Understanding Query Execution

Jay Pipes
Community Relations Manager, North America
(jay@mysql.com)

Setup

- The slides use the Sakila sample database
- Download:

http://downloads.mysql.com/docs/sakila.sql.tar.gz (or.zip)

- The Entity-Relationship diagram is on MySQL Forge:

http://forge.mysql.com/wiki/Image:SakilaSampleDB-0.8.png

- Install onto your local machine:

```
$> tar -xzvf sakila.sql.tar.gz
$> mysql –user=root –password=xxx < sakila-schema.sql
$> mysql –user=root –password=xxx < sakila-data.sql
```

Using EXPLAIN

- Simply append EXPLAIN before any SELECT statement
- Returns a table-ized version of the query execution plan chosen by the MySQL query optimizer
- Read the output from top down; each row in the EXPLAIN output represents a selection of rows from:
  - A real schema table
  - A “virtual” table (a subquery in the FROM clause)
  - A subquery in the SELECT or WHERE clause
  - A UNIONed resultset

Example EXPLAIN

```
mysql> EXPLAIN SELECT * FROM rental
    -> INNER JOIN inventory ON rental.inventory_id = inventory.inventory_id
    -> WHERE rental_date BETWEEN '2006-04-01' AND '2006-07-01' \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: rental
         type: range
possible_keys: rental_date,idx_fk_inventory_id
          key: rental_date
      key_len: 8
          ref: NULL
         rows: 1
        Extra: Using where
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: inventory
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 3
          ref: sakila.rental.inventory_id
         rows: 1
        Extra:
2 rows in set (0.00 sec)
```

The type Column – Avoid the “ALL”

- Perhaps the most important column in EXPLAIN's output
- Tells you the access strategy which MySQL chose to retrieve the specified rows
  - Watch out for the “**ALL**” access type!
    - It means you are doing a full table scan of the table's records
    - Let's see what it looks like...

The ALL access type

```
mysql> EXPLAIN SELECT * FROM rental \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: rental
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 15258
        Extra:
1 row in set (0.01 sec)
```

Here, we see that a **full table scan** is planned. This makes sense, considering we gave MySQL no WHERE clause by which the optimizer could filter the rows or use an index. Also, note the difference between this query, which uses a SELECT * FROM rental, and the next, which selects only the rental_date field...

The type Column – The “index” scan

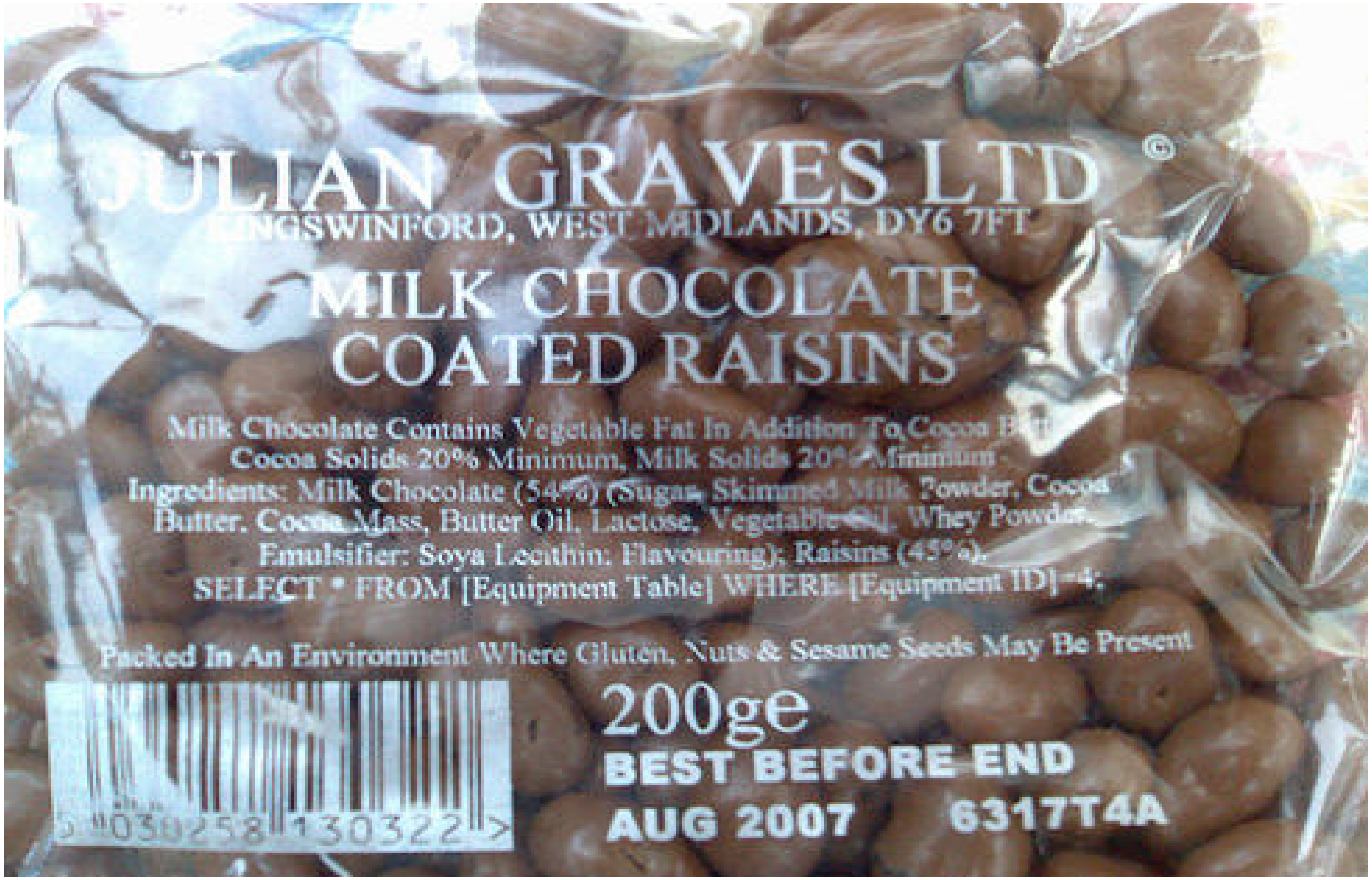
- The “index” access type is **NOT** a good thing!
  - It means you are doing a full index scan of all the index' records
  - Better than a full table scan in most cases, but still requires a LOT of resources
  - Let's see what it looks like...

The index access type

```
mysql> EXPLAIN SELECT rental_date FROM rental \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: rental
         type: index
possible_keys: NULL
          key: rental_date
      key_len: 13
          ref: NULL
         rows: 15258
        Extra: Using index
1 row in set (0.00 sec)
```

Here, we see that a **full index scan** is planned. By specifying that we only wish to see the rental_date column, we are essentially informing the query optimizer that if an index contains the rental_date information, there is no need to pull in the rest of the table fields; instead, the index itself can be used to supply all needed data...

Ahhh, SELECT * ...



JULIAN GRAVES LTD ©
KINGSWINFORD, WEST MIDLANDS, DY6 7FT
MILK CHOCOLATE
COATED RAISINS

Milk Chocolate Contains Vegetable Fat In Addition To Cocoa Butter
Cocoa Solids 20% Minimum, Milk Solids 20% Minimum
Ingredients: Milk Chocolate (54%) (Sugar, Skimmed Milk Powder, Cocoa Butter, Cocoa Mass, Butter Oil, Lactose, Vegetable Oil, Whey Powder, Emulsifier: Soya Lecithin: Flavouring): Raisins (45%).
SELECT * FROM [Equipment Table] WHERE [Equipment ID]=4;

Packed In An Environment Where Gluten, Nuts & Sesame Seeds May Be Present

200g℮
BEST BEFORE END
AUG 2007 6317T4A

http://img.thedailywtf.com/images/200701/pup2/addedsql.jpg

The type Column – The “range”

- The “**range**” access type
  - You have specified a WHERE (or ON) clause that uses a range of filters
    - The BETWEEN operator
    - The IN() operator
    - The >, >=, <, <= operators
  - MySQL has lots of optimizations for range operations, which make this access type generally fast
  - But, you must have an index on the field in question!
  - Let's see what it looks like...

The range access type

```
mysql> EXPLAIN SELECT * FROM rental
    -> WHERE rental_date
    -> BETWEEN '2006-01-01' AND '2006-07-01' \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: rental
         type: range
possible_keys: rental_date
          key: rental_date
      key_len: 8
          ref: NULL
         rows: 2614
        Extra: Using where
1 row in set (0.00 sec)
```

Here, we see that a **range access** is planned. The BETWEEN operator means we want to access rental records corresponding to a range of rental dates. Note that the possible_keys column shows us that an index on rental_date is available for the optimizer to use a range access pattern. But what would happen if there weren't an index on rental_date?

YIKES! Back to a full table scan!

```
mysql> DROP INDEX rental_date ON rental;
Query OK, 16044 rows affected (1.20 sec)
Records: 16044  Duplicates: 0  Warnings: 0

mysql> EXPLAIN SELECT * FROM rental
    -> WHERE rental_date
    -> BETWEEN '2006-01-01' AND '2006-07-01' \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: rental
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 16462
        Extra: Using where
1 row in set (0.01 sec)
```

Uh oh. Because there is no index available on the field we are filtering by, MySQL cannot use a range optimization, and resorts to the (horrible) full table scan, doing a filter on each sequential record to find records meeting our criteria... **so indexes are critically important**!

The type Column – The “index_merge”

- The “index_merge” access type
  - New in MySQL 5.0
  - Optimized data access which can take advantage of **multiple indexes** per query
    - Prior to MySQL 5.0, only **one** index can be used per table!
  - Very useful for OR-type queries
  - Let's see an example...

index_merge example

```
mysql> EXPLAIN SELECT * FROM rental
    -> WHERE rental_id IN (10,11,12)
    -> OR rental_date = '2006-02-01' \G
************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: rental
         type: index_merge
possible_keys: PRIMARY,rental_date
          key: rental_date,PRIMARY
      key_len: 8,4
          ref: NULL
         rows: 4
        Extra: Using sort_union(rental_date,PRIMARY);
Using where
1 r
```

There are two indexes available to the MySQL optimizer which can help it filter records. Prior to MySQL 5.0, the optimizer would have to choose which index would filter the most records for query...

Index Merge (cont'd)

- Can be any of:
  - **sort_union**
    - Previous example. OR conditions with ranges on non-primary key fields
  - **union**
    - OR conditions using constants or ranges on primary key fields (and table is InnoDB)
  - **intersection**
    - AND conditions with constants or range conditions on primary key fields (and table is InnoDB)
- No FULLTEXT support (but planned!)

The ref access type

- A performance benefit when the number of returned rows is relatively small
- When you supply MySQL with a constant value on an indexed field (example ahead)
- Or, when you are accessing a joined table through the joining table's indexed field (example ahead)
- Very, very fast retrieval through the index seek or single sweep multi-join

ref access type example (on indexed field)

```
mysql> EXPLAIN SELECT * FROM rental
    -> WHERE rental_id IN (10,11,12)
    -> AND rental_date = '2006-02-01' \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: rental
         type: ref
possible_keys: PRIMARY,rental_date
          key: rental_date
      key_len: 8
          ref: const
         rows: 1
        Extra: Using where
1 r
```

Again, there are two indexes available to the MySQL optimizer which can help it filter records here. But, the optimizer decides the best access strategy is to seek into the rental_date index and find the appropriate record(s) and then filter that result based on the IN() expression

ref access type example (on join)

```
mysql> EXPLAIN SELECT * FROM rental
    -> INNER JOIN inventory ON inventory.inventory_id=rental.inventory_id \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: inventory
         type: ALL
possible_keys: PRIMARY
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 5007
        Extra:
*************************** 2. row **************
           id: 1
  select_type: SIMPLE
        table: rental
         type: ref
possible_keys: idx_fk_inventory_id
          key: idx_fk_inventory_id
      key_len: 3
          ref: sakila.inventory.inventory_id
         rows: 1
        Extra:
2 rows in set (0.00 sec)
```

Here, an example of the ref access type when the outer table (inventory) is joined to the rental table on an indexed field. Note that even though we entered the rental table first in the FROM clause, the inventory table was chosen as the first table to access. Why? Because it has fewer rows...

The eq_ref access type

- Performance nerdvana
- Applies to situations where the joined column is a primary or unique index
- Über-performance because optimizations are made to quickly locate the single record which will meet the join condition
- Let's see...

eq_ref access type example

```
mysql> EXPLAIN SELECT f.film_id, f.title, c.name
     > FROM film f INNER JOIN film_category fc
     > ON f.film_id=fc.film_id INNER JOIN category c
     > ON fc.category_id=c.category_id WHERE f.title LIKE 'T%' \G
*************************** 1. row ***************************
  select_type: SIMPLE
        table: c
         type: ALL
possible_keys: PRIMARY
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 16
        Extra:
*************************** 2. row ***************
  select_type: SIMPLE
        table: fc
         type: ref
possible_keys: PRIMARY,fk_film_category_category
          key: fk_film_category_category
      key_len: 1
          ref: sakila.c.category_id
         rows: 1
        Extra: Using index
*************************** 3. row ***************
  select_type: SIMPLE
        table: f
         type: eq_ref
possible_keys: PRIMARY,idx_title
          key: PRIMARY
      key_len: 2
          ref: sakila.fc.film_id
         rows: 1
        Extra: Using where
```

Note that the third row shows the eq_ref access. Even though there was an index available which could be used (we do a WHERE on the film.title field, which has an index on it), the optimizer chooses to use the PRIMARY key to fulfill the select request. Why? It viewed the 16 rows in the category table and decided that the quickest way to access the data through the join fields in the film_category table. These 16 rows is less than the 46 rows returned from the title index which match the query filter

The importance of isolating indexed fields

```
mysql> EXPLAIN SELECT * FROM film WHERE title LIKE 'Tr%'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film
         type: range
possible_keys: idx_title
          key: idx_title
      key_len: 767
          ref: NULL
         rows: 15
        Extra: Using where
```

Nice. In the top query, we have a fast range access on the indexed field

```
mysql> EXPLAIN SELECT * FROM film WHERE LEFT(title,2) = 'Tr' \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 951
        Extra: Using where
```

Oops. In the bottom query, we have a slower full table scan because of the function operating on the indexed field (the LEFT() function)

index_subquery and unique_subquery

- Appear when you use subqueries in your SELECT statement
- Subqueries yielding a unique set of data will produce **unique_subquery**, otherwise **index_subquery**
  - unique_subquery is slightly better performing because the optimizer can entirely replace the subquery with a set of constants
    - So it becomes a range condition
- Generally, a join will be better performing though...
- Let's see it in action

unique_subery access type example

```
mysql> EXPLAIN SELECT * FROM rental r
    -> WHERE r.customer_id IN (
    -> SELECT customer_id FROM customer
    -> WHERE last_name LIKE 'S%') \G
*************************** 1. row ******
           id: 1
  select_type: PRIMARY
        table: r
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 15646
        Extra: Using where
*************************** 2. row ******
           id: 2
  select_type: DEPENDENT SUBQUERY
        table: customer
         type: unique_subquery
possible_keys: PRIMARY,idx_last_name
          key: PRIMARY
      key_len: 2
          ref: func
         rows: 1
        Extra: Using index; Using where
```

The outer table (rental) is accessed via a full table scan, and for each record in the table, a WHERE filter is applied against a set of constant customer_id values determined from the unique_subquery result in the bottom select.

Note that the select_type shows DEPENDENT SUBQUERY. This is technically incorrect, as the subquery is not a correlated subquery, but rather returns a list of constants...

Also, the strategy doesn't take advantage of the index on customer.last_name...

Rewriting the subquery as a standard join

```
mysql> EXPLAIN SELECT * FROM rental r
    -> INNER JOIN customer c
    -> ON r.customer_id=c.customer_id
    -> WHERE last_name LIKE 'S%' \G
*************************** 1. row **************
           id: 1
  select_type: SIMPLE
        table: c
         type: range
possible_keys: PRIMARY,idx_last_name
          key: idx_last_name
      key_len: 137
          ref: NULL
         rows: 54
        Extra: Using where
*************************** 2. row **************
           id: 1
  select_type: SIMPLE
        table: r
         type: ref
possible_keys: idx_fk_customer_id
          key: idx_fk_customer_id
      key_len: 2
          ref: sakila.c.customer_id
         rows: 13
        Extra:
```

The standard join gives an entirely different query execution plan (and a faster one) even though the results of the query are identical.

Suggestion:

Rewrite subqueries (especially correlated subqueries) into standard joins

The select_type Column

- select_type is mostly just informative, but watch for the following:
  - **DEPENDENT SUBQUERY**
    - Means you've used a correlated subquery
    - But can show non-correlated subqueries as dependent...
    - Ensure there isn't a more efficient way of joining the information
  - **UNCACHEABLE SUBQUERY**
    - Similar to DEPENDENT SUBQUERY, but means that for each matched row in the outer query, the subquery must be re-evaluated
    - For instance, if you use a user variable (@my_var) in your subquery

Effects of GROUP BY and ORDER BY

- GROUP BY and ORDER BY:
  - Will produce “Using filesort” and/or “Using temporary” in the Extra column only when an index cannot be used for the grouping/sorting
  - Let's see the difference between group by selects on indexed vs. non-indexed fields...

Effects of GROUP BY and ORDER BY

```
mysql> EXPLAIN SELECT r.staff_id, COUNT(*)
    -> FROM rental r GROUP BY r.staff_id \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: r
         type: index
possible_keys: NULL
          key: idx_fk_staff_id
      key_len: 1
          ref: NULL
         rows: 15646
        Extra: Using index

mysql> EXPLAIN SELECT r.return_date, COUNT(*)
    -> FROM rental r GROUP BY r.return_date \G
*************************** 1. row *********************
           id: 1
  select_type: SIMPLE
        table: r
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 15646
        Extra: Using temporary; Using filesort
```

The top query, since it groups on an indexed field, does not require a filesort and temporary storage to do the sort/group that the bottom query does, which groups on a non-indexed field.

Views

- Evaluated into an underlying SELECT statement
- WHERE expression can be applied on the view, which is then translated into a WHERE expression on the underlying table(s)
- But, what does the EXPLAIN return for a view? and...
- Does it differ from an EXPLAIN on an identical SELECT the underlying tables?
- Let's see...

The sales_by_film_category view

```
CREATE VIEW sales_by_film_category
AS
SELECT
c.name AS category
, SUM(p.amount) AS total_sales
FROM payment AS p
INNER JOIN rental AS r ON p.rental_id = r.rental_id
INNER JOIN inventory AS i ON r.inventory_id = i.inventory_id
INNER JOIN film AS f ON i.film_id = f.film_id
INNER JOIN film_category AS fc ON f.film_id = fc.film_id
INNER JOIN category AS c ON fc.category_id = c.category_id
GROUP BY c.name
ORDER BY total_sales DESC;
```

The view joins a number of tables together to make it easier and less cumbersome to query for the information above. Instead of executing the above statement, you would execute SELECT * FROM sales_by_film_category;

EXPLAIN on a view

```
mysql> EXPLAIN SELECT * FROM sales_by_film_category;
+----+-------------+------------+--------+----------------------------------+
| id | select_type | table      | type   | Extra                            |
+----+-------------+------------+--------+----------------------------------+
|  1 | PRIMARY     | <derived2> | ALL    |                                  |
|  2 | DERIVED     | c          | ALL    | Using temporary; Using filesort |
|  2 | DERIVED     | fc         | ref    | Using index                      |
|  2 | DERIVED     | f          | eq_ref | Using index                      |
|  2 | DERIVED     | i          | ref    | Using index                      |
|  2 | DERIVED     | r          | ref    | Using index                      |
|  2 | DERIVED     | p          | ref    | Using where                      |
+----+-------------+------------+--------+----------------------------------+
7 rows in set (0.22 sec)
```

I've abbreviated the output from above for brevity, but notice the plethora (yes, plethora) of **DERIVED** select types? Did you notice any derived tables (subqueries in the FROM clause) in the original view definition? Nope. This is because of the internal rewrite that occurs; the underlying select is turned into a set of derived subqueries that can be independently filtered using a WHERE expression. What happens if we remove the SELECT * and replace with a SELECT category? Will the EXPLAIN output change?

EXPLAIN on a view with specified column

```
mysql> EXPLAIN SELECT category FROM sales_by_film_category;
+----+-------------+------------+--------+------------------------------------+
| id | select_type | table      | type   | Extra                              |
+----+-------------+------------+--------+------------------------------------+
|  1 | PRIMARY     | <derived2> | ALL    |                                    |
|  2 | DERIVED     | c          | ALL    | Using temporary; Using filesort    |
|  2 | DERIVED     | fc         | ref    | Using index                        |
|  2 | DERIVED     | f          | eq_ref | Using index                        |
|  2 | DERIVED     | i          | ref    | Using index                        |
|  2 | DERIVED     | r          | ref    | Using index                        |
|  2 | DERIVED     | p          | ref    | Using where                        |
+----+-------------+------------+--------+------------------------------------+
7 rows in set (0.25 sec)
```

As you can see, there is no change at all in the EXPLAIN output, because of the underlying rewrite happening, there are fewer optimizations that we can use with views.

In general, views will perform worse than an identical SELECT on the underlying tables. However, depending on your situation, the ease of using the view may be worth the performance impact.

Closing Arguments

- EXPLAIN EXTENDED
  - Gives more information on the query execution plan produced for more complex queries. Issue EXPLAIN EXTENDED SELECT ..., and then SHOW WARNINGS;
- SHOW PROFILE
  - Patch committed by Jeremy Cole will be going into the MySQL 5.0.35 Community Server, to be released shortly
  - Provides fine-tuned profiling information on disk I/O and various internal state changes during query parsing and execution. Check out:

  http://jcole.us/blog/archives/category/mysql/

MySQL Conference & Expo 2007

April 23-26, Santa Clara, California

MySQL Conference & Expo

Over 100 Sessions & Tutorials

- Performance Tuning & Benchmarks
- Clustering, Replication, & Scale-Out
- Data Warehousing, Reporting & BI
- Security & Administration
- PHP, LAMP, & MySQL
- Web 2.0, Ajax, Ruby
- And Much More…

REGISTER
BY MARCH 14
And Save $200

http://www.mysqlconf.com