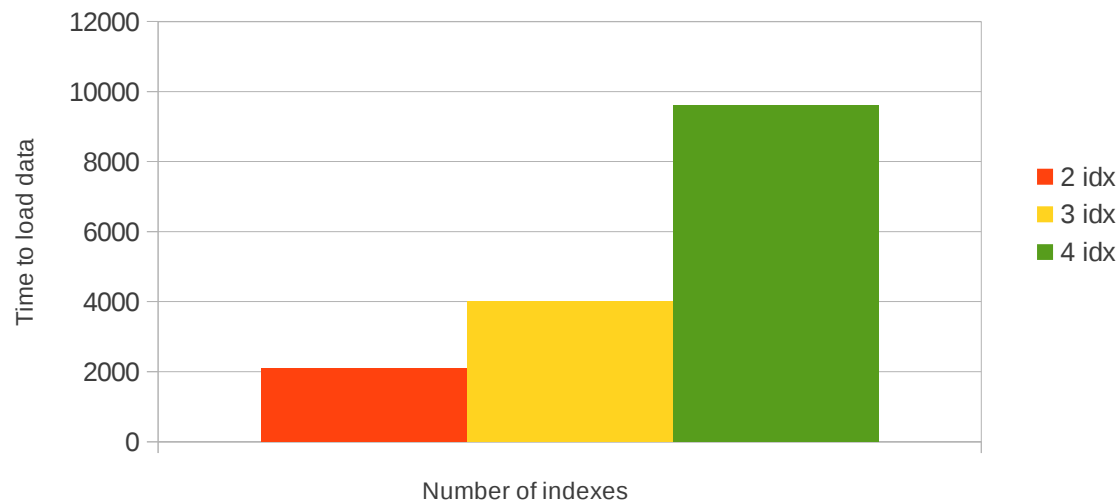
In-memory test



Baseline is 100 for 1 key for both graphs

For in-memory workloads, adding 2 keys makes perf. 2x worse

On-disk test



For on-disk workloads, adding 2 keys make perf. 40x worse!!

So what?

- Removing bad indexes is crucial for perf.
 - Especially for write-intensive workloads
 - Tools will help us
- What if your workload is read-intensive?
 - A few hot tables may handle most of the writes
 - These tables will be write-intensive

Identifying bad indexes

- Before removing bad indexes, identify them!
- What is a bad index?
 - Duplicate indexes: always bad
 - Redondant indexes: generally bad
 - Low-cardinality indexes: depends
 - Unused indexes: always bad

Guidelines for efficient indexes

Before we start...

- Indexing is not an exact science
 - But guessing is not the best way to design indexes
- A few simple rules will help 90% of the time
- Always check your assumptions
 - EXPLAIN does not tell you everything
 - Time your queries with different index combinations
 - SHOW PROFILES is often valuable
- Slow query log is a good place to start!

Rule #1: Filter

Q1: SELECT * FROM t WHERE a = 10 AND b = 20

- Without an index, always a full table scan

```
1. mysql> EXPLAIN SELECT * FROM t WHERE a = 10 AND b = 20\G
2. ***** 1. row *****
3.           id: 1
4.   select_type: SIMPLE
5.           table: t
6.           type: ALL
7. possible_keys: NULL
8.           key: NULL
9.      key_len: NULL
10.           ref: NULL
12.          rows: 1000545
12.      Extra: Using where
```

ALL means
table scan

Estimated #
of rows to read

Post-filtering needed
to discard the non-matching rows

Rule #1: Filter

- Idea: filter as much data as possible by focusing on the WHERE clause
- Candidates for Q1:
 - `key(a)`, `key(b)`, `key(a,b)`, `key(b,a)`
- Condition is on both a and b with an AND
 - A composite index should be better
 - Let's test!

Rule #1: Filter

```
1. mysql> EXPLAIN SELECT * ...
2. ***** 1. row *****
3.      [...]
4.      key: a
5.      key_len: 4
6.      [...]
7.      rows: 20
Exec time: 0.00s
```

```
1. mysql> EXPLAIN SELECT * ...
2. ***** 1. row *****
3.      [...]
4.      key: b
5.      key_len: 4
6.      [...]
7.      rows: 67368
Exec time: 0.20s
```

```
1. mysql> EXPLAIN SELECT * ...
2. ***** 1. row *****
3.      [...]
4.      key: ab
5.      key_len: 8
6.      [...]
7.      rows: 10
Exec time: 0.00s
```

```
1. mysql> EXPLAIN SELECT * ...
2. ***** 1. row *****
3.      [...]
4.      key: ba
5.      key_len: 8
6.      [...]
7.      rows: 10
Exec time: 0.00s
```

Same perf. for this query
Other queries will guide us
to choose between them

Rule #2: Sort

Q2: `SELECT * FROM t WHERE a = 10 ORDER BY b`

- Remember: indexed values are sorted
- An index can avoid costly filesorts
 - Think of filesorts performed on on-disk temp tables
 - ORDER BY clause must be a leftmost prefix of the index
- Caveat: an index scan is fast in itself, but retrieving the rows in index order may be slow
 - Seq. scan on index but random access on table

Rule #2: Sort

- Let's try `key (b)` for Q2 vs full table scan

```
1. mysql> EXPLAIN SELECT * ...
2. ***** 1. row *****
3.      [...]
4.      type: index
5.      key: b
6.      key_len: 4
7.      [...]
8.      rows: 1000638
9.      Extra: Using where
```

Exec time: 1.52s

```
1. mysql> EXPLAIN SELECT * ...
2. ***** 1. row *****
3.      [...]
4.      type: ALL
4.      key: NULL
5.      key_len: NULL
6.      [...]
7.      rows: 1000638
8.      Extra: Using where;
           Using filesort
```

Exec time: 0.37s

EXPLAIN suggest
key(b) is better,
but it's wrong!

Rule #2: Sort

- An index is not always the best for sorting
- If possible, try to sort and filter
- Exception to the leftmost prefix rule:
 - Leading columns appearing in the WHERE clause as **constants** can fill the holes in the index
 - `WHERE a = 10 ORDER BY b: key(a, b)` can filter and sort
 - Not true with `WHERE a > 10 ORDER BY b`

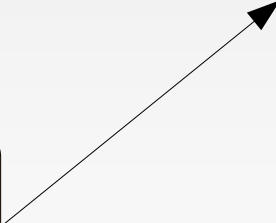
Rule #2: Sort

- With key (a , b)

```
1. mysql> EXPLAIN SELECT * FROM t
WHERE a = 10 ORDER BY b\G
2. ***** 1. row *****
3.      [...]
4.      type: ref
5.      key: ab
6.      key_len: 8
7.      [...]
8.      rows: 20
9.      Extra:
```

```
1. mysql> EXPLAIN SELECT * FROM t
WHERE a > 10 ORDER BY b\G
2. ***** 1. row *****
3.      [...]
4.      type: ALL
5.      key: NULL
6.      key_len: NULL
7.      [...]
8.      rows: 1000638
9.      Extra: Using where;
              Using filesort
```

Could have been a range scan
Depends on the distribution
of the values



Rule #3: Cover

Q3: `SELECT a,b FROM t WHERE a > 100;`

- With `key(a)`, you filter efficiently
- But with `key(a,b)`
 - You filter
 - The index holds all the columns you need
 - Means you don't need to access data
- `key(a,b)` is a covering index

Rule #3: Cover

- Back to InnoDB's clustered index
 - It is always covering
 - SELECT by PK is the fastest access with InnoDB
 - Take care of your PKs!!
- Remember full table scan + filesort vs index?
 - If the index used for sorting is also covering, it will outperform the table scan

Rating an index

- An index can give you 3 benefits: filtering, sorting, covering
- 1-star index: 1 property
- 2-star index: 2 properties
- 3-star index: 3 properties
- This is my own rating, other systems exist

Range queries and ORDER BY

Q4: SELECT * FROM t WHERE a > 10 and b = 20 ORDER BY a

```
mysql> EXPLAIN SELECT * ...\G
***** 1. row *****
  [...]
    type: range
    key: a
    rows: 500319
    Extra: Using where
Exec time: 35.9s
```

Key filters and sorts, but filtering is not efficient. Getting data is very slow (random access + I/O-bound)

```
mysql> EXPLAIN SELECT * ...\G
***** 1. row *****
  [...]
    type: ref
possible_keys: a,b,ab,ba
    key: ba
    rows: 64814
    Extra: Using where;
          Using filesort
Exec time: 0.2s
```

Key filters but doesn't sort. Filtering is efficient so getting data, post-filtering and post-sorting is not too slow

Joins and ORDER BY

- All columns in the ORDER BY clause must refer to the 1st table
- Forcing the join order with SELECT STRAIGHT_JOIN is sometimes useful
- Sometimes you can't fulfill this condition
 - This can be a reason to denormalize

Tools and methods to improve index usage

Userstats v2

- You need Percona Server or MariaDB 5.2+

```
mysql> SELECT s.table_name,s.index_name,rows_read
FROM information_schema.statistics s
LEFT JOIN information_schema.index_statistics i
ON (i.table_schema=s.table_schema
    AND i.table_name=s.table_name
    AND i.index_name=s.index_name)
WHERE s.table_name='comment'
    AND s.table_schema='mydb'
    AND seq_in_index=1;
```

Table added by
this feature

Deals with
composite indexes

table_name	index_name	rows_read
comment	PRIMARY	50361
comment	user_id	NULL
comment	video_comment_idx	18276197
comment	created_language_idx	NULL

Useless

Useless

Workload matters!

- OLAP server

table_name	index_name	rows_read
comment	PRIMARY	50361
comment	user_id	NULL
comment	video_comment_idx	18276197
comment	created_language_idx	NULL

← Never used

- OLTP server

table_name	index_name	rows_read
comment	PRIMARY	12220798
comment	user_id	96674982
comment	video_comment_idx	365691254
comment	created_language_idx	217176

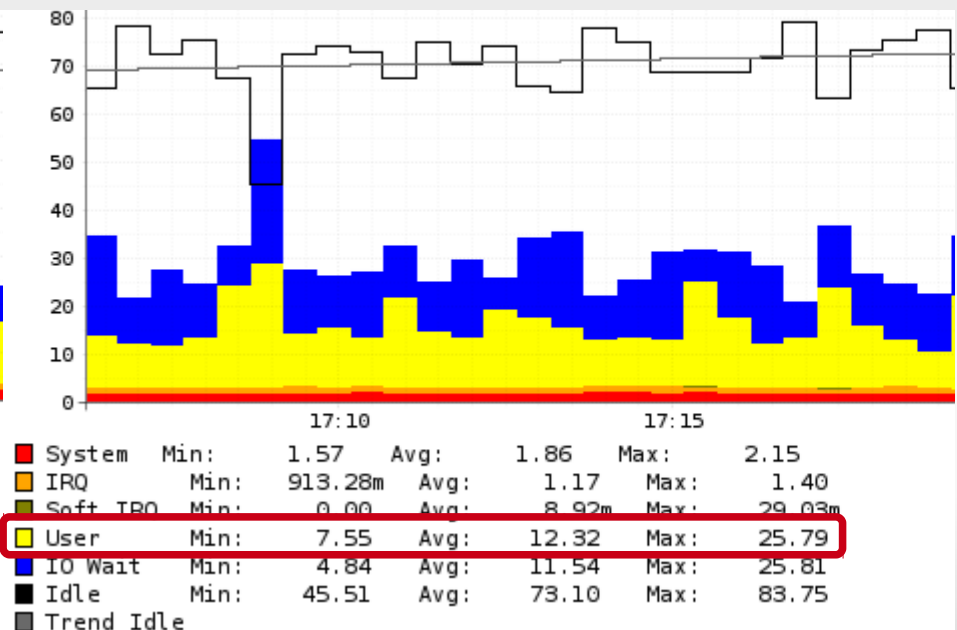
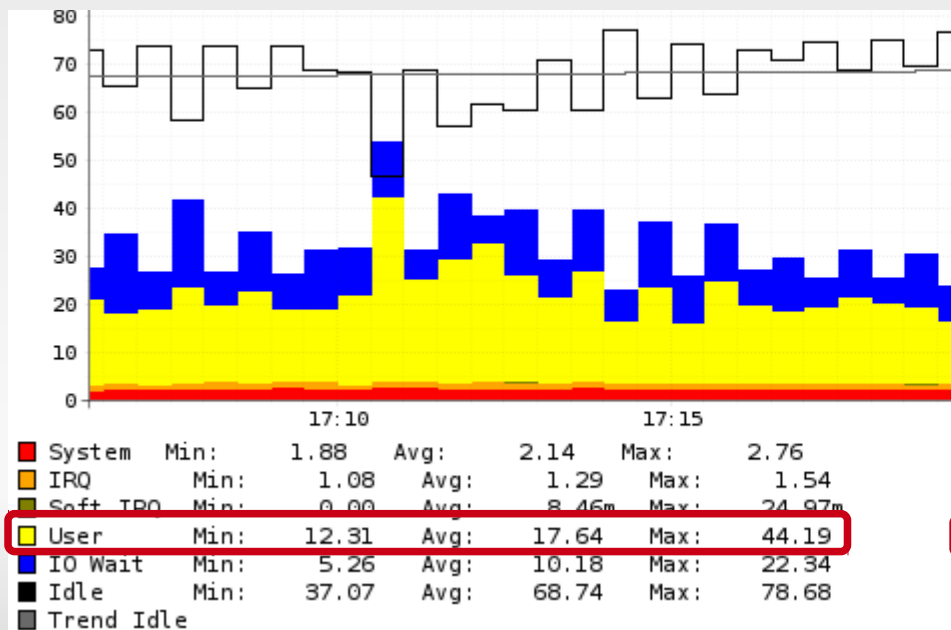
← Useful!

Pros

- Very easy to use
 - Turn on the variable and forget
- Easy to write queries to discover unused indexes automatically

Cons

- Large sample period needed for accurate stats
- Not always obvious to say if index is useful
 - Look at `created_language_idx` in previous slide
- Has some CPU overhead



pt-duplicate-key-checker

- Anything wrong with the keys?

```
CREATE TABLE comment (  
  comment_id int(10) ... AUTO_INCREMENT,  
  video_id int(10) ...,  
  user_id int(10) ...,  
  language char(2) ...,  
  [...]  
  PRIMARY KEY (comment_id),  
  KEY user_id (user_id),  
  KEY video_comment_idx (video_id,language,comment_id)  
) ENGINE=InnoDB;
```

Tool is aware
of InnoDB's
clustered index!

```
$ pt-duplicate-key-checker u=root,h=localhost  
[...]  
# Key video_comment_idx ends with a prefix of the clustered index  
# Key definitions:  
#   KEY video_comment_idx (video_id,language,comment_id)  
#   PRIMARY KEY (comment_id),  
[...]  
# To shorten this duplicate clustered index, execute:  
ALTER TABLE mydb.comment DROP INDEX video_comment_idx, ADD INDEX  
video_comment_idx (video_id,language)
```

Query to remove
the index

pt-index-usage

- Helps answer questions not solved by userstats
 - Are there any queries with a changing exec plan?
 - Is an index necessary for a query?
- Read a slow log file/general log file
- Can give you invaluable information on your index usage
 - See the man page for more

- Thanks for your attention!
- Any questions?