MySQL 5.7: Performance Improvements in Optimizer

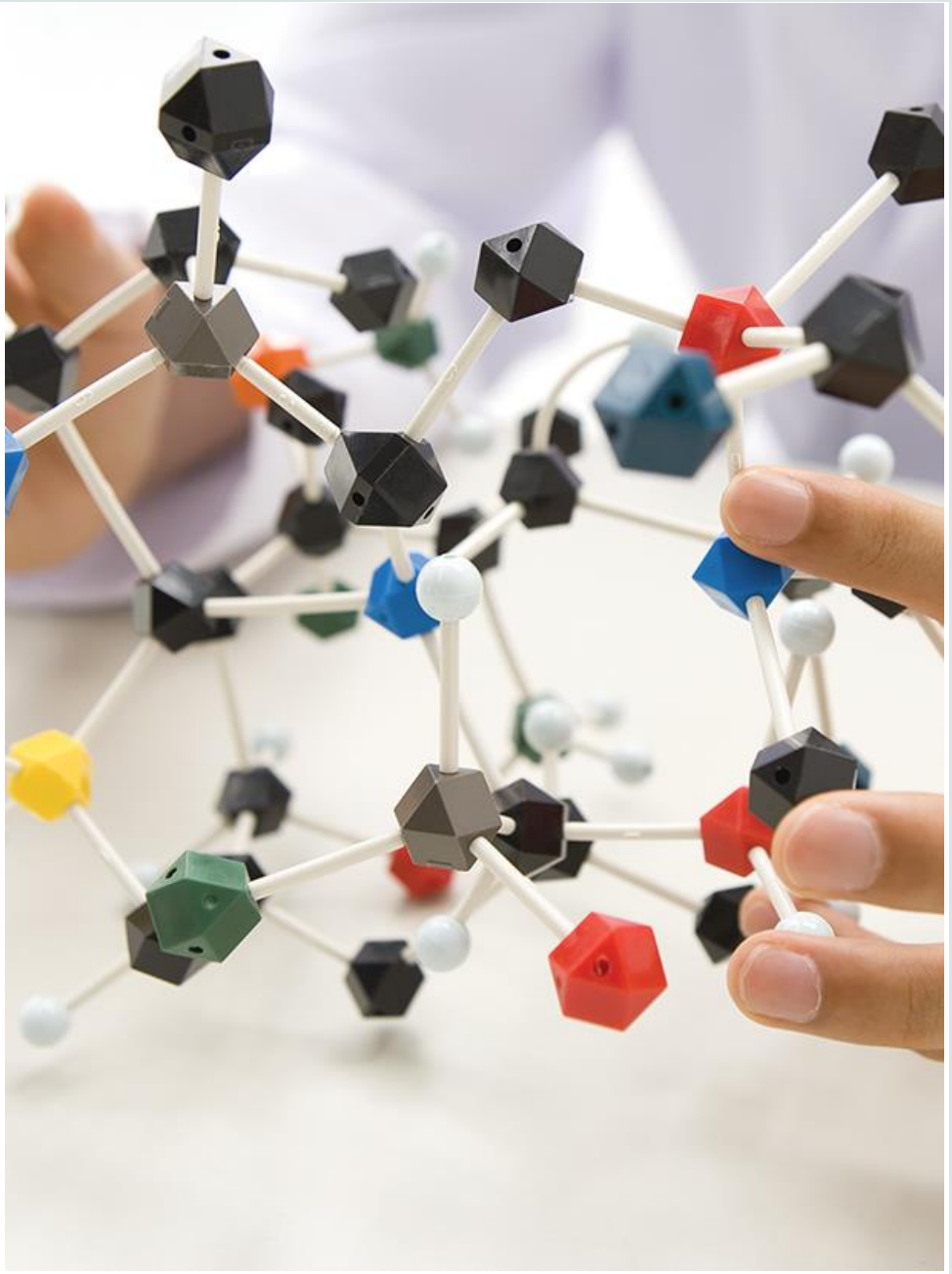
Olav Sandstå
Senior Principal Engineer
MySQL Optimizer Team, Oracle
April 25, 2016

Safe Harbor Statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

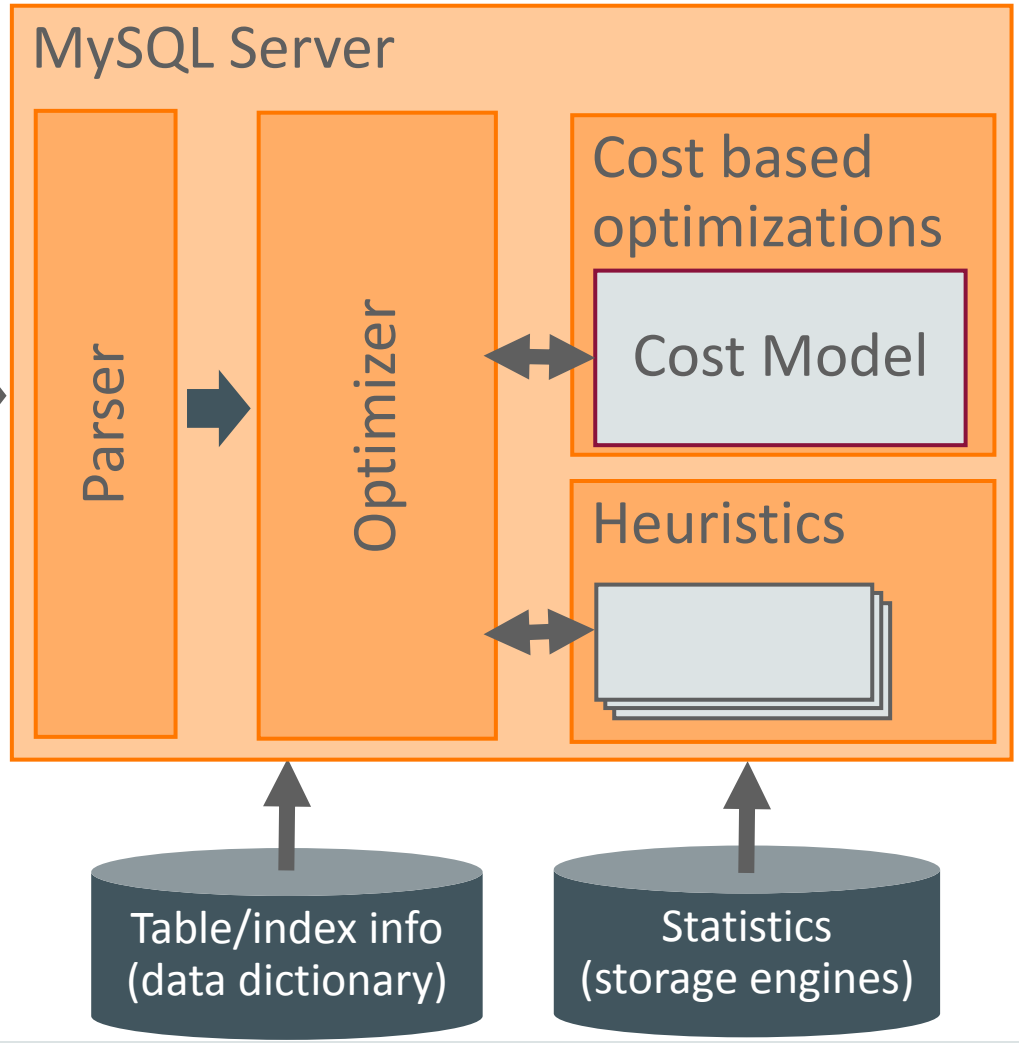
Program Agenda

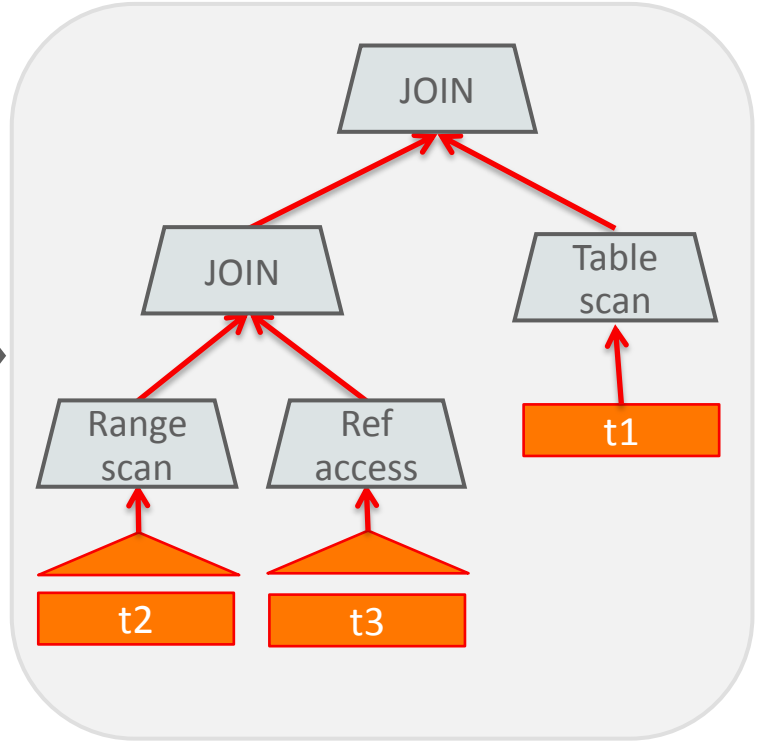
1. Improvements in optimizer
   - Cost model
   - New optimizations
2. Understanding query performance
   - Explain extensions
   - Optimizer trace
3. Tools for improving query plans
   - New hints
   - Query rewrite plugin



MySQL Optimizer

```
SELECT a, b
FROM t1
  JOIN t2
    ON t1.a = t2.b
  JOIN t3
    ON t2.b = t3.c
WHERE t2.d > 20
  AND t2.d < 30;
```





Optimizer Performance Improvements in MySQL 5.7

Cost model improvements:

- Cost model for WHERE conditions (condition filtering effect)
  - Improved JOIN order
- Improved index statistics
  - Better index selection, better join order
- Configurable “cost constants”

New optimizations:

- Merging of derived tables
- Optimization of IN queries
- Union ALL optimization

Condition Filtering Effect

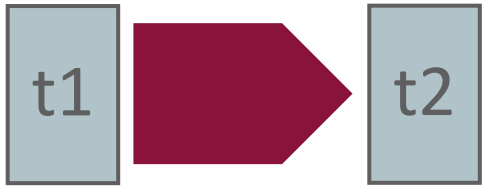
Cost Model for Query Conditions

Goal: Low-fanout tables should be early in the join order

```
SELECT office_name
FROM office JOIN employee ON office.id = employee.office
WHERE employee.name = "John" AND age > 21 AND
      hire_date BETWEEN "2014-01-01" AND "2014-06-01";
```

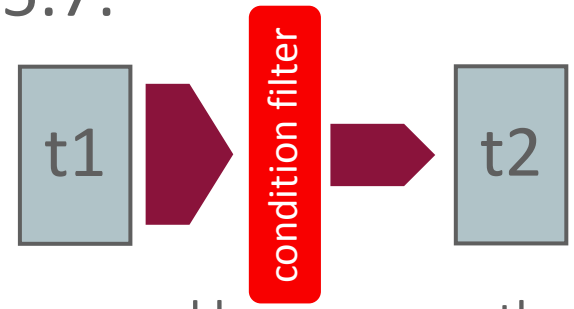
In 5.6 we do not consider the entire WHERE condition when calculating fanout

MySQL 5.6:



Fanout= #rows read by access metod

MySQL 5.7:



Fanout= #rows read by access method * condition filter effect

How to Calculate Condition Filter Effect

```
SELECT office_name
FROM office JOIN employee ON office.id = employee.office
WHERE employee.name = "John" AND age > 21 AND
      hire_date BETWEEN "2014-01-01" AND "2014-06-01";
```

0.1 (guesstimate)

0.89 (range)

0.11 (guesstimate)

Filter estimate based on what is available:

1. Range estimate
2. Index statistics
3. Guesstimate

=	0.1
<=,<,>,>=	1/3
BETWEEN	1/9
NOT <op>	1 – SEL(<op>)
AND	P(A and B) = P(A) * P(B)
OR	P(A or B) = P(A) + P(B) – P(A and B)
…	…

Example: Two Table JOIN in MySQL 5.7

```
SELECT office_name
FROM office JOIN employee ON office.id = employee.office
WHERE employee.name = "John" AND age > 21 AND
      hire_date BETWEEN "2014-01-01" AND "2014-06-01";
```

Explain for 5.6:

Table	Type	Possible keys	Key	Ref	Rows	Filtered	Extra
office	ALL	PRIMARY	NULL	NULL	100	100.00	NULL
employee	ref	office	office	office.id	99	100.00	Using where

Explain for 5.7:

Table	Type	Possible keys	Key	Ref	Rows	Filtered	Extra
employee	ALL	NULL	NULL	NULL	9991	1.23	NULL
office	eq_ref	PRIMARY	PRIMARY	employee.office	1	100.00	Using where

Condition filter estimate

JOIN ORDER HAS CHANGED!

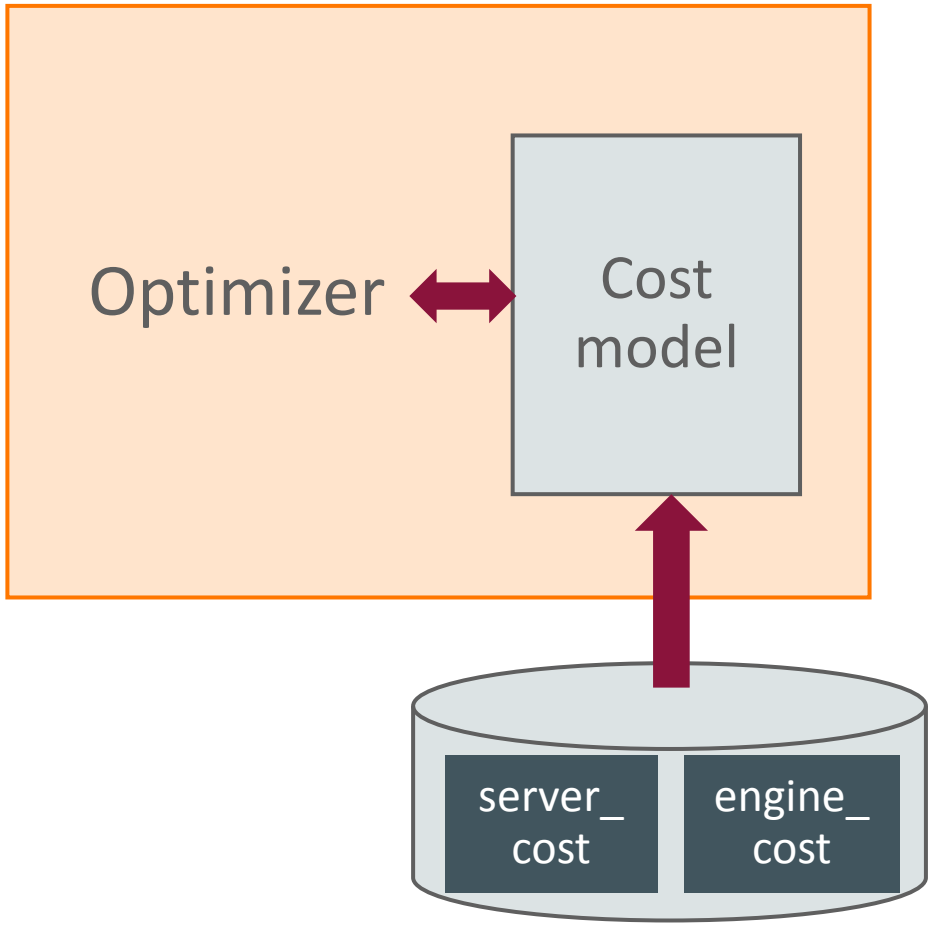
Disable Condition Filtering

- In case of performance regressions:

```
SET optimizer_switch=`condition_fanout_filter=OFF`;
```

Configurable Cost Model

- Replaced hard-coded “cost constants” with configurable “cost constants”
- Stored in “mysql” database:
  - server_cost
  - engine_cost
- “Cost constants” are changed by updating these tables



Configurable Cost Constants

Name	Default value
row_evalute_cost	0.2
key_compare_cost	0.1
memory_temptable_create_cost	2.0
memory_temptable_row_cost	0.2
disk_temptable_create_cost	40.0
disk_temptable_row_cost	1.0
memory_block_read_cost	1.0
io_block_read_cost	1.0

Online update of cost constants:

```
UPDATE mysql.server_cost
SET cost_value=0.1
WHERE cost_name="row_evaluate_cost";

FLUSH OPTIMIZER_COSTS;
```

Tip 1:
All data fits in InnoDB buffer, set:
memory_block_read_cost = 0.5
io_block_read_cost = 0.5

Tip 2:
Working set larger than InnoDB buffer, set:
memory_block_read_cost = 0.5

Merging Derived Tables into Outer Query

```
SELECT * FROM (SELECT * FROM t1 WHERE  .....) AS derived
WHERE .....
```

MySQL 5.6:

- Derived table always materialized in temporary table

MySQL 5.7:

- Merged into outer query or materialized
- Derived table optimized as part of outer query:
  - Faster queries
- Derived tables and views are now optimized the same way

Avoid Creating Temporary Table for UNION ALL


```
SELECT * FROM table_a UNION ALL SELECT * FROM table_b;
```

MySQL 5.6:

- Always materialize results of UNION ALL in temporary tables

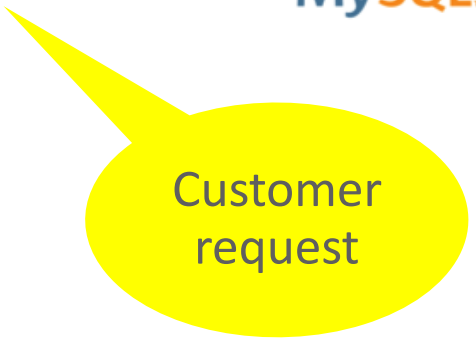
MySQL 5.7:

- Do not materialize in temporary tables unless used for sorting, rows are sent directly to client
- Client will receive the first row faster, no need to wait until the last query block is finished
- Less memory and disk consumption

Optimizations for IN Expressions

```
CREATE TABLE t1 (a INT, b INT, c INT, INDEX idx (a, b));

SELECT a, b FROM t1 WHERE (a, b) IN ((0, 0), (1, 1));
```



MySQL 5.6:

- IN queries with row value expressions can not use index scans or range scans even though all the columns in the query are indexed
- Need to rewrite to de-normalized form:

```
SELECT a, b FROM t1 WHERE ( a = 0 AND b = 0 ) OR ( a = 1 AND b = 1 )
```

MySQL 5.7:

- IN queries with row value expressions executed using range scans

Optimizations for IN Expressions

```
SELECT a, b FROM t1 WHERE (a, b) IN ((0, 0), (1, 1));
```

The table has 10 000 rows, 2 match the where condition

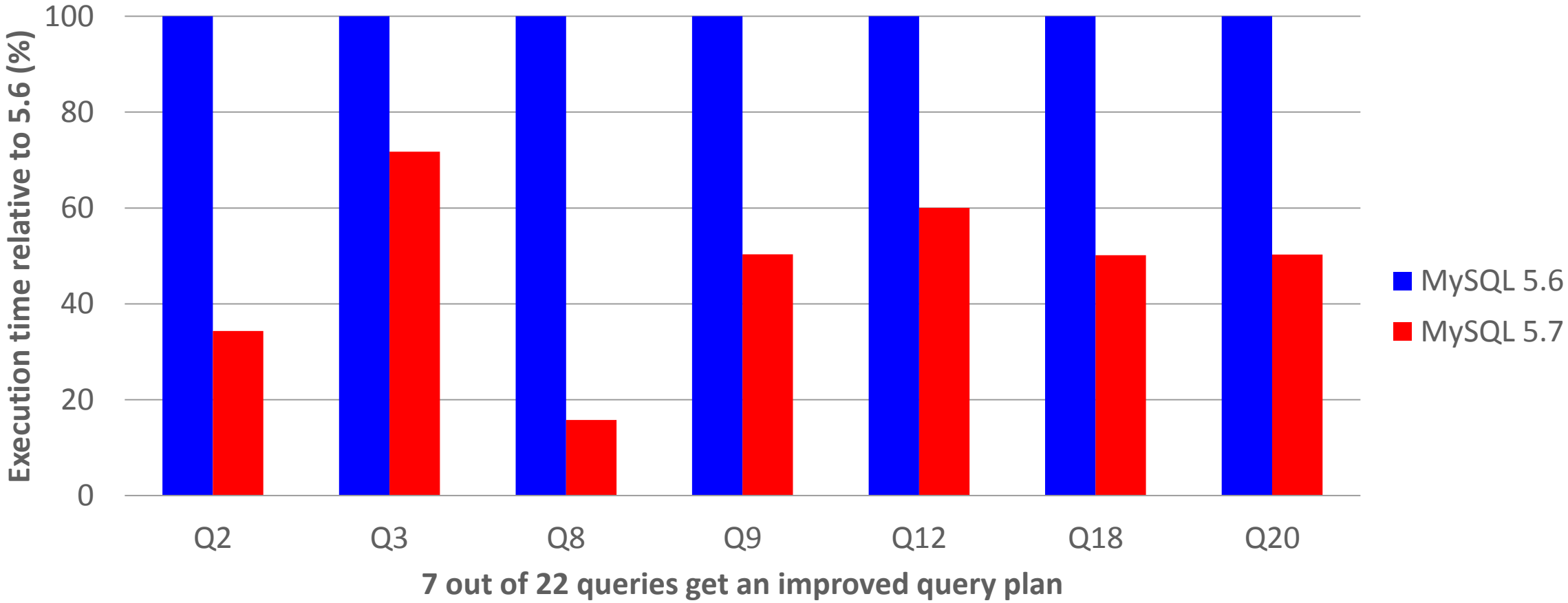
MySQL 5.6:

Table	Type	Possible keys	Key	Key_len	Ref	Rows	Filtered	Extra
t1	index	idx	idx	10	NULL	10000	100.00	Using where; Using index

MySQL 5.7:

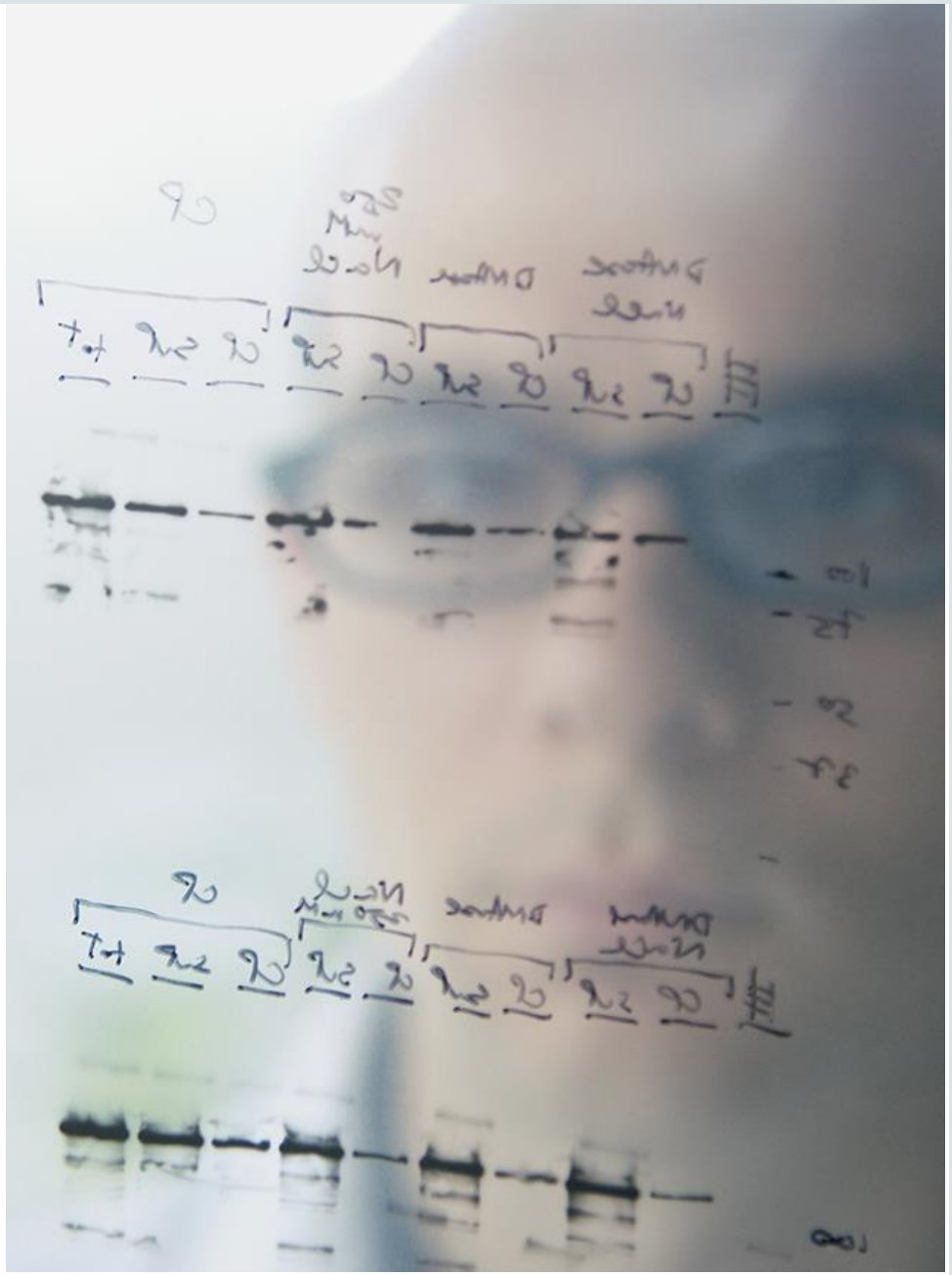
Table	Type	Possible keys	Key	Key_len	Ref	Rows	Filtered	Extra
t1	range	idx	idx	10	NULL	2	100.00	Using where; Using index

Performance improvements: DBT-3 (SF10, CPU bound)



Program Agenda

1. Improvements in optimizer
   - Cost model
   - New optimizations
2. Understanding query performance
   - Explain extensions
   - Optimizer trace
3. Tools for improving query plans
   - New hints
   - Query rewrite plugin

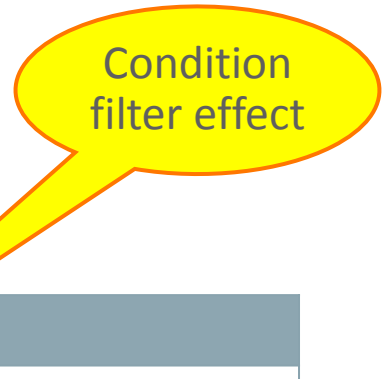


Understanding the Query Plan

EXPLAIN

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;
```



id	type	table	type	possible keys	key	key len	ref	rows	filtered	Extra
1	SIMPLE	t1	range	PRIMARY,idx1	idx1	4	NULL	12	33.33	Using index condition
2	SIMPLE	t2	ref	idx2	idx2	4	t1.a	1	100.00	NULL

Explain on a Running Query

EXPLAIN [FORMAT=(JSON|TRADITIONAL)] FOR CONNECTION <id>;

New in MySQL 5.7

- Shows query plan on connection <id>
- Useful for diagnostic on long running queries
- Plan isn't available when query plan is under creation
- Applicable to SELECT/INSERT/DELETE/UPDATE

Understanding the Query Plan

Structured EXPLAIN

- JSON format:

EXPLAIN FORMAT=JSON SELECT …

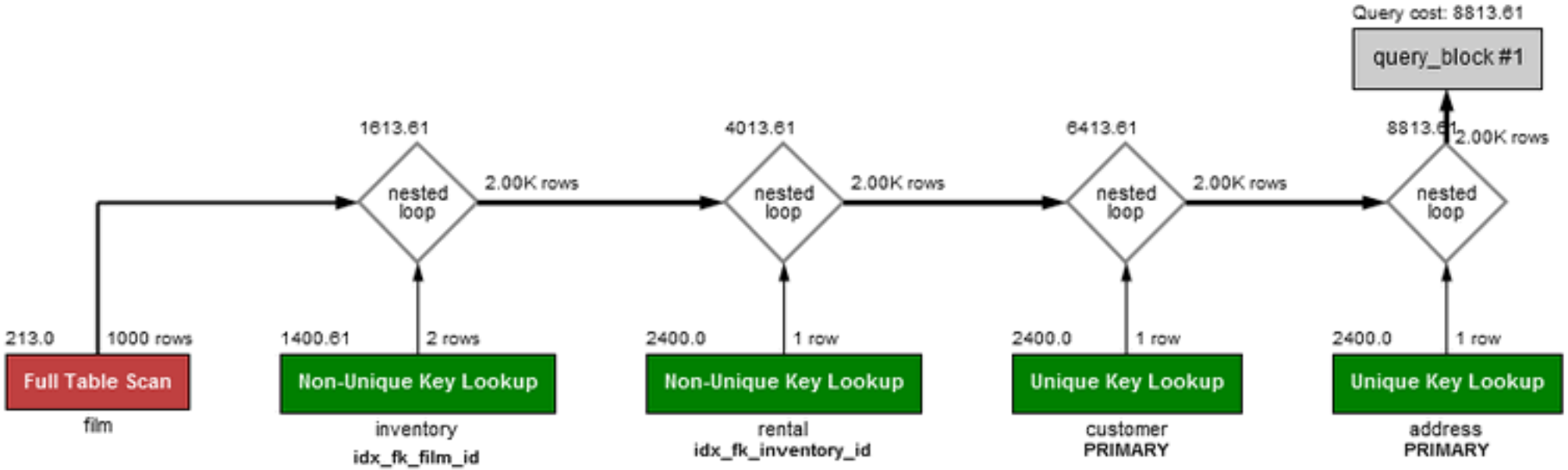
- Contains more information:
  - Used index parts
  - Pushed index conditions
  - Cost estimates
  - Data estimates

Added in MySQL 5.7

```
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",
        "data_read_per_join": "63"
      },
      ……..
      "attached_condition": "(`test`.`t1`.`c` > 10)"
    }
  }
}
```

Understanding the Query Plan

Visual Explain in MySQL Work Bench

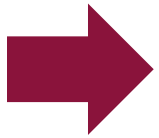


Optimizer Trace

Understand HOW a query is optimized

- Trace of the main steps and decisions done by the optimizer

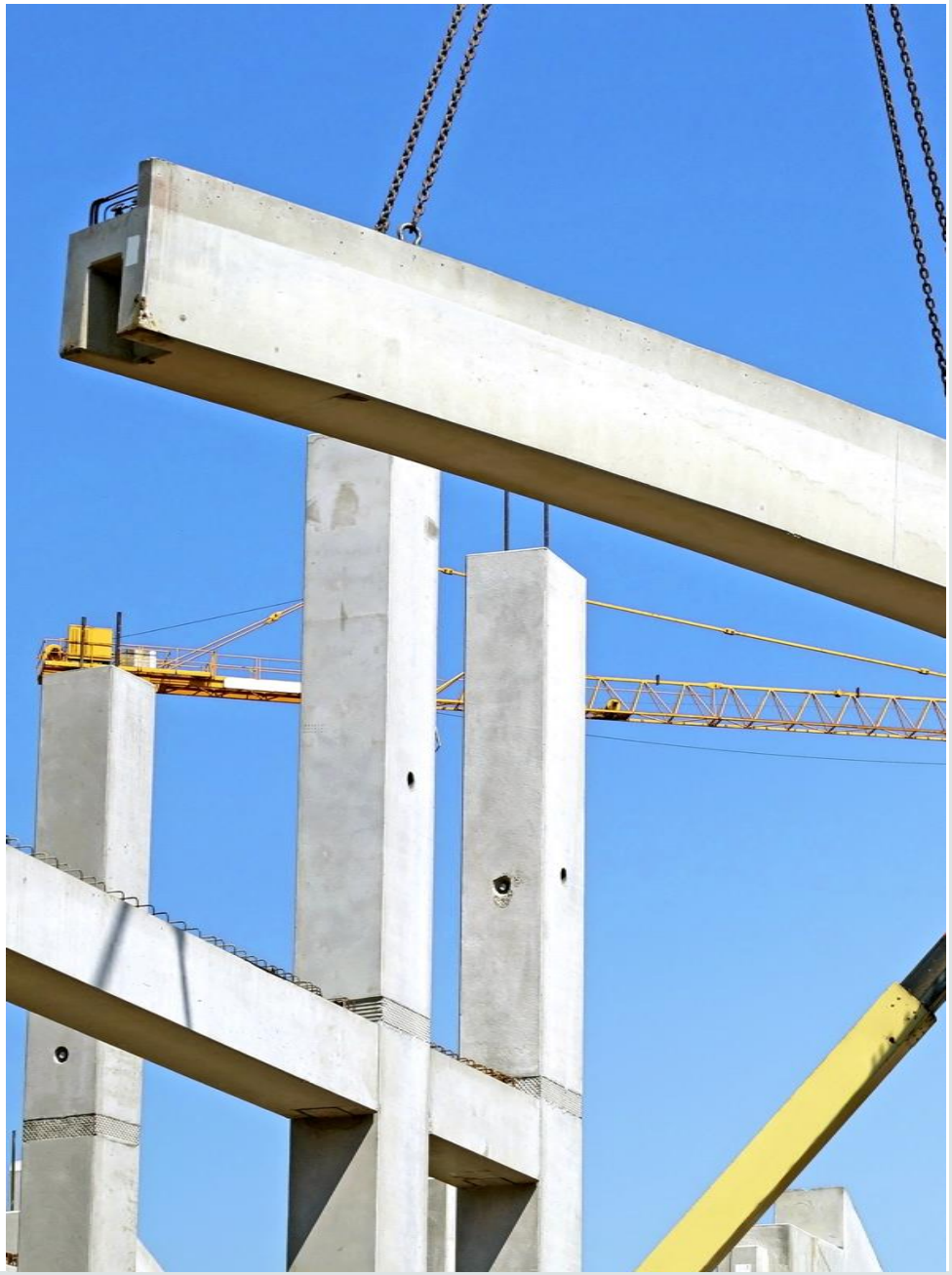
```
SET optimizer_trace=”enabled=on”;
SELECT * FROM t1 WHERE a > 10;
SELECT * FROM
  INFORMATION_SCHEMA.OPTIMIZER_TRACE;
```



```
"table": "`t1`",
"range_analysis": {
   "table_scan": {
     "rows": 54,
     "cost": 13.9
   },
   "best_covering_index_scan": {
     "index": ”idx",
     "cost": 11.903,
     "chosen": true
   },
   "analyzing_range_alternatives": {
     "range_scan_alternatives": [
     {
       "index": ”idx",
       "ranges": [
         "10 < a"
       ],
       "rowid_ordered": false,
       "using_mrr": false,
       "index_only": true,
       "rows": 12,
       "cost": 3.4314,
       "chosen": true
     }
```

Program Agenda

1. Improvements in optimizer
   - Cost model
   - New optimizations
2. Understanding query performance
   - Explain extensions
   - Optimizer trace
3. Tools for improving query plans
   - New hints
   - Query rewrite plugin



Influencing the Optimizer

When the optimizer does not do what you want:

- Add indexes
- Use hints:
  - Index hints: USE INDEX, FORCE INDEX, IGNORE INDEX
  - Join order: STRAIGHT_JOIN
  - Subquery strategy: /*+ SEMIJOIN(FirstMatch) */
  - Join buffer strategy: /*+ BKA(table1) */
- Adjust optimizer_switch flags:
  - set optimizer_switch="condition_fanout_filter=OFF"

New hint syntax and new hints in MySQL 5.7

Improved HINT support

- Introduced new hint syntax: /*+ . . . */
- Examples for new hints:
  - Join buffer strategy (BNL/NO_BNL, BKA/NO_BKA)
  - Semijoin and subquery strategy (SEMIJOIN/NO_SEMIJOIN, SUBQUERY)
  - Multi-range read optimization (MRR/NO_MRR)
  - Max execution time (MAX_EXECUTION_TIME)
- Hints apply at different scope levels: global, query block, table, index
- Most hints are in two forms:
  - Enabling means optimizer should try to use it, but not forced to use it (eg. BKA)
  - Disabling prevents optimizer from using it (eg. NO_BKA)

Hint Example: MAX_EXECUTION_TIME

```
SELECT /*+ MAX_EXECUTION_TIME(1) */ * FROM t1 a, t1 b, t1 c, t1 d, t1 e LIMIT 1;
ERROR 3024 (HY000): Query execution was interrupted, maximum statement execution time exceeded
```

```
SELECT /*+ MAX_EXECUTION_TIME(1000) */ * FROM t1 a, t1 b, t1 c, t1 d, t1 e LIMIT 1;
+---+----+---+----+---+----+---+-- -+---+----+
| a | b  | a | b  | a | b  | a | b  | a |  b |
+---+----+---+----+---+----+---+----+---+----+
| 1 | 10 | 1 | 10 | 1 | 10 | 1 | 10 | 1 | 10 |
+---+----+---+----+---+----+---+----+---+----+
1 row in set (0,00 sec)
```

Hint Example: SEMIJOIN

```
EXPLAIN SELECT * FROM t1 WHERE t1.a IN (SELECT a FROM t2);
```

No hint, optimizer chooses semi-join algorithm loosescan:

id	Select_type	Table	Type	Possible_keys	Key	Key_len	Ref	Rows	Extra
1	simple	t2	index	a	a	4	null	3	Using where; LooseScan
1	simple	t1	ref	a	a	4	test.t2.a	1	Using index

```
EXPLAIN SELECT * FROM t1 WHERE t1.a IN (SELECT /*+ NO_SEMIJOIN() */ a FROM t2);
```

Semi-join disabled with hint, subquery is executed for each row of outer table:

id	Select_type	Table	Type	Possible_keys	Key	Key_len	Ref	Rows	Extra
1	primary	t1	index	null	a	4	null	4	Using where; Using index
2	dependent subquery	t2	index_ subquery	a	a	4	func	1	Using index

Hint Example: SEMIJOIN

```
EXPLAIN SELECT /*+ SEMIJOIN(@subq MATERIALIZATION) */ * FROM t1
WHERE t1.a IN
    (SELECT  /*+ QB_NAME(subq) */ a FROM t2);
```

Hint on a particular algorithm, in this case semi-join materialization

id	Select_type	Table	Type	Possible_keys	Key	Key_len	Ref	Rows	Extra
1	simple	t1	index	a	a	4	null	4	Using where; Using index
1	simple	<subquery2>	eq_ref	<auto_key>	<auto_key>	4	test.t1.a	1	null
2	materialized	t2	index	a	a	4	null	3	Using index

Query Rewrite Plugin

- Problem:
  - Optimizer choses a suboptimal query plan
  - User can change the query plan by adding hints or rewrite the query
  - However, database application code can not be changed
- Solution:
  - Query rewrite plugin
- Rewrite problematic queries without having to change application code
  - Add hints
  - Modify join order
- Rewrite rules are defined in a database table

How Rewrite Plugin works

Problematic query:

```
SELECT *
FROM t1 JOIN t2
ON t1.keycol = t2.keycol
WHERE col1 = 42 AND col2 = 2;
```

Wrong join order
Wrong index

Rewritten query:

```
SELECT *
FROM t2 STRAIGHT_JOIN t1
FORCE INDEX (col1)
ON t1.keycol = t2.keycol
WHERE col1 = 42 AND col2 = 2;
```

Query rewrite plugin

Pattern for matching:

```
SELECT *
FROM t1 JOIN t2
ON t1.keycol = t2.keycol
WHERE col1 = ? AND col2 =?;
```

Replacement rule:

```
SELECT *
FROM t2 STRAIGHT_JOIN t1
FORCE INDEX (col1)
ON t1.keycol = t2.keycol
WHERE col1 = ? AND col2 =?;
```

How to use Query Rewrite Plugin

1. Install query rewrite plugin:

   mysql –u root –p < install_rewriter.sql

2. Insert pattern and replacement rule into query_rewrite.rewrite_rules table:

pattern	replacement	enabled
SELECT * FROM t1 JOIN t2 ON t1.keycol = t2.keycol WHERE col1 = ? AND col2 =?;	SELECT * FROM t2 STRAIGHT_JOIN t1 FORCE INDEX (col1) ON t1.keycol = t2.keycol WHERE col1 = ? AND col2 =?;	Y

3. Reload the new rules into the plugin:

   mysql> CALL query_rewrite.flush_rewrite_rules();

Query Rewrite Plugin

Performance impact:

- Designed for rewriting problematic queries only!
- ~ Zero cost for queries not to be rewritten
  - Statement digest computed for performance schema anyway
- Cost of queries to be rewritten is insignificant compared to performance gain

Benefits:

- Queries can be rewritten without having to change application code
- Easy to test out alternative rewrites of queries
- Easy to temporarily disable rewrite rules to check if the rewrite still is needed

What is on the Optimizer Roadmap?

- Common table expressions (WITH RECURSIVE)
- Window functions
- Improved prepared statement support / Query plan caching
- Cost model:
  - better support for different hardware: data in memory and SSD
- Statistics:
  - Histograms