Improving Query Performance through a better understanding of the Optimizer

By Sasha Pachev, Independent Consultant,
the author of Understanding MySQL Internals

Why Understand the Optimizer?

- No optimizer can fully replace the intelligence of a developer in any database
- MySQL optimizer in particular works well for developers that think along with it
- In MySQL applications, majority of inefficient queries result from an incomplete understanding of what the optimizer would do

Basics of the MySQL Optimizer

- Core algorithm – nested loops join
- No hash joins
- Single-threaded
- Works well with keys, does not work well without especially when joining
- Cost-based – the cost is a guess at the number of disc reads which is largely a function of the number of record combinations to examine

Overview of the algorithm

- Examine several reasonable join order possibilities using greedy or exhaustive search
- For each join order determine the best access path for reading records from each table (data file scan, index read, index range read, index scan, etc)
- Find the plan with the minimum cost and execute it

Role of keys

- Most common optimization mistake is not having the right key, or writing the query in a way that cannot use a key
- Without a key the optimizer does full scan – O(n) reads. Really bad in a join.
- With a B-tree key, O(log n) reads
- Hash key – O(1) reads
- Hash key is the best, but works only when the exact value is known. B-tree works for prefixes and ranges.

Using the keys properly Overview

- Constrain the key
- Allow the optimizer to see the constraint
- Supply a prefix
- Use the correct column order
- Use the correct constant type
- Use LIMIT effectively
- Create keys strategically

Using the keys properly Constrain the key

- SELECT * FROM t1 WHERE n = 20 ; n is constrained to a fixed value, we can use a key – good if the majority of the records have n != 20
- SELECT * FROM t1 WHERE n > 10 AND n < 20 ; n is constrained to the (10,20) range – good if the majority of the records have n <= 10 or n >= 20

Using the keys properly

Allow the optimizer to see the constraint

- Bad query: SELECT * FROM employee WHERE YEAR(hire_date) = 2006
- Although hire_date is constrained to a range, the optimizer cannot see it because hire_date is hidden inside a function. Key on hire_date cannot be used.
- Solution: SELECT * FROM t1 WHERE ts >= '2006-01-01' AND ts <= '2006-12-31'; the optimizer can now see the range constraint

Using the keys properly

Supply the prefix

- Bad query: SELECT * FROM customer WHERE ssn LIKE '%1234'
- Key on ssn cannot be used – only the suffix is available, but not the prefix.
- Solution: ALTER TABLE customer ADD rev_ssn char(9) not null unique key; UPDATE customer SET rev_ssn = REVERSE(ssn); rewrite the query: SELECT * FROM customer WHERE rev_ssn LIKE '4321%'

Using the keys properly

Use the correct column order

- Query: SELECT * FROM customer WHERE lname = 'Jones' AND fname LIKE 'A%'
- (fname,lname) and (lname,fname) are not the same key!
- Key on (fname,lname) is no better than just (fname) – lname = 'Jones' does not help extend the constraining prefix
- Key on (lname,fname) is the best – we get the longest constraining prefix “JonesA”

Using the keys properly

Use the correct constant type

- Suppose id is defined as char(9). Consider: SELECT * FROM employee WHERE id = 12
- The key on ssn cannot be used, because 12 = '12', 12 = '12.0', 12='12.00' – there are many strings that can equal integer 12
- Fix: SELECT * FROM employee WHERE id = '12'

Using the keys properly

Use Limit effectively

- Bad query: SELECT * FROM employee WHERE city = 'New York' ORDER BY ssn LIMIT 200,20
- There is a key on city, 5000 matches. The application is paging through the records
- All 5000 records are fetched and sorted, then LIMIT is applied - slow
- Fix: basic idea - for each page we remember the value of the last ssn on the page.

Using the keys properly

Use limit effectively (cont)

- The query becomes SELECT * FROM employee WHERE city = 'New York' AND ssn > '$last_ssn' ORDER BY ssn LIMIT 20
- If we have a B-tree key on (city,ssn), the optimizer goes to 'New York','$last_ssn' entry, and reads no more than 20 subsequent entries – very fast

Using the keys properly

Create keys strategically

- Each additional key comes at a price – increased disk space use and slower insert/update/delete operations. Look for the right combination of keys to make the most common queries fast.
- Example: need to search by lname, fname and lname, lname and fname prefix, but never just fname
- Need only one key – (lname,fname). (lname) can be used, but most likely a waste since we do need (lname,fname)

Simplify the queries

Basic Guidelines

- 10 optimized queries run faster than one unoptimized.
- Write queries that are simple enough for you to understand. The optimizer will do a good job on them.
- Do not overdo it. MySQL Optimizer can handle several joins just fine.
- Query simplification is an art more than a science. No hard rules – understand the problem and apply common sense.

Simplify the queries
Example

- Problem: Retrieve full records for all customers from California that have ever made a purchase of an electronic item.
- One solution: SELECT * FROM customer WHERE state ='CA' AND id IN (SELECT customer_id FROM orders WHERE purchase_cat = 'Electronics')
- May be good with other databases, but bad with MySQL. The optimizer marks the sub-query as dependent – very slow.

Simplify the queries
Example (cont)

- Better: SELECT DISTINCT c.* FROM customer c,orders o WHERE o.customer_id = c.id AND c.state = 'CA' AND o.product_cat = 'Electronics'
- Works well if we have good keys, eg. (state), and (customer_id,product_cat).

Simplify the queries

Example (cont)

- Even better (assuming a customer typically has placed many orders): CREATE TEMPORARY TABLE tmp_cust SELECT DISTINCT o.customer_id FROM orders o,customer c WHERE o.product_cat = 'Electronics' AND c.state = 'CA'; SELECT c.* FROM customer c, tmp_cust t WHERE c.id = t.customer_id

Simplify the queries

Example (cont)

- Why? Not selecting anything from customer allows the optimizer to use the DISTINCT optimization which moves on to the next record in customer right away once a match has been found in orders. If a typical customer places a lot of orders, the performance gain could exceed the overhead of creating the temporary table and running another query.

Thank you!

Q/A

sasha@asksasha.com