The Optimizer Trace

Introduction

This optimizer trace is aimed at producing output, which is readable by humans and by programs, to aid understanding of decisions and actions taken by the MySQL Optimizer.

Output format

The chosen output format is JSON (JavaScript Object Notation). In JSON there are:

- "objects" (unordered set of key-value pairs); equivalent to Python's dictionary or Perl's associative array or hash or STL's hash_map.
- "arrays" (ordered set of values); equivalent to Python's and Perl's list or STL's vector.
- "values": a value can be a string, number, boolean, null, which we all call "scalars", or be an object, array.

For example (explanations after "<<" are not part of output):

```
{                           << start of top object
  "first_name": "Gustave",  << key/value pair (value is string)
  "last_name": "Eiffel",    << key/value pair (value is string)
  "born": 1832,             << key/value pair (value is integer)
  "contributions_to": [     << key/value pair (value is array)
     {                      << 1st item of array is an object (a building)
       "name": "Eiffel tower",
       "location": Paris
     },                     << end of 1st item of array
     {
       "name": "Liberty statue",
       "location": "New York"
     }                      << end of 2nd item of array
  ]                         << end of array
}                           << end of top object
```

For more details, have a look at the syntax at json.org. Note that indentation and newlines are superfluous, useful only for human-readability. Note also that there is nothing like a "named object": an object, array or value has no name; but if it is the value of a key/value pair in an enclosing, outer object, then the key can be seen as the inner object's "name".

How a user enables/views the trace

```
SET SESSION OPTIMIZER_TRACE="enabled=on"; # enable tracing
<statement to trace>; # like SELECT, EXPLAIN SELECT, UPDATE, DELETE...
SELECT * FROM information_schema.OPTIMIZER_TRACE;
[ repeat last two steps at will ]
SET SESSION OPTIMIZER_TRACE="enabled=off"; # disable tracing
```

`SELECT` and `EXPLAIN` SELECT produce the same trace. But there are exceptions regarding subqueries because the two commands treat subqueries differently, for example in

```
SELECT ... WHERE x IN (subq1) AND y IN (subq2)
```

SELECT terminates after executing the first subquery if the related IN predicate is false, so we won't see `JOIN::optimize()` tracing for subq2; whereas EXPLAIN SELECT analyzes all subqueries (see loop at the end of `select_describe()`).

How a user traces only certain

statements

When tracing is in force, each SQL statement generates a trace; more exactly, so does any of SELECT, EXPLAIN SELECT, INSERT or REPLACE (with VALUES or SELECT), UPDATE/DELETE and their multi-table variants, SET (unless it manipulates @@optimizer_trace), DO, DECLARE/CASE/IF/RETURN (stored routines language elements), CALL. If a command above is prepared and executed in separate steps, preparation and execution are separately traced. By default each new trace overwrites the previous trace. Thus, if a statement contains sub-statements (example: invokes stored procedures, stored functions, triggers), the top statement and sub-statements each generate traces, but at the execution's end only the last sub-statement's trace is visible. If the user wants to see the trace of another sub-statement, she/he can enable/disable tracing around the desired sub-statement, but this requires editing the routine's code, which may not be possible. Another solution is to use

```
SET optimizer_trace_offset=<OFFSET>, optimizer_trace_limit=<LIMIT>
```

where OFFSET is a signed integer, and LIMIT is a positive integer. The effect of this SET is the following:

- all remembered traces are cleared
- a later SELECT on OPTIMIZER_TRACE returns the first LIMIT traces of the OFFSET oldest remembered traces (if OFFSET >= 0), or the first LIMIT traces of the -OFFSET newest remembered traces (if OFFSET < 0).

For example, a combination of OFFSET=-1 and LIMIT=1 will make the last trace be shown (as is default), OFFSET=-2 and LIMIT=1 will make the next-to-last be shown, OFFSET=-5 and LIMIT=5 will make the last five traces be shown. Such negative OFFSET can be useful when one knows that the interesting sub-statements are the few last ones of a stored routine, like this:

```
SET optimizer_trace_offset=-5, optimizer_trace_limit=5;
CALL stored_routine(); # more than 5 sub-statements in this routine
SELECT * FROM information_schema.OPTIMIZER_TRACE; # see only last 5 traces
```

On the opposite, a positive OFFSET can be useful when one knows that the interesting sub-statements are the few first ones of a stored routine.

The more those two variables are accurately adjusted, the less memory is used. For example, OFFSET=0 and LIMIT=5 will use memory to remember 5 traces, so if only the three first are needed, OFFSET=0 and LIMIT=3 is better (tracing stops after the LIMITth trace, so the 4th and 5th trace are not created and don't take up memory). A stored routine may have a loop which executes many sub-statements and thus generates many traces, which would use a lot of memory; proper OFFSET and LIMIT can restrict tracing to one iteration of the loop for example. This also gains speed, as tracing a sub-statement impacts performance.

If OFFSET>=0, only LIMIT traces are kept in memory. If OFFSET<0, that is not true: instead, (-OFFSET) traces are kept in memory; indeed even if LIMIT is smaller than (-OFFSET), so excludes the last statement, the last statement must still be traced because it will be inside LIMIT after executing one more statement (remember than OFFSET<0 is counted from the end: the "window" slides as more statements execute).

Such memory and speed gains are the reason why optimizer_trace_offset/limit, which are restrictions at the trace producer level, are offered. They are better than using

```
SELECT * FROM OPTIMIZER_TRACE LIMIT <LIMIT> OFFSET <OFFSET>;
```

which is a restriction on the trace consumer level, which saves almost nothing.

How a user traces only certain

optimizer features

```
SET OPTIMIZER_TRACE_FEATURES="feature1=on|off,...";
```

where "feature1" is one optimizer feature. For example "greedy_search": a certain Opt_trace_array at the start of `Optimize_table_order::choose_table_order()` has a flag "GREEDY_SEARCH" passed to its constructor: this means that if the user has turned tracing of greedy search off, this array will not be written to the I_S trace, neither will any children structures. All this disabled "trace chunk" will be replaced by an ellipsis "...".

How a developer adds tracing to a function

Check `Opt_trace*` usage in `advance_sj_state()`:

```
Opt_trace_array trace_choices(trace, "semijoin_strategy_choice");
```

This creates an array for key "semijoin_strategy_choice". We are going to list possible semijoin strategy choices.

```
Opt_trace_object trace_one_strategy(trace);
```

This creates an object without key (normal, it's in an array). This object will describe one single strategy choice.

```
trace_one_strategy.add_alnum("strategy", "FirstMatch");
```

This adds a key/value pair to the just-created object: key is "strategy", value is "FirstMatch". This is the strategy to be described in the just-created object.

```
trace_one_strategy.add("cost", *current_read_time).
  add("records", *current_record_count);
trace_one_strategy.add("chosen", (pos->sj_strategy == SJ_OPT_FIRST_MATCH));
```

This adds 3 key/value pairs: cost of strategy, number of records produced by this strategy, and whether this strategy is chosen.

After that, there is similar code for other semijoin strategies.

The resulting trace piece (seen in `information_schema.OPTIMIZER_TRACE`) is

```
          "semijoin_strategy_choice": [
            {
              "strategy": "FirstMatch",
              "cost": 1,
              "records": 1,
              "chosen": true
            },
            {
              "strategy": "DuplicatesWeedout",
              "cost": 1.1,
              "records": 1,
              "duplicate_tables_left": false,
              "chosen": false
            }
          ]
```

For more output examples, check result files of the opt_trace suite in `mysql-test`.

Feature can be un-compiled with

```
cmake -DOPTIMIZER_TRACE=0
```

.

Interaction between trace and DBUG

We don't want to have to duplicate code like this:

```
  DBUG_PRINT("info",("cost %g",cost));
  Opt_trace_object(thd->opt_trace).add("cost",cost);
```

Thus, any optimizer trace operation, *even* if tracing is run-time disabled, has an implicit DBUG_PRINT("opt",...) inside. This way, only the second line above is needed, and several DBUG_PRINT() could be removed from the Optimizer code. When tracing is run-time disabled, in a debug binary, traces are still created in order to catch the `add()` calls and write their text to DBUG, but those traces are not visible into INFORMATION_SCHEMA.OPTIMIZER_TRACE: we then say that they "don't support I_S". A debug binary without optimizer trace compiled in, will intentionally not compile.

Because opening an object or array, or add()-ing to it, writes to DBUG immediately, a key/value pair and its outer object may be 100 lines apart in the DBUG log.

Guidelines for adding tracing

- Try to limit the number of distinct "words". For example, when describing an optimizer's decision, the words "chosen" (true/false value, tells whether we are choosing the said optimization), "cause" (free text value, tells why we are making this choice, when it's not obvious) can and should often be used. Having a restricted vocabulary helps consistency. Use "row" instead of "record". Use "tmp" instead of "temporary".
- Use only simple characters for key names: a-ZA-Z_#, and no space. '#' serves to denote a number, like in "select#" .
- Keep in mind than in an object, keys are not ordered; an application may parse the JSON output and output it again with keys order changed; thus when order matters, use an array (which may imply having anonymous objects as items of the array, with keys inside the anonymous objects, see how it's done in `JOIN::optimize()`). Keep in mind that in an object keys should be unique, an application may lose duplicate keys.

Handling of "out-of-memory" errors

All memory allocations (with exceptions: see below) in the Optimizer trace use `my_error()` to report errors, which itself calls `error_handler_hook`. It is the responsibility of the API user to set up a proper `error_handler_hook` which will alert her/him of the OOM problem. When in the server, this is already the case (`error_handler_hook` is `my_message_sql()` which makes the statement fail). Note that the debug binary may crash if OOM (OOM can cause syntax errors...).

Description of trace-induced security checks.

A trace exposes information. For example if one does SELECT on a view, the trace contains the view's body. So, the user should be allowed to see the trace only if she/he has privilege to see the body, i.e. privilege to do SHOW CREATE VIEW. There are similar issues with stored procedures, functions, triggers.

We implement this by doing additional checks on SQL objects when tracing is on:

- stored procedures, functions, triggers: checks are done when executing those objects
- base tables and views.

Base tables or views are listed in some `LEX::query_tables`. The LEX may be of the executing statement (statement executed by `mysql_execute_command()`, or by `sp_lex_keeper::reset_lex_and_exec_core()`), we check this LEX in the constructor of Opt_trace_start. Or it may be a LEX describing a view, we check this LEX when opening the view (`open_and_read_view()`).

Those checks are greatly simplified by disabling traces in case of security context changes.

See also
opt_trace_disable_if_no_security_context_access().

Those checks must be done with the security context of the connected user. Checks with the SUID context would be useless: assume the design is that the basic user does not have DML privileges on tables, but only EXECUTE on SUID-highly-privileged routines (which implement *controlled approved* DMLs): then the SUID context would successfully pass all additional privilege checks, routine would generate tracing, and the connected user would view the trace after the routine's execution, seeing secret information.

What a developer should read next

The documentation of those classes, in order

```
Opt_trace_context
Opt_trace_context_impl
Opt_trace_stmt
Opt_trace_struct
Opt_trace_object
Opt_trace_array
```

and then **opt_trace.h** as a whole.