MySQL Index Cookbook

Deep & Wide Index Tutorial

Rick James

PERCONA LIVE

TOC

- Preface
- Case Study
- PRIMARY KEY
- Use Cases
- EXPLAIN
- Work-Arounds
- Datatypes
- Tools
- PARTITIONing
- MyISAM
- Miscellany

Preface

Terminology

Engines covered

- InnoDB / XtraDB
- MyISAM
- PARTITIONing
- Not covered:
- NDB Cluster
- MEMORY
- FULLTEXT, Sphinx, GIS

- Except where noted, comments apply to both InnoDB/XtraDB and MyISAM

Index ~= Table

- Each index is stored separately
- Index is very much like a table
  - BTree structure
  - InnoDB leaf: cols of PRIMARY KEY
  - MyISAM leaf: row num or offset into data

("Leaf": a bottom node in a BTree)

BTrees

- Efficient for keyed row lookup
- Efficient for “range” scan by key
- RoT ("Rule of Thumb): Fan-out of about 100 (1M rows = 3 levels)
- Best all-around index type

http://en.wikipedia.org/wiki/B-tree

http://upload.wikimedia.org/wikipedia/commons/thumb/6/65/B-tree.svg/500px-B-tree.svg.png

Index Attributes

- Diff Engines have diff attributes
- Limited combinations (unlike other vendors) of
  - clustering (InnoDB PK)
  - unique
  - method (Btree)

KEY vs INDEX vs …

- KEY == INDEX
- UNIQUE is an INDEX
- PRIMARY KEY is UNIQUE
  - At most 1 per table
  - InnoDB must have one
- Secondary Key = any key but PRIMARY KEY
- FOREIGN KEY implicitly creates a KEY

“Clustering”

- Consecutive things are ‘adjacent’ on disk ☺ , therefore efficient in disk I/O
- “locality of reference” (etc)
- *Index* scans are clustered
  - But note: For InnoDB PK you have to step over rest of data
- *Table* scan of InnoDB PK – clustered by PK (only)

“Table Scan”, “Index Scan”

- What – go through whole data/index
  - Efficient because of way BTree works
  - Slow for big tables
- When – if more than 10-30% otherwise
  - Usually “good” if picked by optimizer
- EXPLAIN says “ALL”

Range / BETWEEN

A "range" scan is a "table" scan, but for less than the whole table

- Flavors of “range” scan
  - `a BETWEEN 123 AND 456`
  - `a > 123`
  - Sometimes: `IN (…)`

Common Mistakes

- “I indexed every column” – usually not useful.
- User does not understand “compound indexes”
- INDEX(a), INDEX(a, b) – redundant
- PRIMARY KEY(id), INDEX(id) – redundant

Size RoT

- 1K rows & fit in RAM: rarely performance problems
- 1M rows: Need to improve datatypes & indexes
- 1B rows: Pull out all stops! Add on Summary tables, SSDs, etc.

Case Study

Building up to a Compound Index

The question

Q: "When was Andrew Johnson president of the US?" Table `Presidents`:

seq	last	first	term
1	Washington	George	1789-1797
2	Adams	John	1797-1801
...			
7	Jackson	**Andrew**	1829-1837
...			
17	**Johnson**	**Andrew**	1865-1869
...			
36	**Johnson**	Lyndon B.	1963-1969
...			

The question – in SQL

```
SELECT  term
        FROM  Presidents
        WHERE  last = 'Johnson'
          AND  first = 'Andrew';
```

What INDEX(es) would be best for that question?

The INDEX choices

- No indexes
- INDEX(first), INDEX(last)
- Index Merge Intersect
- INDEX(last, first) – “compound”
- INDEX(last, first, term) – “covering”
- Variants

No Indexes

The interesting rows in EXPLAIN:

```
  type: ALL          <-- Implies table scan
   key: NULL         <-- Implies that no index is
useful, hence table scan
  rows: 44           <-- That's about how many rows
in the table, so table scan
```

Not good.

INDEX(first), INDEX(last)

Two separate indexes

MySQL rarely uses more than one index

Optimizer will study each index, decide that 2 rows come from each, and pick one.

EXPLAIN:

```
    key: last
key_len: 92    ← VARCHAR(30) utf8: 2+3*30
   rows: 2     ← two "Johnson"
```

INDEX(first), INDEX(last) (cont.)

What's it doing?

1. With INDEX(last), it finds the Johnsons
2. Get the PK from index (InnoDB): [17,36]
3. Reach into data (2 BTree probes)
4. Use "AND first=…" to filter
5. Deliver answer (1865-1869)

Index Merge Intersect

("Index Merge" is rarely used)

1. INDEX(last) → [7,17]
2. INDEX(first) → [17, 36]
3. "AND" the lists → [17]
4. BTree into data for the row
5. Deliver answer

```
           type: index_merge
possible_keys: first, last
            key: first, last
        key_len: 92,92
           rows: 1
     Extra: Using intersect(first,last); Using where
```

INDEX(last, first)

1. Index BTree to the one row: [17]
2. PK BTree for data
3. Deliver answer

```
key_len: 184                ← length of both fields
    ref: const, const       ← WHERE had constants
   rows: 1                  ← Goodie
```

INDEX(last, first, term)

1. Index BTree using last & first; get to leaf
2. Leaf has the answer – Finished!

```
key_len: 184                ← length of both fields
    ref: const, const       ← WHERE had constants
   rows: 1                  ← Goodie
  Extra: Using where; Using index  ← Note
```

“Covering” index – “Using index”

Variants

- Reorder ANDs in WHERE – no diff
- Reorder cols in INDEX – *big* diff
- Extra fields on *end* of index – mostly harmless
- Redundancy: `INDEX(a)` + `INDEX(a,b)` – DROP shorter
- "Prefix" `INDEX(last(5))` – rarely helps; can hurt

Variants – examples

`INDEX(last, first)`

- WHERE last=... – good
- WHERE last=... AND first=… – good
- WHERE first=... AND last=… – good
- WHERE *first*=... – index useless

`INDEX(last)` (applied above):

good, so-so, so-so, useless

Cookbook

SELECT → the optimal compound INDEX to make.

1. all fields in WHERE that are "= const" (any order)
2. One more field (no skipping!):
   1. WHERE Range (BETWEEN, >, …)
   2. GROUP BY
   3. ORDER BY

Cookbook – IN

IN (SELECT ...) – Very poor opt. (until 5.6)

IN (1,2,...) – Works somewhat like “=“.

PRIMARY KEY

Gory details that you really should know

PRIMARY KEY

- By definition: UNIQUE & NOT NULL
- InnoDB PK:
  - Leaf contains the data row
  - So... Lookup by PK goes straight to row
  - So... Range scans by PK are efficient
  - (PK needed for ACID)
- MyISAM PK:
  - Identical structure to secondary index

Secondary Indexes

- BTree
- Leaf item points to data row
  - InnoDB: pointer is copy of PRIMARY KEY
  - MyISAM: pointer is offset to row

"Using Index"

When a SELECT references only the fields in a Secondary index, only the secondary index need be touched. This is a performance bonus.

What should be the PK?

Plan A: A “natural”, such as a unique name; possibly compound

Plan B: An artificial `INT AUTO_INCREMENT`

Plan C: No PK – generally not good

Plan D: UUID/GUID/MD5 – inefficient due to randomness

AUTO_INCREMENT?

```
id INT UNSIGNED NOT NULL
  AUTO_INCREMENT PRIMARY KEY
```

- Better than no key – eg, for maintenance
- Useful when “natural key” is bulky *and* lots of secondary keys; else unnecessary
- Note: each InnoDB secondary key includes the PK columns. (Bulky PK → bulky secondary keys)

Size of InnoDB PK

Each *InnoDB* secondary key includes the PK columns.

- Bulky PK → bulky secondary keys
- "Using index" may kick in – because you have the PK fields implicitly in the Secondary key

No PK?

InnoDB must have a PK:

1. User-provided (best)
2. First UNIQUE NOT NULL key (sloppy)
3. Hidden, inaccessible 6-byte integer (you are better off with your own A_I)

"Trust me, have a PK."

Redundant Index

PRIMARY KEY (id),

INDEX (id, x),

UNIQUE (id, y)

Since the PK is “clustered” in InnoDB, the other two indexes are almost totally useless. Exception: If the index is “covering”.

INDEX (x, id) – a different case

Compound PK - Relationship

```
CREATE TABLE Relationship (
foo_id INT …,
bar_id INT …,
PRIMARY KEY (foo_id, bar_id),
INDEX       (bar_id, foo_id) -- if
  going both directions
) ENGINE=InnoDB;
```

Use Cases

Derived from real life

Normalizing (Mapping) table

Goal: Normalization – id ↔ value

```
id    INT UNSIGNED NOT NULL AUTO_INCREMENT,
name VARCHAR(255),

PRIMARY KEY (id),
UNIQUE (name)
```

In MyISAM add these to “cover”: INDEX(id,name), INDEX(name,id)

Normalizing BIG

```
id MEDIUMINT UNSIGNED NOT NULL AUTO_INCREMENT,
md5 BINARY(16/22/32) NOT NULL,
stuff TEXT/BLOB NOT NULL,

PRIMARY KEY (id),
UNIQUE (md5)

INSERT INTO tbl (md5, stuff) VALUES($m,$s)
  ON DUPLICATE KEY UPDATE id=LAST_INDERT_ID(id);
$id = SELECT LAST_INSERT_ID();
```

Caveat: Dups burn ids.

Avoid Burn

1. `UPDATE ... JOIN ... WHERE id IS NULL` -- Get the ids (old) – Avoids Burn
2. `INSERT IGNORE ... SELECT DISTINCT ...` -- New rows (if any)
3. `UPDATE ... JOIN ... WHERE id IS NULL` -- Get the ids (old or new) – multi-thread is ok.

WHERE lat … AND lng …

- Two fields being range tested
  - Plan A: INDEX(lat), INDEX(lng) – let optimizer pick
  - Plan B: Complex subqueries / UNIONs beyond scope
  - Plan C: Akiban
  - Plan D: Partition on Latitude; PK starts with Longitude:
    `http://mysql.rjweb.org/doc.php/latlng`

Index on MD5 / GUID

- VERY RANDOM! Therefore,
  - Once the index is bigger than can fit in RAM cache, you will be thrashing on disk
- What to do??
  - Normalize
  - Some other key
  - PARTITION by date may help INSERTs
  - http://mysql.rjweb.org/doc.php/uuid (type-1 only)

Key-Value

- Flexible, expandable
- Clumsy, inefficient
- `http://mysql.rjweb.org/doc.php/eav`
- Horror story about RDF...
- Indexes cannot make up for the clumsiness

ORDER BY RAND()

- No built-in optimizations
- Will read all rows, sort by RAND(), deliver the LIMIT
- `http://mysql.rjweb.org/doc.php/random`

Pagination

- `ORDER BY … LIMIT 40,10` – Indexing won't be efficient
- → Keep track of "left off"
- `WHERE x > $leftoff ORDER BY … LIMIT 10`
- LIMIT 11 – to know if there are more
- `http://mysql.rjweb.org/doc.php/pagination`

Latest 10 Articles

- Potentially long list
- of articles, items, comments, etc;
- you want the "latest"

But

- JOIN getting in the way, and
- INDEXes are not working for you

Then build an helper table with a useful index:

`http://mysql.rjweb.org/doc.php/lists`

LIMIT rows & get total count

- `SELECT SQL_CALC_FOUND_ROWS … LIMIT 10`
- `SELECT FOUND_ROWS()`
- *If* INDEX can be used, this is not “too” bad.
- Avoids a second SELECT

ORDER BY x LIMIT 5

- *Only* if you get to the point of using x in the INDEX is the LIMIT going to be optimized.
- Otherwise it will
  1. Collect all possible rows – *costly*
  2. Sort by x – *costly*
  3. Deliver first 5

“It’s not using my index!”

```
SELECT … FROM tbl WHERE x=3;
INDEX (x)
```

- Case: few rows have x=3 – will use INDEX.
- Case: 10-30% match – might use INDEX
- Case: most rows match – will do table scan

The % depends on the phase of the moon

Getting ORDERed rows

Plan A: Gather the rows, filter via WHERE, deal with GROUP BY & DISTINCT, then sort ("filesort").

Plan B: Use an INDEX to fetch the rows in the 'correct' order. (If GROUP BY is used, it must match the ORDER BY.)

The optimizer has trouble picking between them.

INDEX(a,b) vs (b,a)

INDEX (a, b) vs INDEX (b, a)

WHERE a=1 AND b=2 – both work equally well

WHERE a=1 AND b>2 – first is better

WHERE a>1 AND b>2 – each stops after 1st col

WHERE b=2 – 2nd only

WHERE b>2 – 2nd only

Compound “>”

- [assuming] INDEX(hr, min)
- `WHERE (hr, min) >= (7,45)` -- poorly optimized
- `WHERE hr >= 7 AND min >= 45` – *wrong*
- `WHERE (hr = 7 AND min >= 45) OR (hr > 7)` – *slow because of OR*
- `WHERE hr >= 7 AND (hr > 7 OR min >= 45)` – *better; [only needs INDEX(hr)]*
- Use TIME instead of two fields! – *even better*

UNION [ALL | DISTINCT]

- UNION defaults to `UNION DISTINCT`; maybe `UNION ALL` will do? (Avoids dedupping pass)
- Best practice: Explicitly state ALL or DISTINCT

DISTINCT vs GROUP BY

- `SELECT DISTINCT … GROUP BY` → redundant
- To dedup the rows: `SELECT DISTINCT`
- To do aggregates: `GSELECT GROUP BY`

OR --> UNION

- OR does not optimize well
- UNION may do better

```
SELECT ... WHERE a=1 OR b='x'
-->
SELECT ... WHERE a=1
UNION DISTINCT
SELECT ... WHERE b='x'
```

(break)

EXPLAIN SELECT …

To see if your INDEX is useful

http://dev.mysql.com/doc/refman/5.5/en/explain-output.html

EXPLAIN

- Run EXPLAIN SELECT ... to find out how MySQL might perform the query today.
  - Caveat: Actual query may pick diff plan
- Explain says which key it will use; `SHOW CREATE TABLE` shows the INDEXes
- If using compound key, look at byte len to deduce how many fields are used.

EXPLAIN – “using index”

- EXPLAIN says “using index”
- Benefit: Don’t need to hit data ☺
- How to achieve: All fields used are in *one* index
- InnoDB: Remember that PK field(s) are in secondary indexes
- Tip: Sometimes useful to add fields to index:
  - SELECT a,b FROM t WHERE **c=1**
  - SELECT b FROM t WHERE **c=1** ORDER BY a
  - SELECT b FROM t WHERE **c=1** GROUP BY a
  - INDEX (c,a,b)

EXPLAIN EXTENDED

```
EXPLAIN EXTENDED SELECT …;
SHOW WARNINGS;
```

The first gives an extra column.

The second details how the optimizer reformulated the SELECT. LEFT JOIN→JOIN and other xforms.

EXPLAIN – filesort

- Filesort: ☹ But it is just a symptom.
- A messy query will gather rows, write to temp, sort for group/order, deliver
  - Gathering includes all needed columns
  - Write to tmp:
    - Maybe MEMORY, maybe MyISAM
    - Maybe hits disk, *maybe not* -- can't tell easily

“filesort”

These *might* need filesort:

- DISTINCT
- GROUP BY
- ORDER BY
- UNION DISTINCT

Possible to need multiple filsorts (but no clue)

“Using Temporary”

- *if*
  - no BLOB, TEXT, VARCHAR > 512, FULLTEXT, etc (MEMORY doesn’t handle them)
  - estimated data < max_heap_table_size
  - others
- *then* “filesort” is done using the MEMORY engine (no disk)
  - VARCHAR(n) becomes CHAR(n) for MEMORY
  - utf8 takes 3n bytes
- *else* MyISAM is used

EXPLAIN PARTITIONS SELECT

Check whether the “partition pruning” actually pruned.

The “first” partition is always included when the partition key is DATE or DATETIME. This is to deal with invalid dates like 20120500.

Tip: Artificial, empty, “first” partition.

INDEX cost

- An INDEX is a BTree.
- Smaller than data (usually)
- New entry added during INSERT (always up to date)
- UPDATE of indexed col -- juggle index entry
- Benefit to SELECT far outweighs cost of INSERT (usually)

Work-Arounds

Inefficiencies, and what to do about them

Add-an-Index-Cure (not)

- Normal learning curve:
  - Stage 1: Learn to build table
  - Stage 2: Learn to add index
  - Stage 3: Indexes are a panacea, so go wild adding indexes
- Don’t go wild. Every index you add costs something in
  - Disk space
  - INSERT/UPDATE time

OR → UNION

- INDEX(a), INDEX(b) != INDEX(a, b)
- Newer versions *sometimes* use two indexes
- `WHERE a=1 OR b=2 =>`

```
(SELECT ... WHERE a=1)
UNION
(SELECT ... WHERE b=2)
```

Subqueries – Inefficient

Generally, subqueries are less efficient than the equivalent JOIN.

Subquery with GROUP BY or LIMIT *may* be efficient

5.6 and MariaDB 5.5 do an excellent job of making most subqueries perform well

Subquery Types

```
SELECT a, (SELECT …) AS b FROM …;
```

RoT: Turn into JOIN *if* no agg/limit

RoT: Leave as subq. *if* aggregation

```
SELECT … FROM ( SELECT … );
```

Handy for GROUP BY or LIMIT

```
SELECT … WHERE x IN ( SELECT … );
```

```
SELECT … FROM ( SELECT … ) a
           JOIN ( SELECT … ) b ON …;
```

Usually very inefficient – do JOIN instead (Fixed in 5.6 and MariaDB 5.5)

Subquery – example of utility

- You are SELECTing bulky stuff (eg TEXT/BLOB)
- WHERE clause could be entirely indexed, but is messy (JOIN, multiple ranges, ORs, etc)
- → **SELECT a.text, … FROM *tbl* a**

  JOIN (SELECT *id* FROM *tbl* WHERE …) b

  ON a.*id* = b.*id*;
- Why? Smaller “index scan” than “table scan”

Extra filesort

- “ORDER BY NULL” – Eh? “I don’t care what order”
- GROUP BY *may* sort automatically
- ORDER BY NULL skips extra sort if GROUP BY did not sort
- Non-standard

USE, FORCE ("hints")

- `SELECT ... FROM foo USE INDEX(x)`
- RoT: Rarely needed
- Sometimes ANALYZE TABLE fixes the ‘problem’ instead, by recalculating the “statistics”.
- RoT: Inconsistent cardinality → FORCE is a mistake.
- `STRAIGHT_JOIN` forces order of table usage (use sparingly)

Datatypes

little improvements that can be made

Field Sizes

- VARCHAR (utf8: 3x, utf8mb4: 4x) → VARBINARY (1x)
- INT is 4 bytes → SMALLINT is 2 bytes, etc
- DATETIME → TIMESTAMP (8*:4)
- DATETIME → DATE (8*:3)
- Normalize (id instead of string)
- VARCHAR → ENUM (N:1)

Smaller → Cacheable → Faster

- Fatter fields → fatter indexes → more disk space → poorer caching → more I/O → poorer performance
- INT is better than a VARCHAR for a url
  - But this may mean adding a mapping table

WHERE fcn(col) = ‘const’

- No functions!
- `WHERE <fcn>(<indexed col>) = …`
- `WHERE lcase(name) = ‘foo’`
  - Add extra column; index `name`
- Hehe – in this example lcase is unnecessary if using COLLATE *_ci !

Date Range

- `WHERE dt BETWEEN '2009-02-27' AND '2009-03-02'` →
  - "Midnight problem"

```
WHERE dt >= '2009-02-27'
  AND dt <  '2009-02-27' + INTERVAL 4 DAY
```

- `WHERE YEAR(dt) = '2009'` →
  - Function precludes index usage

```
WHERE dt >= '2009-01-01'
  AND dt <  '2009-01-01' + INTERVAL 1 YEAR
```

WHERE utf8 = latin1

- Mixed character set tests (or mixed collation tests) tend not to use INDEX
  - Declare VARCHAR fields consistently
- WHERE foo = _utf8 'abcd'

Don’t index sex

- gender CHAR(1) CHARSET ascii
- INDEX(gender)
- Don’t bother!
- WHERE gender = ‘F’ – if it occurs > 10%, index will not be used

Prefix Index

- `INDEX(a(10))` – Prefixing usually bad
  - May fail to use index when it should
  - May not use subsequent fields
  - Must check data anyway
  - Etc.
- `UNIQUE(a(10))` constrains the first 10 chars to be unique – probably not what you wanted!
- May be useful for TEXT/BLOB

VARCHAR – VARBINARY

- Collation takes some effort
- UTF8 may need 3x the space (utf8mb4: 4x)
- CHAR, TEXT – collated (case folding, etc)
- BINARY, BLOB – simply compare the bytes
- Hence… MD5s, postal codes, IP addresses, etc, should be BINARY or VARBINARY

IP Address

- VARBINARY(39)
  - Avoids unnecessary collation
  - Big enough for Ipv6
- BINARY(16)
  - Smaller
  - Sortable, Range-scannable
- `http://mysql.rjweb.org/doc.php/ipranges`

Tools

Tools

- slow log
- show create table
- status variables
- percona toolkit or others.

SlowLog

- Turn it on
- long_query_time = 2 -- seconds
- pt-query-digest -- to find worst queries
- EXPLAIN – to see what it is doing

Handler_read%

A tool for seeing what is happening…

```
FLUSH STATUS;
SELECT …;
SHOW STATUS LIKE 'Handler_read%';
```

PARTITIONing

Index gotchas, etc.

PARTITION Keys

- Either:
  - No UNIQUE or PRIMARY KEY, *or*
  - *All* Partition-by fields *must* be in *all* UNIQUE/PRIMARY KEYs
    - (Even if artificially added to AI)
- RoT: Partition fields *should not* be first in keys
- Sorta like getting two-dimensional index -- first is partition 'pruning', then PK.

PARTITION Use Cases

- Possible use cases
  - Time series
    - DROP PARTITION much better than DELETE
  - “two” clustered indexes
  - random index and most of effort spent in last partition

PARTITION RoTs

Rules of Thumb

- Reconsider PARTITION – often no benefit
- Don't partition if under 1M rows
- BY RANGE only
- No SUBPARTITIONs

http://mysql.rjweb.org/doc.php/ricksrots#partitioning

PARTITION Pruning

- Uses WHERE to pick some partition(s)
- Sort of like having an extra dimension
- Don't need to pick partition (cannot until 5.6)
- Each "partition" is like a table

MyISAM

The big differences between
MyISAM and InnoDB

MyISAM vs InnoDB Keys

InnoDB PK is “clustered” with the data

- PK lookup finds row
- Secondary indexes use PK to find data

MyISAM PK is just like secondary indexes

- All indexes (in .MYI) point to data (in .MYD) via row number or byte offset

`http://mysql.rjweb.org/doc.php/myisam2innodb`

Caching

- MyISAM: 1KB BTree index blocks are cached in “key buffer”
  - key_buffer_size
  - Recently lifted 4GB limit
- InnoDB: 16KB BTree index and data blocks are cached in buffer pool
  - innodb_buffer_pool_size
  - The 16K is settable (rare cases)
- MyISAM has “delayed key write” – probably rarely useful, especially with RAID & BBWC

4G in MyISAM

- The “pointer” in MyISAM indexes is fixed at N bytes.
  - Old versions defaulted to 4 bytes (4G)
  - 5.1 default: 6 bytes (256T)
- Fixed/Dynamic
  - Fixed length rows (no varchar, etc): Pointer is row number
  - Dynamic: Pointer is byte offset
- Override/Fix: `CREATE/ALTER TABLE ... MAX_ROWS = ...`
  - Alter is slow

Miscellany

you can’t index a kitchen sink

Impact on INSERT / DELETE

- Write operations need to update indexes – sooner or later
- Performance
  - INSERT at end = hot spot there
  - Random key = disk thrashing
- Minimize number of indexes, especially random

WHERE name LIKE ‘Rick%’

- WHERE name LIKE ‘Rick%’
  - INDEX (name) – “range”
- WHERE name LIKE ‘%James’
  - *won’t use index*

WHERE a=1 GROUP BY b

- `WHERE a=1 GROUP BY b`

  `WHERE a=1 ORDER BY b`

  `WHERE a=1 GROUP BY b ORDER BY b`

  - `INDEX(a, b)` – nice for those

- `WHERE a=1 GROUP BY b ORDER BY c`
  - `INDEX(a, b, c)` – no better than (a,b)

WHERE a > 9 ORDER BY a

- WHERE a > 9 ORDER BY a
  - INDEX (a) – will catch both the WHERE and the ORDER BY ☺
- WHERE b=1 AND a > 9 ORDER BY a
  - INDEX (b, a)

GROUP BY, ORDER BY

- *if* there is a compound key such that
  - WHERE is satisfied, and
  - there are more fields in the key,
- *then*, MySQL will attempt to use more fields in the index for GROUP BY and/or ORDER BY
- GROUP BY aa ORDER BY bb → extra "filesort"

ORDER BY, LIMIT

- *If* you get all the way through the ORDER BY, still using the index, and you have LIMIT, then the LIMIT is done efficiently.
- *If not,* it has to gather all the data, sort it, finally deliver what LIMIT says.
  - This is the “Classic Meltdown Query”.

GROUP+ORDER+LIMIT

- Efficient:
  - WHERE a=1 GROUP BY b
    INDEX(a,b)
  - WHERE a=1 ORDER BY b LIMIT 9
    INDEX(a,b)
  - GROUP BY b ORDER BY c
    INDEX(b,c)
- Inefficient:
  - WHERE x.a=1 AND y.c=2 GROUP/ORDER/LIMIT
    - (because of 2 tables)

Index Types (BTree, etc)

- BTree
  - most used, most general
- Hash
  - MEMORY Engine only
  - useless for range scan
- Fulltext
  - Pretty good for “word” searches in text
- GIS (Spatial) (2D)
- No bit, etc.

FULLTEXT index

- “Words”
- Stoplist excludes common English words
- Min length defaults to 4
- Natural
- IN BOOLEAN MODE
- Trumps other INDEXes
- Serious competitors: Lucene, Sphinx
- MyISAM only until 5.6.4
  - Multiple diffs in InnoDB FT

AUTO_INCREMENT index

- AI field must be first in *some* index
- Need not be UNIQUE or PRIMARY
- Can be compound (esp. for PARTITION)
- Could *explicitly* add dup id (unless ...)

- (MyISAM has special case for 2nd field)

RoTs

Rules of Thumb

- 100 I/Os / sec (500/sec for SSD)
- RAID striping (1,5,6,10) – divide time by striping factor
- RAID write cache – writes are “instantaneous” *but* not sustainable in the long haul
- Cached fetch is 10x faster than uncached

- Query Cache is useless (in heavy writes)

Low cardinality, Not equal

- `WHERE deleted = 0`
- `WHERE archived != 1`
- These are likely to be poorly performing queries. Characteristics:
  - Poor cardinality
  - Boolean
  - !=
- Workarounds
  - Move deleted/hidden/etc *rows* into another table
  - Juggle compound index order (rarely works)
- "Cardinality", by itself, is rarely of note

Not NOT

- Rarely uses INDEX:
  - NOT LIKE
  - NOT IN
  - NOT (expression)
  - <>
- NOT EXISTS (SELECT * …) – essentially a LEFT JOIN; often efficient

Replication

- SBR
  - Replays query
  - Slave could be using different Engine and/or Indexes
- RBR
  - PK important

Index Limits

- Index width – 767B per column
- Index width – 3072B total
- Number of indexes – more than you should have
- Disk size – terabytes

Location

- InnoDB, file_per_table – .ibd file
- InnoDB, old – ibdata1
- MyISAM – .MYI
- PARTITION – each partition looks like a separate table

ALTER TABLE

1. copy data to tmp
2. rebuild indexes (on the fly, or separately)
3. RENAME into place

Even ALTERs that should not require the copy do so. (few exceptions)

RoT: Do all changes in a single ALTER. (some PARTITION exceptions)

5.6 fixes most of this

Tunables

- InnoDB indexes share caching with data in innodb_buffer_pool_size – recommend 70% of available RAM
- MyISAM indexes, not data, live in key_buffer_size – recommend 20% of available RAM
- log_queries_not_using_indexes – don’t bother

Closing

- More Questions?
- http://forums.mysql.com/list.php?24