MySQL 8.0: Common Table Expressions

Guilhem Bichot – Lead Software Engineer
MySQL Optimizer Team, Oracle
Who

• With MySQL AB / Sun / Oracle since 2002
• Coder on the Server from 4.0 to 8.0
• Replication, on-line backup, Maria storage engine, optimizer...
• Recent work:
  - 5.7: for queries with GROUP BY, sql_mode=only_full_group_by smarter and enabled by default (see blog links at end)
  - 8.0: [recursive] common table expressions
• Guilhem.bichot@<theBigO>
Program Agenda

1. Non-recursive common table expressions
2. Recursive common table expressions
Program Agenda

1. Non-recursive common table expressions
2. Recursive common table expressions
What is a Common Table Expression?

• In official MySQL 8.0.1+.
• A derived table is a subquery in the FROM clause
  
  SELECT ... FROM (subquery) AS derived, t1 ...

• A Common Table Expression (CTE) is just like a derived table, but its declaration is moved to before the query
  
  WITH cte AS (subquery) SELECT ... FROM cte, t1 ...

• SQL:99: a named (sub)query, “Query Name”. MSSQL, PG: A “Table Expression” (~subquery) which is put in “Common”. ORCL: a factored out subquery, “Subquery Factoring”.

How do I create a CTE?

WITH cte_name [( <list of column names> )] AS
(
  SELECT ...  # Definition
)
[, <any number of other CTE definitions> ]
<SELECT/UPDATE/DELETE statement, referencing the CTEs>
Giving names to columns of the CTE

- Default naming: from selected expressions

  ```sql
  WITH cte AS (SELECT a+2, b FROM t1)
  SELECT cte.`a+2` + 10 FROM cte;
  ```

- Custom naming, alt. 1: give alias to the selected expressions

  ```sql
  WITH cte AS (SELECT a+2 AS a, b FROM t1)
  SELECT cte.a + 10 FROM cte;
  ```

- Custom naming, alt. 2: list names right after CTE's name

  ```sql
  WITH cte(a, b) AS (SELECT a+2, b FROM t2)
  SELECT cte.a + 10 FROM cte;
  ```

  - Side-note: also added same feature to derived table:

    ```sql
    SELECT ... FROM (SELECT a+2,b ...) AS derived(a,b);
    ```

    (views already had that: CREATE VIEW v(a,b) AS SELECT ...)
Two strategies of evaluation for a CTE

WITH cte AS (SELECT a FROM t1 WHERE a<5)
SELECT * FROM cte, t2 WHERE cte.a>3;

- “cte” may be materialized:
  - Evaluate SELECT a FROM t1 WHERE a<5
  - Store that in transient temporary table named “cte” with column named “a”
  - Do: SELECT * FROM cte, t2 WHERE cte.a>3;
  - Optimizer may create indexes on the temporary table to speed up predicates like cte.a=t2.b (look for “auto_key” in “key” column in EXPLAIN).

- Or “cte” may be merged:
  - SELECT * FROM t1, t2 WHERE t1.a>3 AND t1.a<5;
  - good if outer predicate cte.a>3 is selective while inner a<5 is not.
Two strategies of evaluation for a CTE

WITH cte AS (SELECT a FROM t1 WHERE a<5)
SELECT * FROM cte, t2 WHERE cte.a>3;

- Choice between the two strategies, materialization vs merging:
  - Heuristics
  - Technical constraints prevent merging
    - e.g. if CTE contains GROUP BY or is recursive: always materialized
  - Hint:
    - /*+ MERGE(cte) */
    - /*+ NO_MERGE(cte) */
    - Set `@@optimizer_switch=“derived_merge=on|off”`
    - Applicable to CTEs, views and derived tables
CTE can be local to subquery

WITH cte AS (SELECT a+2 AS a, b FROM t2)
SELECT * FROM t1 WHERE t1.a IN
(SELECT a * 3 FROM cte);  # Scope: “cte” is visible to top SELECT

VS

SELECT * FROM t1 WHERE t1.a IN
(WITH cte AS (SELECT a+2 AS a, b FROM t2)
 SELECT a * 3 FROM cte);  # Scope: “cte” is not visible to top SELECT
CTEs allowed in other statements

Can use CTE as read-only source to update other tables:

```
WITH cte AS (SELECT a+2 AS a, b FROM t2)
UPDATE t1, cte SET t1.a=cte.b + 10 WHERE t1.a = cte.a;

WITH cte AS (SELECT a+2 AS a, b FROM t2)
DELETE t1 FROM t1, cte WHERE t1.a = cte.a;

INSERT INTO t2
  WITH cte AS (SELECT a+2 AS a, b FROM t2)
  SELECT * FROM cte;
```
Common Table Expression vs. Derived Table

- Better readability
- Easier chaining
- Can be referenced multiple times
- Improved performance
Better readability

• Derived table:

```sql
SELECT dt.a
FROM t1 LEFT JOIN
    ((SELECT ... FROM ...) AS dt JOIN t2 ON ...) ON ...
```
- If reading top to bottom...
- ... You first see `dt.a` ... so what's `dt` ?
- ... Read on, it's deeply nested somewhere...

• CTE:

```sql
WITH dt AS (SELECT ... FROM ...)
SELECT dt.a
FROM t1 LEFT JOIN (dt JOIN t2 ON ...) ON ...
```
Easier chaining

• Derived table cannot refer to a neighbour derived table:

```sql
SELECT ...
FROM (SELECT ... FROM ...) AS d1,
    (SELECT ... FROM d1 ...) AS d2 ...
```

ERROR: 1146 (42S02): Table ‘db.d1’ doesn’t exist

• Workaround: nest d1 in d2

```sql
SELECT ...
FROM (SELECT ... FROM (SELECT ... FROM ...) AS d1 ...) AS d2
```

• Works but leads to nesting, is not very readable
Easier chaining

• CTE can refer to a neighbour CTE:

```sql
WITH d1 AS (SELECT ... FROM ...),
    d2 AS (SELECT ... FROM d1 ...)
SELECT ...
FROM d1, d2 ...
```

• More readable this way
WITH sales_by_month(month, total) AS
  # first CTE: one row per month, with amount sold on all days of month
  (SELECT MONTH(day_of_sale), SUM(amount) FROM sales_days
   WHERE YEAR(day_of_sale)=2017
   GROUP BY MONTH(day_of_sale)),
best_month(month, total, award) AS # second CTE: best month
  (SELECT month, total, "best" FROM sales_by_month
   WHERE total=(SELECT MAX(total) FROM sales_by_month)),
worst_month(month, total, award) AS # 3rd CTE: worst month
  (SELECT month, total, "worst" FROM sales_by_month
   WHERE total=(SELECT MIN(total) FROM sales_by_month))
# Now show best, worst and difference:
SELECT * FROM best_month UNION ALL SELECT * FROM worst_month
UNION ALL SELECT NULL, bm.total - wm.total, "diff"
FROM best_month bm, worst_month wm;
Example of chaining

Result:

<table>
<thead>
<tr>
<th>month</th>
<th>total</th>
<th>award</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>300</td>
<td>best</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>worst</td>
</tr>
<tr>
<td>NULL</td>
<td>289</td>
<td>diff</td>
</tr>
</tbody>
</table>
Can be referenced multiple times

• If you want to make N references to a derived table you must write N definitions: in the previous query you would need to write (copy) eight times the definition of sales_by_month

• CTE can be referenced multiple times: sales_by_month is referenced four times, best_month two times...
CREATE VIEW revenue0 (r_supplier_no, r_total_revenue) AS SELECT l_suppkey, SUM(l_extendedprice * (1-l_discount)) FROM lineitem WHERE l_shipdate >= '1996-07-01' AND l_shipdate < DATE_ADD('1996-07-01', INTERVAL '90' day) GROUP BY l_suppkey;

SELECT s_suppkey, s_name, s_address, s_phone, r_total_revenue FROM supplier, revenue0 WHERE s_suppkey = r_supplier_no AND r_total_revenue = (SELECT MAX(r_total_revenue) FROM revenue0) ORDER BY s_suppkey;

“Top Supplier Query”
Materialization of view

DBT-3 Query 15

```
SELECT Supplier (eq_ref)
FROM ORDER
JOIN Lineitem (range)
WHERE Revenue0
GROUP Materialize
```

```
SELECT Supplier (eq_ref)
FROM ORDER
JOIN Lineitem (range)
WHERE Revenue0
GROUP Materialize
```
### DBT-3 Query 15

**EXPLAIN**

<table>
<thead>
<tr>
<th>id</th>
<th>select type</th>
<th>table</th>
<th>type</th>
<th>possible keys</th>
<th>key</th>
<th>rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PRIMARY</td>
<td>&lt;derived3&gt;</td>
<td>ALL</td>
<td>NULL</td>
<td>NULL</td>
<td>4801074</td>
</tr>
<tr>
<td>1</td>
<td>PRIMARY</td>
<td>supplier</td>
<td>eq_ref</td>
<td>PRIMARY</td>
<td>PRIMARY</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>DERIVED</td>
<td>lineitem</td>
<td>range</td>
<td>i_l_shipdate, ...</td>
<td>i_l_shipdate</td>
<td>4801074</td>
</tr>
<tr>
<td>2</td>
<td>SUBQUERY</td>
<td>&lt;derived4&gt;</td>
<td>ALL</td>
<td>NULL</td>
<td>NULL</td>
<td>4801074</td>
</tr>
<tr>
<td>4</td>
<td>DERIVED</td>
<td>lineitem</td>
<td>range</td>
<td>i_l_shipdate, ...</td>
<td>i_l_shipdate</td>
<td>4801074</td>
</tr>
</tbody>
</table>
Rewritten to use a CTE

```
WITH revenue0 (r_supplier_no, r_total_revenue) AS
(SELECT l_suppkey, SUM(l_extendedprice * (1-l_discount))
FROM lineitem
WHERE l_shipdate >= '1996-07-01'
AND l_shipdate < DATE_ADD('1996-07-01', INTERVAL '90' day)
GROUP BY l_suppkey) /* same definition as view's */

SELECT s_suppkey, s_name, s_address, s_phone, r_total_revenue
FROM supplier, revenue0
WHERE s_suppkey = r_supplier_no
AND r_total_revenue =
   (SELECT MAX(r_total_revenue) FROM revenue0)
ORDER BY s_suppkey;
```
**DBT-3 Query 15, CTE**

**EXPLAIN**

<table>
<thead>
<tr>
<th>id</th>
<th>select type</th>
<th>table</th>
<th>type</th>
<th>possible keys</th>
<th>key</th>
<th>rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PRIMARY</td>
<td>&lt;derived2&gt;</td>
<td>ALL</td>
<td>NULL</td>
<td>PRIMARY</td>
<td>4801074</td>
</tr>
<tr>
<td>1</td>
<td>PRIMARY</td>
<td>supplier</td>
<td>eq_ref</td>
<td>PRIMARY</td>
<td>PRIMARY</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>SUBQUERY</td>
<td>&lt;derived2&gt;</td>
<td>ALL</td>
<td>NULL</td>
<td>NULL</td>
<td>4801074</td>
</tr>
<tr>
<td>2</td>
<td>DERIVED</td>
<td>lineitem</td>
<td>range</td>
<td>i_l_shipdate,</td>
<td>i_l_shipdate</td>
<td>4801074</td>
</tr>
</tbody>
</table>
DBT-3 Query 15

Materialization of CTE

```
SELECT Supplier (eq_ref)
FROM ORDER
JOIN Lineitem (range)
WHERE Revenue0
GROUP Materialize
```

MySQL
DBT-3 Query 15
Query Performance

Query Execution Time (seconds)
Single-materialized CTE

- N references but one temporary table
- Reduces disk usage, memory usage, execution time
- Possibly a different auto_key index for each reference
  - WITH cte AS (...) SELECT ... FROM cte AS cte1, cte AS cte2 WHERE cte1.a=3 AND cte2.b=5
  - Automatically adds index on cte.a and index on cte.b
  - In EXPLAIN: <auto_key0>, <auto_key1>, ...
- Can be better than CTE merging, if outer predicates are not very selective or base tables in CTE's subquery are poorly indexed
Single-materialized CTE

• Handy if CTE contains some random value to be used twice
• Lottery: find all rows of t1 where column “age” is +-5 around a certain random threshold, and also display the chosen threshold.
  
  WITH cte(rnd) AS (SELECT ROUND(RAND() * 125)
  SELECT t1.name, cte.rnd FROM cte, t1
  WHERE t1.age BETWEEN cte.rnd-5 AND cte.rnd+5
• Single materialization gives same value to all three uses of cte.rnd.
Program Agenda

1. Non-recursive common table expressions
2. Recursive common table expressions
Recursive CTE

WITH RECURSIVE cte AS
( SELECT ... FROM table_name /* "seed" SELECT */
 UNION [DISTINCT|ALL]
 SELECT ... FROM cte, table_name /* recursive SELECT */
 WHERE ...)
SELECT ... FROM cte; /* outer query */

• A recursive CTE refers to itself in the defining subquery
• Always materialized into transient temporary table
• Seed SELECT is executed once to create the first data subset
• Recursive SELECT is repeatedly executed to return new subsets of data: to the eyes of iteration N+1, “cte” = subset produced by iteration N, and only that.
Recursive CTE

• The process stops when an iteration does not generate any row
• The union of all subsets is stored into a transient temporary table; to the eyes of the outer query, “cte” = that table.
• Useful to
  – Generate series (start:end:step)
  – Explore hierarchies (parent/child, part/subpart, trees), connections (graphs)
Recursive CTE

A simple example : start:end:step sequence of values

Print 1 to 10 :

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

| a  | 1 # Seed SELECT | 2 # Recursive SELECT, it. 1 | 3 # Recursive SELECT, it. 2 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | Ø... because 10<10 is false |
Recursive CTE

Note the fixpoint

The final 'cte' satisfies:

cte ==
(SELECT 1 AS a
  UNION ALL
  SELECT 1+a FROM cte
WHERE a<10 )
Recursive CTE

Fixpoint (~10 years of research & discussion in SQL committee)

• Find the fixpoint(s) i.e. the solution(s) to $X = F(X)$, $X$ is a set
• Assuming $F$ is monotonic - if $Y$ contains $X$, $F(Y)$ contains $F(X)$
• $F(\emptyset)$, $F(F(\emptyset))$, $F(F(F(\emptyset)))$, … will stagnate at some point
• So the fixpoint exists and we found it
• Requirement 1: $F$ must be *monotonic*
• But iterating $F$ over growing sets is costly
• Optimization: iterate $F$ over new rows only
  – Works only if $F$ doesn't join $X$ with itself (*linear recursion*)
Recursive CTE

SQL-standard restrictions

• A recursive SELECT must reference the CTE
  - in its FROM clause, and only once (or miss “new JOIN old” ⇒ linear)
  - not in any subquery (NOT EXISTS (subquery) ⇒ monotonic)
  - not as the right table of a left join (row disappears ⇒ monotonic)

• A recursive SELECT must not contain
  - GROUP BY or aggregate/window functions (SUM, LEAD)
    (aggregated SUM changes ⇒ monotonic)

  - ORDER BY, LIMIT or DISTINCT (MySQL-specific restriction)

• My one-line rule: an iteration handles one row in isolation.
Recursive CTE

Implementation: greedy reading over an open-ended tmp table

Seed done

Recursive SELECT

Theory: run the recursive SELECT ten times; implementation: run it once.

And So on

FIFO queue

Read pointer
Write pointer
Recursive CTE

• Numbers 1-10 is a one-level recursive sequence
  - value N+1 = function of value N

• Two-level recursive sequence is possible
  - value N+2 = function of value N and value N+1
  - like Fibonacci numbers
  - However “an iteration handles one row in isolation”
  - So let the row for N+1 also contain a “recall” of N, in an extra column.

• With INSERT SELECT, the sequence can be stored in a table
### Date sequence

#### Missing dates

```sql
SELECT orderdate, SUM(totalprice) sales
FROM orders
GROUP BY orderdate
ORDER BY orderdate;
```

<table>
<thead>
<tr>
<th>orderdate</th>
<th>sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>2016-09-01</td>
<td>43129.83</td>
</tr>
<tr>
<td>2016-09-03</td>
<td>218347.61</td>
</tr>
<tr>
<td>2016-09-04</td>
<td>142568.40</td>
</tr>
<tr>
<td>2016-09-05</td>
<td>299244.83</td>
</tr>
<tr>
<td>2016-09-07</td>
<td>185991.79</td>
</tr>
</tbody>
</table>

Missing September 2nd and 6th...
WITH RECURSIVE dates(date) AS
( SELECT '2016-09-01'
UNION ALL
SELECT DATE_ADD(date,
    INTERVAL 1 DAY)
FROM dates
WHERE date < '2016-09-07' )
SELECT dates.date,
    COALESCE(SUM(totalprice), 0) sales
FROM dates LEFT JOIN orders
    ON dates.date = orders.orderdate
GROUP BY dates.date
ORDER BY dates.date;

<table>
<thead>
<tr>
<th>date</th>
<th>sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>2016-09-01</td>
<td>43129.83</td>
</tr>
<tr>
<td>2016-09-02</td>
<td>0.00</td>
</tr>
<tr>
<td>2016-09-03</td>
<td>218347.61</td>
</tr>
<tr>
<td>2016-09-04</td>
<td>142568.40</td>
</tr>
<tr>
<td>2016-09-05</td>
<td>299244.83</td>
</tr>
<tr>
<td>2016-09-06</td>
<td>0.00</td>
</tr>
<tr>
<td>2016-09-07</td>
<td>185991.79</td>
</tr>
</tbody>
</table>
Hierarchy Traversal

Employee database

CREATE TABLE employees (
  id INT PRIMARY KEY,
  name VARCHAR(100),
  manager_id INT,
  FOREIGN KEY (manager_id)
  REFERENCES employees(id) );

INSERT INTO employees VALUES
(333, "Yasmina", NULL),  # CEO
(198, "John", 333),  # J reports to Y
(692, "Tarek", 333),
(29, "Pedro", 198),
(4610, "Sarah", 29),
(72, "Pierre", 29),
(123, "Adil", 692);
Hierarchy Traversal

Employee database

Yasmina (333)

John (198)  Tarek (692)

Pedro (29)  Adil (123)

Sarah (4610)  Pierre (72)
Hierarchy Traversal

How to walk the tree

Seed

Iteration 1

Iteration 2

Iteration 3

Iteration 4

∅
# Hierarchy Traversal

**Find reporting chain for every employee**

WITH RECURSIVE

\[
\text{emp\_ext (id, name, path) AS (}
\begin{align*}
\text{SELECT id, name, CAST(id AS CHAR(200))} & \text{ FROM employees} \\
\text{WHERE manager\_id IS NULL} & \text{ UNION ALL} \\
\text{SELECT s.id, s.name,} & \text{ CONCAT(m.path, ",", s.id)} \\
& \text{ FROM emp\_ext m JOIN employees s} \\
& \text{ ON m.id=s.manager\_id } \\
\end{align*}
\]

\]

SELECT * FROM emp\_ext ;

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>path</th>
</tr>
</thead>
<tbody>
<tr>
<td>333</td>
<td>Yasmina</td>
<td>333</td>
</tr>
<tr>
<td>198</td>
<td>John</td>
<td>333,198</td>
</tr>
<tr>
<td>692</td>
<td>Tarek</td>
<td>333,692</td>
</tr>
<tr>
<td>29</td>
<td>Pedro</td>
<td>333,198,29</td>
</tr>
<tr>
<td>123</td>
<td>Adil</td>
<td>333,692,123</td>
</tr>
<tr>
<td>4610</td>
<td>Sarah</td>
<td>333,198,29,4610</td>
</tr>
<tr>
<td>72</td>
<td>Pierre</td>
<td>333,198,29,72</td>
</tr>
</tbody>
</table>

∅

Note: no guaranteed output order
## Hierarchy Traversal

**Reporting chain, depth first: “children” before “brothers”**

WITH RECURSIVE
emp_ext (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 emp_ext m JOIN employees s  
      ON m.id=s.manager_id )  
SELECT * FROM emp_ext ORDER BY path;

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>path</th>
</tr>
</thead>
<tbody>
<tr>
<td>333</td>
<td>Yasmina</td>
<td>333</td>
</tr>
<tr>
<td>198</td>
<td>John</td>
<td>333,198</td>
</tr>
<tr>
<td>29</td>
<td>Pedro</td>
<td>333,198,29</td>
</tr>
<tr>
<td>4610</td>
<td>Sarah</td>
<td>333,198,29,4610</td>
</tr>
<tr>
<td>72</td>
<td>Pierre</td>
<td>333,198,29,72</td>
</tr>
<tr>
<td>692</td>
<td>Tarek</td>
<td>333,692</td>
</tr>
<tr>
<td>123</td>
<td>Adil</td>
<td>333,692,123</td>
</tr>
</tbody>
</table>

(tree)
Hierarchy Traversal

Reporting chain, breadth first: “brothers” before “children”

WITH RECURSIVE emp_ext (id, name, path, lvl) AS (  
SELECT id, name,  
CAST(id AS CHAR(200)), 1  
FROM employees  
WHERE manager_id IS NULL  
UNION ALL  
SELECT s.id, s.name,  
CONCAT(m.path, "","", s.id), m.lvl + 1  
FROM emp_ext m JOIN employees s  
ON m.id=s.manager_id )  
SELECT * FROM emp_ext ORDER BY lvl;

id  name  path  lvl
333  Yasmina  333  1
198  John  333,198  2
692  Tarek  333,692  2
29  Pedro  333,198,29  3
123  Adil  333,692,123  3
4610  Sarah  333,198,29,4610  4
72  Pierre  333,198,29,72  4

(tree)
Recursive CTE – complete syntax

WITH RECURSIVE cte_name [ list of column names ] AS
(
    SELECT ...  <!-- specifies initial set
    UNION [DISTINCT|ALL]
    SELECT ...  <!-- specifies initial set
    UNION [DISTINCT|ALL]

    ...  
    SELECT ...  <!-- specifies how to derive new rows
    UNION [DISTINCT|ALL]
    SELECT ...  <!-- specifies how to derive new rows

    ...  
)

[, any number of other CTE definitions ]

<SELECT/UPDATE/DELETE statement, referencing the CTEs>
Recursive CTE – goodies!

- UNION DISTINCT to add only non-duplicate rows
  - Think: walking a graph to find “all direct and indirect connections of A” (a.k.a. build the transitive closure of A): A->B->C->A
  - Several well-known commercial DBMSs don't support UNION DISTINCT
- Flagging abnormal cycles:
  - Leverage the PATH column
  - A new row is closing a cycle if it's already in the PATH (test with FIND_IN_SET)
- More tips about cycles: see my blogs (last slides)
Recursive CTE – goodies!

• Runaway query:
  – Forgot the WHERE in recursive SELECT, or typed it wrong (a<0 in lieu of a>0)
  – Or: unexpected cycle in my data

• You're covered:

  WITH RECURSIVE cte AS
  ( SELECT 1 AS a UNION ALL SELECT 1+a FROM cte )
  SELECT * FROM cte;

ERROR HY000: Recursive query aborted after 1001 iterations. Try increasing @@cte_max_recursion_depth to a larger value.
Want to try it out?

• Available since version 8.0.1

• Install MySQL Server 8.0.3 (Release Candidate)
  - Bonus over 8.0.1:
    • @@cte_max_recursion_depth variable
    • A couple of bugfixes
    • No known bugs
Want to learn more?

- My four blog posts:
  


- Shameless plug 😊 only_full_group_by improvements in 5.7: [http://mysqlserverteam.com/mysql-5-7-only_full_group_by-improved-recognizing-functional-dependencies-enabled-by-default/](http://mysqlserverteam.com/mysql-5-7-only_full_group_by-improved-recognizing-functional-dependencies-enabled-by-default/)
Safe Harbor Statement

The preceding 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.