Run Simple Query Faster....

Dilip Kumar |
2016.05.19

PgCon 2016

Contents

- Background
- What is Simple Query
- Instruction Measurement Of Simple Query
- Instruction Analysis
- Simple Query Optimization Solution
- Performance Results
- Future Optimization Plan

Background

Generic Executor Infrastructure:

❑Current executor is designed to support wide range of queries.

❑Often simple query ends up processing many extra instructions.

- Multi level of processing nodes, for example, update and insert need two level of processing nodes.
- Data structures at different levels.
- Decision making infrastructures.
- Initialization is done for every execution.

What is Simple Query ?

In our experiments, we call a query as simple query if it has following properties:

- ❑Simple target list without any function call or sub-query.
- ❑Simple Qualification clause.
- ❑No Joins.
- ❑No Aggregates.

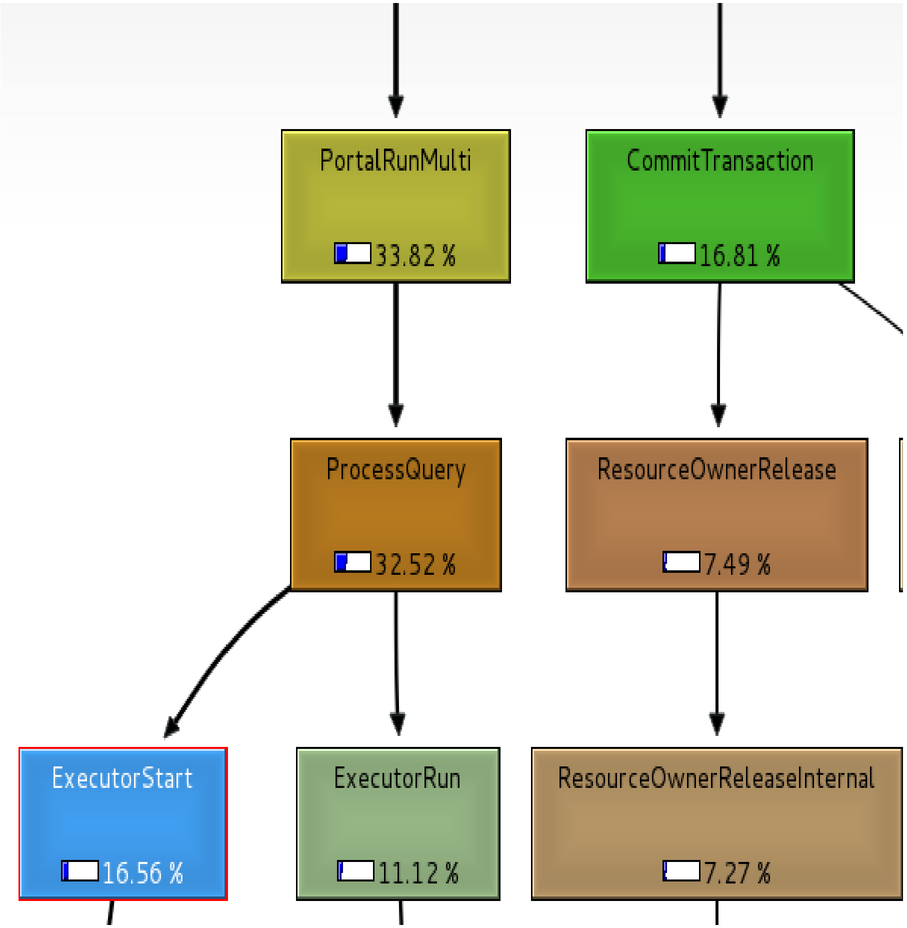
Instructions Measurement for Simple Query

Experiment:

- Execute INSERT query of **pgbench_history** table.
- Measured instructions using callgrind tool, for execution of 1000 transactions.

Results

- Right side call graph shows, instructions for a Insert query.
- Executor is taking almost 28% of total instructions.



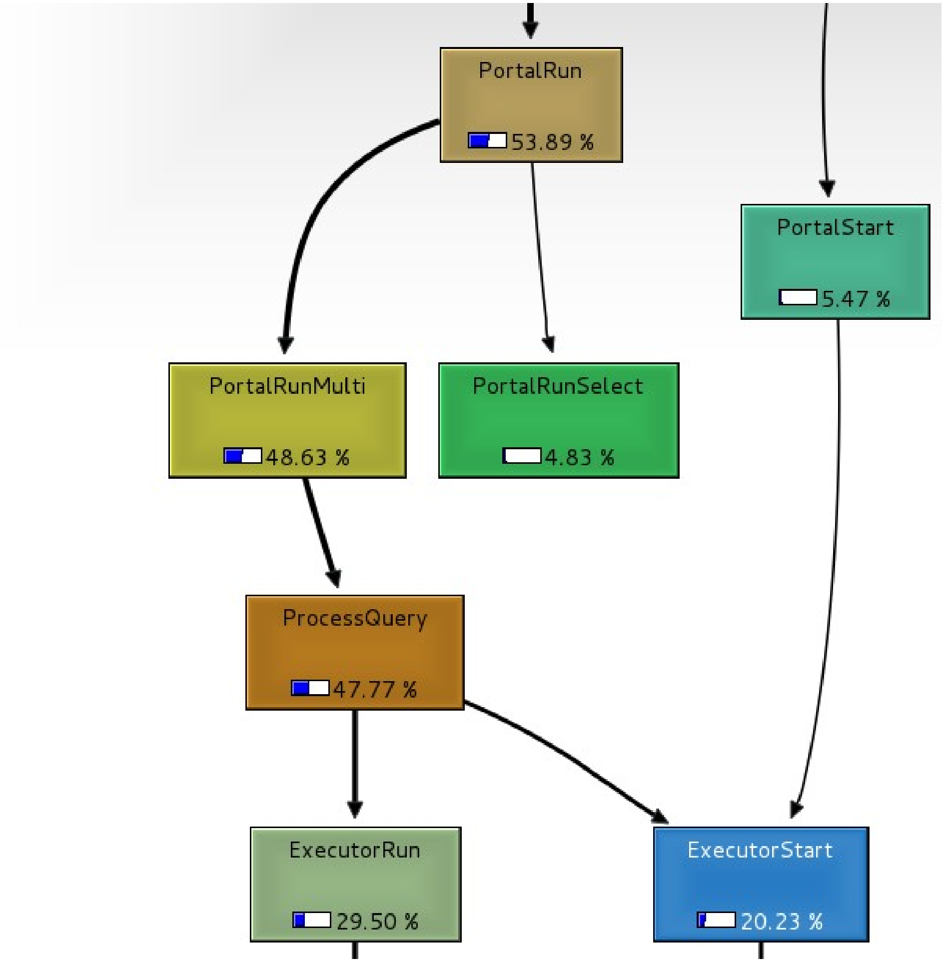
Instructions Measurement for Simple Query

Experiment:

- Execute simple_update of PGBENCH.
- Measured instructions using callgrind, for execution of 1000 transactions.

Results

- Right side call graph shows, instruction for simple_update.
- Executor is taking ~50% of total instructions.



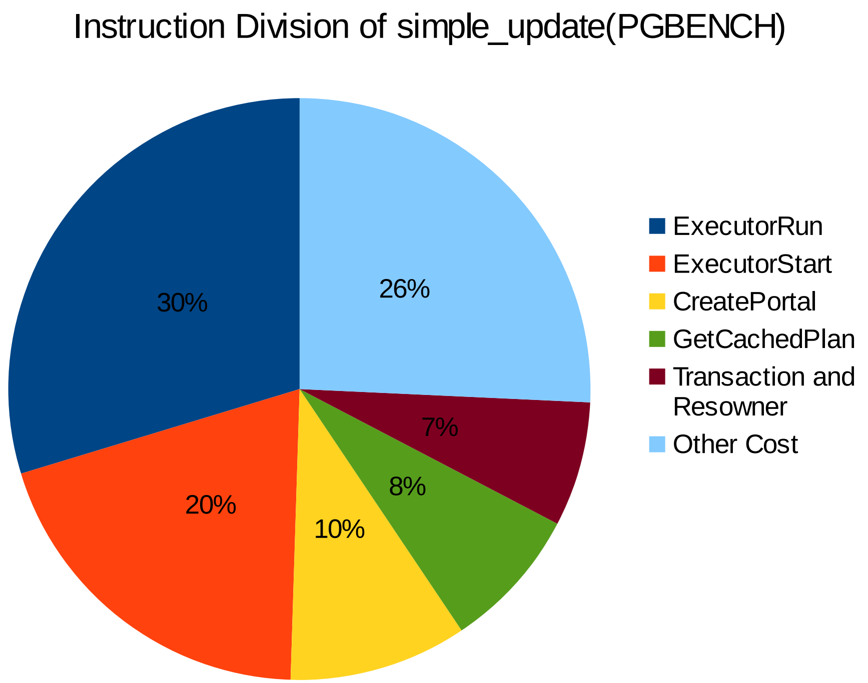
Instructions Analysis of Query Execution

Experiment:

- Executed simple_update of PGBENCH test, and measured instructions for 1000 transactions using callgrind.

Observation:

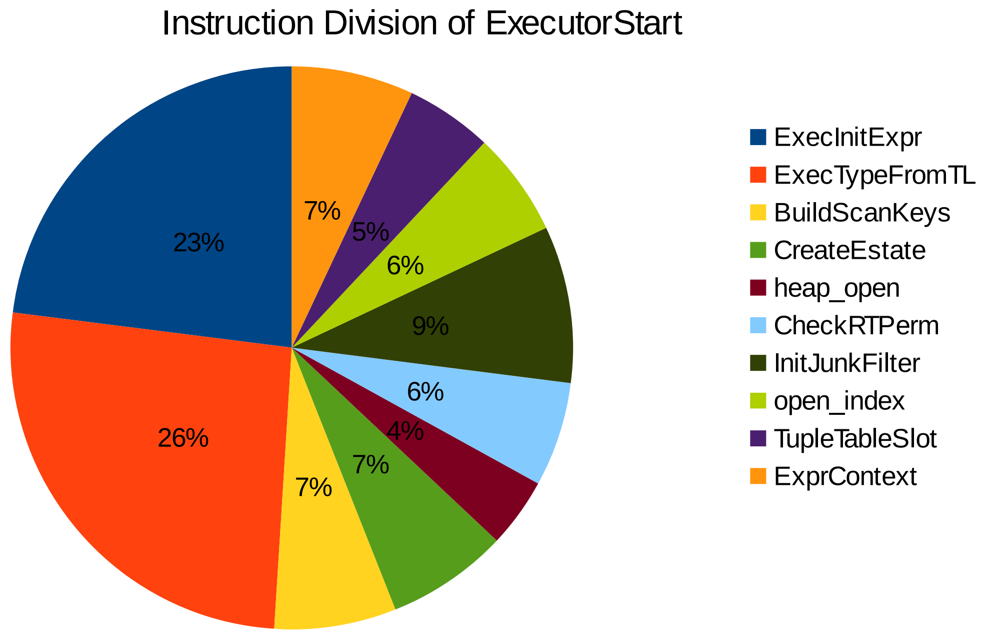
- Below chart shows, instruction division of query execution.
- ~50% instructions are from ExecutorRun and ExecutorStart.



Instructions Analysis of ExecutorStart

Observation:

- In continuation to previous experiment we further divided ExecutorStart insturctions.
- Here we are more interested in ExecutorStart instructions because, most of the initialization operations in ExecutorStart can be done only once and further reused in subsequent execution.
- Here we can see ExecInitExpr and ExecTypeFromTL are main contributors.
- These inputs are used for deriving our optimization.

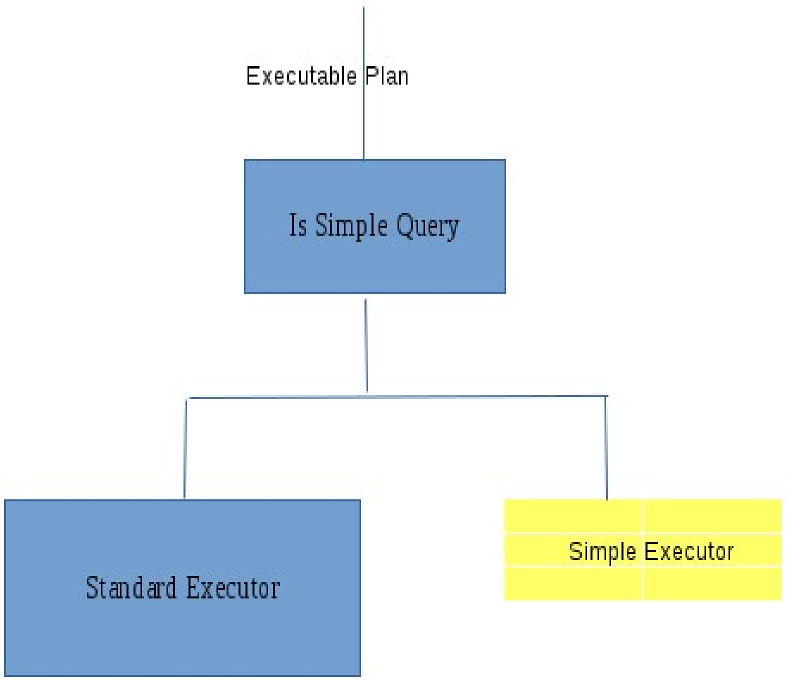


Why Especially Prepared query

- In previous slides we have seen that ExecutorStart is taking > 20% of total and >40% of executor instructions.
- If a query is prepared query then we can reuse executor tree for subsequent execution of same plan and save complete instructions of ExecutorStart.
- Non prepared queries are random, so we can not reuse any previous state, but we can save some infrastructure cost.

Implementation Idea

- Special attention for simple queries, because they don't need very generic infrastructures.
- Provide a simple_executor hook using contrib module.
- If query is identified as simple then execute using simple executor, otherwise fall back to standard executor.



ENTERPRISEDB

Optimization Experiment on Simple Query

- ❑ Push Down Scan key
- ❑ Save Expression Initialization for targetlist and qual
- ❑Save Scan slot
- ❑Save Executor State
- ❑Save Expression Context

Push Down Scan Key

- Since Quals are very simple, we can push down the complete scan key below to the heap.
- Only qualified tuple will be returned from heap.
- Using this experiment we can save 50-60% instructions of total execution.

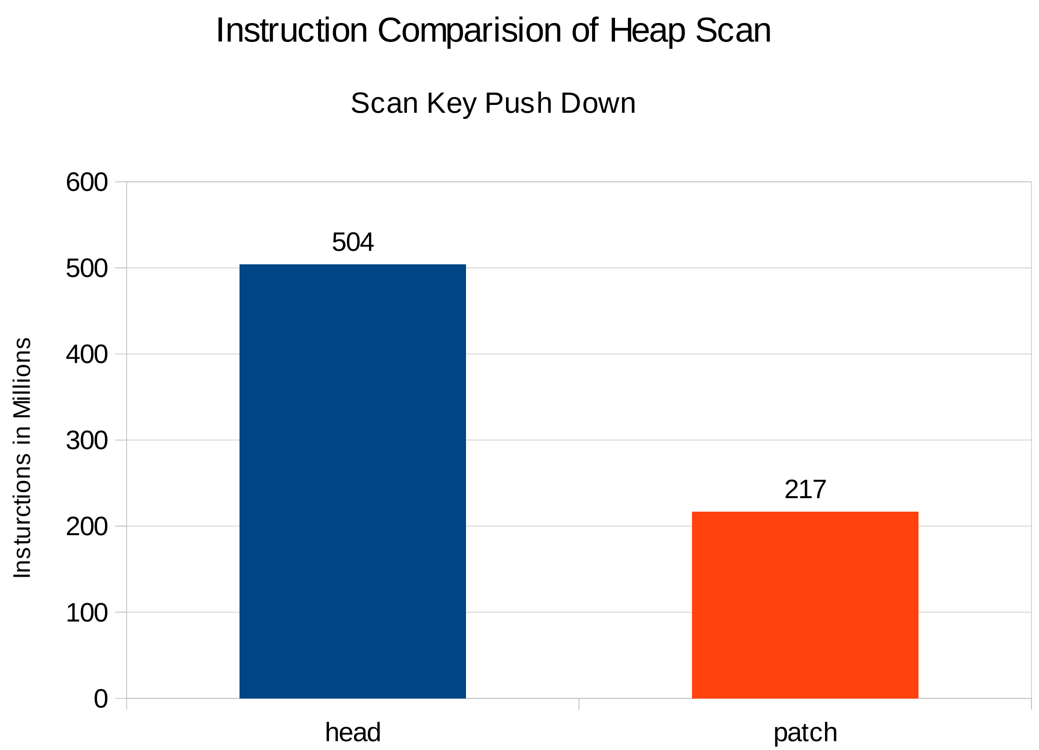
Push Down Scan Key (Instructions)

Experiment:

- Executed select query, with equal qual on an integer column.
  *SELECT * FROM test WHERE c1=10;*
- Selectivity 0.00001

Results:

- ~60% overall instructions reduction.



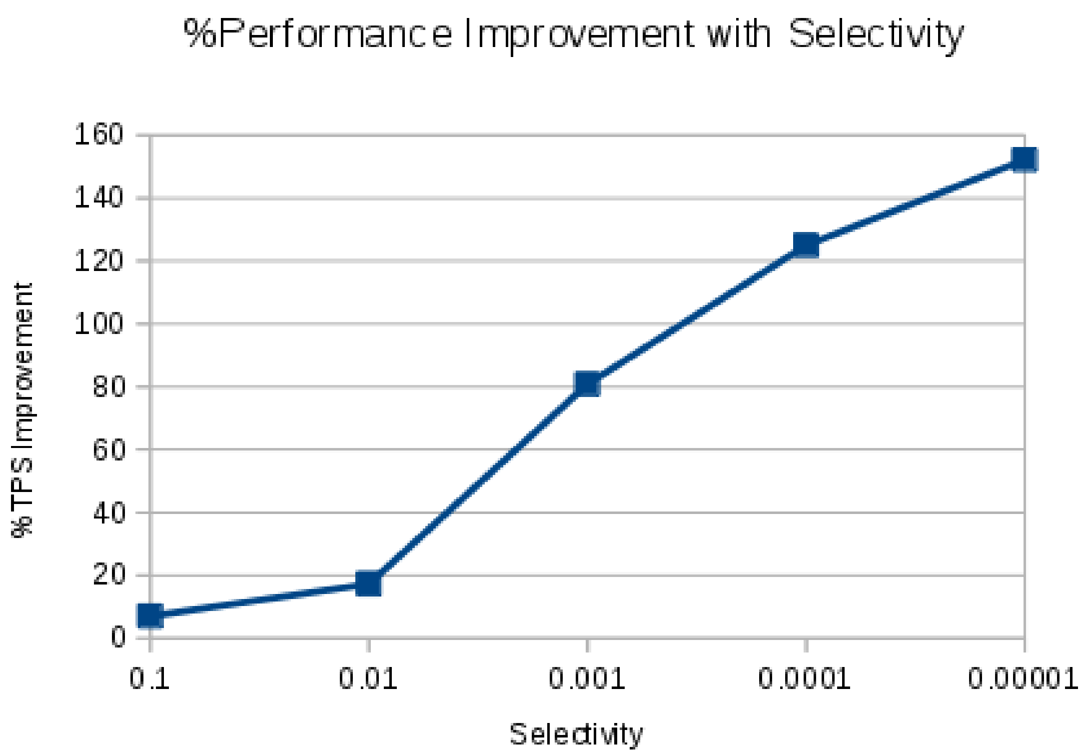
Push Down Scan Key (Performance)

Experiment:

- Executed select query, with equal qual on an integer column.
  *SELECT * FROM test WHERE c1=10;*
- Selectivity vary from 0.1 to 0.00001

Results:

Performance improvement is 7% at selectivity 0.1 which increased up to 150% at selectivity 0.00001.



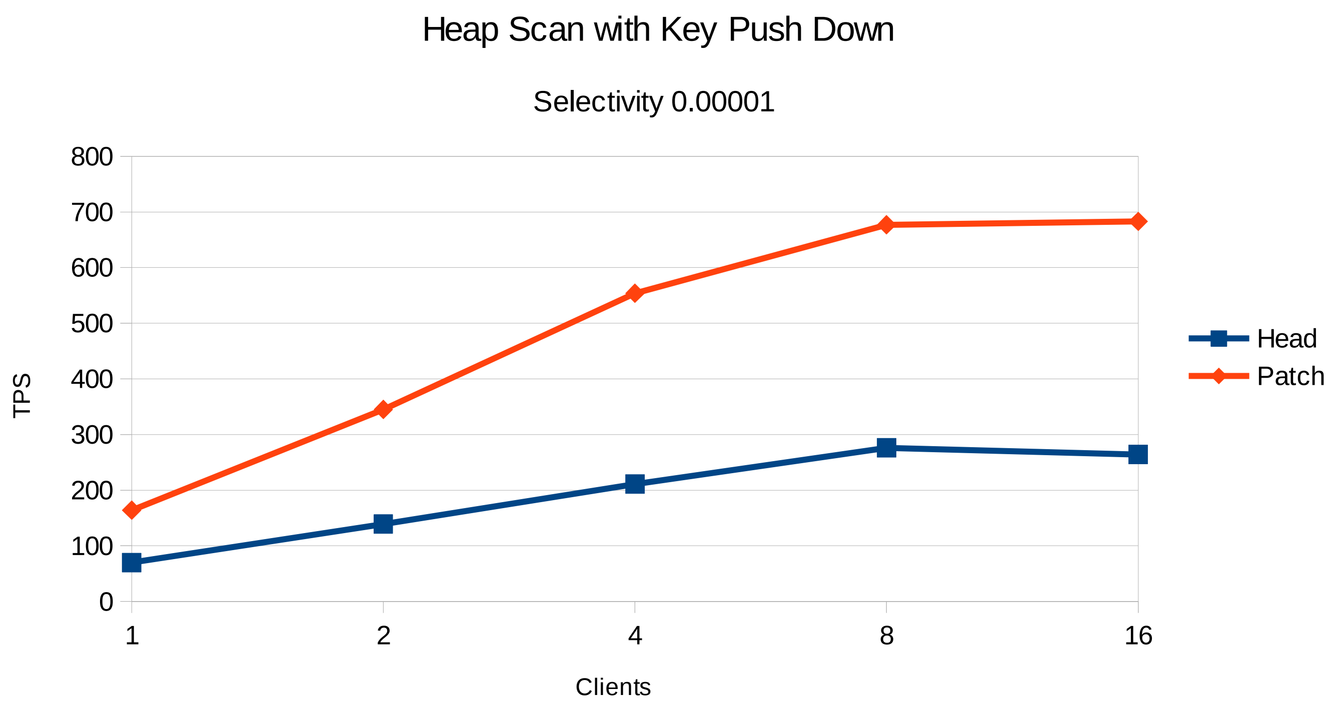
Push down Scan Key (Performance)

Experiment:

- Executed select query, with equal qual on an integer column.
  *SELECT * FROM test WHERE c1=10;*
- Selectivity 0.00001
- Client count vary from 1 to 16

Results:

We observed performance gain of ~150% at different client count.



Qual and Targetlist Initialization

- In case of simple query expressions are easy to store and will not consume huge memory.
- Just by avoiding initialization of qual and tlist, we can save >25 % instructions from ExecutorStart.
- In order to identify a simple query, we need to process qual and targetlist, but this is just one time cost.

Other Optimization

<u>TupleTableSlot</u>

- ❑ExecutorStart creates many TupleTableSlots during every execution.
- ❑If we avoid doing this every time, we can reduce ~5-6% instructions of ExecutorStart.

<u>ExecutorState</u>

- ❑ExecutorStart creates EState for each execution.
- ❑If we avoid this, we can again save 5-6% of ExecutorStart instructions.

Other Optimization (cont..)

Scan Descriptor

- Heap and index scan descriptors can be saved and these can be reused just by resetting some fields.
- Our current experiments don't include this optimization.

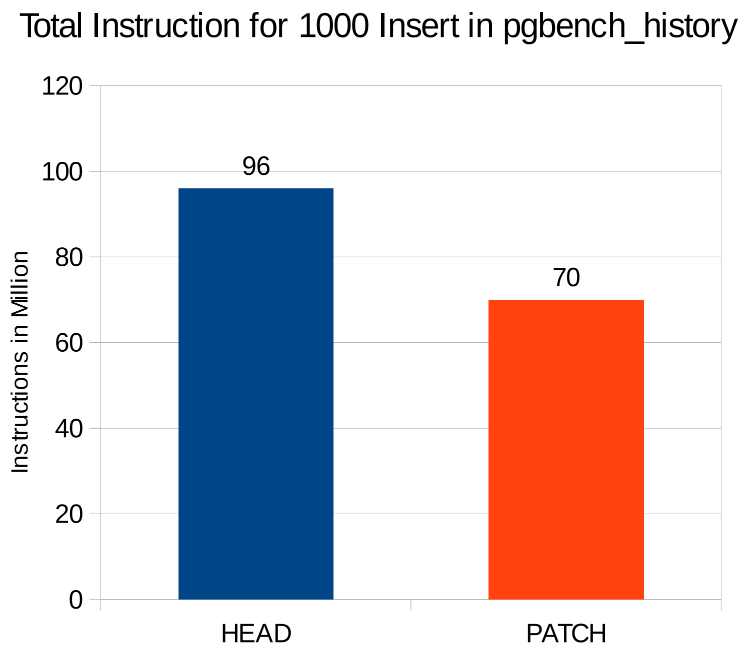
Scan Key

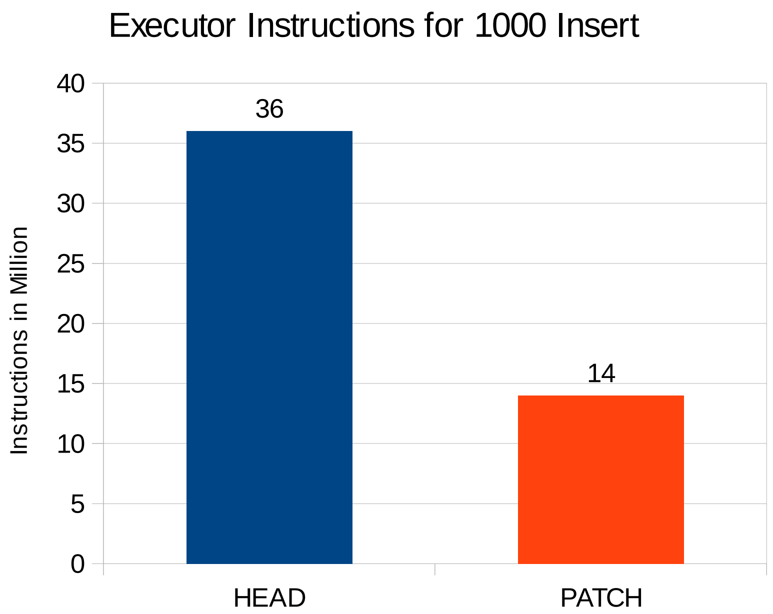
- For index scan, ScanKey can be built only once and can be reused for subsequent executions.
- We can save cost of building scan key every time.

Performance Results (INSERT)

Experiment:
- Execute INSERT query of pgbench_history table
- Measured instructions using callgrind for execution of 1000 transactions.

Results:
We could save > 25% of total instructions and > 60% of executor Instructions.

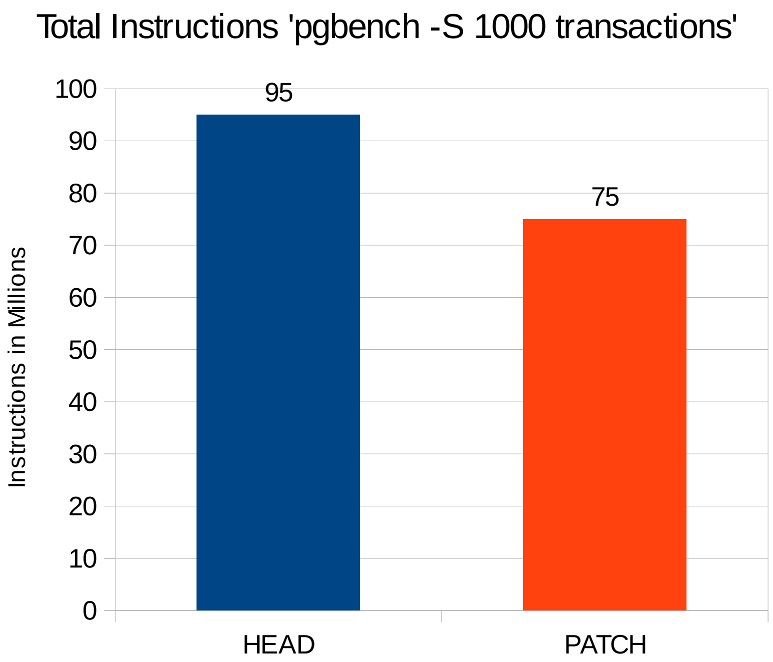


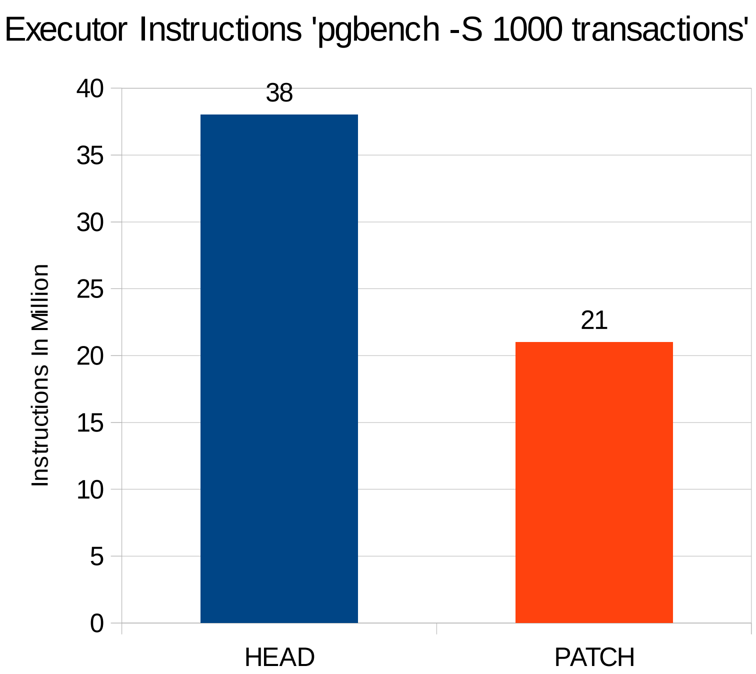


Performance Results (SELECT)

Experiment:
- Executed pgbench read only workload with single client.
- Measured instructions using callgrind for execution of 1000 transactions.

Results:
We could save > 20% of total instructions and > 40% of executor instructions.





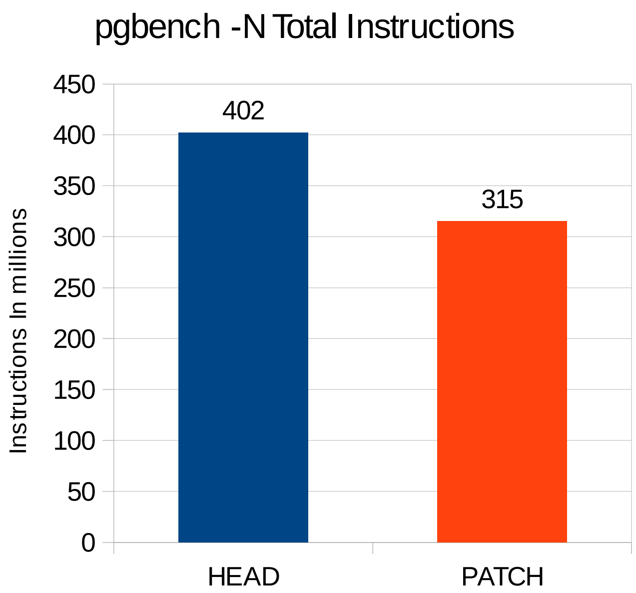
Performance Results (SIMPLE_UPDATE)

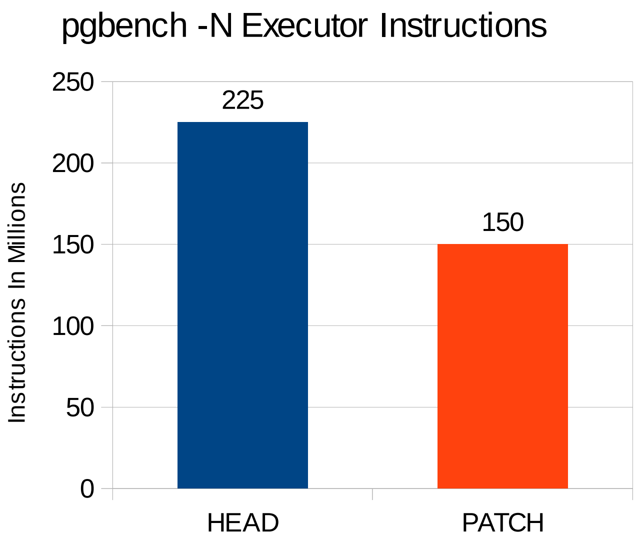
Experiment:

- Executed pgbench simple_update workload (-N).
- Measured instructions using callgrind for execution of 1000 transactions.

Results:

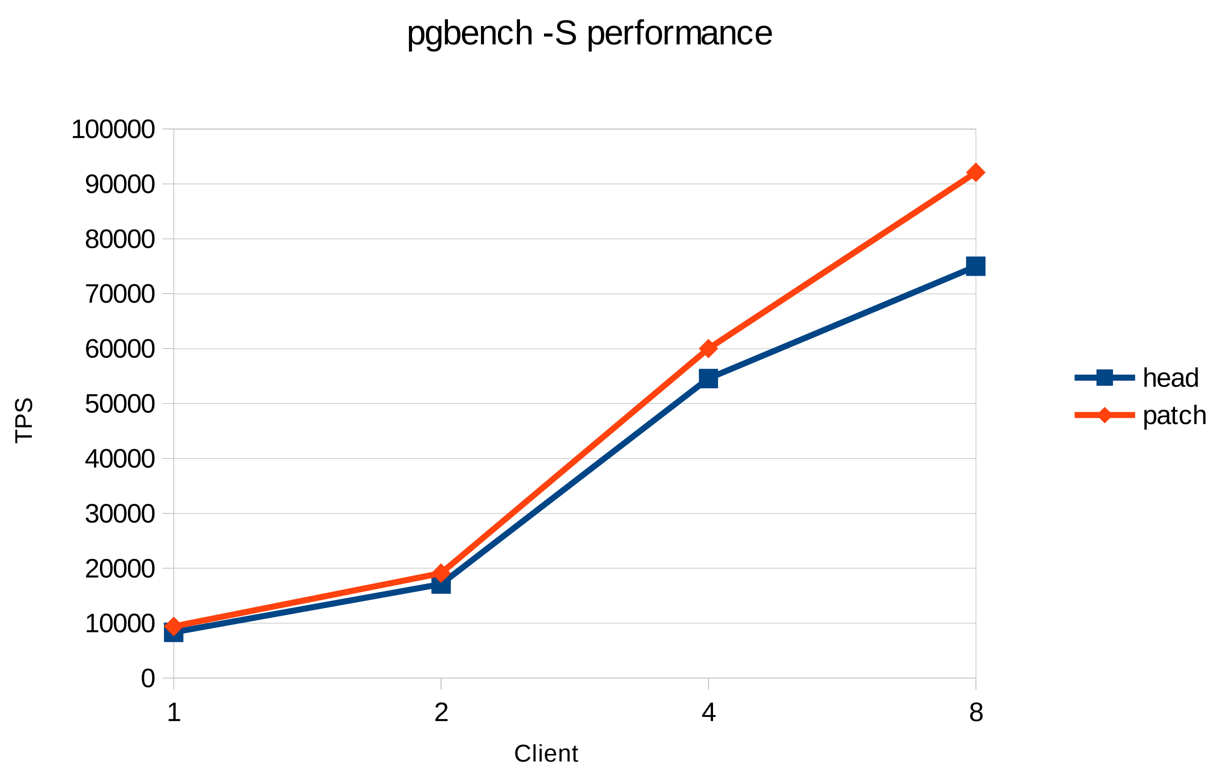
We could save > 20% of total instructions and > 35% of the executor instructions.





Performance Results (SELECT)

In another experiment, we observed that by reducing the instruction count, we could improve scaling, For SELECT, we observed a 12% gain at 1 client and which goes up to 22% at 8 clients.



Future Optimization Plan

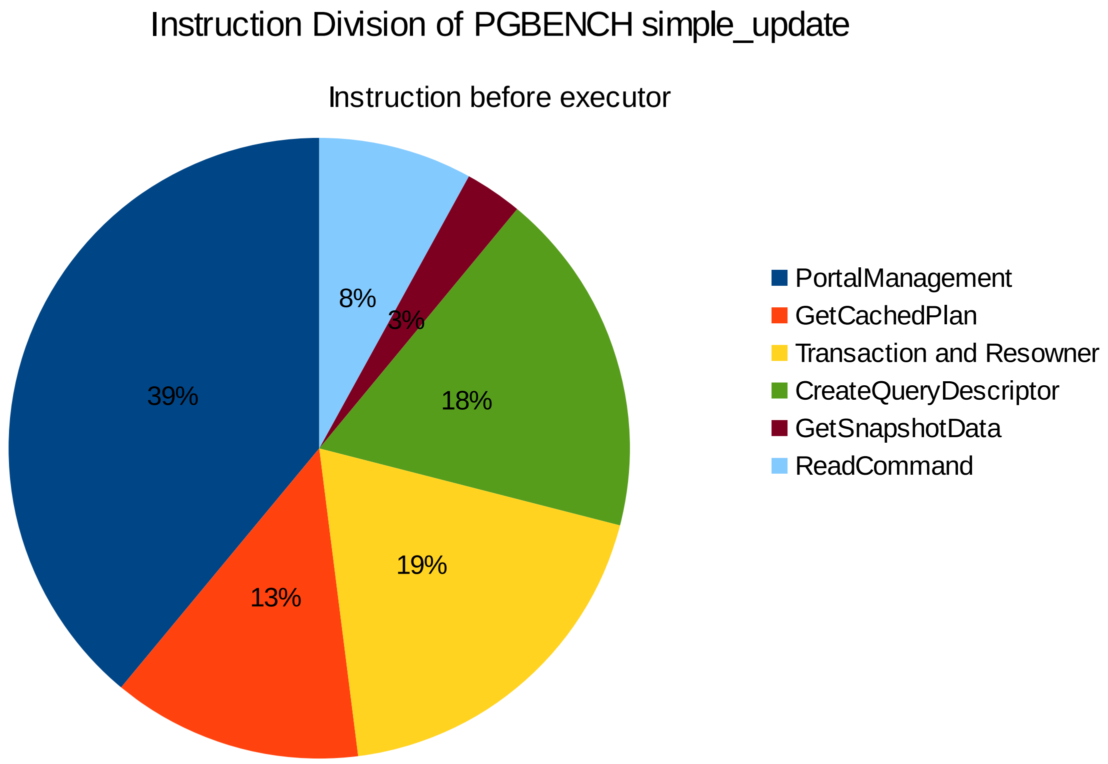
In our initial experiment with simple query we observed that

- ~50% instructions come from executor.
- Remaining 50% are from outside executor.
- For deriving further experiments, we have analyzed remaining instructions, which are outside executor.

Future Optimization Plan

Experiment:

- Executed simple_update of PGBENCH.
- Measured instructions using callgrind.
- Analyzed all the instructions, which executed before hitting actual executor.



Future Optimization Plan

Results:

- Most of these instructions are from portal management infrastructures.
  ~ 39% instructions, that is ~15-20% of total execution instructions.
- 18% instructions are from CreateQueryDescriptor, that is ~10% of total execution instructions.
- Remaining are distributed across various functions like ReadCommand, GetSnapshotData and many more.

Conclusion:

- In next level of optimization, we can further reduce 25-30% of total execution instructions.
- So by including existing experiment, we can save 40-50% of total execution instructions.

Questions?

Thanks!