Programmatic Queries

Things you can code with SQL

Shlomi Noach
openark.org

Percona Live, London 2011

SQL

- SQL, or the Structured Query Language, is often referred to as a *declarative language*.
  - From **Wikipedia**:

> ***declarative programming*** *is a programming paradigm that expresses the logic of a computation without describing its control flow.*

- SQL and the relational model are in part based on *relational algebra*, which allows for mathematical expressions and conclusions on the model.

Is SQL really declarative?

- Even in standard SQL, there are hints of algorithmic behavior.
  - Does **ORDER BY** imply an algorithm?
  - Does MySQL's **ORDER BY... LIMIT** imply an algorithm?
  - How about *Window Functions* & running totals?

```
SELECT SUM(sale_amount) OVER (ORDER BY sale_date)
FROM sales
```

Query behavior in MySQL

- There are certain aspects to query behavior in MySQL, that imply *programmatic* nature.
- We discuss a few:
  - Row evaluation order
  - Control flow evaluation order
  - Table and query materialization order
  - Time & time suspension

User Defined Variables

- Perhaps the most obvious programmatic feature in MySQL; stored routines aside.
- One is allowed to define, assign, calculate & reassign variable values throughout a query's execution.
- Variables can be used to generate counters, running totals, ranking, sequences, and more.

```
SET @counter := 0;
SELECT (@counter := @counter + 1) AS counter, Name
  FROM world.City;
```

User Defined Variables

counter	Name
1	Kabul
2	Qandahar
3	Herat
4	Mazar-e-Sharif
5	Amsterdam
6	Rotterdam
7	Haag
8	Utrecht
9	Eindhoven
10	Tilburg

User Defined Variables

- We rely on a programmatic nature of MySQL:
  - MySQL evaluates the result set row by row
  - The order of rows is sometimes easily predictable; other times unexpected
- The following query will assign lower counter values to larger cities:

```
SET @counter := 0;
SELECT (@counter := @counter + 1) AS counter, Name
  FROM world.City ORDER BY Population DESC;
```

Derived tables

- The above query requires us to remember to **SET** the **@counter** variable to zero each time.
  - We look for a solution that will do it all in one single query.
- The common solution is to use *derived tables*:

```
SELECT (@counter := @counter + 1) AS counter, Name
  FROM world.City, (SELECT @counter := 0) s_init;
```

Derived tables

- A *derived table* is a query passed in place of a table.
- Derived table is materialized before the outer query is evaluated.
  - See sql_base.cc
- This makes for a programmatic nature of the query, where we can expect certain parts to execute before others.
  - In the above example, we used that knowledge to set a session variable.
  - Other use cases include: instantiating a subquery; taking advantage of indexes from other tables, etc.

Order of evaluation

- The following query produces the *Fibonacci sequence*.
- It *happens* to work.

```
SELECT
  @c3 := @c1 + @c2 AS value,
  @c1 := @c2,
  @c2 := @c3
FROM
  mysql.help_topic,
  (SELECT @c1 := 1, @c2 := 0) s_init
LIMIT 10;
```

Order of evaluation

value	@c1 := @c2	@c2 := @c3
1	0	1
1	1	1
2	1	2
3	2	3
5	3	5
8	5	8
13	8	13
21	13	21
34	21	34
55	34	55

Order of evaluation

- However, it relies on the first column to be evaluated first, second column to come next etc.
- This is not guaranteed, and fails to work on more complex queries.
- We look for a construct that can guarantee order of evaluation.

Order of evaluation: CASE … WHEN … ELSE

- A **CASE … WHEN … ELSE** statement:

> *… returns the result for the first condition that is true. If there was no matching … the result after ELSE is returned …* ***[The MySQL Manual]***

- This is a *declarative* explanation. However, the easiest way to *implement* this also happens to be the most efficient way.

Order of evaluation: CASE … WHEN … ELSE

- CASE evaluation algorithm:
  - Try (evaluate) the first **WHEN** statement.
  - Is it *True*? Quit.
  - Otherwise try the second **WHEN**.
  - Is it *True*? Quit.
  - Otherwise …
  - No luck? Evaluate the **ELSE** statement.
    - See item_cmpfunc.cc
- We can utilize this known programmatic nature of evaluation for writing our code.

Order of evaluation

- The following query works correctly, and the order of evaluation is *predicted*:

```
SELECT
  CASE
    WHEN (@c3 := @c1 + @c2) IS NULL THEN NULL
    WHEN (@c1 := @c2) IS NULL THEN NULL
    WHEN (@c2 := @c3) IS NULL THEN NULL
    ELSE @c3
  END AS seq
FROM
  mysql.help_topic,
  (SELECT @c1 := 1, @c2 := 0) s_init
LIMIT 10;
```

UNION ALL

- A **UNION ALL** query concatenates results from multiple query parts.
- The declaration of **UNION ALL** says nothing about the order of rows in the result set.
- The MySQL manual explicitly suggests that order of rows is unknown.
- However the above says nothing about the *time* at which the statements are executed.

UNION ALL execution order

- As expected, when no derived tables are involved, statements are executed by order of appearance in the **UNION ALL** query.
  - See sql_union.cc

```
SELECT
  'This will be executed first' AS description
UNION ALL
  SELECT 'This will be executed second'
UNION ALL
  SELECT 'This will be executed last'
;
```

UNION ALL: Time delayed statements

- The following reads **com_select**, sleeps for **10** seconds, then reads **com_select** again.

```
SELECT VARIABLE_VALUE AS value
  FROM INFORMATION_SCHEMA.GLOBAL_STATUS
  WHERE VARIABLE_NAME = 'com_select'
UNION ALL
  SELECT SLEEP(10) FROM DUAL
UNION ALL
  SELECT VARIABLE_VALUE
  FROM INFORMATION_SCHEMA.GLOBAL_STATUS
  WHERE VARIABLE_NAME = 'com_select'
;
```

UNION ALL: Time delayed statements

- We may not know in advance the order of rows in the result set.
- We can force the order by providing an extra sorting column; but we don't always need to.

```
+-------+
| value |
+-------+
| 10397 |
| 0     |
| 10562 |
+-------+
```

UNION ALL: Time delayed statements

- The following computes **com_select** per second:

```
SELECT SUM(value) AS diff, SUM(value)/10 AS rate
FROM (
  SELECT 0 - VARIABLE_VALUE AS value
    FROM INFORMATION_SCHEMA.GLOBAL_STATUS
    WHERE VARIABLE_NAME = 'com_select'
  UNION ALL
    SELECT SLEEP(10) FROM DUAL
  UNION ALL
   SELECT 0 + VARIABLE_VALUE
    FROM INFORMATION_SCHEMA.GLOBAL_STATUS
    WHERE VARIABLE_NAME = 'com_select'
) s_inner;
```

Monitoring queries

- Last example provides SQL with monitoring capabilities
  - A META query over META data
- Why stop with a single variable?
- Using our knowledge of evaluation order, we can combine derived tables, time delay & **UNION ALL** execution order into a general purpose monitoring query.
- common_schema provides with a view which queries for any changes for status variables.

Monitoring MySQL

```
SELECT
  gs0.VARIABLE_NAME,
  (gs1.VARIABLE_VALUE - gs0.VARIABLE_VALUE),
  (gs1.VARIABLE_VALUE - gs0.VARIABLE_VALUE) / 10
FROM (
  SELECT
    VARIABLE_NAME, VARIABLE_VALUE
    FROM INFORMATION_SCHEMA.GLOBAL_STATUS
  UNION ALL
    SELECT '', SLEEP(10) FROM DUAL
  ) AS gs0
  JOIN INFORMATION_SCHEMA.GLOBAL_STATUS gs1
    USING (VARIABLE_NAME)
;
```

Monitoring MySQL

VARIABLE_NAME	diff	rate
BYTES_RECEIVED	56	5.6
COM_CALL_PROCEDURE	325	32.5
COM_SELECT	162	16.2
COM_SET_OPTION	648	64.8
HANDLER_READ_RND_NEXT	587	58.7
HANDLER_WRITE	878	87.8
QUESTIONS	1	0.1
SELECT_FULL_JOIN	1	0.1
SELECT_SCAN	2	0.2

Pop quiz

- What would be the result of the following query?

```
SELECT NOW()
 UNION ALL
SELECT SLEEP(10)
 UNION ALL
SELECT NOW()
;
```

Time evaluation

- **NOW()** indicates the moment the query started (or routine start of execution)
- **SYSDATE()** indicates the time the operating system reports, moment of function execution.
- This makes for a programmatic nature.
- Given a query, how much time does it take to evaluate?
  - On your Python/PHP/Perl/... code it's easy to read start/end times, and do the math.
  - Is it possible to get the time by a query?

Time evaluation

- Assume the following query:

```
SELECT * FROM t1 JOIN t2 USING (c) JOIN t3 USING (d)
```

- We can get just the time it took to execute, or we can pass along the time within the query. For example:

```
SELECT TIMESTAMPDIFF(
  MICROSECOND, NOW(), MAX(SYSDATE())) / 1000000.0
FROM (
  SELECT * FROM t1 JOIN t2 USING (c) JOIN t3 USING (d)
) s_orig;
```

Imagine

- Suppose **LIMIT** could accept a non-constant value; an expression to be re-evaluated.
- Imagine the following:

```
SELECT lots_of_data FROM
  many, many_tables
WHERE
  some_conditions_are_met
LIMIT
  IF(
    TIMESTAMPDIFF(SECOND, NOW(), SYSDATE()) < 60,
    99999999999,
    0
  );
```

Homework

- *(Hey, be thankful there's no surprise exam!)*
- Assume a very long running **SELECT** query (e.g. a reporting query).
  - For simplicity, assume it scans a single table.
- The query makes for a significant I/O impact.
- Task: make the query self throttling!
  - *Step 1*: make the query **SLEEP()** for **5** seconds for every **1,000** rows scanned
  - *Step 2*: make the query **SLEEP()** for the duration of last **1,000** rows scanned (effectively doubling the total runtime of the query)

Thank you!

- I blog at http://openark.org
- Find open source projects on http://code.openark.org/forge/
- Contact me at shlomi@[you-know-where].org
- Questions?