How to Analyze and Tune MySQL Queries for Better Performance

Øystein Grøvlen
Senior Principal Software Engineer
MySQL Optimizer Team, Oracle
Program Agenda

1. Cost-based query optimization in MySQL
2. Tools for monitoring, analyzing, and tuning queries
3. Data access and index selection
4. Join optimizer
5. Sorting
6. Influencing the optimizer
Program Agenda

1. Cost-based query optimization in MySQL
2. Tools for monitoring, analyzing, and tuning queries
3. Data access and index selection
4. Join optimizer
5. Sorting
6. Influencing the optimizer
MySQL Optimizer

SELECT a, b
FROM t1, t2, t3
WHERE t1.a = t2.b
AND t2.b = t3.c
AND t2.d > 20
AND t2.d < 30;
Cost-based Query Optimization

General idea

• Assign cost to operations
• Assign cost to partial or alternative plans
• Search for plan with lowest cost

• Cost-based optimizations:
  - Access method
  - Join order
  - Subquery strategy
Optimizer Cost Model

Cost Model

Cost formulas
- Access methods
- Join
- Subquery

Cost constants
- CPU
- IO

Metadata:
- Row and index size
- Index information
- Uniqueness

Statistics:
- Table size
- Cardinality
- Range estimates

Cost model configuration

New in MySQL 5.7

Cost estimate + Row estimate
Cost Model Example

SELECT SUM(o_totalprice) FROM orders
WHERE o_orderdate BETWEEN '1994-01-01' AND '1994-12-31';

Table scan:
• IO-cost: #pages in table * IO_BLOCK_READ_COST
• CPU cost: #rows * ROW_EVALUATE_COST

Range scan (on secondary index):
• IO-cost: #rows_in_range * IO_BLOCK_READ_COST
• CPU cost: #rows_in_range * ROW_EVALUATE_COST
Cost Model Example

EXPLAIN SELECT SUM(o_totalprice) FROM orders
WHERE o_orderdate BETWEEN '1994-01-01' AND '1994-12-31';

<table>
<thead>
<tr>
<th>id</th>
<th>select type</th>
<th>table</th>
<th>type</th>
<th>possible keys</th>
<th>key</th>
<th>key len</th>
<th>rows</th>
<th>filtered</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>orders</td>
<td>ALL</td>
<td>i_o_orderdate</td>
<td>NULL</td>
<td>NULL</td>
<td>15000000</td>
<td>29.93</td>
<td>Using where</td>
</tr>
</tbody>
</table>

EXPLAIN SELECT SUM(o_totalprice) FROM orders
WHERE o_orderdate BETWEEN '1994-01-01' AND '1994-06-30';

<table>
<thead>
<tr>
<th>id</th>
<th>select type</th>
<th>table</th>
<th>type</th>
<th>possible keys</th>
<th>key</th>
<th>key len</th>
<th>rows</th>
<th>filtered</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>orders</td>
<td>range</td>
<td>i_o_orderdate</td>
<td>i_o_orderdate</td>
<td>4</td>
<td>2235118</td>
<td>100.00</td>
<td>Using index condition</td>
</tr>
</tbody>
</table>
Cost Model Example: Optimizer Trace

```
join_optimization / row_estimation / table : orders / range_analysis

"table_scan": {
  "rows": 15000000,
  "cost": 3.12e6
} // table_scan * /

"potential_range_indices": [
  {
    "index": "PRIMARY",
    "usable": false,
    "cause": "not_applicable"
  },
  {
    "index": "i_o_orderdate",
    "usable": true,
    "key_parts": ["o_orderDATE", "o_orderkey"]
  }
] // potential_range_indices */

"analyzing_range_alternatives": {
  "range_scan_alternatives": {
    "index": "i_o_orderdate",
    "ranges": ["1994-01-01 <= o_orderDATE <= 1994-12-31"],
    "index_dives_for_eq_ranges": true,
    "rowid_ordered": false,
    "using_mrr": false,
    "index_only": false,
    "rows": 4489990,
    "cost": 5.39e6,
    "chosen": false,
    "cause": "cost"
  }
} // analyzing_range_alternatives */
```

...
### Cost Model vs Real World

#### Measured Execution Times

<table>
<thead>
<tr>
<th></th>
<th>Data in Memory</th>
<th>Data on Disk</th>
<th>Data on SSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table scan</td>
<td>6.8 seconds</td>
<td>36 seconds</td>
<td>15 seconds</td>
</tr>
<tr>
<td>Index scan</td>
<td>5.2 seconds</td>
<td>2.5 hours</td>
<td>30 minutes</td>
</tr>
</tbody>
</table>

**Force Index Scan:**

```sql
SELECT SUM(o_totalprice)
FROM orders FORCE INDEX (i_o_orderdate)
WHERE o_orderdate BETWEEN '1994-01-01' AND '1994-12-31';
```
Performance Schema

Disk I/O

SELECT event_name, count_read, avg_timer_read/1000000000.0 "Avg Read Time (ms)",
    sum_number_of_bytes_read  "Bytes Read"
FROM performance_schema.file_summary_by_event_name
WHERE event_name='wait/io/file/innodb/innodb_data_file';

<table>
<thead>
<tr>
<th>event_name</th>
<th>count_read</th>
<th>Avg Read Time (ms)</th>
<th>Bytes Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>wait/io/file/innodb/innodb_data_file</td>
<td>115769</td>
<td>0.0342</td>
<td>1896759296</td>
</tr>
</tbody>
</table>

Table Scan

Index Scan

<table>
<thead>
<tr>
<th>event_name</th>
<th>count_read</th>
<th>Avg Read Time (ms)</th>
<th>Bytes Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>wait/io/file/innodb/innodb_data_file</td>
<td>2188853</td>
<td>4.2094</td>
<td>35862167552</td>
</tr>
</tbody>
</table>
Program Agenda

1. Cost-based query optimization in MySQL
2. Tools for monitoring, analyzing, and tuning queries
3. Data access and index selection
4. Join optimizer
5. Sorting
6. Influencing the optimizer
Useful tools

• MySQL Enterprise Monitor (MEM), Query Analyzer
  – Commercial product
• Performance schema, MySQL sys schema
• EXPLAIN
  – Tabular EXPLAIN
  – Structured EXPLAIN (FORMAT=JSON)
  – Visual EXPLAIN (MySQL Workbench)
• Optimizer trace
• Slow log
• Status variables (SHOW STATUS LIKE 'Sort%')
Query Analyzer Query Details

The query with the longest execution time during the Time Span (usually the slowest but not always).

Sampled Query
truncated | full | formatted

```
SELECT
  mysqlserve0..id AS id124_0,
  mysqlserve0..id AS id2_124_0,
  mysqlserve0..hasLastContact AS lastCont1_124_0,
  mysqlserve0..startTime AS startTime5_124_0,
  mysqlserve0..hasStartTime AS hasStart7_124_0,
  mysqlserve0..timestamp AS time123_0,
  mysqlserve0..capabilities AS cap118_124_0,
  mysqlserve0..hasCapabilities AS hasCap109_124_0,
  mysqlserve0..CharacterSet AS char13_124_0,
  mysqlserve0..hasCharacterSet AS hasChar11_124_0,
  mysqlserve0..collation AS collation12_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
  hasColl13_124_0,
Performance Schema

Some useful tables

- `events_statements_history`
  - `events_statements_history_long`
    - Most recent statements executed

- `events_statements_summary_by_digest`
  - Summary for similar statements (same statement digest)

- `file_summary_by_event_name`
  - Interesting event: `wait/io/file/innodb/innodb_data_file`

- `table_io_waits_summary_by_table`
  - `table_io_waits_summary_by_index_usage`
    - Statistics on storage engine access per table and index
Performance Schema

Statement events

• Tables:
  - `events_statements_current` (Current statement for each thread)
  - `events_statements_history` (10 most recent statements per thread)
  - `events_statements_history_long` (10000 most recent statements)

• Columns:

```
THREAD_ID, EVENT_ID, END_EVENT_ID, EVENT_NAME, SOURCE, TIMER_START, TIMER_END, TIMER_WAIT,
LOCK_TIME, SQL_TEXT, DIGEST, DIGEST_TEXT, CURRENT_SCHEMA, OBJECT_TYPE, OBJECT_SCHEMA,
OBJECT_NAME, OBJECT_INSTANCE_BEGIN, MYSQL_ERRNO, RETURNED_SQLSTATE, MESSAGE_TEXT, ERRORS,
WARNINGS, ROWS_AFFECTED, ROWS_SENT, ROWS_EXAMINED, CREATED_TMP_DISK_TABLES,
CREATED_TMP_TABLES, SELECT_FULL_JOIN, SELECT_FULL_RANGE_JOIN, SELECT_RANGE,
SELECT_RANGE_CHECK, SELECT_SCAN, SORT_MERGE_PASSES, SORT_RANGE, SORT_ROWS, SORT_SCAN,
NO_INDEX_USED, NO_GOOD_INDEX_USED, NESTING_EVENT_ID, NESTING_EVENT_TYPE
```
Performance Schema

Statement digest

• Normalization of queries to group statements that are similar to be grouped and summarized:

  SELECT * FROM orders WHERE o_custkey=10 AND o_totalprice>20
  SELECT * FROM orders WHERE o_custkey = 20 AND o_totalprice > 100

  SELECT * FROM orders WHERE o_custkey = ? AND o_totalprice > ?

• `events_statements_summary_by_digest`

  DIGEST, DIGEST_TEXT, COUNT_STAR, SUM_TIMER_WAIT, MIN_TIMER_WAIT, AVG_TIMER_WAIT,
  MAX_TIMER_WAIT, SUM_LOCK_TIME, SUM_ERRORS, SUM_WARNINGS, SUM_ROWS_AFFECTED,
  SUM_ROWS_SENT, SUM_ROWS_examined, SUM_CREATED_TMP_DISK_TABLES, SUM_CREATED_TMP_TABLES,
  SUM_SELECT_FULL_JOIN, SUM_SELECT_FULL_RANGE_JOIN, SUM_SELECT_RANGE, SUM_SELECT_RANGE_CHECK,
  SUM_SELECT_SCAN, SUM_SORT_MERGE_PASSES, SUM_SORT_RANGE, SUM_SORT_ROWS, SUM_SORT_SCAN,
  SUM_NO_INDEX_USED, SUM_NO_GOOD_INDEX_USED, FIRST_SEEN, LAST_SEEN
MySQL sys Schema

• A collection of views, procedures and functions, designed to make reading raw Performance Schema data easier

• Implements many common DBA and Developer use cases
  – File IO usage per user
  – Which indexes is never used?
  – Which queries use full table scans?

• Examples of very useful functions:
  – format_time(), format_bytes(), format_statement()

• Included with MySQL 5.7

• Bundled with MySQL Workbench
MySQL sys Schema

Example

statement_analysis: Lists a normalized statement view with aggregated statistics, ordered by the total execution time per normalized statement

```sql
mysql> SELECT * FROM sys.statement_analysis LIMIT 1\G
*************************** 1. row ***************************
query: INSERT INTO `mem__quan` . `nor ... nDuration` = IF ( VALUES ( ...
db: mem
full_scan: 0
exec_count: 1110067
err_count: 0
warn_count: 0
total_latency: 1.93h
max_latency: 5.03 s
avg_latency: 6.27 ms
lock_latency: 00:18:29.18
rows_sent: 0
rows_sent_avg: 0
rows_examined: 0
rows_examined_avg: 0
tmp_tables: 0
tmp_disk_tables: 0
rows_sorted: 0
sort_merge_passes: 0
digest: d48316a218e95b1b8b72db5e6b177788!
```
EXPLAIN
Understand the query plan

• Use **EXPLAIN** to print the final query plan:

**EXPLAIN SELECT * FROM t1 JOIN t2 ON t1.a = t2.a WHERE b > 10 AND c > 10;**

<table>
<thead>
<tr>
<th>id</th>
<th>select_type</th>
<th>table</th>
<th>partitions</th>
<th>type</th>
<th>possible_keys</th>
<th>key</th>
<th>key_len</th>
<th>ref</th>
<th>rows</th>
<th>filtered</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>t1</td>
<td>NULL</td>
<td>range</td>
<td>PRIMARY, idx1</td>
<td>idx1</td>
<td>4</td>
<td>NULL</td>
<td>12</td>
<td>33.33</td>
<td>Using index condition</td>
</tr>
<tr>
<td>2</td>
<td>SIMPLE</td>
<td>t2</td>
<td>NULL</td>
<td>ref</td>
<td>idx2</td>
<td>idx2</td>
<td>4</td>
<td>t1.a</td>
<td>1</td>
<td>100.00</td>
<td>NULL</td>
</tr>
</tbody>
</table>

• Explain for a running query **(New in MySQL 5.7):**

**EXPLAIN FOR CONNECTION connection_id;**
Structured EXPLAIN

- JSON format:
  - Used index parts
  - Pushed index conditions
  - Cost estimates
  - Data estimates

EXPLAIN FORMAT=JSON
SELECT * FROM t1 WHERE b > 10 AND c > 10;
EXPLAIN
{
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "17.81"
    },
    "table": {
      "table_name": "t1",
      "access_type": "range",
      "possible_keys": ["idx1"],
      "key": "idx1",
      "used_key_parts": ["b"],
      "key_length": "4",
      "rows_examined_per_scan": 12,
      "rows_produced_per_join": 3,
      "filtered": "33.33"
    },
    "index_condition": "(`test`.`t1`.`b` > 10)",
    "cost_info": {
      "read_cost": "17.01",
      "eval_cost": "0.80"
    },
    "prefix_cost": "17.81"
  }
}

```
EXPLAIN FORMAT=JSON
SELECT ...
```
Structured EXPLAIN
Assigning Conditions to Tables

EXPLAIN FORMAT=JSON SELECT * FROM t1, t2
WHERE t1.a=t2.a AND t2.a=9 AND (NOT (t1.a > 10 OR t2.b > 3) OR (t1.b=t2.b+7 AND t2.b = 5));

EXPLAIN

```json
{"query_block": {
  "select_id": 1,
  "nested_loop": [
    {
      "table": {
        "table_name": "t1",
        "access_type": "ALL",
        "rows": 10,
        "filtered": 100,
        "attached_condition": "((t1.a = 9) and (t1.a = 9))"
      }, /* table */
    }, /* table */
    {
      "table": {
        "table_name": "t2",
        "access_type": "ALL",
        "rows": 10,
        "filtered": 100,
        "using_join_buffer": "Block Nested Loop",
        "attached_condition": "((t2.a = 9) and ((t2.b <= 3) or ((t2.b = 5) and (t1.b = 12))))"
      }, /* table */
    } /* nested_loop */
  } /* query_block */
}
```
Visual EXPLAIN (MySQL Workbench)

Accumulated cost

Total query cost

Cost per table

Table and index

Rows per lookup
Optimizer Trace: Query Plan Debugging

• EXPLAIN shows the selected plan
• Optimizer trace shows WHY the plan was selected

```
SET optimizer_trace="enabled=on";
SELECT * FROM t1,t2 WHERE f1=1 AND f1=f2 AND f2>0;
SELECT trace FROM information_schema.optimizer_trace
  INTO OUTFILE <filename> LINES TERMINATED BY ";
SET optimizer_trace="enabled=off";
```

<table>
<thead>
<tr>
<th>QUERY</th>
<th>SELECT * FROM t1,t2 WHERE f1=1 AND f1=f2 AND f2&gt;0;</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRACE</td>
<td>&quot;steps&quot;: [ { &quot;join_preparation&quot;: { &quot;select#&quot;: 1,... } ... } ... ]</td>
</tr>
<tr>
<td>MISSING_BYTES_BEYOND_MAX_MEM_SIZE</td>
<td>0</td>
</tr>
<tr>
<td>INSUFFICIENT_PRIVILEGES</td>
<td>0</td>
</tr>
</tbody>
</table>
Program Agenda

1. Cost-based query optimization in MySQL
2. Tools for monitoring, analyzing, and tuning queries
3. Data access and index selection
4. Join optimizer
5. Sorting
6. Influencing the optimizer
Selecting Access Method

Finding the optimal method to read data from storage engine

• For each table, find the best access method:
  – Check if the access method is useful
  – Estimate cost of using access method
  – Select the cheapest to be used

• Choice of access method is cost based

Main access methods:
- Table scan
- Index scan
- Index look-up (ref access)
- Range scan
- Index merge
- Loose index scan
Ref Access
Single Table Queries

EXPLAIN SELECT * FROM customer WHERE c_custkey = 570887;

<table>
<thead>
<tr>
<th>id</th>
<th>select type</th>
<th>table</th>
<th>type</th>
<th>possible keys</th>
<th>key</th>
<th>key len</th>
<th>ref</th>
<th>rows</th>
<th>extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>customer</td>
<td>const</td>
<td>PRIMARY</td>
<td>PRIMARY</td>
<td>4</td>
<td>const</td>
<td>1</td>
<td>NULL</td>
</tr>
</tbody>
</table>

EXPLAIN SELECT * FROM orders WHERE o_orderdate = '1992-09-12';

<table>
<thead>
<tr>
<th>id</th>
<th>select type</th>
<th>table</th>
<th>type</th>
<th>possible keys</th>
<th>key</th>
<th>key len</th>
<th>ref</th>
<th>rows</th>
<th>extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>orders</td>
<td>ref</td>
<td>i_o_orderdate</td>
<td>i_o_orderdate</td>
<td>4</td>
<td>const</td>
<td>6271</td>
<td>NULL</td>
</tr>
</tbody>
</table>
Ref Access
Join Queries

EXPLAIN SELECT *
FROM orders JOIN customer ON c_custkey = o_custkey
WHERE o_orderdate = '1992-09-12';

<table>
<thead>
<tr>
<th>id</th>
<th>select type</th>
<th>table</th>
<th>type</th>
<th>possible keys</th>
<th>key</th>
<th>key len</th>
<th>ref</th>
<th>rows</th>
<th>extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>orders</td>
<td>ref</td>
<td>i_o_orderdate, i_o_custkey</td>
<td>i_o_orderdate</td>
<td>4</td>
<td>const</td>
<td>6271</td>
<td>Using where</td>
</tr>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>customer</td>
<td>eq_ref</td>
<td>PRIMARY</td>
<td>PRIMARY</td>
<td>4</td>
<td>dbt3.orders.o_custkey</td>
<td>1</td>
<td>NULL</td>
</tr>
</tbody>
</table>
Range Optimizer

• Goal: find the "minimal" ranges for each index

• Example:

  SELECT * FROM t1 WHERE (key1 > 10 AND key1 < 20) AND key2 > 30

• Range scan using INDEX(key1):

  ![Range scan using INDEX(key1)]

• Range scan using INDEX(key2):

  ![Range scan using INDEX(key2)]
Range Optimizer: Case Study

Why table scan?

SELECT * FROM orders
WHERE YEAR(o_orderdate) = 1997 AND MONTH(o_orderdate) = 5
AND o_clerk = 'Clerk#000001866';

<table>
<thead>
<tr>
<th>id</th>
<th>select</th>
<th>type</th>
<th>table</th>
<th>type</th>
<th>possible keys</th>
<th>key</th>
<th>key len</th>
<th>ref</th>
<th>rows</th>
<th>extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>orders</td>
<td>ALL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>15000000</td>
<td>Using where</td>
</tr>
</tbody>
</table>

Index not considered

mysql> SELECT * FROM orders WHERE year(o_orderdate) = 1997 AND MONTH(...
...
15 rows in set (8.91 sec)
Range Optimizer: Case Study

Rewrite query to avoid functions on indexed columns

```
SELECT * FROM orders
WHERE o_orderdate BETWEEN '1997-05-01' AND '1997-05-31'
AND o_clerk = 'Clerk#000001866';
```

```
<table>
<thead>
<tr>
<th>id</th>
<th>select type</th>
<th>table</th>
<th>type</th>
<th>possible keys</th>
<th>key</th>
<th>key len</th>
<th>ref</th>
<th>rows</th>
<th>extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>orders</td>
<td>range</td>
<td>i_o_orderdate</td>
<td>i_o_orderdate</td>
<td>4</td>
<td>NULL</td>
<td>376352</td>
<td>Using index condition; Using where</td>
</tr>
</tbody>
</table>

mysql> SELECT * FROM orders WHERE o_orderdate BETWEEN '1997-05-01' AND ...
...
15 rows in set (0.91 sec)
Range Optimizer: Case Study

Adding another index

```sql
CREATE INDEX i_o_clerk ON orders(o_clerk);
```

```sql
SELECT * FROM orders WHERE o_orderdate BETWEEN '1997-05-01' AND '1997-05-31'
   AND o_clerk = 'Clerk#000001866';
```

<table>
<thead>
<tr>
<th>id</th>
<th>select type</th>
<th>table</th>
<th>type</th>
<th>possible keys</th>
<th>key</th>
<th>key len</th>
<th>ref</th>
<th>rows</th>
<th>extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>orders</td>
<td>range</td>
<td>i_o_orderdate,</td>
<td>i_o_clerk</td>
<td>16</td>
<td>NULL</td>
<td>1504</td>
<td>Using index condition; Using where</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>i_o_clerk</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

mysql> SELECT * FROM orders WHERE o_orderdate BETWEEN '1997-05-01' AND ... 
... 
15 rows in set (0.01 sec)
Range Access for Multi-Column Indexes

Example table with multi-part index

• Table:

  ![Diagram of table with multi-part index]

  - INDEX idx(a, b, c);

• Logical storage layout of index:
Range Access for Multi-Column Indexes, cont

• Equality on 1ˢᵗ index column?
  – Can add condition on 2ⁿᵈ index column to range condition

• Example:

  SELECT * from t1 WHERE a IN (10,11,13) AND (b=2 OR b=4)

  • Resulting range scan:
Range Access for Multi-Column Indexes, cont

• Non-Equality on 1st index column:
  – Can NOT add condition on 2nd index column to range condition

• Example:

  SELECT * from t1 WHERE a > 10 AND a < 13 AND (b=2 OR b=4)

  • Resulting range scan:

    a > 10 AND a < 13
Range Optimizer: Case Study

Create multi-column index

```
CREATE INDEX i_o_clerk_date ON orders(o_clerk, o_orderdate);

SELECT * FROM orders
WHERE o_orderdate BETWEEN '1997-05-01' AND '1997-05-31'
AND o_clerk = 'Clerk#000001866';
```

<table>
<thead>
<tr>
<th>id</th>
<th>select type</th>
<th>table</th>
<th>type</th>
<th>possible keys</th>
<th>key</th>
<th>key len</th>
<th>ref</th>
<th>rows</th>
<th>extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>orders</td>
<td>range</td>
<td>i_o_orderdate, i_o_clerk, i_o_clerk_date</td>
<td>i_o_clerk_date</td>
<td>20</td>
<td>NULL</td>
<td>14</td>
<td>Using index condition</td>
</tr>
</tbody>
</table>

mysql> SELECT * FROM orders WHERE o_orderdate BETWEEN '1997-05-01' AND ... ...
15 rows in set (0.00 sec)
### Performance Schema: Query History

**UPDATE performance_schema.setup_consumers**

```sql
SET enabled='YES' WHERE name = 'events_statements_history';
```

**MySQL 5.7:** Enabled by default

```sql
mysql> SELECT sql_text, (timer_wait)/1000000000.0 "t (ms)", rows_examined rows
    FROM performance_schema.events_statements_history ORDER BY timer_start;
```

<table>
<thead>
<tr>
<th>sql_text</th>
<th>t (ms)</th>
<th>rows</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SELECT * FROM orders WHERE o_orderdate BETWEEN '1997-05-01' ...</code></td>
<td>8.1690</td>
<td>1505</td>
</tr>
<tr>
<td><code>SELECT * FROM orders WHERE o_orderdate BETWEEN '1997-05-01' ...</code></td>
<td>7.2120</td>
<td>1505</td>
</tr>
<tr>
<td><code>SELECT * FROM orders WHERE o_orderdate BETWEEN '1997-05-01' ...</code></td>
<td>8.1613</td>
<td>1505</td>
</tr>
<tr>
<td><code>SELECT * FROM orders WHERE o_orderdate BETWEEN '1997-05-01' ...</code></td>
<td>7.0535</td>
<td>1505</td>
</tr>
<tr>
<td><code>CREATE INDEX i_o_clerk_date ON orders(o_clerk,o_orderdate)</code></td>
<td>82036.4190</td>
<td>0</td>
</tr>
<tr>
<td><code>SELECT * FROM orders WHERE o_orderdate BETWEEN '1997-05-01' ...</code></td>
<td>0.7259</td>
<td>15</td>
</tr>
<tr>
<td><code>SELECT * FROM orders WHERE o_orderdate BETWEEN '1997-05-01' ...</code></td>
<td>0.5791</td>
<td>15</td>
</tr>
<tr>
<td><code>SELECT * FROM orders WHERE o_orderdate BETWEEN '1997-05-01' ...</code></td>
<td>0.5423</td>
<td>15</td>
</tr>
<tr>
<td><code>SELECT * FROM orders WHERE o_orderdate BETWEEN '1997-05-01' ...</code></td>
<td>0.6031</td>
<td>15</td>
</tr>
<tr>
<td><code>SELECT * FROM orders WHERE o_orderdate BETWEEN '1997-05-01' ...</code></td>
<td>0.2710</td>
<td>15</td>
</tr>
</tbody>
</table>
Program Agenda

1. Cost-based query optimization in MySQL
2. Tools for monitoring, analyzing, and tuning queries
3. Data access and index selection
4. Join optimizer
5. Sorting
6. Influencing the optimizer
Join Optimizer

"Greedy search strategy"

• Goal: Given a JOIN of N tables, find the best JOIN ordering

• Strategy:
  – Start with all 1-table plans (Sorted based on size and key dependency)
  – Expand each plan with remaining tables
    • Depth-first
  – If “cost of partial plan” > “cost of best plan”:
    • “prune” plan
  – Heuristic pruning:
    • Prune less promising partial plans
    • May in rare cases miss most optimal plan (turn off with `set optimizer_prune_level = 0`)

N! possible plans
Join Optimizer Illustrated

SELECT city.name AS capital, language.name
FROM city
JOIN country ON city.country_id = country.country_id
JOIN language ON country.country_id = language.country_id
WHERE city.city_id = country.capital
Join Optimizer

Example

EXPLAIN SELECT *
FROM customers JOIN orders ON c_custkey = o_custkey
WHERE c_acctbal < -1000 AND o_orderdate < '1993-01-01';

<table>
<thead>
<tr>
<th>id</th>
<th>select type</th>
<th>table</th>
<th>type</th>
<th>possible keys</th>
<th>key</th>
<th>key len</th>
<th>ref</th>
<th>rows</th>
<th>filtered</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>orders</td>
<td>ALL</td>
<td>i_o_orderdate, i_o_custkey</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>15000000</td>
<td>31.19</td>
<td>Using where</td>
</tr>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>customer</td>
<td>eq_ref</td>
<td>PRIMARY</td>
<td>PRIMARY</td>
<td>4</td>
<td>dbt3.orders.o_custkey</td>
<td>1</td>
<td>33.33 Using where</td>
<td></td>
</tr>
</tbody>
</table>
Join Optimizer

Change join order with STRAIGHT_JOIN

EXPLAIN SELECT STRAIGHT_JOIN *
FROM customer JOIN orders ON c_custkey = o_custkey
WHERE c_acctbal < -1000 AND o_orderdate < '1993-01-01';

<table>
<thead>
<tr>
<th>id</th>
<th>select type</th>
<th>table</th>
<th>type</th>
<th>possible keys</th>
<th>key</th>
<th>key len</th>
<th>ref</th>
<th>rows</th>
<th>filtered</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>customer</td>
<td>ALL</td>
<td>PRIMARY</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>1500000</td>
<td>33.33</td>
<td>Using where</td>
</tr>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>orders</td>
<td>ref</td>
<td>i_o_orderdate, i_o_custkey</td>
<td>i_o_custkey</td>
<td>5</td>
<td>dbt3. customer. c_custkey</td>
<td>15</td>
<td>31.19</td>
<td>Using where</td>
</tr>
</tbody>
</table>
Join Order

Performance

Query Execution Time (seconds)

- orders → customer
- customer → orders
Join Optimizer: Case study

DBT-3 Query 8: National Market Share Query

```
SELECT o_year, SUM(CASE WHEN nation = 'FRANCE' THEN volume ELSE 0 END) / SUM(volume) AS mkt_share
FROM (SELECT EXTRACT(YEAR FROM o_orderdate) AS o_year,
       l_extendedprice * (1 - l_discount) AS volume, n2.n_name AS nation
       FROM part
       JOIN lineitem ON p_partkey = l_partkey
       JOIN supplier ON s_suppkey = l_suppkey
       JOIN orders ON l_orderkey = o_orderkey
       JOIN customer ON o_custkey = c_custkey
       JOIN nation n1 ON c_nationkey = n1.n_nationkey
       JOIN region ON  n1.n_regionkey = r_regionkey
       JOIN nation n2 ON s_nationkey = n2.n_nationkey
       WHERE r_name = 'EUROPE' AND o_orderdate BETWEEN '1995-01-01' AND '1996-12-31'
       AND p_type = 'PROMO BRUSHED STEEL'
) AS all_nations GROUP BY o_year ORDER BY o_year;
```
Join Optimizer: Case Study

MySQL Workbench: Visual EXPLAIN (MySQL 5.6)

Execution time: 21 seconds
Join Optimizer: Case study

Force early processing of high selectivity predicates

```sql
SELECT o_year, SUM(CASE WHEN nation = 'FRANCE' THEN volume ELSE 0 END) / SUM(volume) AS mkt_share
FROM (SELECT EXTRACT(YEAR FROM o_orderdate) AS o_year, l_extendedprice * (1 - l_discount) AS volume, n2.n_name AS nation
FROM part
STRAIGHT_JOIN lineitem ON p_partkey = l_partkey
JOIN supplier ON s_suppkey = l_suppkey
JOIN orders ON l_orderkey = o_orderkey
JOIN customer ON o_custkey = c_custkey
JOIN nation n1 ON c_nationkey = n1.n_nationkey
JOIN region ON n1.n_regionkey = r_regionkey
JOIN nation n2 ON s_nationkey = n2.n_nationkey
WHERE r_name = 'EUROPE' AND o_orderdate BETWEEN '1995-01-01' AND '1996-12-31' AND p_type = 'PROMO BRUSHED STEEL'
) AS all_nations GROUP BY o_year ORDER BY o_year;
```

part before lineitem

Highest selectivity
Join Optimizer: Case study

Improved join order

Execution time: 3 seconds
MySQL 5.7: Improved join order

Improvements to Query 8 in MySQL 5.7:

- Filtering on non-indexed columns are taken into account
  - No need for hint to force `part` table to be processed early
- Merge derived tables into outer query
  - No temporary table
Program Agenda

1. Cost-based query optimization in MySQL
2. Tools for monitoring, analyzing, and tuning queries
3. Data access and index selection
4. Join optimizer
5. Sorting
6. Influencing the optimizer
ORDER BY Optimizations

• General solution; “Filesort”:
  – Store query result in temporary table before sorting
  – If data volume is large, may need to sort in several passes with intermediate storage on disk.

• Optimizations:
  – Take advantage of index to generate query result in sorted order
  – For "LIMIT n" queries, maintain priority queue of n top items in memory instead of filesort. (MySQL 5.6)
### Filesort

SELECT * FROM orders ORDER BY o_totalprice;

<table>
<thead>
<tr>
<th>id</th>
<th>select type</th>
<th>table</th>
<th>type</th>
<th>possible keys</th>
<th>key</th>
<th>key len</th>
<th>ref</th>
<th>rows</th>
<th>extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>orders</td>
<td>ALL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>15000000</td>
<td>Using filesort</td>
</tr>
</tbody>
</table>

SELECT c_name, o_orderkey, o_totalprice
FROM orders JOIN customer ON c_custkey = o_custkey
WHERE c_acctbal < -1000 ORDER BY o_totalprice;

<table>
<thead>
<tr>
<th>id</th>
<th>select type</th>
<th>table</th>
<th>type</th>
<th>possible keys</th>
<th>key</th>
<th>key len</th>
<th>ref</th>
<th>rows</th>
<th>extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>customer</td>
<td>ALL</td>
<td>PRIMARY</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>1500000</td>
<td>Using where;</td>
</tr>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>orders</td>
<td>ref</td>
<td>i_o_custkey</td>
<td>i_o_custkey</td>
<td>5</td>
<td>...</td>
<td>7</td>
<td>Using temporary; Using filesort</td>
</tr>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>orders</td>
<td>ref</td>
<td>i_o_custkey</td>
<td>i_o_custkey</td>
<td>5</td>
<td>...</td>
<td>7</td>
<td>NULL</td>
</tr>
</tbody>
</table>
Filesort

Status variables

Status variables related to sorting:

```sql
mysql> show status like 'Sort%';
+-------------------+--------+
| Variable_name     | Value  |
+-------------------+--------+
| Sort_merge_passes | 1      |
| Sort_range        | 0      |
| Sort_rows         | 136170 |
| Sort_scan         | 1      |
+-------------------+--------+
```

>0: Intermediate storage on disk.
Consider increasing `sort_buffer_size`

Number of sort operations (range scan or table/index scans)

Number of rows sorted
Filesort

Performance Schema

Sorting status per statement available from Performance Schema

```sql
mysql> SELECT sql_text, sort_merge_passes, sort_range, sort_rows, sort_scan
FROM performance_schema.events_statements_history
ORDER BY timer_start DESC LIMIT 1;
```

```
+-----------------+-----------------+-----------------+-----------------+-----------------+
| sql_text        | sort_merge_passes | sort_range      | sort_rows       | sort_scan       |
+-----------------+-----------------+-----------------+-----------------+-----------------+
| SELECT ...      | 1               | 0               | 136170          | 1               |
+-----------------+-----------------+-----------------+-----------------+-----------------+
```
Filesor: Case Study

mysql> FLUSH STATUS;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT AVG(o_totalprice) FROM (SELECT * FROM orders ORDER BY o_totalprice DESC LIMIT 100000) td;
+-------------------+
| AVG(o_totalprice) |
+-------------------+
| 398185.986158     |
+-------------------+
1 row in set (24.65 sec)

Unnecessary large data volume!

mysql> SHOW STATUS LIKE 'sort%';
+-------------------+
| Variable_name     | Value  |
|-------------------+---------|
| Sort_merge_passes | 1432    |
| Sort_range        | 0       |
| Sort_rows         | 100000  |
| Sort_scan         | 1       |
+-------------------+
4 rows in set (0.00 sec)

Many intermediate sorting steps!
Filesor: Case Study

Reduce amount of data to be sorted

```
mysql> SELECT AVG(o_totalprice) FROM (SELECT o_totalprice FROM orders ORDER BY o_totalprice DESC LIMIT 100000) td;

+-------------------+
| AVG(o_totalprice) |
+-------------------+
| 398185.986158     |
+-------------------+
1 row in set (8.18 sec)

mysql> SELECT sql_text, sort_merge_passes FROM performance_schema.events_statements_history ORDER BY timer_start DESC LIMIT 1;

+-----------------+----------------+-----------------+
| sql_text        | sort_merge_passes |
+-----------------+-----------------+
| SELECT AVG(o_totalprice) FROM (SELECT o_totalprice FROM orders ORDER BY o_totalprice DESC LIMIT 100000) td; | 229 |
+-----------------+-----------------+
```
Filesort: Case Study

Increase sort buffer (1 MB)

mysql> SET sort_buffer_size = 1024*1024;

mysql> SELECT AVG(o_totalprice) FROM (SELECT o_totalprice FROM orders ORDER BY o_totalprice DESC LIMIT 100000) td;

+-------------------+
| AVG(o_totalprice) |
| 398185.986158     |
+-------------------+
1 row in set (7.24 sec)

mysql> SELECT sql_text, sort_merge_passes FROM performance_schema.events_statements_history ORDER BY timer_start DESC LIMIT 1;

+-----------------------------------------------+-------------------+
| sql_text                                      | sort_merge_passes |
| SELECT AVG(o_totalprice) FROM (SELECT o_totalprice |
| 57                                            |                   |
+-----------------------------------------------+-------------------+
Filesort: Case Study

Increase sort buffer even more (8 MB)

mysql> SET sort_buffer_size = 8*1024*1024;

mysql> SELECT AVG(o_totalprice) FROM (SELECT o_totalprice FROM orders ORDER BY o_totalprice DESC LIMIT 100000) td;

+-------------------+
| AVG(o_totalprice) |
+-------------------+
| 398185.986158     |
+-------------------+
1 row in set (6.30 sec)

mysql> SELECT sql_text, sort_merge_passes FROM performance_schema.events_statements_history ORDER BY timer_start DESC LIMIT 1;

+-------------------------------------------------------------------------------------------------+-------------------+
| sql_text                                                                                       | sort_merge_passes  |
+-------------------------------------------------------------------------------------------------+-------------------+
| SELECT AVG(o_totalprice) FROM (SELECT o_totalprice FROM orders ORDER BY o_totalprice DESC LIMIT 100000) td; | 0                 |
+-------------------------------------------------------------------------------------------------+
Using Index to Avoid Sorting

CREATE INDEX i_o_totalprice ON orders(o_totalprice);

SELECT o_orderkey, o_totalprice FROM orders ORDER BY o_totalprice ;

<table>
<thead>
<tr>
<th>id</th>
<th>select type</th>
<th>table</th>
<th>type</th>
<th>possible keys</th>
<th>key</th>
<th>key len</th>
<th>ref</th>
<th>rows</th>
<th>extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>orders</td>
<td>index</td>
<td>NULL</td>
<td>i_o_totalprice</td>
<td>6</td>
<td>NULL</td>
<td>15000000</td>
<td>Using index</td>
</tr>
</tbody>
</table>

However, still (due to total cost):

SELECT * FROM orders ORDER BY o_totalprice ;

<table>
<thead>
<tr>
<th>id</th>
<th>select type</th>
<th>table</th>
<th>type</th>
<th>possible keys</th>
<th>key</th>
<th>key len</th>
<th>ref</th>
<th>rows</th>
<th>extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIMPLEx</td>
<td>orders</td>
<td>ALL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>15000000</td>
<td>Using filesort</td>
</tr>
</tbody>
</table>
Using Index to Avoid Sorting

Case study revisited

```sql
SELECT AVG(o_totalprice) FROM
(SELECT o_totalprice FROM orders ORDER BY o_totalprice DESC LIMIT 100000) td;
```

<table>
<thead>
<tr>
<th>id</th>
<th>select type</th>
<th>table</th>
<th>Type</th>
<th>possible keys</th>
<th>key</th>
<th>key len</th>
<th>ref</th>
<th>rows</th>
<th>extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PRIMARY</td>
<td>&lt;derived2&gt;</td>
<td>ALL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>100000</td>
<td>NULL</td>
</tr>
<tr>
<td>2</td>
<td>DERIVED</td>
<td>orders</td>
<td>index</td>
<td>NULL</td>
<td>i_o_totalprice</td>
<td>6</td>
<td>NULL</td>
<td>15000000</td>
<td>Using index</td>
</tr>
</tbody>
</table>

mysql> SELECT AVG(o_totalprice) FROM (SELECT o_totalprice FROM orders ORDER BY o_totalprice DESC LIMIT 100000) td;

... 
1 row in set (0.06 sec)
Program Agenda

1. Cost-based query optimization in MySQL
2. Tools for monitoring, analyzing, and tuning queries
3. Data access and index selection
4. Join optimizer
5. Sorting
6. Influencing the optimizer
Influencing the Optimizer

When the optimizer does not do what you want

• Add indexes

• Force use of specific indexes:
  – USE INDEX, FORCE INDEX, IGNORE INDEX

• Force specific join order:
  – STRAIGHT_JOIN

• Adjust session variables
  – optimizer_switch flags: set optimizer_switch="index_merge=off"
  – Buffer sizes: set sort_buffer=8*1024*1024;
  – Other variables: set optimizer_search_depth = 10;
MySQL 5.7: New Optimizer Hints

• Ny hint syntax:
  – SELECT /*+ HINT1(args) HINT2(args) */ ... FROM ...

• New hints:
  – BKA(tables)/NO_BKA(tables), BNL(tables)/NO_BNL(tables)
  – MRR(table indexes)/NO_MRR(table indexes)
  – SEMIJOIN/NO_SEMIJOIN(strategies), SUBQUERY(strategy)
  – NO_ICP(table indexes)
  – NO_RANGE_OPTIMIZATION(table indexes)
  – QB_NAME(name)

• Finer granularity than optimizer_switch session variable
MySQL 5.7: Hint Example: SEMIJOIN

• No hint, optimizer chooses semi-join algorithm LooseScan:

```sql
EXPLAIN SELECT * FROM t2 WHERE t2.a IN (SELECT a FROM t3);
```

<table>
<thead>
<tr>
<th>id</th>
<th>select type</th>
<th>table</th>
<th>type</th>
<th>possible keys</th>
<th>key</th>
<th>key len</th>
<th>ref</th>
<th>rows</th>
<th>extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>t3</td>
<td>index</td>
<td>a</td>
<td>a</td>
<td>4</td>
<td>NULL</td>
<td>3</td>
<td>Using where; LooseScan</td>
</tr>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>t2</td>
<td>ref</td>
<td>a</td>
<td>a</td>
<td>4</td>
<td>test.t3.a</td>
<td>1</td>
<td>Using index</td>
</tr>
</tbody>
</table>

• Disable semi-join with hint:

```sql
EXPLAIN SELECT * FROM t2 WHERE t2.a IN (SELECT /*+ NO_SEMIJOIN() */ a FROM t3);
```

<table>
<thead>
<tr>
<th>id</th>
<th>select type</th>
<th>table</th>
<th>type</th>
<th>possible keys</th>
<th>key</th>
<th>key len</th>
<th>ref</th>
<th>rows</th>
<th>extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PRIMARY</td>
<td>t2</td>
<td>index</td>
<td>null</td>
<td>a</td>
<td>4</td>
<td>NULL</td>
<td>4</td>
<td>Using where; Using index</td>
</tr>
<tr>
<td>2</td>
<td>DEPENDENT_SUBQUERY</td>
<td>t3</td>
<td>Index_subquery</td>
<td>a</td>
<td>a</td>
<td>4</td>
<td>func</td>
<td>1</td>
<td>Using index</td>
</tr>
</tbody>
</table>
MySQL 5.7: Hint Example: SEMIJOIN

• Force Semi-join Materialization to be used

```sql
EXPLAIN SELECT /*+ SEMIJOIN(@subq MATERIALIZATION) */ * FROM t2
WHERE t2.a IN (SELECT /*+ QB_NAME(subq) */ a FROM t3);
```

<table>
<thead>
<tr>
<th>id</th>
<th>select type</th>
<th>table</th>
<th>type</th>
<th>possible keys</th>
<th>key</th>
<th>key len</th>
<th>ref</th>
<th>rows</th>
<th>extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>t2</td>
<td>index</td>
<td>a</td>
<td>a</td>
<td>4</td>
<td>NULL</td>
<td>4</td>
<td>Using where; Using index</td>
</tr>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>&lt;subquery2&gt;</td>
<td>eq_ref</td>
<td>&lt;auto_key&gt;</td>
<td>&lt;auto_key&gt;</td>
<td>4</td>
<td>test.t2.a</td>
<td>1</td>
<td>NULL</td>
</tr>
<tr>
<td>2</td>
<td>MATERIALIZED</td>
<td>t3</td>
<td>index</td>
<td>a</td>
<td>a</td>
<td>4</td>
<td>NULL</td>
<td>3</td>
<td>Using index</td>
</tr>
</tbody>
</table>
Optimizer Hints

Future

• New hints in 8.0
  – Enable/disable merge of views and derived tables:
    • MERGE(derived_table) NO_MERGE(derived_table)
  – Join order
    • JOIN_ORDER(tables) JOIN_PREFIX(tables) JOIN_SUFFIX(tables) JOIN_FIXED_ORDER()
  – Force/ignore index_merge alternatives
    • INDEX_MERGE(table indexes) NO_INDEX_MERGE(table indexes)

• Hints we consider to add
  – Reimplement index hints using the new syntax
  – Temporarily set session variables for the duration of the query
MySQL 5.7: Query Rewrite Plugin

• Rewrite problematic queries without the need to make application changes
  – Add hints
  – Modify join order
  – Much more ...

• Add rewrite rules to table:
  
  INSERT INTO query_rewrite.rewrite_rules (pattern, replacement ) VALUES
  ("SELECT * FROM t1 WHERE a > ? AND b = ?",
   "SELECT * FROM t1 FORCE INDEX (a_idx) WHERE a > ? AND b = ?");

• New pre- and post-parse query rewrite APIs
  – Users can write their own plug-ins
MySQL 5.7: Adjustable Cost Constants

Use with caution!

EXPLAIN SELECT SUM(o_totalprice) FROM orders
WHERE o_orderdate BETWEEN '1994-01-01' AND '1994-12-31';

<table>
<thead>
<tr>
<th>id</th>
<th>select type</th>
<th>table</th>
<th>type</th>
<th>possible keys</th>
<th>key</th>
<th>key len</th>
<th>rows</th>
<th>filtered</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>orders</td>
<td>ALL</td>
<td>i_o_orderdate</td>
<td>NULL</td>
<td>NULL</td>
<td>15000000</td>
<td>29.93</td>
<td>Using where</td>
</tr>
</tbody>
</table>

UPDATE mysql.engine_cost SET cost_value=0.25 Default: 1.0
WHERE cost_name='memory_block_read_cost';

FLUSH OPTIMIZER_COSTS; Make server read new cost constants
### MySQL 5.7: Adjustable Cost Constants

**Improved plan**

```sql
EXPLAIN SELECT SUM(o_totalprice) FROM orders
WHERE o_orderdate BETWEEN '1994-01-01' AND '1994-12-31';
```

<table>
<thead>
<tr>
<th>Id</th>
<th>select type</th>
<th>table</th>
<th>type</th>
<th>possible keys</th>
<th>key</th>
<th>key len</th>
<th>rows</th>
<th>filtered</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>orders</td>
<td>range</td>
<td>i_o_orderdate</td>
<td>i_o_orderdate</td>
<td>4</td>
<td>4489990</td>
<td>100.00</td>
<td>Using index condition</td>
</tr>
</tbody>
</table>

**Note:**

- Heuristic used: If table is smaller than 20% of database buffer, all pages are in memory
- Only new connections will see updated cost constants
MySQL 8.0: Disk vs memory access

• New defaults for const constants:

<table>
<thead>
<tr>
<th>Cost</th>
<th>MySQL 5.7</th>
<th>MySQL 8.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read a random disk page</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Read a data page from memory buffer</td>
<td>1.0</td>
<td>0.25</td>
</tr>
<tr>
<td>Evaluate query condition</td>
<td>0.2</td>
<td>0.1</td>
</tr>
<tr>
<td>Compare keys/records</td>
<td>0.1</td>
<td>0.05</td>
</tr>
</tbody>
</table>

• InnoDB reports for each table/index percentage of data cached in buffer pool

• Note: Query plan may change between executions
More information

• MySQL Webinar:

• My blog:
  – http://oysteing.blogspot.com/

• MySQL Server Team blog
  – http://mysqlserverteam.com/

• MySQL forums:
Q+A