MySQL 8.0 What's New in the Optimizer

Manyi Lu

Director
MySQL Optimizer & GIS Team, Oracle
October 2016

ORACLE

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.

New Features in 8.0.0

- **Invisible index**
- Descending index
- Common table expressions
- Improved performance of scans
- JSON aggregation
- Hints
- Better IPv6 and UUID support

Invisible Index

```
ALTER TABLE t1 ALTER INDEX idx INVISIBLE;

mysql> SHOW INDEXES FROM t1;
+--------+----------------+-------------------------------------+
| Table  |  Key_name      |  Column_name  | Visible             |
+--------+----------------+---------------------+---------------+
| t1     |  idx           |   a                 |  NO           |
+--------+----------------- +--------------------+--------------+
```

- Maintained by the SE, ignored by the Optimizer
- Primary key cannot be INVISIBLE
- Use case: Check for performance drop BEFORE dropping an index

New Features in 8.0

- Invisible index
- **Descending index**
- Common table expressions
- Improved performance of scans
- JSON aggregation
- Hints
- Better IPv6 and UUID support

Descending Index (lab release 8.0.0 Optimizer)

```
CREATE TABLE t1 (
      a INT,
      b INT,
      INDEX a_b (a DESC, b ASC)
 );
```

- In 5.7: Index in ascending order is created, server scans it backwards
- In 8.0: Index in descending order is created, server scans it forwards
- Works on btree index only

Benefits:

- Forward index scan is faster than backward index scan
- Use indexes instead of filesort for ORDER BY clause with ASC/DESC sort key

Download from http://labs.mysql.com/

New Features in 8.0

- Invisible index
- Descending index
- **Common table expressions**
- Improved performance of scans
- JSON aggregation
- Hints
- Better UUID and IPv6support

Common Table Expression (8.0.0 optimizer lab release)

```
SELECT … FROM (subquery) AS derived, t1...

WITH derived AS subquery SELECT … FROM derived, t1...;
```

- A derived table is a subquery in the FROM clause
- CTE is just a derived table, but you put its declaration before the outer SELECT instead of in FROM clause.

Common Table Expression VS Derived Table

Advantages over derived table

- Better readability
  - Derived table requires inside out reading
  - SELECT … FROM

    (SELECT … FROM (SELECT … FROM # understand this, go up
- Can be referenced more than once
  - Derived table can't be referenced twice in

    FROM SELECT …

    FROM (subquery) AS derived, derived

Common Table Expression VS Derived Table

Advantages over Derived table

- Better performance
  - SELECT … FROM (subquery) AS derived, (same subquery again) AS derived1
  - Two independent derived tables: if materialization is used, two materializations are done. Performance problem (more space, more time, longer locks)
- Chainable
  - Derived table doesn’t allow chaining
  - SELECT … FROM (subquery) AS derived, (SELECT … FROM derived,...) AS derived1
    Error : Derived table can't refer to previously created derived table

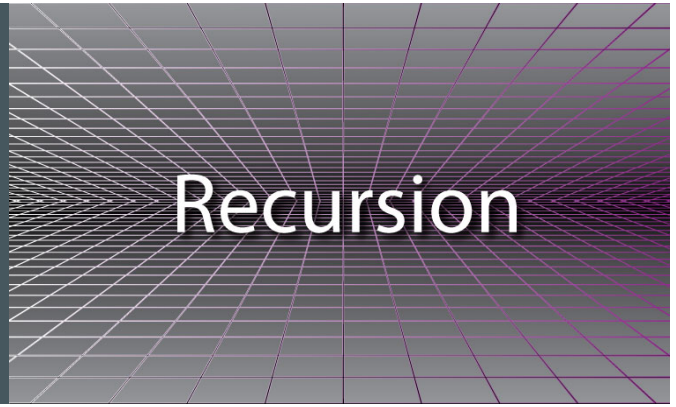
Common Table Expression VS Derived Table

```
WITH cte1(txt)  AS  (SELECT "This "),
        cte2(txt)  AS  (SELECT CONCAT(cte1.txt,"is a ") FROM cte1),
        cte3(txt)  AS  (SELECT "nice query" UNION
                        SELECT "query that rocks" UNION
                        SELECT "query"),
        cte4(txt)  AS  (SELECT concat(cte2.txt, cte3.txt) FROM cte2, cte3)
SELECT MAX(txt), MIN(txt) FROM cte4;
```

```
+----------------------------------+--------------------------+
| MAX(txt)                         | MIN(txt)                 |
+----------------------------------+--------------------------+
| This is a query that rocks       | This is a nice query     |
+----------------------------------+--------------------------+
```

Recursive CTE

```
WITH RECURSIVE cte AS
( SELECT ... FROM table_name /* "seed" SELECT */
   UNION ALL
   SELECT ... FROM cte, table_name) /* "recursive"  SELECT */
SELECT ... FROM cte;
```



- A recursive CTE refers to itself in a subquery
- The “seed” SELECT is executed once to create the initial data subset, the recursive SELECT is repeatedly executed to return subsets of data until the complete result set is obtained.
- Useful to dig in hierarchies (parent/child, part/subpart)
- Similar to Oracle's CONNECT BY

Recursive CTE – Simple Example

Print 1 to 10 :

```
WITH RECURSIVE qn AS
  ( SELECT 1 AS a
    UNION ALL
    SELECT 1+a FROM qn WHERE a<10
  )
SELECT * FROM qn;
```

```
a
1
2
3
4
5
6
7
8
9
10
```

Recursive CTE – Example Fibonacci Numbers

Fibonacci numbers:

```
WITH RECURSIVE qn AS
  (  SELECT 1 AS n, 1 AS un, 1 AS unp1
     UNION ALL
    SELECT 1+n, unp1, un+unp1
        FROM qn WHERE n<10)
SELECT * FROM qn;
```

n	un	unp1
1	1	1
2	1	2
3	2	3
4	3	5
5	5	8
6	8	13
7	13	21
8	21	34
9	34	55
10	55	89

Recursive CTE – Example Hierarchy Traversal

```
CEATE TABLE EMPLOYEES (
        ID INT PRIMARY KEY,
        NAME VARCHAR(100),
        MANAGER_ID INT,
        INDEX (MANAGER_ID),
        FOREIGN KEY (MANAGER_ID) REFERENCES EMPLOYEES(ID) );
```

```
INSERT INTO EMPLOYEES VALUES
(333,   “Yasmina",   NULL), #  CEO , Manager ID = NULL
(198,   "John",       333), # John is 198 reports to 333, Yasmina
(692,   "Tarek",      333),
(29,    "Pedro",      198),  # Pedro is 29, reports to 198, John
(4610,  "Sarah",      29),
(72,    "Pierre",     29),
(123,   "Adil",       692);
```

Recursive CTE – Example Hierarchy Traversal

Print all employees and their management chain:

```
WITH RECURSIVE EMPLOYEES_EXTENDED (ID, NAME, PATH) AS (
  SELECT ID, NAME, CAST(ID AS CHAR(200))
  FROM EMPLOYEES  WHERE MANAGER_ID IS NULL
  UNION ALL
  SELECT S.ID, S.NAME, CONCAT(M.PATH, ",", S.ID)
  FROM EMPLOYEES_EXTENDED M JOIN   EMPLOYEES S ON M.ID=S.MANAGER_ID )
SELECT * FROM EMPLOYEES_EXTENDED ORDER BY PATH;
```

ID	NAME	PATH
333	Yasmina	333
198	John	333,198
29	Pedro	333,198,29
4610	Sarah	333,198,29,4610 # Sarah->Pedro->John->Yasmina
72	Pierre	333,198,29,72
692	Tarek	333,692
123	Adil	333,692,123

New Features in 8.0

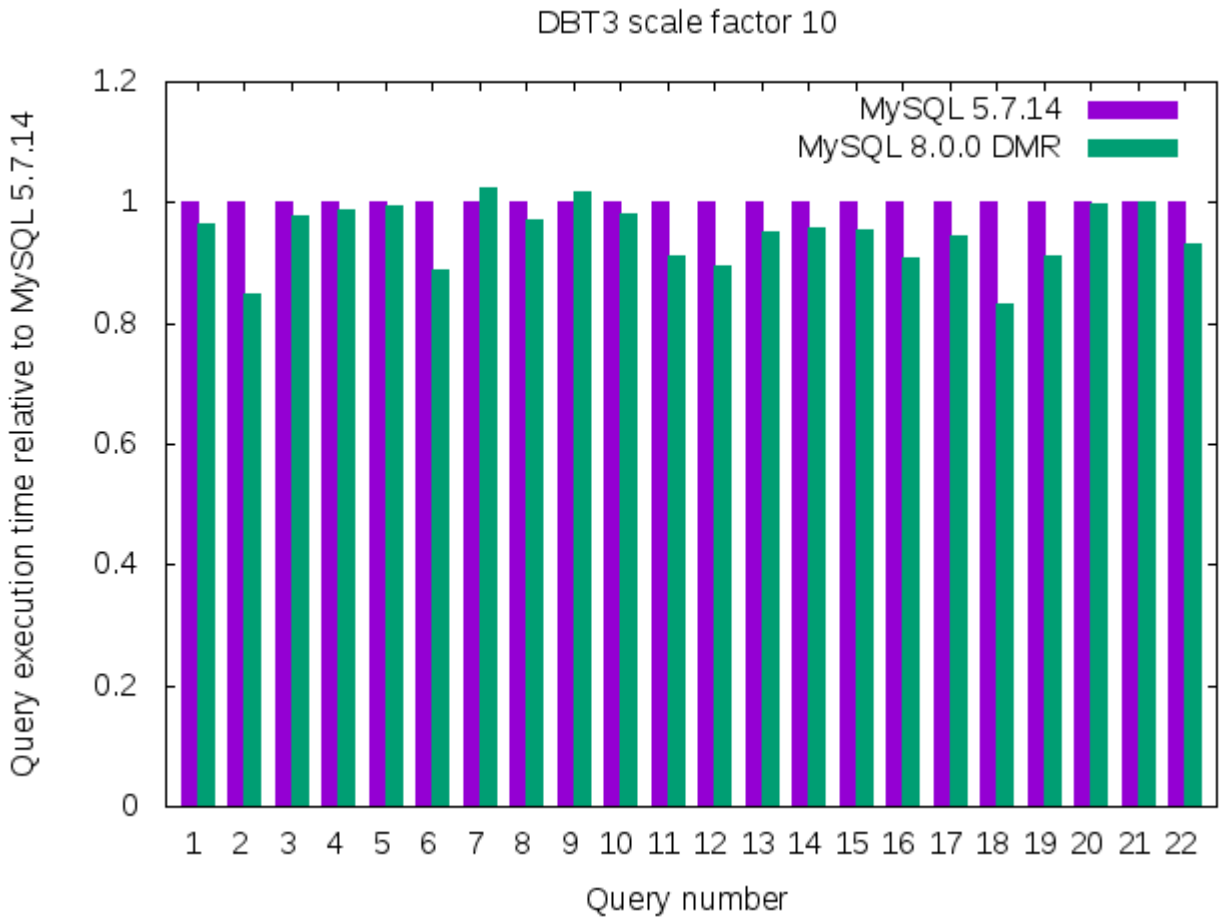
- Invisible index
- Descending index
- CTE
- **Improved performance of scans**
- JSON aggregation
- Hints
- Better UUID and IPv6support

Improved Performance of Scans

- Motivation:
  - InnoDB already uses an internal buffer to fetch records in batches
  - Optimizer knows the operation and the estimated number of records to read
  - Why cannot optimizer tell InnoDB how much to read??
- 8.0 extends the handler API: provides a buffer for each table or index where optimizer expects more than one row
  - So far only InnoDB uses this API
- Queries with improved performance:

```
SELECT * FROM t;
SELECT * FROM t WHERE pk BETWEEN 1000 and  10 000;
SELECT * FROM t1, t2 WHERE t1.pk = t2.pk;  /* buffer for the outer table, no need for inner table which is accessed using eq_ref)
```

Improved Performance of Scans - DBT3



- 18 out of 22 queries got improved performance
- 4 queries have more than 10% improvement, 2 queries more than 15% improvements

New Features in 8.0

- Invisible index
- Descending index
- CTE
- Improved performance of scans
- **JSON Aggregation**
- Hint
- Better UUID and IPv6support

JSON Support

- 5.7:
  - JSON datatype
  - JSON functions
  - Indexing of JSON through virtual column
- 8.0:
  - Add more functions
  - Improve performance

JSON Aggregation (8.0.0 optimizer lab release)

Combine JSON documents in multiple rows into a JSON array

```
CREATE TABLE t1(id INT, grp INT, jsoncol JSON);

 INSERT INTO t1 VALUES(1, 1,
'{"key1":"value1","key2":"value2"}');

INSERT INTO t1 VALUES(2, 1,
'{"keyA":"valueA","keyB":"valueB"}');

INSERT INTO t1 VALUES(3, 2,
'{"keyX":"valueX","keyY":"valueY"}');
```

```
SELECT JSON_ARRAYAGG(jsoncol) AS
json FROM t1;


[{"key1":"value1","key2":"value2"},
{"keyA":"valueA","keyB":"valueB"},
{"keyX":"valueX","keyY":"valueY"}]
```

JSON Aggregation (8.0.0 optimizer lab release)

Combine JSON documents in multiple rows into a JSON object

```
CREATE TABLE t1(id INT, grp INT,
jsoncol JSON);

 INSERT INTO t1 VALUES(1, 1,
'{"key1":"value1","key2":"value2"}');

INSERT INTO t1 VALUES(2, 1,
'{"keyA":"valueA","keyB":"valueB"}');

INSERT INTO t1 VALUES(3, 2,
'{"keyX":"valueX","keyY":"valueY"}');
```

```
SELECT JSON_OBJECTAGG(id, jsoncol) AS json
FROM t1 GROUP BY grp;


1 | {"1":{"key1":"value1","key2":"value2"},
"2":{"keyA":"valueA","keyB":"valueB"}}
2 | {"3":{"keyX":"valueX","keyY":"valueY"}}
```

New Features in 8.0

- Invisible index
- Descending index
- CTE
- Improved performance of scans
- **Hints**
- Better UUID and IPv6support

Improved HINTs

- 5.7: Introduced new hint syntax /*+ ...*/
  - Flexibility over optimizer switch, effect individual statement only
  - Hints within statement take precedence over optimizer switch
  - Hints apply at different scope levels: global, query block, table, index
- 8.0: Add more hints, gradually replace the old hint syntax

Hint: Join Order (8.0.0 optimizer lab Release)

- Hints to control table order for join execution
- 5.7: STRAIGHT_JOIN to force the listed order in FROM clause
- 8.0:
  - JOIN_FIXED_ORDER /* replacement for STRAIGHT_JOIN*/
  - JOIN_ORDER /* use specified order */
  - JOIN_PREFIX /* use specified order for first tables */
  - JOIN_SUFFIX /* use specified order for last tables */

Join Order Hints

Original query

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

id	select type	table	type	possible keys	key	key len	ref	rows	filtered	extra
1	SIMPLE	orders	ALL	i_o_orderdate, i_o_custkey	NULL	NULL	NULL	15000000	31.19	Using where
1	SIMPLE	customer	eq_ref	PRIMARY	PRIMARY	4	dbt3.orders.o_custkey	1	33.33	Using where

Join Order Hints

Change join order with hint

EXPLAIN SELECT /*+ JOIN_ORDER(customer, orders) */ *
FROM customer JOIN orders ON c_custkey = o_custkey
WHERE c_acctbal < -1000 AND o_orderdate < '1993-01-01';

id	select type	table	type	possible keys	key	key len	ref	rows	filtered	extra
1	SIMPLE	customer	ALL	PRIMARY	NULL	NULL	NULL	1500000	31.19	Using where
1	SIMPLE	orders	ref	i_o_orderdate, i_o_custkey	i_o_custkey	5	dbt3. customer. c_custkey	15	33.33	Using where

Alternatives with same effect for this query:

JOIN_PREFIX(customer) JOIN_SUFFIX(orders) JOIN_FIXED_ORDER()

Hint: Merge/Materialize Derived Table

```
SELECT /*+ NO_MERGE(dt) */ *
FROM t1 JOIN (SELECT x, y FROM t2) dt ON  t1.x = dt.x;
```

- Derived table is subquery in a FROM clause
- 5.6: Always materialize
- 5.7: Merged into outer queries in most cases, materialized in some cases
- 8.0: Users can override default behavior with hints
  - /*+ MERGE() */
  - /*+ NO_MERGE() */

New Features in 8.0

- Invisible index
- Descending index
- CTE
- Improved performance of scans
- Hints
- **Better UUID and IPv6support**

Improved Support for UUID

```
mysql> select uuid();
+--------------------------------------+
| uuid()                               |
+--------------------------------------+
| aab5d5fd-70c1-11e5-a4fb-b026b977eb28 |
+--------------------------------------+
```

- Five “-” separated hexadecimal numbers
- MySQL uses version 1, The first three numbers are generated from the low, middle, and high parts of a timestamp.
- 36 characters, inefficient for storage

➡ Convert to BINARY(16) datatype, only 16 bytes

Improved Support for UUID

```
IS_UUID ()  /* Check validity */

UUID_TO_BIN (arg1)   /* Convert UUID formatted text to binary(16) */
UUID_TO_BIN (arg1, arg2) /* If arg2 = true, shuffle  low and high time parts)

BIN_TO_UUID  (arg1)  /* Convert binary(16) to UUID formatted text */
BIN_TO_UUID  (arg1, arg2) /* If arg2 = true, shuffle  low and high time parts)
```

- Ease of use!
- More efficient indexing
  - Low part of timestamp being first makes index inserts slow
  - Shuffling low part with high part improves index efficiency

IPv4 vs IPv6

Getting all the networks that contain the given IP address

```
SELECT inet_ntoa(network) AS network, inet_ntoa(netmask) AS netmask
FROM network_table
WHERE (inet_aton('192.168.0.30') & netmask) = network;

SELECT inet6_ntoa(network) AS network, inet6_ntoa(netmask) AS netmask
FROM network_table
WHERE (inet6_aton('2001:0db8:85a3:0000:0000:8a2e:0370:7334') & netmask) = network;
```

- IPv4 address commonly stored as INT
- IPv6 address 128 bit, commonly stored as BINARY(16)
- The secondary query would give wrong results in 5.7, because & operator converts both operands from VARBINARY to BIGINT

Bitwise Operations on Binary Data Types

Bit operators and functions:

& | ^ ~ << >>

BIT_COUNT

Aggregate bit functions:

BIT_AND

BIT_XOR

BIT_OR

- Prior to 8.0, bitwise operations only worked on integer, truncation beyond 64 bits
- 8.0: Extends bit-wise operations to work with BINARY/BLOB
- 8.0: More functions to be added ...

Improved Support for IPv6

Use BINARY[16] and Bitwise-operations

```
SET @cidr = '2606:b400:85c:1040::1/64';

# Extract network length
SET @net_len = SUBSTRING_INDEX(@cidr, '/', -1 );
SELECT @net_len;
->  64

# Netmask
SET @net_mask= ~INET6_ATON('::') << (128 - @net_len);
SELECT INET6_NTOA(@net_mask);
->  ffff:ffff:ffff:ffff::

SET @network = INET6_ATON(SUBSTRING_INDEX(@cidr, '/', 1 )) & @net_mask;
# Print it in human-readable format
SELECT INET6_NTOA(@network);
-> 2606:b400:85c:1040::
```

Improved Support for IPv6

```
#host mask
SET @host_mask= ~INET6_ATON('::') >> @net_len;
SELECT INET6_NTOA(@host_mask);
-> ::ffff:ffff:ffff:ffff

# Generate network range IPv6 address.
SET @range_start = @network & @net_mask;
SELECT INET6_NTOA(@range_start);
-> 2606:b400:85c:1040::

SET @range_to = @network | @host_mask;
SELECT INET6_NTOA(@range_to);
-> 2606:b400:85c:1040:ffff:ffff:ffff:ffff

#Check if address in the range
SET @address1 = INET6_ATON('2606:b400:85c:1040:c5e4:2bb8:cc09:c4e2‘;)
SELECT @address1 between @range_start and @range_to:
-> 1
```

What is on Our Roadmap?

- Advanced JSON functions
- Cost model improvements: histogram, data in memory vs disk
- Advanced SQL features e.g WINDOWING functions
- Improve prepared statement/cache query plan
- Optimization for GIS operations

Hardware and Software
Engineered to Work Together