EXPLAIN ANALYZE Is All We Need

But what if we could always get actual rows without it?

Hironobu SUZUKI

Agenda

1. What if we could always get actual rows?
2. Discussion: How to get actual rows without EXPLAIN ANALYZE
3. Conclusion

Who am I

Name

Hironobu SUZUKI

Ex-Director of JPUG

Author

"The Internals of PostgreSQL" https://www.interdb.jp/pg/

"The Engineer's Guide To Deep Learning" https://www.interdb.jp/dl/

✅ Perceptrons to Transformers with scratch code and high school math

✅ 99 figures included ✅ Ready to read real papers

EXPLAIN ANALYZE

shows actual rows and other detailed runtime statistics:

```
testdb=# EXPLAIN ANALYZE SELECT count(*) FROM test1 AS a, test3 AS c WHERE a.id = c.id;
                                            QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=498.79..498.80 rows=1 width=8) (actual time=5.616..5.618 rows=1.00 loops=1)
   Buffers: shared hit=373
   ->  Merge Join  (cost=0.73..485.60 rows=5274 width=0) (actual time=0.030..4.774 rows=10000.00 loops=1)
         Merge Cond: (a.id = c.id)
         Buffers: shared hit=373
         ->  Index Only Scan using test1_id_idx on test1 a  (cost=0.42..4043.36 rows=155000 width=4) (actual
time=0.011..1.477 rows=10001.00 loops=1)
               Heap Fetches: 452
               Index Searches: 1
               Buffers: shared hit=355
         ->  Index Only Scan using test3_pkey on test3 c  (cost=0.28..145.28 rows=5000 width=4) (actual
time=0.014..0.775 rows=5000.00 loops=1)
               Heap Fetches: 452
               Index Searches: 1
               Buffers: shared hit=18
 Planning:
   Buffers: shared hit=10
 Planning Time: 0.323 ms
 Execution Time: 5.670 ms
(17 rows)
```

What if

we could always get actual rows without EXPLAIN ANALYZE?

What if

we could always get actual rows without EXPLAIN ANALYZE?

Pros:

1. Detection of Cardinality Estimation Errors

    > Useful for improving the query optimizer

2. Query Progress Monitoring

3. Anomaly Detection

Cons:

?

What if

we could always get actual rows?

Pros:

1. Detection of Cardinality Estimation Errors > For Optimizer Improvement

   Enables incremental improvements in cardinality estimation using Machine Learning and Deep Learning methods.

   Examples:

   - "Adaptive Query Optimizer" by PostgresPro: https://github.com/postgrespro/aqo

     > https://arxiv.org/pdf/1711.08330

   - pg_plan_inspector: https://github.com/s-hironobu/pg_plan_inspector

     > Inspired by pg_plan_advsr.

   - Google Scholar shows thousands papers:

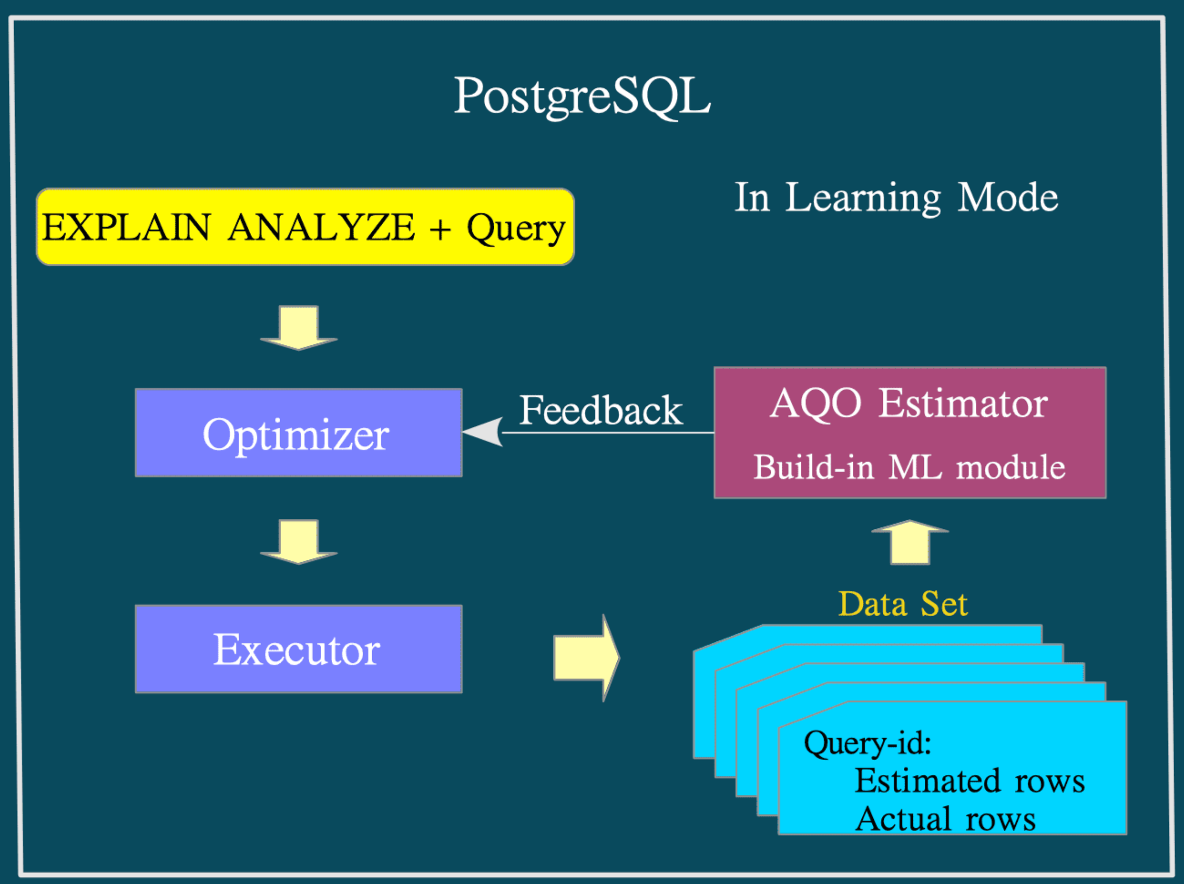
     > Keywords: "Machine OR Deep Learning Cardinality Estimation"

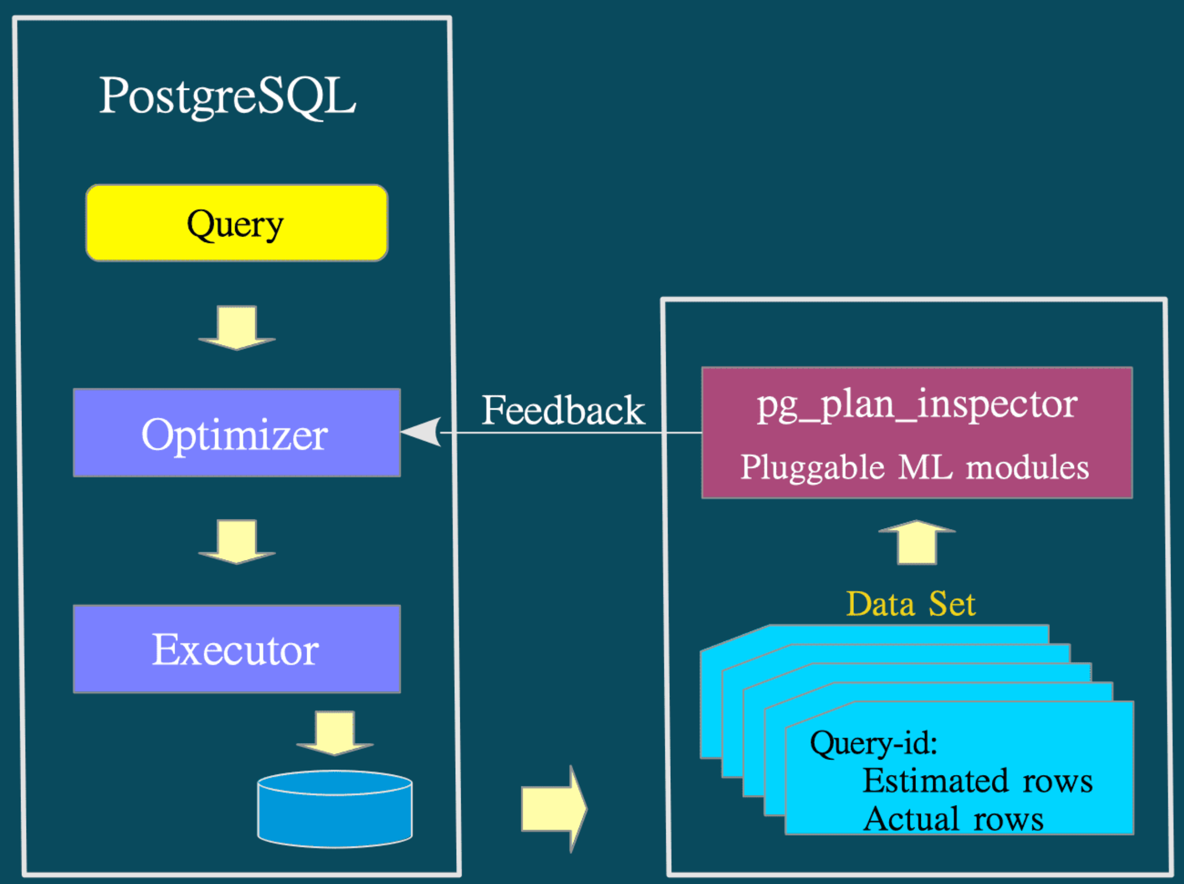
Adaptive Query Optimization (AQO)

- EXPLAIN ANALYZE must be explicitly issued in learning mode.
- Supports a Build-in ML module described in https://arxiv.org/pdf/1711.08330
- Requires patching PostgreSQL.

pg_plan_inspector

- No EXPLAIN ANALYZE or learning mode needed; the instrument module is invoked automatically.
- Supports pluggable ML modules (currently only linear regression).
- Pure extension with external model support.





What if

we could always get actual rows?

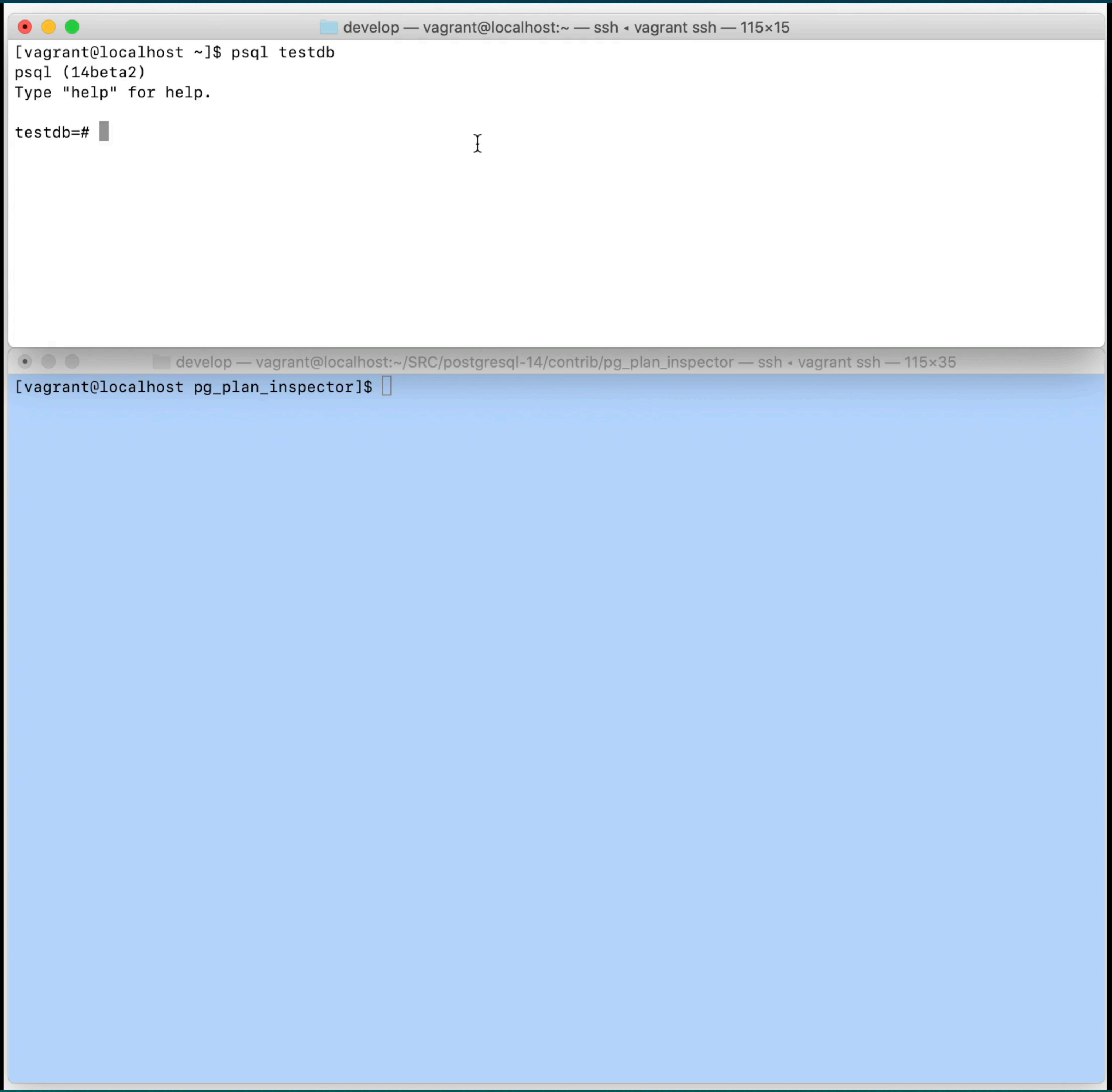
Pros:

2. Query Progress Monitoring

   Enables real-time monitoring of query execution progress.

   > pg_plan_inspector provides this capability.

Query Progress Monitoring



Original video: https://github.com/s-hironobu/pg_plan_inspector

What if

we could always get actual rows and other runtime statistics?

Pros:

3. Anomaly Detection

   Sudden changes in runtime statistics may indicate data corruption or other anomalies.

   > A first step toward achieving Observability.

   Observability is the ability to understand a system's internal state through metrics (quantitative data) and logs (event data).

   > OpenTelemetry is a standard for collecting metrics and logs.

What if

we could always get actual rows?

Pros:

1. Detection of Cardinality Estimation Errors
2. Query Progress Monitoring
3. Anomaly Detection

> At least counting the actual rows would be very useful.

> A step towards addressing the technology trend of Observability.

What if

we could always get actual rows?

Cons:

- Instrumentation Overhead

```
testdb=# SELECT count(*) FROM test1 AS a, test2 AS b, test3 AS c WHERE a.id = c.id;
  count
-----------
 500000000
(1 row)

Time: 16672.551 ms (00:16.673)
```

```
testdb=# EXPLAIN (ANALYZE TRUE, BUFFERS FALSE, TIMING FALSE) SELECT count(*) FROM test1 AS a,
test2 AS b, test3 AS c WHERE a.id = c.id;
                                  QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------
--
 Aggregate  (cost=3969476.05..3969476.06 rows=1 width=8) (actual rows=1.00 loops=1)
   ->  Nested Loop  (cost=0.83..3308101.05 rows=264550000 width=0) (actual rows=500000000.00
loops=1)
         ->  Seq Scan on test2 b  (cost=0.00..722.00 rows=50000 width=0) (actual rows=50000.00 loops=1)
         ->  Materialize  (cost=0.83..517.28 rows=5291 width=0) (actual rows=10000.00 loops=50000)
               Storage: Memory  Maximum Storage: 363kB
               ->  Merge Join  (cost=0.83..490.82 rows=5291 width=0) (actual rows=10000.00 loops=1)
                     Merge Cond: (a.id = c.id)
                     ->  Index Only Scan using test1_id_idx on test1 a  (cost=0.42..4047.35 rows=155000 width=4)
(actual rows=10001.00 loops=1)
                           Heap Fetches: 678
                           Index Searches: 1
                     ->  Index Only Scan using test3_pkey on test3 c  (cost=0.28..146.28 rows=5000 width=4)
(actual rows=5000.00 loops=1)
                           Heap Fetches: 678
                           Index Searches: 1
 Planning Time: 2.828 ms
 Execution Time: 21530.715 ms
(15 rows)

Time: 21539.742 ms (00:21.540)
```

```
testdb=# EXPLAIN (ANALYZE TRUE, BUFFERS FALSE, TIMING TRUE) SELECT count(*) FROM test1 AS a, test2 AS b,
test3 AS c WHERE a.id = c.id;
                                  QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------
----------
 Aggregate  (cost=3969476.05..3969476.06 rows=1 width=8) (actual time=65191.610..65191.622 rows=1.00 loops=1)
   ->  Nested Loop  (cost=0.83..3308101.05 rows=264550000 width=0) (actual time=0.084..42770.779
rows=500000000.00 loops=1)
         ->  Seq Scan on test2 b  (cost=0.00..722.00 rows=50000 width=0) (actual time=0.016..6.777 rows=50000.00
loops=1)
         ->  Materialize  (cost=0.83..517.28 rows=5291 width=0) (actual time=0.000..0.312 rows=10000.00
loops=50000)
               Storage: Memory  Maximum Storage: 363kB
               ->  Merge Join  (cost=0.83..490.82 rows=5291 width=0) (actual time=0.061..6.881 rows=10000.00 loops=1)
                     Merge Cond: (a.id = c.id)
                     ->  Index Only Scan using test1_id_idx on test1 a  (cost=0.42..4047.35 rows=155000 width=4) (actual
time=0.026..2.217 rows=10001.00 loops=1)
                           Heap Fetches: 678
                           Index Searches: 1
                     ->  Index Only Scan using test3_pkey on test3 c  (cost=0.28..146.28 rows=5000 width=4) (actual
time=0.029..1.173 rows=5000.00 loops=1)
                           Heap Fetches: 678
                           Index Searches: 1
 Planning Time: 0.500 ms
 Execution Time: 65191.858 ms
(15 rows)

Time: 65193.945 ms (01:05.194)
```

Note: gprof only profiles user-mode code and does not capture time spent executing system calls in kernel mode.

```
$ gprof ./bin/postgres data/gprof/3158509/gmon.out
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 20.66      7.14     7.14 1000010003     0.00     0.00  ExecInterpExpr
 16.52     12.85     5.71                             _init
 11.49     16.82     3.97 500000001     0.00     0.00  ExecNestLoop
  7.61     19.45     2.63 500040016     0.00     0.00  tuplestore_gettupleslot
  6.39     21.66     2.21 1000125014     0.00     0.00  InstrStartNode
  6.28     23.83     2.17 500050000     0.00     0.00  tuplestore_ateof
  5.24     25.64     1.81 500050000     0.00     0.00  ExecMaterial
  5.21     27.44     1.80 1000125014     0.00     0.00  InstrStopNode
  4.05     28.84     1.40 500040016     0.00     0.00  tuplestore_gettuple
  3.70     30.12     1.28 1000125015     0.00     0.00  ExecProcNodeInstr
  2.69     31.05     0.93 1000125033     0.00     0.00  MemoryContextReset
  2.66     31.97     0.92 500000001     0.00     0.00  fetch_input_tuple
  2.49     32.83     0.86        2     0.43     2.53  ExecAgg
  2.43     33.67     0.84 499990015     0.00     0.00  ExecStoreMinimalTuple
  1.56     34.21     0.54 500025044     0.00     0.00  tts_virtual_clear
  0.93     34.53     0.32 500000000     0.00     0.00  int8inc
  0.03     34.54     0.01    50007     0.00     0.00  InstrEndLoop
  0.03     34.55     0.01    10001     0.00     0.00  ExecMergeJoin
  0.03     34.56     0.01        3     0.00     0.00  GetCurrentFDWTuplestore
  0.00     34.56     0.00   150018     0.00     0.00  tts_minimal_clear
  0.00     34.56     0.00    51575     0.00     0.00  ExecStoreBufferHeapTuple
  0.00     34.56     0.00    50003     0.00     0.00  heap_getnextslot
  0.00     34.56     0.00    50003     0.00     0.00  heapgettup_pagemode
  0.00     34.56     0.00    50001     0.00     0.00  ExecSeqScan
  0.00     34.56     0.00    50001     0.00     0.00  SeqNext
  0.00     34.56     0.00    50000     0.00     0.00  ExecReScan
```

```c
/*
 * ExecProcNode wrapper that performs instrumentation
 * calls.  By keeping  this a separate function, we avoid
 * overhead in the normal case where  no instrumentation
 *  is wanted.
*/
static TupleTableSlot *
ExecProcNodeInstr(PlanState *node)
{
    TupleTableSlot *result;

    InstrStartNode(node->instrument);

    result = node->ExecProcNodeReal(node);

    InstrStopNode(node->instrument, TupIsNull(result) ? 0.0 : 1.0);

    return result;
}
```

What if

we could always get actual rows?

Cons:

- Instrumentation Overhead
  - The Instrument module collects statistics during query execution.

    Example: In SeqScan, InstrStartNode() and InstrStopNode() are called for each row.

  - The instrument module can consume 25-30% of total execution time, when counting rows.

Note

Most of the overhead comes from calling the system clock (e.g., clock_gettime()) when the TIMING option is set to TRUE.

```c
/* Entry to a plan node */
void
InstrStartNode(Instrumentation *instr)
{
    if (instr->need_timer &&
        !INSTR_TIME_SET_CURRENT_LAZY(instr->starttime))
        elog(ERROR, "InstrStartNode called twice in a row");

    /* save buffer usage totals at node entry, if needed */
    if (instr->need_bufusage)
        instr->bufusage_start = pgBufferUsage;

    if (instr->need_walusage)
        instr->walusage_start = pgWalUsage;
}


/* Finish a run cycle for a plan node */
void
InstrEndLoop(Instrumentation *instr)
{
    double      totaltime;

    /* Skip if nothing has happened, or already shut down */
    if (!instr->running)
        return;

    if (!INSTR_TIME_IS_ZERO(instr->starttime))
        elog(ERROR, "InstrEndLoop called on running node");

    /* Accumulate per-cycle statistics into totals */
    totaltime = INSTR_TIME_GET_DOUBLE(instr->counter);

    instr->startup += instr->firsttuple;
    instr->total += totaltime;
    instr->ntuples += instr->tuplecount;
    instr->nloops += 1;

    /* Reset for next cycle (if any) */
    instr->running = false;
    INSTR_TIME_SET_ZERO(instr->starttime);
    INSTR_TIME_SET_ZERO(instr->counter);
    instr->firsttuple = 0;
    instr->tuplecount = 0;
}
```

```c
/* Exit from a plan node */
void
InstrStopNode(Instrumentation *instr, double nTuples)
{
    double      save_tuplecount = instr->tuplecount;
    instr_time  endtime;

    /* count the returned tuples */
    instr->tuplecount += nTuples;

    /* let's update the time only if the timer was requested */
    if (instr->need_timer)
    {
        if (INSTR_TIME_IS_ZERO(instr->starttime))
            elog(ERROR, "InstrStopNode called without start");

        INSTR_TIME_SET_CURRENT(endtime);
        INSTR_TIME_ACCUM_DIFF(instr->counter, endtime, instr->starttime);

        INSTR_TIME_SET_ZERO(instr->starttime);
    }

    /* Add delta of buffer usage since entry to node's totals */
    if (instr->need_bufusage)
        BufferUsageAccumDiff(&instr->bufusage,
                             &pgBufferUsage, &instr->bufusage_start);

    if (instr->need_walusage)
        WalUsageAccumDiff(&instr->walusage,
                          &pgWalUsage, &instr->walusage_start);

    /* Is this the first tuple of this cycle? */
    if (!instr->running)
    {
        instr->running = true;
        instr->firsttuple = INSTR_TIME_GET_DOUBLE(instr->counter);
    }
    else
    {
        /*
         * In async mode, if the plan node hadn't emitted any tuples before,
         * this might be the first tuple
         */
        if (instr->async_mode && save_tuplecount < 1.0)
            instr->firsttuple = INSTR_TIME_GET_DOUBLE(instr->counter);
    }
}
```

Only a very small portion of these C functions relates to actual row counting.

```
/* Entry to a plan node */
void
InstrStartNode(Instrumentation *instr)
{
    Timer
    Buffer
    WAL
}

/* Finish a run cycle for a plan node */
void
InstrEndLoop(Instrumentation *instr)
{

    /* Skip if nothing has happened, or already shut down */
    if (!instr->running)
        return;
    Timer
    instr->ntuples += instr->tuplecount;
    instr->nloops += 1;

    /* Reset for next cycle (if any) */
    instr->running = false;
    Timer
    instr->tuplecount = 0;
}
```

```
/* Exit from a plan node */
void
InstrStopNode(Instrumentation *instr, double nTuples)
{
    Timer

    /* count the returned tuples */
    instr->tuplecount += nTuples;

    Timer
    Buffer
    WAL
    /* Is this the first tuple of this cycle? */
    if (!instr->running)
    {
        instr->running = true;

    }
    else
    {
        Timer
    }
}
```

What if

we could always get actual rows?

Pros:

1. Detection of Cardinality Estimation Errors

   > Useful for improving the query optimizer.

2. Query Progress Monitoring

   > Enables real-time monitoring of query execution progress.

3. Anomaly Detection

   > A first step toward achieving Observability.

Cons:

- Instrumentation Overhead

  > The instrument module can consume 25-30% of total execution time, when counting rows.

Discussion

How to get actual rows without EXPLAIN ANALYZE

Discussion

How to get actual rows without EXPLAIN ANALYZE

How to reduce the overhead of counting rows

Discussion

Reducing the overhead of counting rows

Out of Scope:

Actual time, WAL, BufferUsage.

Initial Ideas:

1. Systematic Sampling (Throttled Sampling)

2. Modify All Plan Nodes

Discussion

Reducing the overhead of counting rows

1. Systematic Sampling (Throttled Sampling) > We don't need to be perfect.

   Selects every k-th element from a stream (e.g., every 100th row)

   - Theoretically, overhead would be 1/k (e.g., 1/100).

     > However, the accuracy would also decrease.

   - Practically, it might not be possible to reduce the number of ExecProcNodeInstr() calls.

     > The overhead might shift to the systematic sampling condition check.

   - While this is a poor approach, the concept of statistical sampling may be useful in other contexts (e.g., estimating actual runtime).

Discussion

Reducing the overhead of counting rows

2. Modify All Plan Nodes (Without using Instrument module)

   Count and save the number of processed rows in each Plan Node.

   - Feasibility study not yet conducted.
   - When EXPLAIN ANALYZE is used, actual rows can be passed to Instrument nodes, avoiding overlapping functionality.

Discussion

Reducing the overhead of counting rows

Initial Ideas:

1. Systematic Sampling (Throttled Sampling)
2. Modify All Plan Nodes (Without using Instrument module)

Any other ideas?

Discussion

Other Point(s):

1. How can the results be read?
   - By expanding auto_explain.
   - By storing tables.
   - By creating a command (function):

     It shows the plan of the specified PID, sharing it via DSM.

     > Optionally, it could also write the result to the log.

     > It may be easier to gain community acceptance if we implement a "show plan" feature first, and then an "actual row count" feature.

2. ?

Conclusion

What if we could always get actual rows without EXPLAIN ANALYZE?

Benefits:

- Optimizer Improvement

  Providing actual rows enables AI engineers to enhance the query optimizer.

  > No fully satisfactory ML/DL methods currently exist.

  I suspect this is because we don't provide sufficient internal statistics. Implementing this feature could help stimulate further research.

- Query Progress Monitoring
- Observability

Thank You