

OpenSQLCamp 2009

Minimizing data access with covering indexes

Stéphane Combaudon

stephane.combaudon@gmail.com

Basic features of an index

- Data structure intended to speed up SELECTs
- Similar in principle to an index in a book
- Good to know:
 - Possibility to have one index for several columns
 - Overhead for every write
 - But usually negligible / boost for SELECTs
 - MySQL specific:
 - Storage engine dependant
 - Only one index used per query per table

Different types of index

- `mysql> SHOW INDEX FROM store\G`

```
***** 1. row *****
```

Table: store

Non_unique: 0

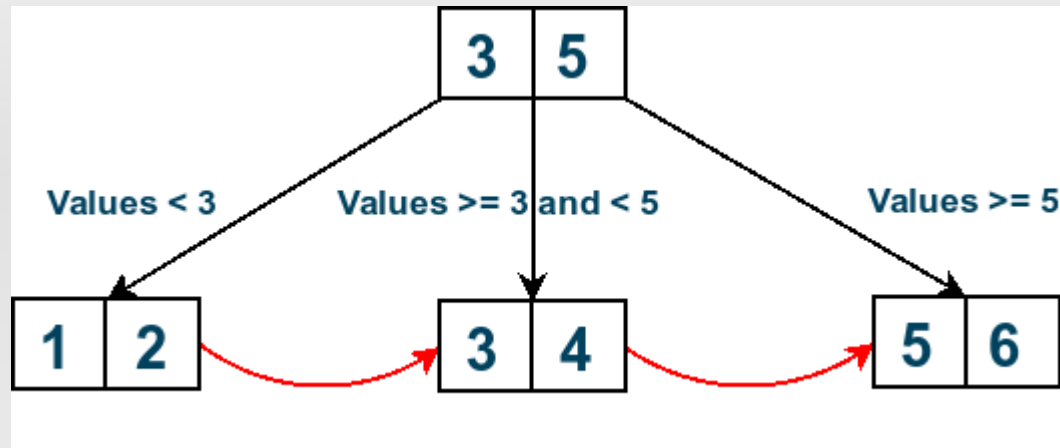
Key_name: PRIMARY

...

Index_type: BTREE

Comment:

Design of a B-Tree index



- All leaves at the same distance from the root
- Efficient insertions, deletions and lookups
- Values are sorted
- B+Trees
 - Efficient range scans
 - Values stored in the leaves

Usage of a B-Tree index

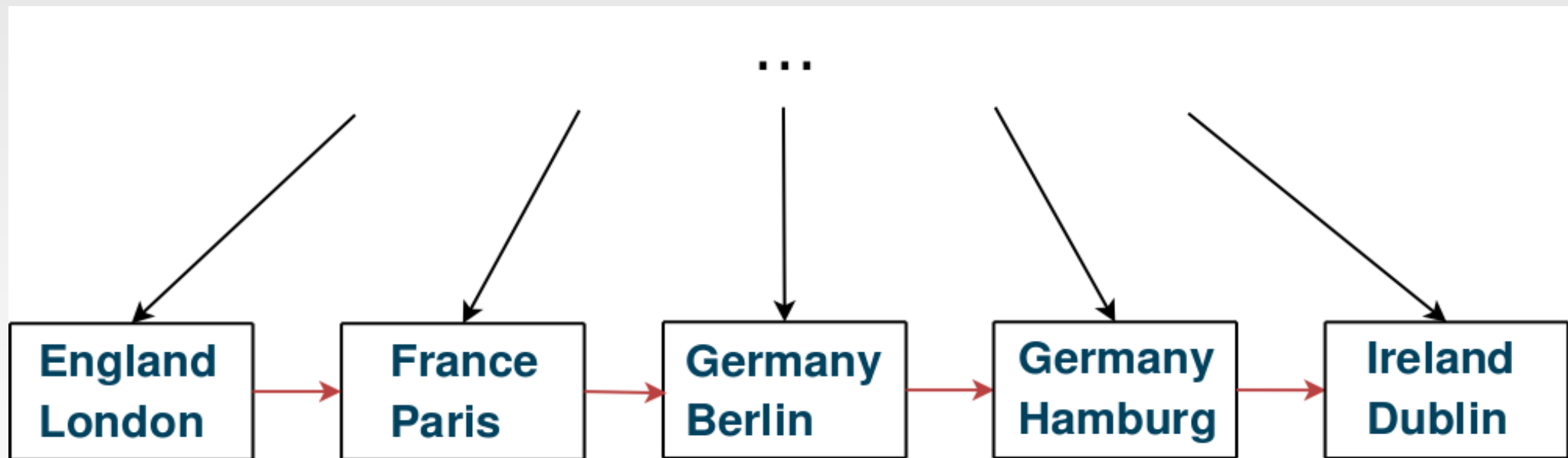
- Most kinds of lookups:
 - Exact full value (= xxx)
 - Range of values (BETWEEN xx AND yy)
 - Column prefix (LIKE 'xx%')
 - Leftmost prefix
- Sorting
 - But this can cause random I/O

Off-topic (but useful)

- Accessing data on disk : cheap but slow
 - ~ 100 random I/O ops/s
 - ~ 500,000 sequential I/O ops/s
- Accessing data in RAM : quick but expensive
 - ~ 250,000 random accesses/s
 - ~ 5,000,000 sequential accesses/s
- Disks are extremely slow for random accesses
- Not much difference for sequential accesses

Limitations of a B-Tree index

- Not useful for 'LIKE %xxx' or LIKE '%xx%'
- The columns' order is important for a multi-column index
- You can't skip columns in a multi-column index

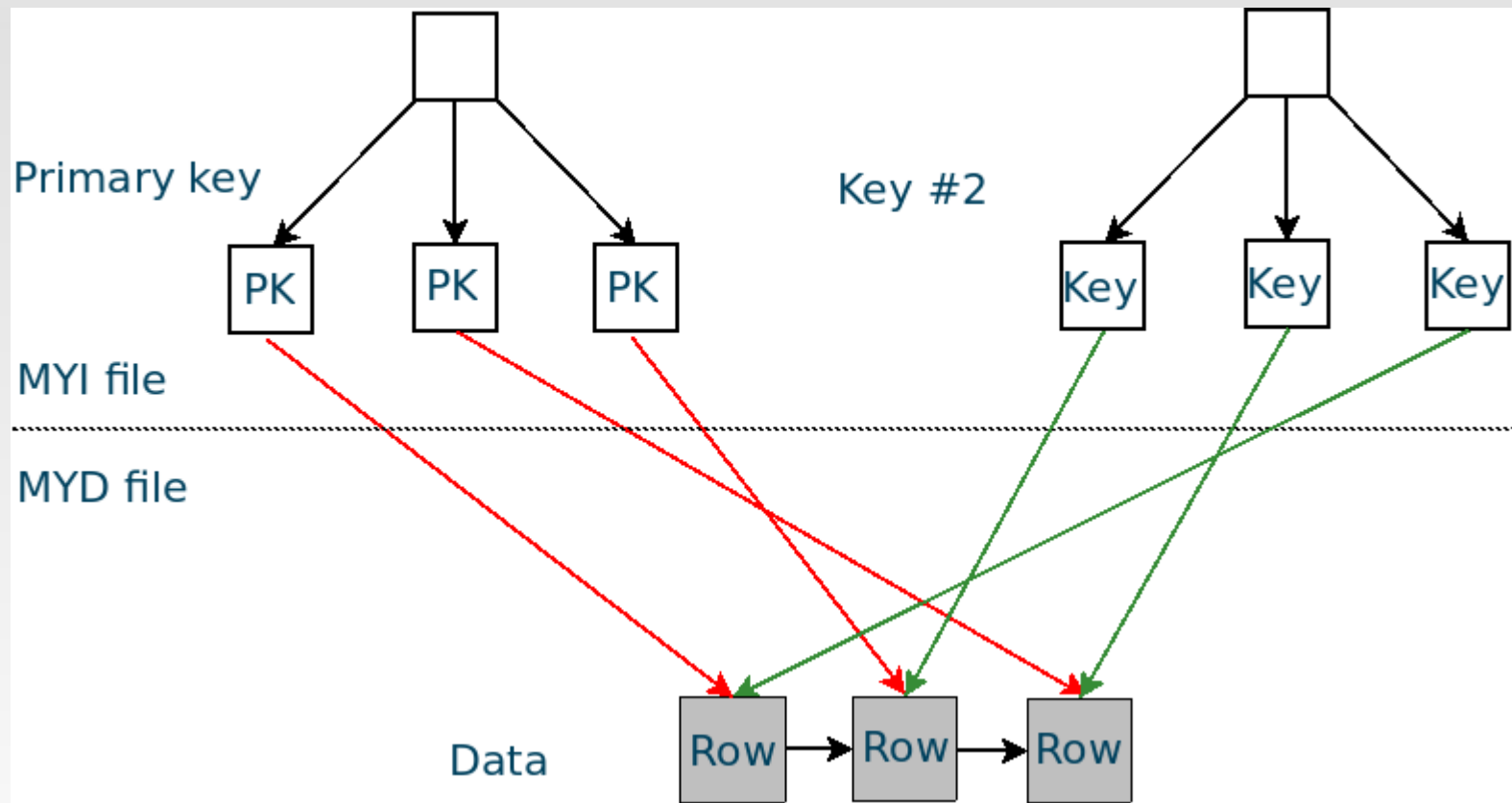


Other types of indexes

- Hash
 - Table with hash and pointer to row
 - Not supported by InnoDB or MyISAM
 - Default for the Memory storage engine
- R-Tree - T-Tree
 - Same principle as Btree
 - Used for MyISAM spatial indexes (R-Tree)
 - Used in NDB Cluster (T-Tree)

Data and indexes for MyISAM

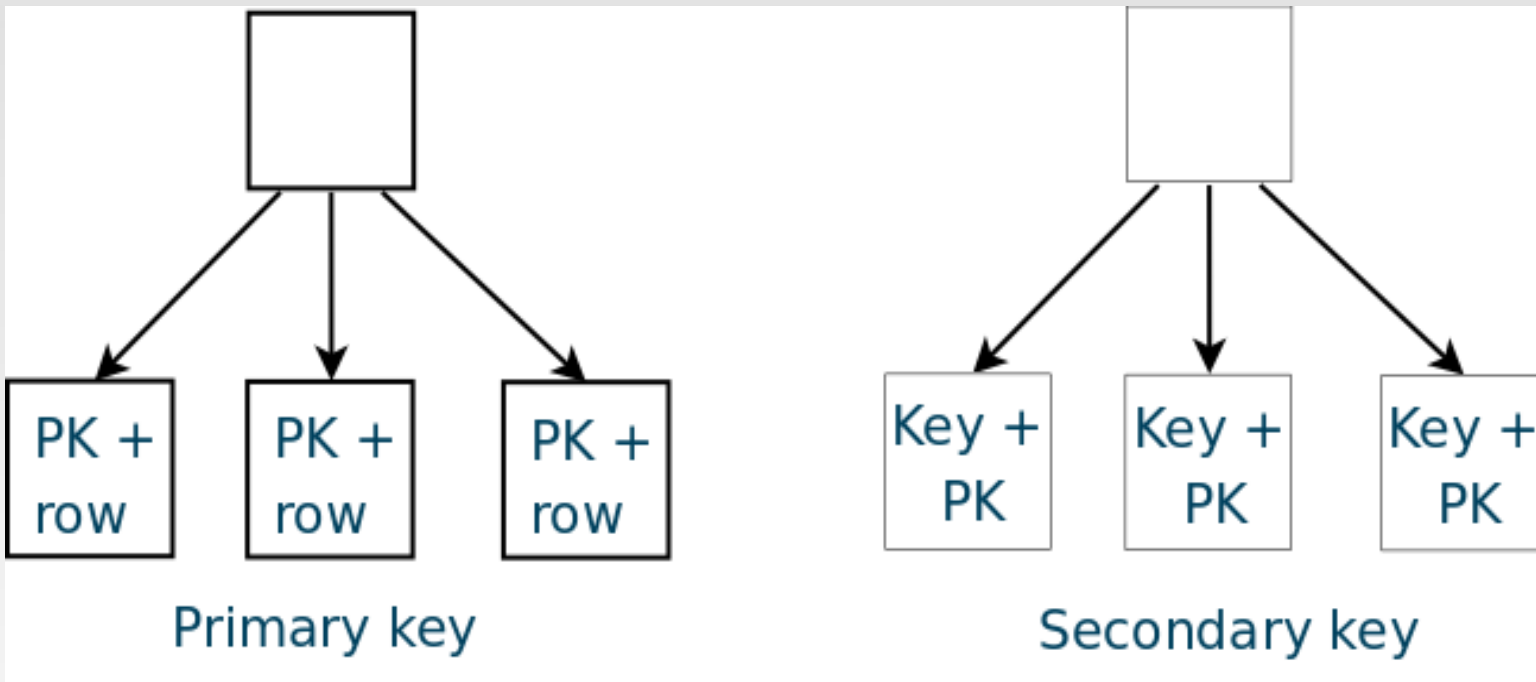
- Data, primary key and secondary key (simplified)



- No structural difference between PK and secondary key

Data and indexes for InnoDB

- Data, primary key and secondary key (simplified)



Interesting facts :

- A primary key lookup is efficient
- Two lookups needed to get row data from secondary key

Covering indexes

- If all the requested columns are part of the index
- If you index contains data
- Then:
 - You don't need to fetch data anymore
- Your query is covered by an index (=index-only query)
- Your index is covering

Execution path

- Query with traditional index:
 - Get right rows with index
 - Get data from rows
 - Send data back to client

- Index-covered query:
 - Get right rows with index
 - ~~Get data from rows~~
 - Send data back to client

Covering index and EXPLAIN

```
mysql> EXPLAIN SELECT ID FROM world.City\G
```

```
***** 1. row *****
```

```
...
```

```
type: index
```

```
possible_keys: NULL
```

```
key: PRIMARY
```

```
key_len: 4
```

```
ref: NULL
```

```
rows: 4079
```

```
Extra: Using index
```

Advantages of a covering index

- You access the index only, not the data
- Indexes are smaller and easier to cache than data
- Indexes are sorted by values: random access can become sequential access
- InnoDB can make your life easier (more later)
- => Covering indexes are very beneficial for I/O bound workloads

When you can't use a covering index

- `SELECT *`
- Indexes that don't store the values:
 - Indexes different from B-Tree indexes
 - B-Tree indexes with `MEMORY` tables
 - Indexes on a column's prefix

A case study

```
CREATE TABLE `customer` (  
    `id` int(11) NOT NULL AUTO_INCREMENT,  
    `name` varchar(20) NOT NULL DEFAULT "",  
    `age` tinyint(4) DEFAULT NULL,  
    `subscription` date NOT NULL,  
    PRIMARY KEY (`id`)  
) ENGINE=MyISAM
```


A case study

- Table populated with 5 million rows
- Name of people who subscribed on 2009-01-01 ?
- We want this list to be sorted by name

- The query:

```
mysql> SELECT name FROM customer WHERE  
subscription='2009-01-01' ORDER BY name;
```

- How to optimize it?

Without index

```
mysql> EXPLAIN SELECT name FROM customer WHERE  
subscription='2009-01-01' ORDER BY name\G
```

```
***** 1. row *****
```

```
...
```

```
type: ALL
```

```
possible_keys: NULL
```

```
key: NULL
```

```
...
```

```
rows: 5000000
```

```
Extra: Using where; Using filesort
```

First try ...

```
mysql> CREATE INDEX idx_name ON customer(name);
```

```
mysql> EXPLAIN SELECT name FROM customer WHERE  
subscription='2009-01-01' ORDER BY name\G
```

```
***** 1. row *****
```

```
...
```

```
type: ALL
```

```
...
```

```
rows: 5000000
```

```
Extra: Using where; Using filesort
```

Better ...

```
mysql> CREATE INDEX idx_sub ON customer (subscription);
```

```
mysql> EXPLAIN SELECT name FROM customer WHERE  
subscription='2009-01-01' ORDER BY name\G
```

```
***** 1. row *****
```

```
...
```

```
key: idx_sub
```

```
rows: 4370
```

```
Extra: Using where; Using filesort
```

The ideal way

```
mysql> ALTER TABLE customer ADD INDEX  
      idx_sub_name (subscription,name);
```

```
mysql> EXPLAIN SELECT name FROM customer WHERE  
subscription='2009-01-01' ORDER BY name\G
```

```
***** 1. row *****
```

...

key: idx_sub_name

rows: 4363

Extra: Using where; **Using index**

Benchmarks

- Avg number of sec to run the query
 - Without index: 3.743
 - Index on subscription: 0.435
 - Covering index: 0.012

- Covering index
 - 35x faster than index on subscription
 - 300x faster than full table scan

Off-topic (but interesting)

- We can keep the covering index in memory

```
mysql> SET GLOBAL
```

```
customer_cache.key_buffer_size = 130000000;
```

```
mysql> CACHE INDEX customer IN customer_cache;
```

```
mysql> LOAD INDEX INTO CACHE customer;
```

- Avg number of sec to run the query: 0.007
- This step is specific to MyISAM!

What about InnoDB?

- InnoDB secondary keys hold primary key values
- `mysql> EXPLAIN SELECT name,id FROM customer WHERE subscription='2009-01-01' ORDER BY name`

```
***** 1. row *****
```

```
possible_keys: idx_sub_name
```

```
key: idx_sub_name
```

```
Extra: Using where; Using index
```


2nd case study (harder)

- Same table : customer
- List people who subscribed on 2009-01-01 AND whose name ends with xx?
- `SELECT * FROM customer WHERE subscription='2009-01-01' AND name LIKE '%xx'`
- Let's add an index on (subscription,name) ...

2nd case study (harder)

```
mysql> EXPLAIN SELECT * FROM customer WHERE  
subscription='2009-01-01' AND name LIKE '%xx'
```

```
***** 1. row *****
```

```
...
```

```
key: idx_sub_name
```

```
...
```

```
rows: 500272
```

```
Extra: Using where
```

- The index is not covering anymore

Query rewriting - Indexing

- Rewriting the query

```
SELECT * FROM customer
```

```
INNER JOIN (
```

```
    SELECT id FROM customer
```

```
    WHERE subscription='2009-01-01'
```

```
    AND name LIKE '%xx'
```

```
) AS t USING(id)
```

- Adding an index

```
CREATE INDEX idx_sni ON customer (subscription,name,id)
```

Running EXPLAIN

***** 1. row *****

select_type: PRIMARY

table: <derived2>

***** 2. row *****

select_type: PRIMARY

table: customer

***** 3. row *****

select_type: DERIVED

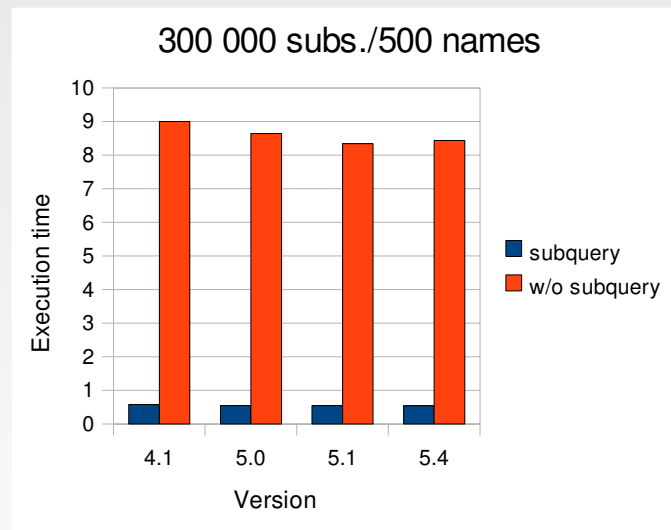
table: customer

key: idx_sni

Extra: Using where; **Using index**

Efficiency of the optimization

- 10 subs./3 names with %xx
 - Execution time is always 0.000s
- 300,000 subs./500 names with %xx



- Many intermediate situations
- Always benchmark !

InnoDB ?

- The index on (subscription,name) is already covering for the subquery
- Your work is easier: just rewrite the query if need be
- But you still need to benchmark