Understanding Indexing
Without Needing to Understand Data Structures

MySQL UC 2011 - April 12, 2011

Zardosht Kasheff

Tokutek™

A *dictionary* is a set of (key, value) pairs.

- We'll assume you can **update** the dictionary (insertions, deletions, updates) and **query** the dictionary (point queries, range queries)

- B-Trees and Fractal Trees are examples of dictionaries

- Hashes are not (range queries are not supported)

Tokutek™

A *table* is a set of dictionaries.

Example:

```
create table foo (a
int, b int, c int,
primary key(a));
```

Then we insert a bunch of data and get...

a	b	c
100	5	45
101	92	2
156	56	45
165	6	2
198	202	56
206	23	252
256	56	2
412	43	45

Tokutek™

The sort order of a dictionary is defined by the key

- For the data structures/storage engines we'll think about, range queries on the sort order are FAST
- Range queries on any other order require a table scan = SLOW
- Point queries -- retrieving the value for one particular key -- is SLOW
  - ▸ A single point query is fast, but reading a bunch of rows this way is going to be 2 orders of magnitude slower than reading the same number of rows in range query order

Tokutek™

A *Index I* on table T is itself a dictionary

- We need to define the (key, value) pairs.
- The key in index I is a subset of fields in the primary dictionary T.
- The value in index I is the primary key in T.
  ▸ There are other ways to define the value, but we're sticking with this.

Example:

```
alter table foo add key(b);
```

Then we get...

Tokutek™

What's an Index?

Primary

a	b	c
100	5	45
101	92	2
156	56	45
165	6	2
198	202	56
206	23	252
256	56	2
412	43	45

`key(b)`

b	a
5	100
6	165
23	206
43	412
56	156
56	256
92	101
202	198

Q: `count(*) where a<120;`

a	b	c
100	5	45
101	92	2
156	56	45
165	6	2
198	202	56
206	23	252
256	56	2
412	43	45

b	a
5	100
6	165
23	206
43	412
56	156
56	256
92	101
202	198

Tokutek™

Q: count(*) where a<120;

a	b	c
100	5	45
101	92	2
156	56	45
165	6	2
198	202	56
206	23	252
256	56	2
412	43	45

b	a
5	100
6	165
23	206
43	412
56	156
56	256
92	101
202	198

100	5	45
101	92	2

2

Q: count(*) where b>50;

a	b	c
100	5	45
101	92	2
156	56	45
165	6	2
198	202	56
206	23	252
256	56	2
412	43	45

b	a
5	100
6	165
23	206
43	412
56	156
56	256
92	101
202	198

Tokutek™

Q: count(*) where b>50;

a	b	c
100	5	45
101	92	2
156	56	45
165	6	2
198	202	56
206	23	252
256	56	2
412	43	45

b	a
5	100
6	165
23	206
43	412
56	156
56	256
92	101
202	198

56	156
56	256
92	101
202	198

4

Q: `sum(c) where b>50;`

a	b	c
100	5	45
101	92	2
156	56	45
165	6	2
198	202	56
206	23	252
256	56	2
412	43	45

b	a
5	100
6	165
23	206
43	412
56	156
56	256
92	101
202	198

Tokutek™

Q: `sum(c) where b>50;`

a	b	c
100	5	45
101	92	2
156	56	45
165	6	2
198	202	56
206	23	252
256	56	2
412	43	45

b	a
5	100
6	165
23	206
43	412
56	156
56	256
92	101
202	198

Fast

56	156
56	256
92	101
202	198

Slow

156	56	45
256	56	2
101	92	2
198	202	56

105

What are indexes good for?

Indexes make queries go fast

- Each index will speed up some subset of queries

Design indexes with queries in mind

- Pick most important queries and set up indexes for those
- Consider the cost of maintaining the indexes

Tokutek ™

Indexes make queries go fast

- Each index will speed up some subset of queries

Design indexes with queries in mind

- Pick most important queries and set up indexes for those
- Consider the cost of maintaining the indexes

Tokutek™

3 simple rules for designing good indexes

Avoid details of any data structure

- B-trees and Fractal Trees are interesting and fun for the algorithmically minded computer scientist, but the 3 rules will apply equally well to either data structure.

- All we need to care about is that range queries are fast (per row) and that point queries are much slower (per row).

Tokutek ™

There is no absolute rule

- Indexing is like a math problem
- Rules help, but each scenario is its own problem, which requires problem-solving analysis

That said, rules help a lot

Tokutek ™

1. Retrieve less data

- Less bandwidth, less processing, …

2. Avoid point queries

- Not all data access cost is the same
- Sequential access is MUCH faster than random access

3. Avoid Sorting

- **GROUP BY** and **ORDER BY** queries do post-retrieval work
- Indexing can help get rid of this work

Tokutek ™

Rule 1

Retrieve less data

Example TABLE (1B rows, no indexes):

- `create table foo (a int, b int, c int);`

Query (1000 rows match):

- `select sum(c) from foo where b=10 and a<150;`

Query Plan:

- Rows where `b=10` and `a<150` can be anywhere in the table
- Without an index, entire table is scanned

Slow execution:

- Scan 1B rows just to count 1000 rows

Tokutek™

What should we do?

- Reduce the data retrieved
- Analyze much less than 1B rows

How (for a simple select)?

- **Design index by focusing on the `WHERE` clause**
  - ▸ This defines what rows the query is interested in
  - ▸ Other rows are not important for this query

Tokutek ™

What should we do?

- Reduce the data retrieved
- Analyze much less than 1B rows

How (for a simple select)?

- **Design index by focusing on the `WHERE` clause**
  - ▸ This defines what rows the query is interested in
  - ▸ Other rows are not important for this query

```
select sum(c) from foo where b=10 and a<150;
```

Tokutek ™

Option 1: `key(a)`

Option 2: `key(b)`

Which is better? Depends on *selectivity*:
- If there are fewer rows where `a<150`, then `key(a)` is better
- If there are fewer rows where `b=10`, then `key(b)` is better

Option 3: `key(a)` AND `key(b)`, then MERGE
- We'll come to this later

Tokutek™

Neither `key(a)` nor `key(b)` is optimal

Suppose:

- 200,000 rows exist where `a<150`
- 100,000 rows exist where `b=10`
- 1000 rows exist where `b=10` and `a<150`

Then either index retrieves too much data

For better performance, indexes should try to optimize over as many pieces of the where clause as possible

- We need a *composite index*

Composite indexes reduce data retrieved

Where clause: `b=5` and `a<150`

- Option 1: `key(a,b)`
- Option 2: `key(b,a)`

Which one is better?

- `key(b,a)`!

KEY RULE:

- When making a composite index, place equality checking columns first. Condition on **b** is equality, but not on **a**.

Tokutek™

a	b	c
100	5	45
101	6	2
156	5	45
165	6	2
198	6	56
206	5	252
256	5	2
412	6	45

b,a	a
5,100	100
5,156	156
5,206	206
5,256	256
6,101	101
6,165	165
6,198	198
6,412	412

Tokutek™

Q: `where b=5 and a>150;`

a	b	c
100	5	45
101	6	2
156	5	45
165	6	2
198	6	56
206	5	252
256	5	2
412	6	45

b,a	a
5,100	100
5,156	156
5,206	206
5,256	256
6,101	101
6,165	165
6,198	198
6,412	412

Tokutek™

Composite Indexes: No equality clause

What if `where` clause is:

- `where a>100 and  a<200 and b>100;`

Which is better?

- `key(a), key(b), key(a,b), key(b,a)`?

KEY RULE:

- As soon as a column on a composite index is NOT used for equality, the rest of the composite index no longer reduces data retrieved.

  ▸ `key(a,b)` is no better* than `key(a)`

  ▸ `key(b,a)` is no better* than `key(b)`

Tokutek™

What if `where` clause is:

- `where a>100 and  a<200 and b>100;`

Which is better?

- `key(a), key(b), key(a,b), key(b,a)`?

KEY RULE:

- As soon as a column on a composite index is NOT used for equality, the rest of the composite index no longer reduces data retrieved.

  ▸ `key(a,b)` is no better* than `key(a)`

  ▸ `key(b,a)` is no better* than `key(b)`

  * Are there corner cases where it helps? Yes, but rare.

Tokutek™

Q: `where b>=5 and a>150;`

a	b	c
100	5	45
101	6	2
156	5	45
165	6	2
198	6	56
206	5	252
256	5	2
412	6	45

b,a	a
5,100	100
5,156	156
5,206	206
5,256	256
6,101	101
6,165	165
6,198	198
6,412	412

Tokutek

Q: where b>=5 and a>150;

a	b	c
100	5	45
101	6	2
156	5	45
165	6	2
198	6	56
206	5	252
256	5	2
412	6	45

b,a	a
5,100	100
5,156	156
5,206	206
5,256	256
6,101	101
6,165	165
6,198	198
6,412	412

Tokutek™

a	b	c
100	5	45
101	6	2
156	5	45
165	6	2
198	6	56
206	5	252
256	5	2
412	6	45

b,a	a
5,100	100
5,156	156
5,206	206
5,256	256
6,101	101
6,165	165
6,198	198
6,412	412

b,a	a
5,156	156
5,206	206
5,256	256
6,101	101
6,165	165
6,198	198
6,412	412

Tokutek™

Composite Indexes: Another example

WHERE clause: `b=5 and c=100`

- `key(b,a,c)` is as good as `key(b)`, because `a` is not used in clause, so having `c` in index doesn't help. `key(b,c,a)` would be much better.

a	b	c
100	5	100
101	6	200
156	5	200
165	6	100
198	6	100
206	5	200
256	5	100
412	6	100

b,a,c	a
5,100,100	100
5,156,200	156
5,206,200	206
5,256,100	256
6,101,200	101
6,165,100	165
6,198,100	6,198
6,412,100	412

Tokutek™

Composite Indexes: Another example

WHERE clause: `b=5 and c=100`

- `key(b,a,c)` is as good as `key(b)`, because `a` is not used in clause, so having `c` in index doesn't help. `key(b,c,a)` would be much better.

a	b	c
100	5	100
101	6	200
156	5	200
165	6	100
198	6	100
206	5	200
256	5	100
412	6	100

b,a,c	a
5,100,100	100
5,156,200	156
5,206,200	206
5,256,100	256
6,101,200	101
6,165,100	165
6,198,100	6,198
6,412,100	412

5,100,100	100
5,156,200	156
5,206,200	206
5,256,100	256

Tokutek™

Goal is to <u>reduce rows retrieved</u>

- Create composite indexes based on where clause
- Place equality-comparison columns at the beginning
- Make first non-equality column in index as selective as possible
- Once first column in a composite index is not used for equality, or not used in where clause, the rest of the composite index does not help reduce rows retrieved
  - ▸ Does that mean they aren't helpful?
  - ▸ They might be very helpful... on to Rule 2.

Tokutek™

Rule 2

Avoid Point Queries

Table:

- ```
  create table foo (a int, b int, c int,
  primary key(a), key(b));
  ```

Query:

- ```
  select sum(c) from foo where b>50;
  ```

Query plan: use `key(b)`

- retrieval cost of each row is high because random point queries are done

Q: `sum(c) where b>50;`

a	b	c
100	5	45
101	92	2
156	56	45
165	6	2
198	202	56
206	23	252
256	56	2
412	43	45

b	a
5	100
6	165
23	206
43	412
56	156
56	256
92	101
202	198

Tokutek™

Q: sum(c) where b>50;

a	b	c
100	5	45
101	92	2
156	56	45
165	6	2
198	202	56
206	23	252
256	56	2
412	43	45

b	a
5	100
6	165
23	206
43	412
56	156
56	256
92	101
202	198

56	156
56	256
92	101
202	198

156	56	45
256	56	2
101	92	2
198	202	56

105

Table:

- ```
  create table foo (a int, b int, c int,
  primary key(a), key(b));
  ```

Query:

- ```
  select sum(c) from foo where b>50;
  ```

Query plan: scan primary table

- retrieval cost of each row is CHEAP!
- But you retrieve too many rows

a	b	c
100	5	45
101	92	2
156	56	45
165	6	2
198	202	56
206	23	252
256	56	2
412	43	45

b	a
5	100
6	165
23	206
43	412
56	156
56	256
92	101
202	198

Tokutek™

Q: `sum(c) where b>50;`

a	b	c
100	5	45
101	92	2
156	56	45
165	6	2
198	202	56
206	23	252
256	56	2
412	43	45

b	a
5	100
6	165
23	206
43	412
56	156
56	256
92	101
202	198

100	5	45
101	92	2
156	56	45
165	6	2
198	202	56
206	23	252
256	56	2
412	43	45

↓

105

Table:

- `create table foo (a int, b int, c int, primary key(a), key(b));`

Query:

- `select sum(c) from foo where b>50;`

What if we add another index?

- What about `key(b,c)`?
- Since we index on `b`, we retrieve only the rows we need.
- Since the index has information about `c`, we don't need to go to the main table. **No point queries!**

Tokutek ™

Q: sum(c) where b>50;

a	b	c
100	5	45
101	92	2
156	56	45
165	6	2
198	202	56
206	23	252
256	56	2
412	43	45

b,c	a
5,45	100
6,2	165
23,252	206
43,45	412
56,2	256
56,45	156
92,2	101
202,56	198

Tokutek™

Q: `sum(c) where b>50;`

a	b	c
100	5	45
101	92	2
156	56	45
165	6	2
198	202	56
206	23	252
256	56	2
412	43	45

b,c	a
5,45	100
6,2	165
23,252	206
43,45	412
56,2	256
56,45	156
92,2	101
202,56	198

56,2	256
56,45	156
92,2	101
202,56	198

↓

105

Tokutek™

An index *covers* a query if the index has enough information to answer the query.

Examples:

Q: `select sum(c) from foo where b<100;`

Q: `select sum(d) from foo where b<100;`

Indexes:

- **`key(b,c)`** -- covering index for first query
- **`key(b,d)`** -- covering index for second query
- **`key(b,c,d)`** -- covering index for both

How to build a covering index

Add *every* field from the select

- Not just where clause

Q: `select c,d from foo where a=10 and b=100;`

Mistake: `add index(a,b);`

- This doesn't cover the query. You still need point queries to retrieve **c** and **d** values.

Correct: `add index(a,b,c,d);`

- Includes all referenced fields
- Place **a** and **b** at beginning by Rule 1

Tokutek ™

What if Primary Key matches `where`?

Q: `select sum(c) from foo where b>100 and b<200;`

Schema: `create table foo (a int, b int, c int, ai int auto_increment, primary key (b,ai));`

- Query does a range query on primary dictionary
- Only one dictionary is accessed, in sequential order
- This is fast

Primary key covers all queries

- If sort order matches where clause, problems solved

Tokutek™

What if primary key doesn't match the `where` clause?

- Ideally, you should be able to declare secondary indexes that carry all fields

- AFAIK, storage engines don't let you do this

- With one exception... TokuDB

- TokuDB allows you to declare any index to be `CLUSTERING`

- A `CLUSTERING` index covers all queries, just like the primary key

Q: `select sum(c) from foo where b<100;`

Q: `select sum(d) from foo where b>200;`

Q: `select c,e from foo where b=1000;`

Indexes:

- `key(b,c)` covers first query
- `key(b,d)` covers second query
- `key(b,c,e)` covers first and third queries
- `key(b,c,d,e)` covers all three queries

Indexes require a lot of analysis of queries

Tokutek ™

Q: `select sum(c) from foo where b<100;`

Q: `select sum(d) from foo where b>200;`

Q: `select c,e from foo where b=1000;`

What covers all queries?

- **clustering key(b)**

Clustering keys let you focus on the where clause

- They eliminate point queries and make queries fast

Tokutek ™

We had example:

- `create table foo(a int, b int, c int);`
- `select sum(c) from foo where b=10 and a<150;`

Suppose

- 200,000 rows have **a<150**
- 100,000 rows have **b=10**
- 1000 rows have **b=10** and **a<150**

What if we use key(a) and key(b) and merge results?

Merge plan:

- Scan 200,000 rows in `key(a)` where `a<150`
- Scan 100,000 rows in `key(b)` where `b=10`
- Merge the results and find 1000 row identifiers that match query
- Do point queries with 1000 row identifiers to retrieve `c`

Better than no indexes

- Reduces number of rows scanned compared to no index
- Reduces number of point queries compared to not merging

Tokutek™

Suppose `key(a)` is clustering

Query plan:

- Scan `key(a)` for 200,000 rows where `a<150`
- Scan resulting rows for ones where `b=10`
- Retrieve `c` values from 1000 remaining rows

Once again, no point queries

What's even better?

- `clustering key(b,a)`!

Tokutek ™

Avoid Point Queries

Make sure index covers query

- By mentioning all fields in the query, not just those in the where clause

Use clustering indexes

- Clustering indexes cover all queries

- Allows user to focus on where clause

- Speeds up more (and unforeseen) queries -- simplifies database design

Tokutek™

Rule 3

Avoid Sorting

Simple selects require no post-processing

- `select * from foo where b=100;`

Just get the data and return to user

More complex queries do post-processing

- `GROUP BY` and `ORDER BY` sort the data

Index selection can avoid this sorting step

Tokutek ™

Q1: `select count(c) from foo;`

Q2: `select count(c) from foo group by b, order by b;`

Q1 plan:

- While doing a table scan, count rows with **c**

Q2 plan

- Scan table and write data to a temporary file
- Sort tmp file data by **b**
- Rescan sorted data, counting rows with **c**, for each **b**

Q2: `select count(c) from foo group by b, order by b;`

Q2: what if we use key(b,c)?

- By adding all needed fields, we cover query. FAST!
- By sorting first by b, we avoid sort. FAST!

Take home:

- Sort index on **group by** or **order by** fields to avoid sorting

Tokutek™

Use indexes to pre-sort for `order by` and `group by` queries

Tokutek™

Putting it all together

Sample Queries

```
select count(*) from foo where c=5,
group by b;
```

```
select count(*) from foo where c=5,
group by b;

key(c,b):
```

- First have **c** to limit rows retrieved (R1)
- Then have remaining rows sorted by **b** to avoid sort (R3)
- Remaining rows will be sorted by **b** because of equality test on **c**

```
select sum(d) from foo where c=100,
group by b;
```

```
select sum(d) from foo where c=100,
group by b;

key(c,b,d):
```

- First have **c** to limit rows retrieved (R1)
- Then have remaining rows sorted by **b** to avoid sort (R3)
- Make sure index covers query, to avoid point queries (R2)

Tokutek ™

Sometimes, there is no clear answer

- Best index is data dependent

Q: `select count(*) from foo where c<100, group by b;`

Indexes:

- `key(c,b)`
- `key(b,c)`

Tokutek™

Q: `select count(*) from foo where c<100, group by b;`

Query plan for `key(c,b):`

- Will filter rows where **`c<100`**

- Still need to sort by **`b`**

  ▸ Rows retrieved will not be sorted by **`b`**

  ▸ **`where`** clause does not do an equality check on **`c`**, so **`b`** values are scattered in clumps for different **`c`** values

Tokutek™

Q: `select count(*) from foo where`
`c<100, group by b;`

Query plan for `key(b,c)`:

- Sorted by b, so R3 is covered
- Rows where **`c>=100`** are also processed, so not taking advantage of R1

Tokutek ™

Which is better?

- Answer depends on the data
- If there are many rows where `c>=100`, saving time by not retrieving these useless rows helps. Use `key(c,b)`.
- If there aren't so many rows where `c>=100`, the time to execute the query is dominated by sorting. Use `key(b,c)`.

The point is, in general, rules of thumb help, but often they help us think about queries and indexes, rather than giving a recipe.

Tokutek ™

Why not just load up with indexes?

Need to keep up with insertion load
More indexes = smaller max load

Space

- Issue
  - ▸Each index adds storage requirements
- Options
  - ▸Use compression (i.e., aggressive compression of 5-15x always on for TokuDB)

Performance

- Issue
  - ▸B-trees while fast for certain indexing tasks (in memory, sequential keys), are over 20x slower for other types of indexing
- Options
  - ▸Fractal Tree indexes (TokuDB's data structure) is fast at all kinds of indexing (i.e., random keys, large tables, wide keys, etc...)
    - ▸ No need to worry about what type of index you are creating.
    - ▸ Fractal trees enable customers to index early, index often

Tokutek™

Range Query Performance

- Issue
  - ▸ Rule #2 (range query performance over point query performance) depends on range queries being fast
  - ▸ However B-trees can get **fragmented**
    - ▸ from deletions, from random insertions, ...
    - ▸ Fragmented B-trees get slow for range queries
- Options
  - ▸ For B-trees, optimize tables, dump and reload, (ie, time consuming and offline maintenance) ...
  - ▸ For Fractal Tree indexing (TokuDB), not an issue
    - ▸ Fractal trees don't fragment

Tokutek ™

For more information...

- Please contact me at <u>zardosht@tokutek.com</u> for any thoughts or feedback

- Please visit <u>Tokutek.com</u> for a copy of this <u>presentation</u> to learn more about the power of indexing, read about <u>Fractal Tree indexes</u>, or to download a free <u>eval copy</u> of TokuDB

Tokutek ™