



PERCONA
Performance Consulting Experts

EXPLAIN Demystified

Baron Schwartz

Percona Inc

Outline

- What is EXPLAIN?
- How MySQL executes queries
- How the execution plan becomes EXPLAIN
- How to reverse-engineer EXPLAIN
- Hopelessly complex stuff you'll never remember
- Cool tricks

What is EXPLAIN?

- Shows MySQL's estimated query plan
- Only works for SELECT queries

```
mysql> explain select title from sakila.film where film_id=5\G
*****
1. row *****
      id: 1
  select_type: SIMPLE
        table: film
         type: const
possible_keys: PRIMARY
          key: PRIMARY
    key_len: 2
         ref: const
         rows: 1
      Extra:
```

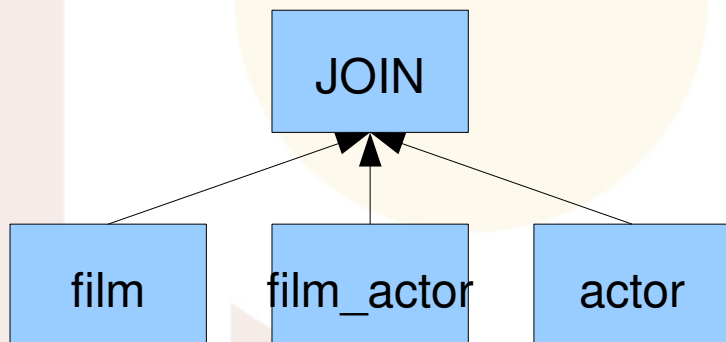
But first...

- How does MySQL execute queries?
- SQL => Parse Tree => Execution Plan
- Executioner looks at Execution Plan
- Executioner makes calls to Storage Engines
- MySQL does NOT generate byte-code!

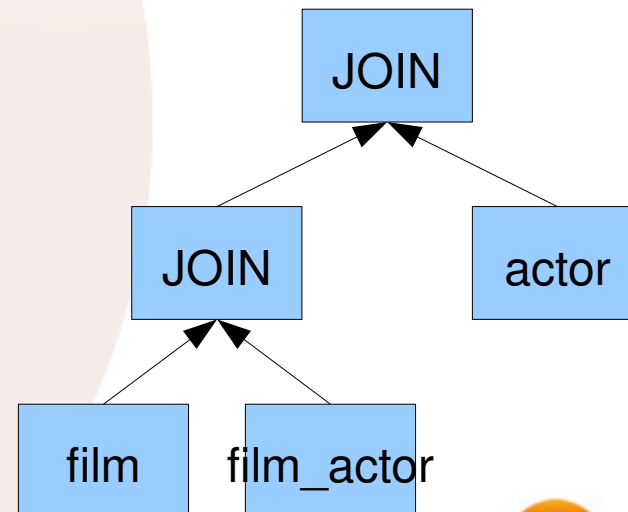
The Execution Plan

- `SELECT... sakila.film`
`JOIN sakila.film_actor USING(film_id)`
`JOIN sakila.actor USING(actor_id)`

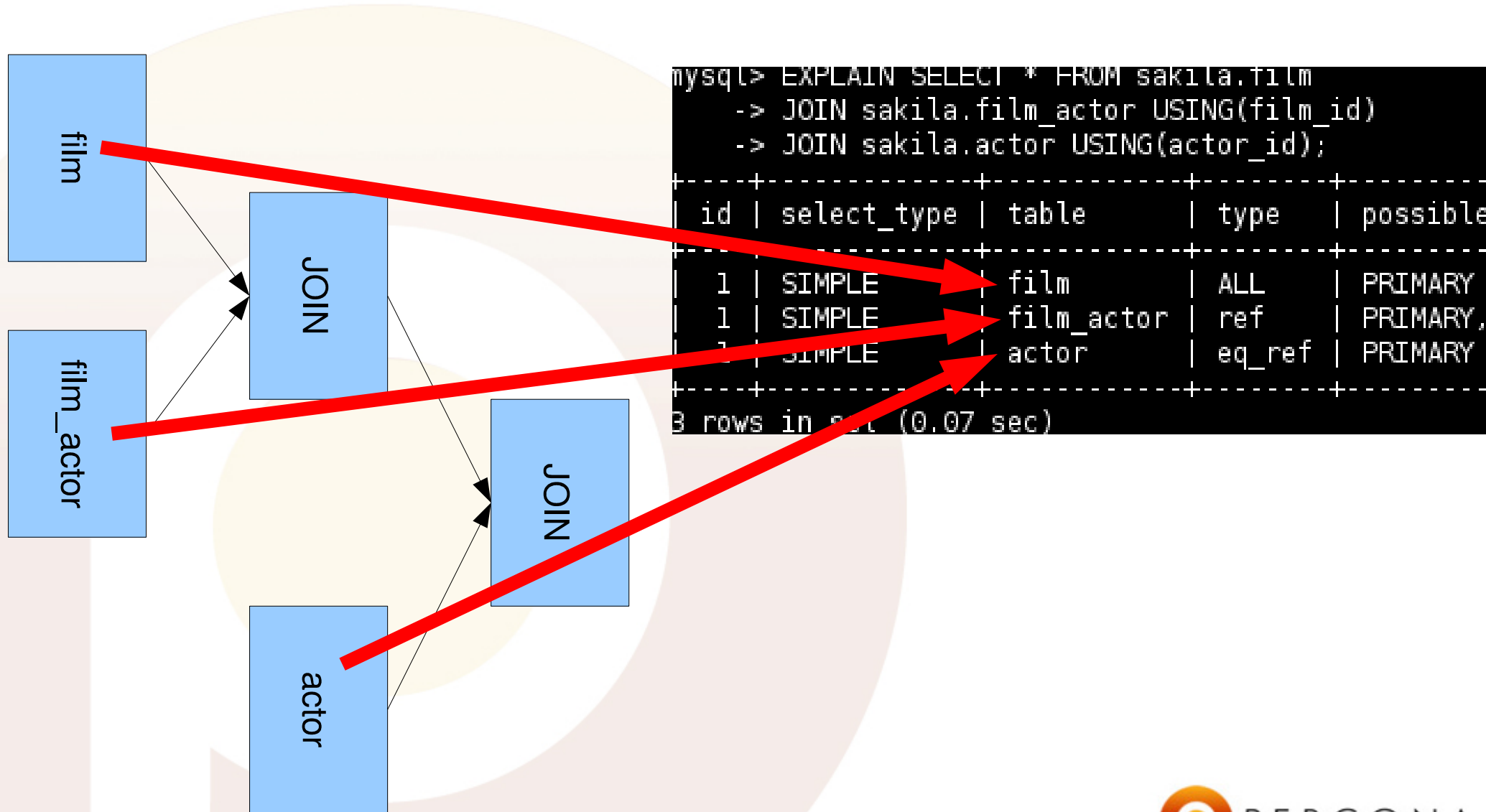
One way to do it



The MySQL Way (TM)



Where EXPLAIN comes from



Generating EXPLAIN

- MySQL actually executes the query
- But at each JOIN, instead of executing, it fills the EXPLAIN result set
- What is a JOIN?
 - Everything is a JOIN, because MySQL always uses nested-loops
 - Even a single-table SELECT or a UNION or a subquery

The Columns in EXPLAIN

- id: which SELECT the row belongs to
 - If only one SELECT with no subquery or UNION, then everything is 1
 - Otherwise, generally numbered sequentially
 - Simple/complex types
 - simple: there is only one SELECT in the whole query
 - 3 subtypes of complex: subquery, derived, union.
 - subquery: numbered according to position in SQL text
 - derived (subquery in the FROM clause): executed as a temp table
 - union: rows are spooled into a temp table, then read out with a NULL id in a row that says UNION RESULT

The Columns in EXPLAIN

- simple subquery

```
mysql> EXPLAIN SELECT (SELECT 1  
FROM sakila.actor LIMIT 1) FROM  
sakila.film;
```

```
+-----+-----+-----+...  
| id | select_type | table |...  
+-----+-----+-----+...  
| 1 | PRIMARY | film |...  
| 2 | SUBQUERY | actor |...  
+-----+-----+-----+...
```

The Columns in EXPLAIN

- derived table

```
mysql> EXPLAIN SELECT film_id FROM  
(SELECT film_id FROM sakila.film) AS der;
```

```
+-----+-----+-----+...  
| id | select_type | table |...  
+-----+-----+-----+...  
| 1 | PRIMARY | <derived2> |...  
| 2 | DERIVED | film |...  
+-----+-----+-----+...
```

The Columns in EXPLAIN

- Union

```
mysql> EXPLAIN SELECT 1 UNION ALL  
SELECT 1;
```

id	select_type	table	...
1	PRIMARY	NULL	...
2	UNION	NULL	...
NULL	UNION RESULT	<union1,2>	...

The Columns in EXPLAIN

- `select_type` shows whether it's a simple or complex select, and which type of complex select (PRIMARY, SUBQUERY, DERIVED, UNION, UNION RESULT)
- Special UNION rules: first contained SELECT has the same type as the outer context
 - e.g. the first row in a UNION contained within a subquery in the FROM clause says “DERIVED”
- Dependences and uncacheability
 - {DEPENDENT,UNCACHEABLE} {SUBQUERY,UNION}
 - Uncacheable refers to the `Item_cache`, not query cache

The Columns in EXPLAIN

- table: the table accessed, or its alias
- More complicated when there's a derived table
 - <derivedN>, where N is the subquery's id column
 - Always a forward reference: the child rows are later in the output
- Also complicated by a UNION
 - <union1,2,3...> in the UNION RESULT, where the referenced ids are parts of the UNION
 - Always a backwards reference: the referenced ids are earlier in the output

Are You Ready For This?

id	select_type	table	...
1	PRIMARY	<derived3>	...
3	DERIVED	actor	...
2	DEPENDENT SUBQUERY	film_actor	...
4	UNION	<derived6>	...
6	DERIVED	film	...
7	SUBQUERY	store	...
5	UNCACHEABLE SUBQUERY	rental	...
NULL	UNION RESULT	<union1,4>	...

Are You Ready For This?

id	select_type	table	...
1	PRIMARY	<derived3>	...
3	DERIVED	actor	...
2	DEPENDENT SUBQUERY	film_actor	...
4	UNION	<derived6>	...
6	DERIVED	film	...
7	SUBQUERY	store	...
5	UNCACHEABLE SUBQUERY	rental	...
NULL	UNION RESULT	<union1,4>	...

Are You Ready For This?

- Boundaries of UNION: first id, last id (back ref)
- Boundaries of DERIVED: every subsequent id (forward ref)
- \geq to the DERIVED id

id	select_type		
1	PRIMARY		
3	DERIVED		
2	DEPENDENT SUBQUERY		
4	UNION		
6	DERIVED		
7	SUBQUERY	store	...
5	UNCACHEABLE SUBQUERY	rental	...
NULL	UNION RESULT	<union1,4>	...
			...

Huh?

SQL, If You Want To Study

```
EXPLAIN
SELECT actor_id,
       (SELECT 1 FROM sakila.film_actor
        WHERE film_actor.actor_id = der_1.actor_id LIMIT 1)
FROM (
      SELECT actor_id
      FROM sakila.actor LIMIT 5
) AS der_1
UNION ALL
SELECT film_id,
       (SELECT @var1 FROM sakila.rental LIMIT 1)
FROM (
      SELECT film_id,
             (SELECT 1 FROM sakila.store LIMIT 1)
      FROM sakila.film LIMIT 5
) AS der_2;
```

The Columns in EXPLAIN

- type: the “join type”
- Really, the access type: how MySQL will access the rows to find results
- From worst to best
 - ALL, index, range, ref, eq_ref, const, system, NULL

```
mysql> EXPLAIN SELECT ...  
      id: 1  
select_type: SIMPLE  
  table: film  
   type: range
```

The Columns in EXPLAIN

- **possible_keys**: which indexes looked useful to the optimizer
 - the indexes that can help make row lookups efficient
- **key**: which index(es) the optimizer chose
 - the index(es) the optimizer chose to minimize overall query cost
 - not the same thing as making row lookups efficient!
 - optimizer cost metric is based on disk reads

The Columns in EXPLAIN

- `key_len`: the number of bytes of the index MySQL will use
 - MySQL uses only a leftmost prefix of the index
 - multibyte character sets make byte \neq character

```
mysql> EXPLAIN SELECT ...  
      table: film  
      type: range  
possible_keys: PRIMARY  
      key: PRIMARY  
      key_len: 2
```

The Columns in EXPLAIN

- ref: which columns/constants from preceding tables are used for lookups in the index named in the key column

```
mysql> EXPLAIN
-> SELECT STRAIGHT_JOIN f.film_id
-> FROM sakila.film AS f
->     INNER JOIN sakila.film_actor AS fa
->         ON f.film_id=fa.film_id AND fa.actor_id = 1
->     INNER JOIN sakila.actor AS a USING(actor_id);

...+-----+...+-----+-----+-----+...
...| table |...| key          | key_len | ref          |...
...+-----+...+-----+-----+-----+...
...| a     |...| PRIMARY     | 2       | const       |...
...| f     |...| idx_fk_language_id | 1       | NULL        |...
...| fa    |...| PRIMARY     | 4       | const,sakila.f.film_id |...
...+-----+...+-----+-----+-----+...
```

The Columns in EXPLAIN

- rows: estimated number of rows to read
 - for every loop in the nested-loop join plan
 - doesn't reflect LIMIT in 5.0 and earlier
- NOT the number of rows in the result set!

```
mysql> EXPLAIN SELECT * FROM
sakila.film WHERE film_id > 50
      rows: 511
      Extra: Using where
```

The Columns in EXPLAIN

- filtered: percentage of rows that satisfy a condition, in 5.1 only
- in most cases will be 0 or 100
- too complicated to explain

The Columns in EXPLAIN

- The Extra column: very important!
- Some possible values
 - Using index: covering index
 - Using where: server post-filters rows from storage engine
 - Using temporary: an implicit temporary table (for sorting or grouping rows, DISTINCT)
 - No indication of whether the temp table is on disk or in memory
 - Using filesort: external sort to order results
 - No indication of whether this is an on-disk filesort or in-memory
 - No indication of which filesort algorithm MySQL plans to use

An Example

```
mysql> EXPLAIN SELECT film_id FROM  
sakila.film WHERE film_id > 50
```

```
    id: 1  
  select_type: SIMPLE  
    table: film  
    type: range  
possible_keys: PRIMARY  
    key: PRIMARY  
  key_len: 2  
    ref: NULL  
   rows: 511  
  Extra: Using where; Using index
```

Demo: Visual Explain

- Maatkit includes a tool called mk-visual explain
- It can apply the rules I've shown (plus many others) to construct a tree that might approximate the execution plan

```
baron@kanga:~$ mk-visual-explain -c
select f.film_id from sakila.film f join sakila.film_actor using(film_id) join s
akila.actor using(actor_id)
JOIN
+- Unique index lookup
| key          f->PRIMARY
| possible_keys PRIMARY
| key_len      2
| ref          sakila.film_actor.film_id
| rows         1
+- JOIN
+- Index lookup
| key          film_actor->PRIMARY
| possible_keys PRIMARY,idx_fk_film_id
| key_len      2
| ref          sakila.actor.actor_id
| rows         13
+- Index scan
  key          actor->PRIMARY
  possible_keys PRIMARY
  key_len      2
  rows         200
baron@kanga:~$
baron@kanga:~$
```