

# Collecting Runtime Statistics - Profiling

- [Overview](#)
- [Usage](#)
- [Examining Profiling Output](#)
- [Caveats for Runtime Statistics](#)
- [Making Runtime Statistics Persistent](#)
- [Examples of Profiling Output](#)
- [Additional Performance Measurements](#)

## Overview

ClustrixDB includes the capability to profile SQL statements. When profiling is enabled, fine-grained statistics are collected for every query. These statistics can be used to provide insight into query execution and into how time is being spent during that execution. Statistics include [invocations](#), query [fragments](#), [forwards](#), [broadcasts](#), etc.

Profiling output can contain sensitive information such as usernames and passwords. Avoid enabling profiling when running queries or commands that may contain this type of sensitive information.

## Usage

To enable profiling, simply set the session variable:

```
> SET SESSION PROFILING = 1;
```

Then run your query or queries and then turn profiling off:

```
> SET SESSION PROFILING = 0;
```

During the time that profiling is enabled, ClustrixDB will collect detailed statistics on query execution and how resources were spent, including time spent waiting for CPU, number and types of operations performed on rows, and how data was forwarded between nodes.

Runtime statistics are persisted only at a session level and are no longer available once the session ends. To make statistics persist beyond a session, please see the section below on Retaining Runtime Statistics.

## Examining Profiling Output

### View list of Profiled Statements

Here are some sample queries you can use to examine the output of query profiling.

#### view the list of queries profiled

```
> SELECT statement
FROM   system.profiled_statements
WHERE  session_id = @@session_id
ORDER BY event_id;
```

### Examining Runtime Statistics

### Query to examine runtime statistics for a given query and session

```
> SELECT Concat(LEFT(REPLACE(Min(statement.statement), "\n", " "), 50), "...") AS statement,
      invocation.iid,
      stats.fragment,
      Sum(stats.fragment_executions)           AS executions,
      Sum(stats.cpu_runtime_ns)                AS cpu_runtime_ns,
      Sum(stats.cpu_waits)                     AS cpu_waits,
      Sum(stats.cpu_waittime_ns)              AS cpu_waittime_ns,
      Count(DISTINCT stats.nodeid)             AS nodes_used,
      Count(DISTINCT stats.nodeid, stats.cpu) AS cpus_used,
      Sum(stats.forwards)                      AS forwards,
      Sum(stats.broadcasts)                   AS broadcasts,
      Sum(stats.rows_read)                     AS rows_read,
      Sum(stats.rows_output)                   AS rows_output,
      Sum(stats.inserts)                       AS inserts,
      Sum(stats.updates)                       AS updates,
      Sum(stats.deletes)                       AS deletes
FROM   profiled_statements statement
JOIN   profiled_invocations invocation
ON     invocation.nodeid = statement.nodeid
AND    invocation.session_id = statement.session_id
AND    invocation.event_id = statement.event_id
JOIN   profiling stats
ON     stats.session_id = invocation.session_id
AND    stats.xid = invocation.xid
AND    stats.iid = invocation.iid
WHERE  statement.session_id = @@session_id
AND    statement.statement LIKE <query snippet here>
AND    stats.fragment IS NOT NULL
GROUP BY invocation.iid,
         stats.fragment
ORDER BY invocation.iid,
         stats.fragment;
```

## Getting EXPLAIN output via profiling

Setting profiling = 1 also records the query plan used to execute EXPLAIN:

```
> SELECT Concat(LEFT(REPLACE(query.statement, "\n", " "), 50), "...") AS statement,
      query.program_id,
      plan.operation,
      plan.est_cost,
      plan.est_rows
from   (select distinct statement.statement,
      invocation.program_id
      FROM   system.profiled_statements statement
      JOIN   system.profiled_invocations invocation
      ON     invocation.nodeid = statement.nodeid
      AND    invocation.session_id = statement.session_id
      AND    invocation.event_id = statement.event_id
      WHERE  statement.session_id = @@session_id
      AND    statement.statement LIKE "% test.example JOIN test.example %") query
JOIN   system.profiled_plans plan
USING  (program_id)
ORDER BY query.statement,
         query.program_id,
         plan.plan_order;
```

## Caveats for Runtime Statistics

Enabling runtime statistics globally can lead to system instability.

- This feature is intended for use only at the session level.
- Setting profiling = 1 can lead to longer runtimes due to the overhead of additional statistics collection.

## Making Runtime Statistics Persistent

The runtime statistics collected with the profiling feature are stored in memory and not guaranteed to be available beyond the session. To make them persistent, run the following from a SQL prompt.

```
CREATE DATABASE profiling_hist;
CREATE TABLE profiling_hist.profiled_invocations AS SELECT * FROM system.profiled_invocations;
CREATE TABLE profiling_hist.profiled_transactions AS SELECT * FROM system.profiled_transactions;
CREATE TABLE profiling_hist.profiled_statements AS SELECT * FROM system.profiled_statements;
CREATE TABLE profiling_hist.profiled_til AS SELECT * FROM system.profiled_til;
CREATE TABLE profiling_hist.profiled_plans AS SELECT * FROM system.profiled_plans;
CREATE TABLE profiling_hist.profilng AS SELECT * FROM system.profilng;
```

## Examples of Profiling Output

### Sample Query

Here is a simple table and a few queries being profiled:

```
> CREATE TABLE test.example (e INT PRIMARY KEY);
> INSERT INTO test.example VALUES (1), (2), (3);
> SET SESSION profiling = 1;
> SELECT * FROM test.example;
> SELECT COUNT(*) FROM test.example;
> SELECT COUNT(e) FROM test.example JOIN test.example e2 USING (e) WHERE e > 1;
> SET SESSION profiling = 0;
```

### Get the list of queries profiled

First, we examine the list of statements recorded in this profiling session:

```
> SELECT statement FROM system.profiled_statements WHERE session_id = @@session_id ORDER BY event_id;
```

Here is that query's output, where we see the 4 queries we ran:

```
+-----+
| statement                                                                 |
+-----+
| SELECT * FROM test.example                                               |
| SELECT COUNT(*) FROM test.example                                        |
| SELECT COUNT(e) FROM test.example JOIN test.example e2 USING (e) WHERE e > 1 |
| SET SESSION profiling = 0                                               |
+-----+
4 rows in set (0.01 sec)
```

### Runtime Statistics for a specific Query

Let's focus on the query with the JOIN and look more closely at some runtime statistics:

## Runtime Statistics for the example JOIN query

```
> SELECT Concat(LEFT(REPLACE(Min(statement.statement), "\n", " "), 50), "...") AS statement,
      invocation.iid,
      stats.fragment,
      Sum(stats.fragment_executions)           AS executions,
      Sum(stats.cpu_runtime_ns)                AS cpu_runtime_ns,
      Sum(stats.cpu_waits)                     AS cpu_waits,
      Sum(stats.cpu_waittime_ns)              AS cpu_waittime_ns,
      Count(DISTINCT stats.nodeid)            AS nodes_used,
      Count(DISTINCT stats.nodeid, stats.cpu) AS cpus_used,
      Sum(stats.forwards)                      AS forwards,
      Sum(stats.broadcasts)                   AS broadcasts,
      Sum(stats.rows_read)                    AS rows_read,
      Sum(stats.rows_output)                  AS rows_output,
      Sum(stats.inserts)                      AS inserts,
      Sum(stats.updates)                      AS updates,
      Sum(stats.deletes)                      AS deletes
FROM   profiled_statements statement
JOIN   profiled_invocations invocation
ON     invocation.nodeid = statement.nodeid
AND    invocation.session_id = statement.session_id
AND    invocation.event_id = statement.event_id
JOIN   profiling_stats
ON     stats.session_id = invocation.session_id
AND    stats.xid = invocation.xid
AND    stats.iid = invocation.iid
WHERE  statement.session_id = @@session_id
AND    statement.statement LIKE "% test.example JOIN test.example %"
AND    stats.fragment IS NOT NULL
GROUP BY invocation.iid,
         stats.fragment
ORDER BY invocation.iid,
         stats.fragment;
```

These results were obtained using the sample query available above.

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| statement                                                                 | iid                | fragment | executions |
cpu_runtime_ns | cpu_waits | cpu_waittime_ns | nodes_used | cpus_used | forwards | broadcasts | rows_read |
rows_output | inserts | updates | deletes |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| SELECT COUNT(e) FROM test.example JOIN test.exempl... | 6277791923705589764 | 0 | 1
| 2 | 0 | 0 | 0 | 1 | 3 | 3 | 1 | 0
| 3 | 0 | 0 | 0 | 3 | 3 | 3 | 0 | 3
| SELECT COUNT(e) FROM test.example JOIN test.exempl... | 6277791923705589764 | 1 | 3
| 3 | 0 | 0 | 0 | 3 | 3 | 3 | 0 | 3
| 0 | 0 | 0 | 0 | 3 | 3 | 0 | 0 | 3
| SELECT COUNT(e) FROM test.example JOIN test.exempl... | 6277791923705589764 | 2 | 3
| 0 | 0 | 0 | 0 | 3 | 3 | 0 | 0 | 3
| 0 | 0 | 0 | 0 | 3 | 3 | 0 | 0 | 3
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

## Getting EXPLAIN Output

Instead of running EXPLAIN, you can also run the following query to get the EXPLAIN output for a given query.

```

> SELECT Concat(LEFT(REPLACE(query.statement, "\n", " "), 50), "...") AS statement,
      query.program_id,
      plan.operation,
      plan.est_cost,
      plan.est_rows
from   (select distinct statement.statement,
      invocation.program_id
      FROM      system.profiled_statements statement
      JOIN      system.profiled_invocations invocation
      ON        invocation.nodeid = statement.nodeid
      AND       invocation.session_id = statement.session_id
      AND       invocation.event_id = statement.event_id
      WHERE     statement.session_id = @@session_id
      AND       statement.statement LIKE "% test.example JOIN test.example %") query
JOIN   system.profiled_plans plan
USING  (program_id)
ORDER BY query.statement,
      query.program_id,
      plan.plan_order;

```

```

+-----+-----+-----+-----+
+-----+-----+-----+-----+
| statement                                | program_id |
operation                                | est_cost | est_rows |
+-----+-----+-----+-----+
+-----+-----+-----+-----+
| SELECT COUNT(e) FROM test.example JOIN test.exempl... | 6277618986029830146 | row_count
"expr0"                                | 17.88      | 1.00      |
| SELECT COUNT(e) FROM test.example JOIN test.exempl... | 6277618986029830146 |
nljoin                                | 17.78      | 1.02      |
| SELECT COUNT(e) FROM test.example JOIN test.exempl... | 6277618986029830146 |
stream_combine                          | 13.31      | 1.01      |
| SELECT COUNT(e) FROM test.example JOIN test.exempl... | 6277618986029830146 | index_scan 2 := example.
__idx_example__PRIMARY                   | 4.07       | 0.34      |
| SELECT COUNT(e) FROM test.example JOIN test.exempl... | 6277618986029830146 | filter (1.e > param
(0))                                     | 4.43       | 1.01      |
| SELECT COUNT(e) FROM test.example JOIN test.exempl... | 6277618986029830146 | index_scan 1 := example.
__idx_example__PRIMARY, e = 2.e | 4.41       | 1.00      |
+-----+-----+-----+-----+
+-----+-----+-----+-----+
6 rows in set (0.01 sec)

```

## Additional Performance Measurements

The following are examples of various statistics and their associated VRELS.

During execution of a profiled invocation, we keep separate stats for every fragment of the query on every CPU used during execution. These measurements are available both during and after execution, via the profiling virtual relation.

### system.profiling

system.profiling is a mapping from (node, CPU, session, transaction, invocation, query fragment) key to corresponding stats counts.

```

> SELECT nodeid,
       cpu,
       session_id,
       xid,
       iid,
       fragment,
       operation_id,
       replica,
       fragment_executions,
       forwards,
       cpu_runtime_ns,
       bm_fixes,
       rows_read,
       rows_output
from   system.profiling
WHERE  iid = 6277791923705589764
ORDER BY nodeid,
         cpu,
         session_id,
         xid,
         iid,
         fragment,
         operation_id,
         replica;

```

nodeid	cpu	session_id	xid	iid	fragment	operation_id	replica	fragment_executions	forwards	cpu_runtime_ns	bm_fixes	rows_read	rows_output
1	3	3246082	6277791923697008644	6277791923705589764	1	NULL	NULL						
		1	1	1	2	1	0						
1	4	3246082	6277791923697008644	6277791923705589764	2	NULL	NULL						
		1	0	0	2	1	0						
2	1	3246082	6277791923697008644	6277791923705589764	0	NULL	NULL						
		0	0	0	0	0	0						
2	1	3246082	6277791923697008644	6277791923705589764	1	NULL	NULL						
		1	1	1	2	1	0						
2	2	3246082	6277791923697008644	6277791923705589764	0	NULL	NULL						
		0	0	0	0	0	0						
2	2	3246082	6277791923697008644	6277791923705589764	2	NULL	NULL						
		1	0	0	2	1	0						
2	7	3246082	6277791923697008644	6277791923705589764	0	NULL	NULL						
		1	3	2	0	0	3						
3	3	3246082	6277791923697008644	6277791923705589764	1	NULL	NULL						
		1	1	1	2	1	0						
3	4	3246082	6277791923697008644	6277791923705589764	2	NULL	NULL						
		1	0	0	2	1	0						

9 rows in set (0.00 sec)

Querying system.profiling will often require a GROUP BY clause, depending on the dimensions used.

## By CPU

```

> SELECT  nodeid,
          cpu,
          Sum(cpu_runtime_ns),
          Sum(cpu_waits),
          Sum(cpu_waittime_ns)
from      system.profiling
WHERE     iid = 6277791923705589764
GROUP BY nodeid,
          cpu
ORDER BY nodeid,
          cpu;

```

```

+-----+-----+-----+-----+-----+
| nodeid | cpu | Sum(cpu_runtime_ns) | Sum(cpu_waits) | Sum(cpu_waittime_ns) |
+-----+-----+-----+-----+-----+
|      1 |  3 |                1 |                0 |                0 |
|      1 |  4 |                0 |                0 |                0 |
|      2 |  1 |                1 |                0 |                0 |
|      2 |  2 |                0 |                0 |                0 |
|      2 |  7 |                2 |                0 |                0 |
|      3 |  3 |                1 |                0 |                0 |
|      3 |  4 |                0 |                0 |                0 |
+-----+-----+-----+-----+-----+
7 rows in set (0.01 sec)

```

## History

In addition to performance measurements, we also keep a history of all profiled invocations in the session. This history includes the statement, plan, and query program for the invocation. The following four vrels are available for use.

### profiled\_invocations

system.profiled\_invocations holds a history of every profiled invocation. You will likely join to this table whenever querying multiple profiling vrels.

```

> SELECT *
FROM    system.profiled_invocations
WHERE   session_id = @@session_id
ORDER  BY iid;

```

```

+-----+-----+-----+-----+-----+-----+-----+-----+
| nodeid | session_id | event_id | program_id | xid | iid | fanout |
| priority | started | elapsed_ms | status | | | |
+-----+-----+-----+-----+-----+-----+-----+
|      2 | 3246082 | 5 | 6277618985999765506 | 6277791923645485060 | 6277791923679819780 | 7 |
|      0 | 2016-04-25 22:07:49 | 0 | Ok: 3 rows selected | | | |
|      2 | 3246082 | 6 | 6277618986012650498 | 6277791923684120580 | 6277791923688409092 | 7 |
|      0 | 2016-04-25 22:07:49 | 1 | Ok: 1 rows selected | | | |
|      2 | 3246082 | 7 | 6277618986029830146 | 6277791923697008644 | 6277791923705589764 | 7 |
|      0 | 2016-04-25 22:07:49 | 2 | Ok: 1 rows selected | | | |
|      2 | 3246082 | 8 | 6264986681145192450 | 6277791941108854788 | 6277791941108858884 | 0 |
|      0 | 2016-04-25 22:07:53 | 0 | Ok: 1 rows selected | | | |
+-----+-----+-----+-----+-----+-----+-----+
4 rows in set (0.01 sec)

```

### profiled\_transactions

```

> SELECT *
FROM system.profiled_transactions
WHERE session_id = @@session_id
ORDER BY xid;
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+
| nodeid | session_id | xid          | isolation      | started          | elapsed_ms | committed |
commit_id |
+-----+-----+-----+-----+-----+-----+-----+-----+
|      2 |    3246082 | 6277791923645485060 | REPEATABLE-READ | 2016-04-25 22:07:49 |      8 |      1 |
6277791923645485060 |
|      2 |    3246082 | 6277791923684120580 | REPEATABLE-READ | 2016-04-25 22:07:49 |      2 |      1 |
6277791923684120580 |
|      2 |    3246082 | 6277791923697008644 | REPEATABLE-READ | 2016-04-25 22:07:49 |      4 |      1 |
6277791923697008644 |
|      2 |    3246082 | 6277791941108854788 | REPEATABLE-READ | 2016-04-25 22:07:53 |      0 |      1 |
6277791941108854788 |
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+
4 rows in set (0.00 sec)

```

## profiled\_statements

Statements have a many-to-many relationship with transactions, and a one-to-many relationship with invocations, so we break statements into their own table. Usually, this is the table you will examine first when analyzing profiled queries.

```

> SELECT *
FROM system.profiled_statements
WHERE session_id = @@session_id
ORDER BY event_id;
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+
| nodeid | session_id | event_id | statement
| started          | elapsed_ms | finished |
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+
|      2 |    3246082 |      5 | SELECT * FROM test.example
| 2016-04-25 22:07:49 |      8 |      1 |
|      2 |    3246082 |      6 | SELECT COUNT(*) FROM test.example
| 2016-04-25 22:07:49 |      2 |      1 |
|      2 |    3246082 |      7 | SELECT COUNT(e) FROM test.example JOIN test.example e2 USING (e) WHERE e > 1
| 2016-04-25 22:07:49 |      4 |      1 |
|      2 |    3246082 |      8 | SET SESSION profiling = 0
| 2016-04-25 22:07:53 |      0 |      1 |
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+
4 rows in set (0.00 sec)

```

event\_id is the event ID used by SBR.

## profiled\_plans

We also save the plan of each profiled invocation by program ID, so that you don't have to run EXPLAIN for each query. Unlike the other profiling history tables, system.profiled\_plans has multiple rows per history item, one for each row in EXPLAIN. This makes it easier to join to system.profiling.



```

> SELECT *
FROM   system.profiled_plans
WHERE  session_id = @@session_id
ORDER  BY program_id,
        plan_order;

```

```

+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+
| nodeid | session_id | program_id          | plan_order | operation_id |
operation |           |                   |            |             |
| est_cost | est_rows |
+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+
|      2 | 3246082 | 6264986681145192450 | 0 | 4 | compute expr0 := param
(1)      |         |                   |   |   |
|         |         |                   | 0.02 | 0.10 |
|      2 | 3246082 | 6264986681145192450 | 1 | 3 | row_limit LIMIT := param
(0)      |         |                   |   |   |
|         |         |                   | 0.02 | 0.10 |
|      2 | 3246082 | 6264986681145192450 | 2 | 2 |
dual     |         |                   |   |   |
|         |         |                   | 0.02 | 1.00 |
|      2 | 3246082 | 6277618985999765506 | 0 | 5 |
stream_combine |         |                   |   |   |
|         |         |                   | 13.31 | 1.01 |
|      2 | 3246082 | 6277618985999765506 | 1 | 4 | index_scan 1 := example.
__idx_example_PRIMARY |         |                   | 4.07 | 0.34 |
|      2 | 3246082 | 6277618986012650498 | 0 | 8 | row_count
"expr1" |         |                   | 13.43 | 1.00 |
|      2 | 3246082 | 6277618986012650498 | 1 | 7 |
stream_combine |         |                   | 13.32 | 1.01 |
|         |         |                   |   |   |
|      2 | 3246082 | 6277618986012650498 | 2 | 6 | compute expr0 := param
(0)      |         |                   | 4.07 | 0.34 |
|         |         |                   |   |   |
|      2 | 3246082 | 6277618986012650498 | 3 | 5 | index_scan 1 := example.
__idx_example_PRIMARY |         |                   | 4.07 | 0.34 |
|      2 | 3246082 | 6277618986029830146 | 0 | 17 | row_count
"expr0" |         |                   | 17.88 | 1.00 |
|      2 | 3246082 | 6277618986029830146 | 1 | 16 |
nljoin |         |                   | 17.78 | 1.02 |
|         |         |                   |   |   |
|      2 | 3246082 | 6277618986029830146 | 2 | 15 |
stream_combine |         |                   | 13.31 | 1.01 |
|         |         |                   |   |   |
|      2 | 3246082 | 6277618986029830146 | 3 | 14 | index_scan 2 := example.
__idx_example_PRIMARY |         |                   | 4.07 | 0.34 |
|         |         |                   |   |   |
|      2 | 3246082 | 6277618986029830146 | 4 | 9 | filter (1.e > param
(0))      |         |                   | 4.43 | 1.01 |
|         |         |                   |   |   |
|      2 | 3246082 | 6277618986029830146 | 5 | 8 | index_scan 1 := example.
__idx_example_PRIMARY, e = 2.e | 4.41 | 1.00 |
+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+
15 rows in set (0.00 sec)

```