Covering indexes

Stéphane Combaudon - SQLI
Indexing basics

- Data structure intended to speed up SELECTs
- Similar to an index in a book
- Overhead for every write
  - Usually negligible / speed up for SELECT
- Possibility to have one index for several columns
Index types
SHOW INDEX info

mysql> SHOW INDEX FROM City\G
************************** 1. row
  Table: City
  Non_unique: 0
  Key_name: PRIMARY
  Seq_in_index: 1
  Column_name: ID
  Collation: A
  Cardinality: 4079
  Sub_part: NULL
  Packed: NULL
  Null:
  Index_type: BTREE
  Comment:
1 row in set (0.00 sec)
BTree indexes

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

- Ok for most kinds of lookups:
  - Exact full value (= xxx)
  - Range of values (BETWEEN xx AND yy)
  - Column prefix (LIKE 'xx%')
  - Leftmost prefix

- Ok for sorting too

- But
  - Not useful for 'LIKE %xxx' or LIKE '%xx%'
  - You can't skip columns
Hash indexes

- Hash table with hash and pointer to row

<table>
<thead>
<tr>
<th>Hash</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>25356</td>
<td>Link to row 61</td>
</tr>
<tr>
<td>29651</td>
<td>Link to row 2</td>
</tr>
<tr>
<td>47238</td>
<td>Link to row 16</td>
</tr>
</tbody>
</table>

Drawbacks
- Useful only for exact lookups (=, IN)
- Not supported by InnoDB or MyISAM

Benefits
- Very fast
- Compact
R-Tree and T-Tree indexes

- **R-Tree Indexes**
  - Same principle as B-Tree indexes
  - Used for spatial indexes
  - Requires the use of GIS functions
  - MyISAM only

- **T-Tree indexes**
  - Same principle as B-Tree indexes
  - Specialized for in-memory storage engines
  - Used in NDB Cluster
Index and data layouts
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)

```
  Secondary key
     /    |    |
  Key + PK  Key + PK  Key + PK
```

- Two lookups needed to get row from secondary key

```
  /    |    |
PK + row  PK + row  PK + row
```
Accessing data
Different methods to access data

- **Disk** : cheap but slow
  - ~ 100 random I/O ops/s
  - ~ 500,000 sequential I/O ops/s

- **RAM** : quick but expensive
  - ~ 250,000 random accesses/s
  - ~ 5,000,000 sequential accesses/s

- **Remember** :
  - Disks are extremely slow for random accesses
  - Not much difference for sequential accesses
Covering indexes
Index-covered queries

- When performance problems occur:
  - Add indexes
  - Rewrite your queries
  - Or both

- Do you need to fetch data (often on disk)?

- If the index contains the data, you don't

- If you don't, your query is covered by an index (=index-only query)
Index-covered queries

- 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 ***************************
    id: 1
  select_type: SIMPLE
    table: City
    type: index
possible_keys: NULL
    key: PRIMARY
  key_len: 4
    ref: NULL
  rows: 4079
Extra: Using index
Advantages of a covering index

- No access to the rows anymore!
- Indexes smaller and easier to cache than data
- Indexes sorted by values: random access can become sequential access
- Additional trick with InnoDB (more later)
- Covering indexes are very beneficial for I/O bound workloads
When you can't use a covering idx

- SELECT *

- Indexes that don't store the values:
  - Indexes different from BTree indexes
  - BTree indexes with MEMORY tables
  - Indexes on a column's prefix
Case studies
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

- Name of people who subscribed on 2009-01-01 ?
- We want this list to be sorted by name
The naïve way

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

*************************** 1. row ***************************
   id: 1
   select_type: SIMPLE
   table: customer
   type: ALL
   possible_keys: NULL
   key: NULL
   key_len: NULL
   ref: NULL
   rows: 5000000
Extra: Using where; Using filesort
First try ...

mysql> ALTER TABLE customer ADD INDEX idx_name (name)

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

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

type: ALL
possible_keys: NULL
key: NULL
key_len: NULL
ref: NULL
rows: 5000000
Extra: Using where; Using filesort
mysql> ALTER TABLE customer ADD INDEX idx_sub (subscription)

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

*************************** 1. row ***************************
...
  type: ref
    key: idx_sub
  rows: 4370
Extra: Using where; Using filesort
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

*************************** 1. row ***************************
...
   type: ref
   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
Even better for MyISAM

- We can keep the covering index in memory

```sql
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!
Even better for InnoDB

- InnoDB secondary keys hold primary key values

```sql
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
```
Another (harder) case study

- Same table: customer

- List people who subscribed on 2009-01-01 AND whose name ends up with xx?

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

- Let's add an index on (subscription, name) ...
Another (harder) case study

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

```
*************************** 1. row ***************************
...  
key: idx_sub_name
key_len: 3  
ref: const  
rows: 500272  
Extra: Using where
```

- The index is not covering anymore
Query rewriting - Indexing

- Rewriting the query
  
  ```sql
  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
  
  ```sql
  ALTER TABLE customer ADD INDEX idx_sub_name_id (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_sub_name_id
Extra: Using where; Using index
Efficiency of the optimization

- Beware of the subquery

- 10 subs./3 names with %xx
  - Original query: 0.000 s
  - Rewritten query: 0.000 s

- 300,000 subs./500 names with %xx
  - Original query: 1.284 s
  - Rewritten query: 0.553 s

- 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