MySQL Queries Tuning Hints

Shlomi Noach
openark.org
shlomi@openark.org

MySQL Users Group Meeting
Israel, April 2nd. 2009

Query Tuning Hints & Tips

- Query tuning is of the most important points when considering database performance.
- Untuned queries may lead to slow execution times, long held locks, the creation of temporary tables, disk-based sorting, and more.

What is there to tune?

- Common problems arise from improper use of data types and from missing/unused indexes.
- Some tuning is done by setting data types.
- Some by improving indexing.
- MySQL allows for index usage hints.
- And some hints can simply be embedded in normal SQL.

Example #1

- Consider the following table, where we track the Ipv4 of connected users:

```
CREATE TABLE online_user (
  online_user_id INT UNSIGNED AUTO_INCREMENT,
  session_id INT UNSIGNED,
  ip_address VARCHAR(15) CHARSET ascii,
  …
  PRIMARY KEY(online_user_id),
  KEY(ip_address)
);
```

1st attempt

- A common query would be: “Which users are currently connected from *Kookoo Islands*?”
- We base our search on a known IP range.
- Assume the range is 212.143.0.0 - 212.143.255.255.

```
SELECT online_user_id
FROM online_user
WHERE ip_address LIKE '212.143.%'
```

1st problem

- Now assume the range is 212.143.20.184 – 212.143.112.101.
- LIKE will not work now.
- Nor will BETWEEN, since '2' > '1'.

```
SELECT online_user_id
FROM online_user
WHERE ip_address
  BETWEEN '212.143.20.184' AND '212.143.112.101'
```

2nd attempt

- INET_ATON to the rescue!

```
SELECT online_user_id
FROM online_user
WHERE INET_ATON(ip_address)
  BETWEEN INET_ATON('212.143.20.184')
  AND INET_ATON('212.143.112.101')
```

2nd problem

- The KEY(is_address) cannot be used now.
- To complete this query, MySQL will have to do a full table scan.

Schema change

- Ipv4 is best described as 4 bytes. That's an INT.

```
CREATE TABLE online_user (
  online_user_id INT UNSIGNED AUTO_INCREMENT,
  session_id INT UNSIGNED,
  ip_address INT UNSIGNED,
  ...
  PRIMARY KEY(online_user_id),
  KEY(ip_address)
);
```

3rd attempt

- This time the index on ip_address is utilized.

```
SELECT online_user_id
FROM online_user
WHERE ip_address
  BETWEEN INET_ATON('212.143.20.184')
  AND INET_ATON('212.143.112.101')
```

Example #2

- We wish to find out where a user came from. We have the Ipv4 ranges for all countries and regions.

```
CREATE TABLE regions_ip_range (
  regions_ip_range_id INT UNSIGNED AUTO_INCREMENT,
  country VARCHAR(64) CHARSET utf8,
  region VARCHAR(64) CHARSET utf8,
  start_ip INT UNSIGNED,
  end_ip INT UNSIGNED
  …
  PRIMARY KEY(regions_ip_range_id),
  ...
);
```

1st attempt

- Add the following index:

```
KEY(start_ip, end_ip)
```

- Use the following query:

```
SELECT * FROM regions_ip_range
WHERE start_ip <= INET_ATON('212.143.80.165')
AND end_ip >= INET_ATON('212.143.80.165')
```

1st problem

- The 'end_ip' part of the index cannot be utilized, as there is a range condition on the first column – 'start_ip'
- The query will essentially behave as if we only have KEY(start_ip), and will most probably execute a full table scan, as on average half the rows match our criteria.

2nd attempt

- Define two indexes:

KEY(start_ip), KEY(end_ip)

- Use the same query, and hope for index_merge.

2nd problem

- index_merge is not guaranteed.
- In our case, it will probably not be used, since both conditions return a large number of rows. MySQL may choose to do full table scan.

3rd attempt

- We realize that IP ranges are *mutually exclusive*.
- Define just one index:

```
KEY(start_ip)
```

- Run the following query:

```
SELECT * FROM regions_ip_range
WHERE start_ip <= INET_ATON('212.143.80.165')
ORDER BY start_ip DESC LIMIT 1
```

Success

- By understanding the values and indexing strategies, we manage to rewrite queries which make for a dramatic performance boost.

Sometimes MySQL is wrong

- It happens that MySQL produces the wrong query plan.
- This may happen even for a simple two-table join. It may happen for a single table.
- ANALYZE table may solve the problem, but still, not always.
- How can we ask/force MySQL to use the correct plan?

Example #3: true story

- Consider the following table and query:

```
CREATE TABLE data (
  id INT UNSIGNED AUTO_INCREMENT,
  type INT UNSIGNED,
  level TINYINT UNSIGNED,
  …
  PRIMARY KEY(id),
  KEY(type),
  ...
);

SELECT id FROM data WHERE  type=12345 AND level > 3
ORDER BY id
```

Facts

- Table 'data' is very large (tens of millions of rows).
- Filtering by 'type' is good: for said query, only 110 rows have 'type=12345'.
- Query takes a very long time to complete.
- EXPLAIN shows MySQL chose using the PRIMARY KEY instead of KEY(type).

1st solution: IGNORE INDEX

- We can instruct MySQL to ignore specific keys:

```
SELECT id FROM data IGNORE INDEX(PRIMARY)
WHERE  type=12345 AND level > 3
ORDER BY id
```

2nd solution: USE INDEX

- We can instruct MySQL to only consider specific keys:

```
SELECT id FROM data USE INDEX(type)
WHERE  type=12345 AND level > 3
ORDER BY id
```

Are there other alternatives?

- It's best if we can enter MySQL's mind.
- Obviously it was wrong in choosing the query plan.
- But what caused the confusion?

… ORDER BY id ...

- When dropping the 'ORDER BY', MySQL chooses the 'good' plan.

3rd solution: avoid ORDER BY

- We can drop the 'ORDER BY' part, and let the application logic handle the sorting.
- Requires coding on the application side.

4th solution: disable use of key

- Instead of using IGNORE INDEX, we can negate the use of the primary key in the following manner:

```
SELECT id FROM data WHERE  type=12345 AND level > 3
ORDER BY id+0
```

- Id+0, IFNULL(id,id) etc. are functions on id. Functions disable keys in MySQL

5th solution: make it seem harder

- We modify the ODER BY:

```
SELECT id FROM data WHERE  type=12345 AND level > 3
ORDER BY id, type, level
```

- Since id is PRIMARY KEY, it is UNIQUE. Therefore, the order of results is unchanged.
- However, we make MySQL *think* it's more complicated. It will avoid using the PRIMARY KEY.

Thank you!

Hope to see you in the next MySQL Users Group meeting!