Target Practice

A Workshop in Tuning MySQL Queries

OSCON 2007

Jay Pipes
Community Relations Manager, North America
MySQL, Inc.

Setup

Download materials and MySQL Community Server

Download workshop materials

Download the presentation materials from http://jpipes.com/target-practice/materials.tar.gz (or .zip)

```
shell> cd ~
shell> wget http://jpipes.com/presentations/target-practice/target-practice.tar.gz 
(or .zip)
```

Untar/unzip to a directory

```
shell> tar -xzf target-practice.tar.gz
```

Download MySQL Community Server

Download the latest MySQL *Community* Server (binary, not source) package to this directory
mysql-5.0.45-[os]-[platform]-[compiler].tar.gz
Untar/unzip the MySQL server binary

```
shell> tar -xzf mysql-5.*
```

Install the server:

```
shell> groupadd mysql
shell> useradd -g mysql mysql
shell> usermod -aG mysql [yourusername]
shell> ln -s FULL-PATH-TO-MYSQL-VERSION-OS mysql
shell> cd mysql
shell> sudo chown -R mysql .
shell> sudo chgrp -R mysql .
shell> sudo ./scripts/mysql_install_db --user=mysql
shell> sudo chown -R mysql:mysql data
shell> sudo ./bin/mysqld_safe --user=mysql --port=3307 &
```

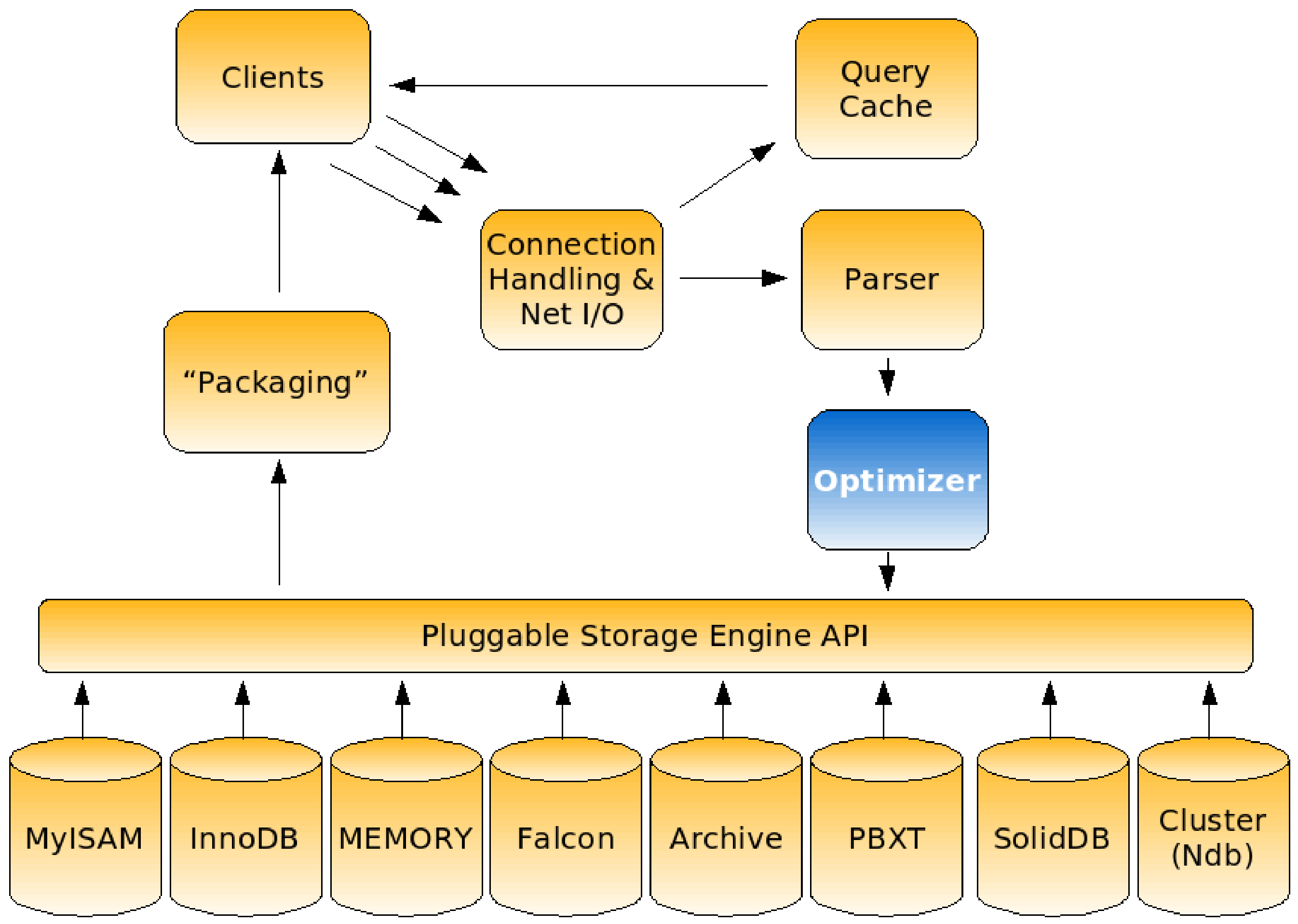
Install the sakila sample database:

```
shell> ./bin/mysql --user=root < ../sakila-db/sakila-schema.sql
shell> ./bin/mysql--user=root < ../sakila-db/sakila-data.sql
```

Use the mysql command line client and log into the sakila DB

```
shell> ./bin/mysql--user=root sakila
```

Graphical overview of the MySQL Server



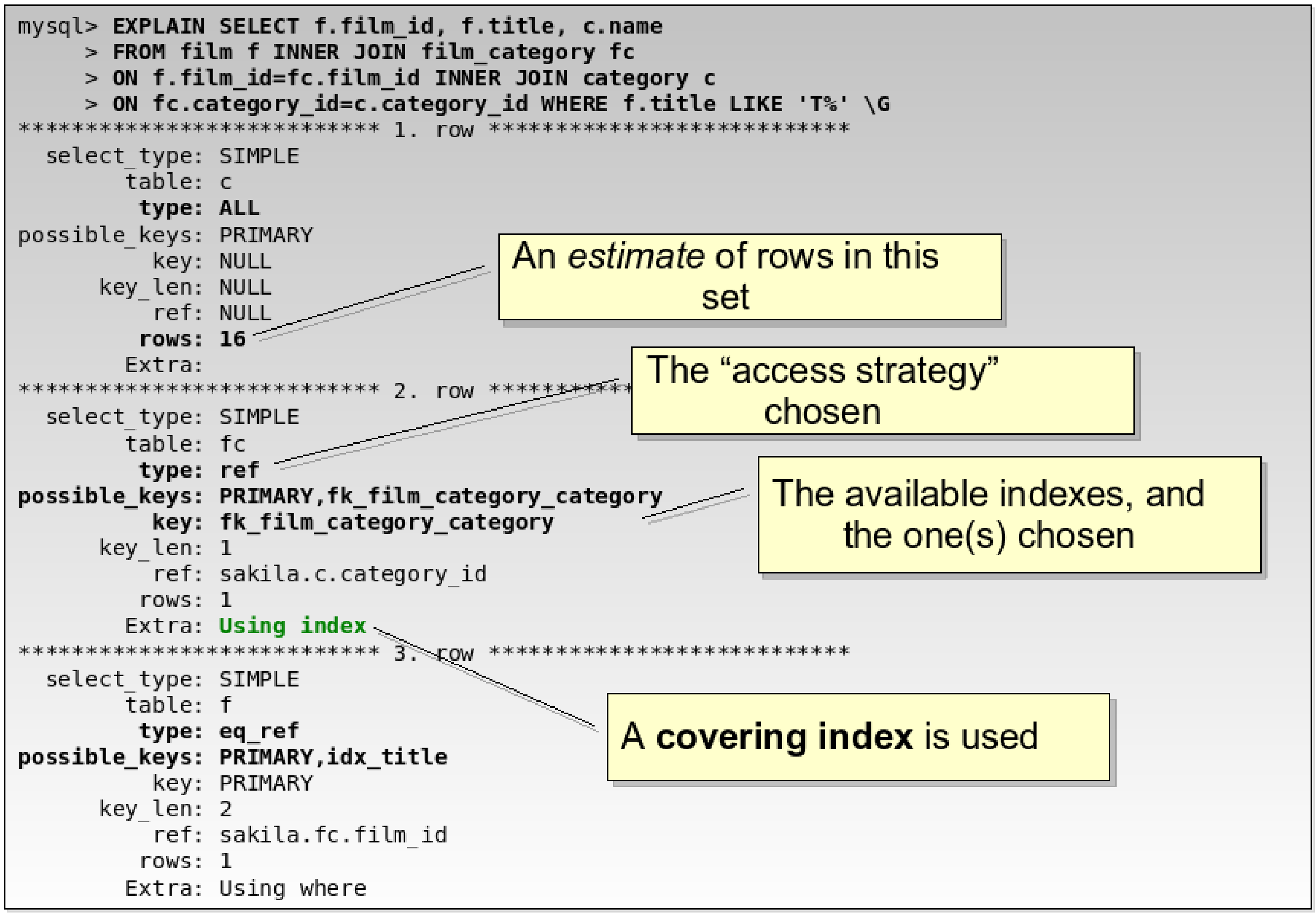
The EXPLAIN Command

- Provides the execution plan chosen by the MySQL optimizer for a specific SELECT statement
- Simply append the word EXPLAIN to the beginning of your SELECT statement

The Basics of EXPLAIN

Each row in output represents a set of information used in the SELECT

- A real schema table
- A virtual table (derived table) or temporary table
- A subquery in SELECT or WHERE
- A unioned set



```
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
```

An *estimate* of rows in this set

The "access strategy" chosen

The available indexes, and the one(s) chosen

A **covering index** is used

Hey Jay, what's up with that \G switch?

The mysql command line client has a **\G** switch which re-arranges the output into a vertical format, making wide rows easy to read in a terminal.

Important columns in EXPLAIN output

- select_type
  - The type of “set” the data in this row contains
  - Common values
    - SIMPLE: Regular table access
    - DERIVED: Result of a derived table (materialized - kind of...)
    - DEPENDENT SUBQUERY: A correlated subquery (sometimes...)
    - SUBQUERY: A non-correlated subquery
    - UNION RESULT: A unioned result
- table
  - The ***alias*** (or full table name if no alias) of the table *or derived table* from which the data in this set comes
- type
  - The “access strategy” used to grab the data in this set
- possible_keys
  - Lists any keys available to optimizer to choose to use during its creation of an execution plan
- keys
  - Lists the keys chosen by the optimizer (or a single key if < MySQL 5.0)
- rows
  - An **estimate** of the number of rows contained in this set
    - Accurate for some engines
    - *Wildly* inaccurate for other engines
- Extra
  - Extra information the *optimizer* chooses to give you
  - Lots of good stuff, which we'll be covering in a bit
- ref
  - Not as important, but shows the column used in join relations

Extra info:

http://dev.mysql.com/doc/refman/5.0/en/explain.html

The access strategies (type column values)

The best way to learn about the access strategies is to just go into the command line client and start executing queries.

Let's go and get the rental record for the rental with a rental_id of 13.

Open up a mysql client session to the sakila database and enter the following:

```
SELECT * FROM rental WHERE rental_id = 13\G
```

You should get the following returned:

```
*************************** 1. row ***************************
   rental_id: 13
 rental_date: 2005-05-25 00:22:55
inventory_id: 2294
 customer_id: 334
 return_date: 2005-05-30 04:28:55
    staff_id: 1
 last_update: 2006-02-15 21:30:53
1 row in set (0.09 sec)
```

listing 1.

Now find out what the MySQL optimizer chose as an access strategy:

```
EXPLAIN SELECT * FROM rental WHERE rental_id = 13\G
```

You should see the following:

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: rental
         type: const
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: const
         rows: 1
        Extra:
1 row in set (0.00 sec)
```

listing 2.

The const and system access strategies

In listing 2, you will notice that the access strategy chosen was **const**. The const access strategy is just about as good as you can get from the optimizer.

It means that a WHERE clause was provided in the SELECT statement that used:

- an equality operator
- on a field *indexed with a* ***unique non-nullable*** *key*
- and a *constant value* was supplied

The access strategy of ***system*** is related to const and refers to when a table with only a single row is referenced in the SELECT

Constant propogation

Let's suppose we want to include additional information from another table to our query above. Instead of just a customer ID in the output, let's grab the customer's name as well.

Enter the following into the mysql command line client:

```
SELECT r.*, c.first_name, c.last_name
FROM rental r
INNER JOIN customer c
ON r.customer_id = c.customer_id
WHERE r.rental_id = 13\G
```

You should end up with the following:

```
*************************** 1. row ***************************
   rental_id: 13
 rental_date: 2005-05-25 00:22:55
inventory_id: 2294
 customer_id: 334
 return_date: 2005-05-30 04:28:55
    staff_id: 1
 last_update: 2006-02-15 21:30:53
  first_name: RAYMOND
   last_name: MCWHORTER
1 row in set (0.01 sec)
```

listing 3.

OK, easy enough. Take a look at the EXPLAIN output now. The results should look like listing 4:

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: r
         type: const
possible_keys: PRIMARY,idx_fk_customer_id
          key: PRIMARY
      key_len: 4
          ref: const
         rows: 1
        Extra:
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: c
         type: const
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 2
          ref: const
         rows: 1
        Extra:
2 rows in set (0.00 sec)
```

listing 4.

What is going on here?

Well, the optimizer *reduced* the query through something called *constant propogation*.

The optimizer determined that the PRIMARY KEY index on rental (rental_id) would yield a single row for the criteria supplied (rental_id = 13). This single row will always yield a single value for the customer_id field, and so the optimizer first retrieves the single row from the rental table and then replaces the join condition:

```
ON r.customer_id = c.customer_id
```

with a row retrieval into the customer table on it's PRIMARY KEY index (customer_id) using the value from the rental.customer_id field.

Because of this retrieval –> row data --> constant value supplied to unique index scenario, this is called constant propogation.

The range access strategy

Continuing with our example, let's assume that instead of getting just a single rental record, we need to find all rentals that were made between the 14th and 16th of June, 2005. We'll need to make a change to our original SELECT statement to use a BETWEEN operator:

```
SELECT * FROM rental
WHERE rental_date BETWEEN '2005-06-14' AND '2005-06-16'\G
```

I won't output the results here, for brevity, but you should notice a total of 364 records returned from the above query.

Take a look at the EXPLAIN output. You should see the following:

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: rental
         type: range
possible_keys: rental_date
          key: rental_date
      key_len: 8
          ref: NULL
         rows: 364
        Extra: Using where
1 row in set (0.00 sec)
```

listing 5.

As you can see, the access strategy chosen by the optimizer is the **range** type. This makes perfect sense, since we are using a BETWEEN operator in the WHERE clause. The BETWEEN operator deals with ranges, as do <, <=, IN, >, >=.

The MySQL optimizer is highly optimized to deal with range optimizations.

Generally, range operations are very quick, but here's some things you may not be aware of regarding the range access strategy:

- An index must be available on the field operated upon by a range operator (see page 12-13)
- If too many records are estimated to be returned by the condition, the range operator won't be used (see next page)
  - an index or a full table scan will instead be preferred (see page 12-13)
- The field must not be operated on by a function call (see page XXX)

The scan vs seek dilemma

Behind the scenes, the MySQL optimizer has to decide what access strategy to use in order to retrieve information from the storage engine.

One of the decisions it must make is whether to do a *seek* operation or a *scan* operation.

A **seek** operation, generally speaking, jumps into a random place -- either on disk or in memory -- to fetch the data needed. The operation is *repeated for each piece of data needed from disk or memory*.

A **scan** operation, on the other hand, will jump to the start of a chunk of data, and sequentially read data -- either from disk or from memory -- until the end of the chunk of data.

For large amounts of data, scan operations tend to be more efficient than multiple seek operations.

Therefore, the optimizer determines which operation to perform depending on whether the *estimated* number of matched rows in the data sets is more than a certain *threshold percentage* of the total number of records in the table or index.

Note that this threshold is not a static number and has changed and may change in various versions of MySQL

To demonstrate this scan versus seek choice, let's modify our range query from above to include a larger range of rental_dates.

Enter this modified SELECT into the mysql command line client:

```
SELECT COUNT(*) FROM rental
WHERE rental_date BETWEEN '2005-06-14' AND '2005-06-21';
```

Notice I've just stretched the date range out to a week; no other modifications were made to the query. You should get the number 2,036 from the above query. There are a total of 16,044 records in the rental table.

Removing the COUNT(*) and replacing with *, get the execution plan for the above query:

```
EXPLAIN SELECT * FROM rental
WHERE rental_date BETWEEN '2005-06-14' AND '2005-06-21'\G
```

You should see the following:

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: rental
         type: ALL
possible_keys: rental_date
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 16298
        Extra: Using where
1 row in set (0.00 sec)
```

listing 6.

Clearly, the optimizer is no longer using the range access strategy. Instead, because the number of rows estimated to be matched by the condition:

```
WHERE rental_date BETWEEN '2005-06-14' AND '2005-06-21'
```

broke the internal threshold the optimizer uses to determine whether to perform a single scan or a seek operation for each matched record. In this case, the optimizer chose to perform a ***full table scan***, which corresponds to the **ALL** access strategy you see in the type column of the EXPLAIN output in listing 6.

The ALL access strategy (Full Table Scan)

The full table scan (**ALL** type column value) is definitely something you want to watch out for, particularly if:

- You are ***not*** running a data warehouse scenario
- You ***are*** supplying a WHERE clause to the SELECT
- You have very large data sets

Sometimes, full table scans cannot be avoided -- and sometimes they can perform better than other access strategies -- but *generally they are a sign of a lack of proper indexing on your schema*.

For instance, let's return to our range access example:

```
SELECT * FROM rental
WHERE rental_date BETWEEN '2005-06-14' AND '2005-06-16'\G
```

which had the EXPLAIN output:

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: rental
         type: range
possible_keys: rental_date
          key: rental_date
      key_len: 8
          ref: NULL
         rows: 364
        Extra: Using where
1 row in set (0.00 sec)
```

listing 5 (repeated).

Remember that I noted one of the requirements for getting the optimized range access strategy was that an index needed to be available on the field operated on by the range operator (rental_date in the example above).

Let's remove the index on rental_date and see what happens to our execution plan.

To demonstrate, let's drop the rental_date index on rental:

```
mysql> DROP INDEX rental_date ON rental;
Query OK, 16044 rows affected (1.22 sec)
Records: 16044  Duplicates: 0  Warnings: 0
```

and re-execute our SELECT which produced a range access strategy from before:

```
EXPLAIN SELECT * FROM rental
WHERE rental_date BETWEEN '2005-06-14' AND '2005-06-16'\G
```

you should now see the following output:

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: rental
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 16355
        Extra: Using where
1 row in set (0.02 sec)
```

Oops.

Remember: *If you don't have an appropriate index, no range optimization*

To reset our data, let's add our rental_date index back to the schema...

```
mysql> ALTER TABLE rental ADD INDEX rental_date (rental_date, inventory_id,
customer_id);
Query OK, 16044 rows affected (1.00 sec)
Records: 16044  Duplicates: 0  Warnings: 0
```

Yes, the rental_date index did originally have 3 columns in it...in case you didn't notice before. :)

The eq_ref access strategy

Now let's take a look at some of the access strategies that come up when doing joins of various sorts.

The first one we'll examine is the eq_ref access strategy, which comes up when two tables are related (joined) on a field which has a **unique, non-nullable index on one side** of the join condition, and an **indexed field on the other side**.

Let's augment our range SELECT from before to include the customer's first and last name (like we did on page 8):

```
EXPLAIN SELECT r.*, c.first_name, c.last_name
FROM rental r
INNER JOIN customer c
ON r.customer_id = c.customer_id
WHERE r.rental_date BETWEEN '2005-06-14' AND '2005-06-16'\G
```

you should get the following execution plan:

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: r
         type: range
possible_keys: idx_fk_customer_id,rental_date
          key: rental_date
      key_len: 8
          ref: NULL
         rows: 364
        Extra: Using where
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: c
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 2
          ref: sakila.r.customer_id
         rows: 1
        Extra:
2 rows in set (0.00 sec)
```

The top set should be no surprise. It's identical to that of listing 5.

The bottom set displays the access strategy chosen by the optimizer to fulfill the join on the customer table. An **eq_ref** access strategy was chosen because a unique, non-nullable index is available on customer.customer_id *and* an index is available on the rental.customer_id field.

Note that in the “ref” column of the bottom set shows the indexed field in the rental table (sakila.r.customer_id) which enables the eq_ref to occur on the customer.customer_id PRIMARY KEY field.

Caution: Geek content here.

The MySQL optimizer will perform the above execution plan using a ***nested-loops join algorithm***.

For each record the optimizer fetches in the top set (the rental table), the optimizer will seek into the customer.customer_id index for the customer_id value in the rental record. A hash table of pointers to the customer table's matched records then allows the first name and last name data to be pulled from the storage engine to complete the SELECT request.

The ref and ref_or_null access strategies

Two variations on the eq_ref access strategies that you may see pop up are the **ref** and **ref_or_null** access strategies. These strategies, while fast, are not as well performing as the eq_ref strategy because the eq_ref strategy can rely on the uniqueness of values on one side of the join condition.

The ref_or_null strategy usually appears when joining against a column that can be NULL. Generally, you should avoid joining against NULLable columns.

The ref strategy can give some good performance gains when the number of rows returned from the data access is relatively small. The ref strategy appears in the following situations:

1. when joining against two non-unique, but indexed fields
2. when supplying a non-unique index with a constant value

Execute the following code in the mysql command line client:

```
EXPLAIN SELECT * FROM rental
WHERE rental_id IN (10,11,12)
AND rental_date = '2006-02-01' \G
```

You should see the following returned:

```
*************************** 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 row in set (0.02 sec)
```

listing 7.

Here you can see the ref access strategy being chosen. The optimizer had two possible indexes (PRIMARY and rental_date) to choose from when determining an optimal execution plan. Here, it chooses to supply the constant “2006-02-01” to the rental_date index and retrieve the matched records, and then apply the rental_id filter afterwards.

We get the ref access strategy (instead of a const strategy) because the rental_date values in the rental_date index are not unique. If the index record values for rental_date were also NULLable, we might have seen the ref_or_null here.

The index_merge access strategy

The **index_merge** access strategy is the *biggest improvement to the MySQL optimizer since MySQL 5.0*.

Before MySQL 5.0, the optimizer could **only use a single index** per table in a SELECT expression, *even if more than one index could fulfill part of the query's needs.*

With MySQL 5.0+, the optimizer can make use of multiple indexes on a table's fields within a single query.

As an example, execute the following EXPLAIN SELECT in the client:

```
EXPLAIN SELECT * FROM rental
WHERE rental_id IN (10,11,12)
OR rental_date = '2006-02-01' \G
```

Notice the only difference between this query and the previous one is the use of OR instead of AND in the WHERE expression.

You should see the following output from EXPLAIN:

```
*************************** 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 row in set (0.02 sec)
```

listing 8.

It may not look like much, but the execution plan shown above is a *major* improvement over <= MySQL 4.1. The index_merge access strategy sees that there are **two** indexes (PRIMARY and rental_date) that *could* be used to filter -- or winnow, in geek terms -- the resulting data set.

Before MySQL 5.0, the optimizer would be forced to choose one or the other index to use for evaluating the SELECT. In the case of OR conditions, however, the optimizer would be forced to use a full table scan because an additional pass over the result set is necessary to fulfill the OR (union) condition. Here, however, both indexes can be queried together in a single pass.

Suggestion: If your SELECTs contain OR conditions and you are not on MySQL 5.0, consider upgrading to MySQL 5.+

The unique_subquery and index_subquery access strategies

These access strategies appear when you use subqueries in your SELECT statement (duh.)

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...
  - Don't believe me? OK.. I'll show ya.

Here is an example of a query which gets some customer information and payment amount for payments made on rentals between June 14 and 16, 2005:

```
SELECT
  c.customer_id
, c.first_name
, c.last_name
, p.amount
FROM customer c
INNER JOIN payment p
ON c.customer_id = p.customer_id
WHERE p.rental_id IN (
  SELECT rental_id FROM rental
  WHERE rental_date BETWEEN '2005-06-14' AND '2005-06-16'
)\G
```

If you execute the above query, you will find that 364 records are returned. Likely, you will see average execution times of around .09 to .10 seconds. Let's examine the EXPLAIN output for the above query. You should see the output in listing 10.

```
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: c
         type: ALL
possible_keys: PRIMARY
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 541
        Extra:
*************************** 2. row ***************************
           id: 1
  select_type: PRIMARY
        table: p
         type: ref
possible_keys: idx_fk_customer_id
          key: idx_fk_customer_id
      key_len: 2
          ref: sakila.c.customer_id
         rows: 15
        Extra: Using where
*************************** 3. row ***************************
           id: 2
  select_type: DEPENDENT SUBQUERY
        table: rental
         type: unique_subquery
possible_keys: PRIMARY,rental_date
          key: PRIMARY
      key_len: 4
          ref: func
         rows: 1
        Extra: Using where
3 rows in set (0.00 sec)
```

listing 10.

There's a number of interesting things to note about the execution plan chosen by the optimizer in listing 10:

1) The optimizer chooses to first access the customer table, c, using a full table scan strategy
2) The IN() subquery (non-correlated) in the WHERE clause yields a unique_subquery access strategy that generates a list of unique rental_ids
3) In the unique_subquery set (row 3), both the PRIMARY and the rental_date indexes are available to the optimizer to reduce the set to an appropriate set of rental_id values. The optimizer chooses to use the PRIMARY key on rental_id, even though a range access could be used on rental_date index

4) In row 2, which represents the payment table set, we see a ref access strategy deployed on the customer_id index. This makes sense, since the index customer_id is non-unique and not-nullable. However, notice that in the Extra column you see “Using where”. Why?

The reason “Using where” shows up in row 2's Extra column is because once the ref access strategy is employed on the customer_id column, an additional check must be made to match on the range of rental_id values generated from the subquery in row 3.

OK, now let's rewrite the query using a standard join to the rental table in place of the IN() subquery and show the EXPLAIN output:

```
EXPLAIN SELECT
  c.customer_id
, c.first_name
, c.last_name
, p.amount
FROM customer c
INNER JOIN payment p
ON c.customer_id = p.customer_id
INNER JOIN rental r
ON p.rental_id = r.rental_id
WHERE r.rental_date BETWEEN '2005-06-14' AND '2005-06-16'\G
)\G
```

You should see the output in listing 11.

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: r
         type: range
possible_keys: PRIMARY,rental_date
          key: rental_date
      key_len: 8
          ref: NULL
         rows: 364
        Extra: Using where; Using index
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: p
         type: ref
possible_keys: idx_fk_customer_id,fk_payment_rental
          key: fk_payment_rental
      key_len: 5
          ref: sakila.r.rental_id
         rows: 1
        Extra: Using where
*************************** 3. row ***************************
           id: 1
  select_type: SIMPLE
        table: c
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 2
          ref: sakila.p.customer_id
         rows: 1
        Extra:
3 rows in set (0.00 sec)
```

listing 11.

Working through listing 11, we now see that the range access strategy is first used on the rental table, using the rental_date index. Then, a nested loops join algorithm works its way through a ref access into the payment table, p, and then finally into the customer table, c, via an eq_ref access strategy.

The optimizer is telling us that the estimated number of rows to be returned is 364 * 1 * 1 = 364. This is, actually, the correct number of rows returned by the query.

Go ahead and execute the actual query above and compare the speed vs. the one with the IN() subquery. You should notice a significant increase in performance.

Correlated Subqueries and why they're evil

Here's an example of a standard subquery in the WHERE clause. The query grabs the last payment made by a customer, along with the customer's first and last name.

The subquery references a field in the main (outer, or primary) result set. This reference is called a *correlation*, and thus subqueries of this type are called *correlated subqueries*.

```
EXPLAIN SELECT
  p.payment_id, p.amount, p.payment_date, c.first_name, c.last_name
FROM payment p
INNER JOIN customer c
ON p.customer_id = c.customer_id
WHERE p.payment_date = (
 SELECT MAX(payment_date)
 FROM payment
 WHERE payment.customer_id = p.customer_id
)\G
```

The output from EXPLAIN should look like that of listing 12.

```
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: c
         type: ALL
possible_keys: PRIMARY
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 541
        Extra:
*************************** 2. row ***************************
           id: 1
  select_type: PRIMARY
        table: p
         type: ref
possible_keys: idx_fk_customer_id
          key: idx_fk_customer_id
      key_len: 2
          ref: sakila.c.customer_id
         rows: 14
        Extra: Using where
*************************** 3. row ***************************
           id: 2
  select_type: DEPENDENT SUBQUERY
        table: payment
         type: ref
possible_keys: idx_fk_customer_id
          key: idx_fk_customer_id
      key_len: 2
          ref: sakila.p.customer_id
         rows: 14
        Extra:
3 rows in set (0.00 sec)
```

listing 12.

In listing 12 you will notice there are three rows in the EXPLAIN output. The optimizer has chosen, in this case, to first access the customer table using a full table scan strategy, then seek into the payment.idx_fk_customer_id index using a ref access strategy. Finally, in the third row, you see the correlated subquery, denoted as select_type = DEPENDENT SUBQUERY. The ref column shows the column used as the correlation: sakila.p.customer_id.

What is actually going on with the dependent subquery, however, is not particularly optimal, and has to do with deficiencies in the MySQL optimizer with regards to correlated subqueries.

The correlated subquery will be executed *once for each matched row in the second set of information*, which happens to be the entire payments table. Therefore, in our sakila sample database, the correlated subquery will be executed **16,044 times**!

Go ahead and execute the above SELECT statement without the EXPLAIN a few times. I think you'll find that the average execution time is about .5 seconds.

How can we re-write the correlated subquery to remove this performance impact?

One way of doing it is to use a derived table -- which is a subquery in the FROM clause:

```
EXPLAIN SELECT
  p.payment_id, p.amount, p.payment_date, c.first_name, c.last_name
FROM payment p
INNER JOIN (
  SELECT customer_id, MAX(payment_date) AS payment_date
  FROM payment
  GROUP BY customer_id
) AS last_orders
ON p.customer_id = last_orders.customer_id
AND p.payment_date = last_orders.payment_date
INNER JOIN customer c
ON p.customer_id = c.customer_id\G
```

You should see something very similar to listing 13 in the output.

```
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: <derived2>
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 599
        Extra:
*************************** 2. row ***************************
           id: 1
  select_type: PRIMARY
        table: c
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 2
          ref: last_orders.customer_id
         rows: 1
        Extra:
*************************** 3. row ***************************
           id: 1
  select_type: PRIMARY
        table: p
         type: ref
possible_keys: idx_fk_customer_id
          key: idx_fk_customer_id
      key_len: 2
          ref: sakila.c.customer_id
         rows: 15
        Extra: Using where
*************************** 4. row ***************************
           id: 2
  select_type: DERIVED
        table: payment
         type: index
possible_keys: NULL
          key: idx_fk_customer_id
      key_len: 2
          ref: NULL
         rows: 16451
        Extra:
4 rows in set (0.11 sec)
```
listing 13.

Anyone notice how much time the optimizer took to generate the execution plan in listing 13? Yep, that's .11 seconds...

So, let's walk through what listing 13 is telling us. Unlike listing 12, we actually start reading this EXPLAIN output with the *last* row in the output -- the row which represents the derived table containing the customer_id and last payment date set.

The EXPLAIN output in row 4 tells us that the derived table is generated by using an index scan on the payment.customer_id column. We see an index scan because we're not supplying any WHERE condition, and optimizer sees that it can use the index on customer_id to fulfill the GROUP BY in the derived table.

For a derived table of this sort, the optimizer creates a temporary table to store the result of the derivation. This temporary table is denoted in the EXPLAIN output in the first row and given the name <derived2>.

From here on, the EXPLAIN output is fairly simple. The optimizer joins the derived table result to the customer table, c, using an *eq_ref* access strategy on the c.customer_id column, and then to the payment table, p, using a ref access strategy on the p.customer_id column. You will notice in the Extra column in row 3 the words "Using where" show up. Why?

Well, the join from the derived table to the payment table is on both customer_id *and* payment_date. This "Using where" refers to the process of filtering the payment records via the payment_date *after* the customer_id join has occurred.

So, go ahead and try running the derived table version of our query without EXPLAIN in the command line client. I think you will find that the query executes much faster than the correlated subquery version.

The reason for the better performance is because fewer accesses are being done in the case of the derived table.

Discussion point:

How can we make the derived table query even faster?

Covering indexes

When MySQL can locate every field needed for a specific table within an index (as opposed to the full table records) the index is known as a ***covering index***.

Covering indexes are critically important for performance of certain queries and joins. When a covering index is located and used by the optimizer, you will see "Using index" show up in the Extra column of the EXPLAIN output.

Important distinction

Remember that "index" in the type column means a full index scan. "Using index" in the Extra column means a covering index is being used.

The benefit of a covering index is that MySQL can grab the data directly from the index records and does not need to do a lookup operation into the data file or memory to get additional fields from the main table records.

One of the reasons that using SELECT * is not a recommended practice is because by specifying columns instead of *, you have a better chance of hitting a covering index. Consider these two statements, only differing by the * vs field list in SELECT:

```
EXPLAIN SELECT * FROM rental WHERE rental_date = '2005-06-14'\G
```

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: rental
         type: ref
possible_keys: rental_date
          key: rental_date
      key_len: 8
          ref: const
         rows: 1
        Extra:
1 row in set (0.00 sec)
```

listing 14.

```
EXPLAIN SELECT rental_id, customer_id, inventory_id
FROM rental WHERE rental_date = '2005-06-14'\G
```

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: rental
         type: ref
possible_keys: rental_date
          key: rental_date
      key_len: 8
          ref: const
         rows: 1
        Extra: Using index
1 row in set (0.00 sec)
```

listing 15.

By just specifying rental_id, customer_id, and inventory_id the optimizer is able to use the rental_date index as a covering index.

Win a book question:

Why is this a covering index? The rental_date index contains rental_date, customer_id, and inventory_id. But, we asked for those fields AND rental_id...

Using temporary and Using filesort

Let's find out the number of English-language films in each price rate, ordered by the rental rate.

Before we do, however, let's first issue the following commands in the client:

FLUSH STATUS;
SHOW STATUS LIKE 'Created_tmp%';

You should see the following output:

```
+-------------------------+-------+
| Variable_name           | Value |
+-------------------------+-------+
| Created_tmp_disk_tables | 0     |
| Created_tmp_files       | 0     |
| Created_tmp_tables      | 1     |
+-------------------------+-------+
3 rows in set (0.00 sec)
```

The FLUSH STATUS command will reset certain counters that we're going to be keeping an eye on. Two of those counters is the **Created_tmp_tables** and **Created_tmp_disk_tables** status counters, which you see displayed above. Note that the SHOW STATUS command itself creates a temporary table, as evidenced by the value of 1 in Created_tmp_tables. OK, on to the SQL code.

Execute the following code in the mysql client:

```
SELECT rental_rate, COUNT(*) AS num_films
FROM film
WHERE language_id = 1
GROUP BY rental_rate
ORDER BY rental_rate DESC;
```

you'll see the following result set:

```
+-------------+-----------+
| rental_rate | num_films |
+-------------+-----------+
|        4.99 |       336 |
|        2.99 |       323 |
|        0.99 |       341 |
+-------------+-----------+
3 rows in set (0.05 sec)
```

Before we look at the execution plan, let's first take a look to see what happened to our status counter variables. Re-execute the SHOW STATUS command, and you should see the following:

```
+-------------------------+-------+
| Variable_name           | Value |
+-------------------------+-------+
| Created_tmp_disk_tables | 0     |
| Created_tmp_files       | 0     |
| Created_tmp_tables      | 3     |
+-------------------------+-------+
3 rows in set (0.00 sec)
```

Subtracting 2 from the counter for the two executions of SHOW STATUS, we see that our SELECT expression caused MySQL to created a temporary table ***in memory***.

Now, let's take a look at the EXPLAIN output:

```

*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film
         type: ref
possible_keys: idx_fk_language_id
          key: idx_fk_language_id
      key_len: 1
          ref: const
         rows: 511
        Extra: Using where; Using temporary; Using filesort
1 row in set (0.00 sec)
```
listing 16.

There should be nothing new about the EXPLAIN output above except for the phrases “**Using temporary**” and “**Using filesort**” which appear in the Extra column.

I think it should be obvious by now what the “Using temporary” phrase refers to: the temporary in-memory table we noticed had been created via the SHOW STATUS output. The “Using filesort” phrase means that MySQL did not have data from the idx_fk_language_id index in an order needed by the query – in this case, the optimizer needed a sorted list of rental_rate values since the query is grouped and ordered by that field.

So, you might be wondering at what point you see the Created_tmp_disk_tables counter increase, instead of the Created_tmp_tables counter. The Created_tmp_disk_tables status counter is incremented when size of the temporary table created to do grouping and/or sorting is greater than the **max_heap_table_size** variable or the **tmp_table_size** variable.

If you notice the Created_tmp_disk_tables counter increasing dramatically, it is likely that you need to increase both of those server variables.

If you notice the Created_tmp_tables counter increasing dramatically, typically you have a case of a necessary index not being available for a frequently executed query.

Let's assume that we execute our grouping query on rental_rates quite frequently. How do we remove the need for a temporary table and filesort from our execution plan?

Well, one strategy we can try is to add the rental_rate field to our idx_fk_language_id index. Let's do that:

```
ALTER TABLE film
DROP INDEX idx_fk_language_id
, ADD INDEX idx_language_rental_rate (language_id, rental_rate);
```

Let's see how our EXPLAIN output changed:

```
mysql> EXPLAIN SELECT rental_rate, COUNT(*) AS num_films
    -> FROM film
    -> WHERE language_id = 1
    -> GROUP BY rental_rate
    -> ORDER BY rental_rate DESC\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film
         type: ref
possible_keys: idx_language_rental_rate
          key: idx_language_rental_rate
      key_len: 1
          ref: const
         rows: 469
        Extra: Using where; Using index
1 row in set (0.02 sec)
```

listing 17.

Voila! No more temporary table or filesort. Let's execute the query and double check the STATUS counters:

```
mysql> SELECT rental_rate, COUNT(*) AS num_films
    -> FROM film
    -> WHERE language_id = 1
    -> GROUP BY rental_rate
    -> ORDER BY rental_rate DESC;
+-------------+-----------+
| rental_rate | num_films |
+-------------+-----------+
|        4.99 |       336 |
|        2.99 |       323 |
|        0.99 |       341 |
+-------------+-----------+
3 rows in set (0.01 sec)

mysql> SHOW STATUS LIKE 'Created_tmp%';
+-------------------------+-------+
| Variable_name           | Value |
+-------------------------+-------+
| Created_tmp_disk_tables | 0     |
| Created_tmp_files       | 0     |
| Created_tmp_tables      | 4     |
+-------------------------+-------+
3 rows in set (0.00 sec)
```

Excellent. Our tmp tables only increased by one – for the SHOW STATUS command. So, we have successfully eliminated temporary table usage by creating an index on language_id and rental_rate.

But, wait. What if we wanted to see the number of films, in any language, grouped by the rental_rate?

Let's see what happens if we remove the WHERE language_id = 1 clause. See listing 17 for the EXPLAIN output.

```
mysql> EXPLAIN SELECT rental_rate, COUNT(*) AS num_films
    -> FROM film
    -> GROUP BY rental_rate\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film
         type: index
possible_keys: NULL
          key: idx_language_rental_rate
      key_len: 3
          ref: NULL
         rows: 938
        Extra: Using index; Using temporary; Using filesort
1 row in set (0.00 sec)
```

listing 17.

Oops. The temporary table showed up again along with the filesort.

Discussion point:

Why, if the optimizer has a covering index with the idx_language_rental_rate index does the temporary table and filesort happen?

Effects of functions operating on indexed columns

A final topic of discusssion when looking at the execution plans generated by the optimizer is the effect of using a function upon an indexed column.

Let's say we want to see the films which begin with “Tr”. In the film table, we have an index on the title column. Here are two queries which produce identical results, but which have very different execution plans:

```
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
1 row in set (0.03 sec)
```

listing 18.

```
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: 938
        Extra: Using where
1 row in set (0.00 sec)
```

listing 19.

As you can see, using a function on an indexed column eliminates the ability of the optimizer to use the index on that column.

Always attempt to isolate indexed columns without being operated on by a function.