# Intelligent Utilization Aware Scheduling for Impala Virtual Compute Clusters

Kurt Deschler, Gokul Kolady, Abhishek Rawat, David Rorke, Andrew Sherman, Riza Suminto

# Outline

- Introduction to Apache Impala
- Scaling Impala
- New Utilization Aware AutoScaling
- Impala Threading Model Review
- New Cpu Cost Model
- Evaluation
- Future Work

# What is Apache Impala?

- Distributed, massively parallel SQL database engine
- Main focus is speed
  - frontend (query planning, optimisation) is in Java
  - backend (distributed query execution) is written in C++
    - Uses LLVM runtime code generation for speed
    - Data Caching for remote storage

# What is Apache Impala?

- Flexible
  - Storage Systems: HDFS, Ozone, S3, ADLS, Kudu, …
  - File Format : Parquet, Text, Sequence, Avro, ORC, …
  - Table Formats : External, ACID, Iceberg
- Supports Intel and ARM Processors
- Enterprise-grade
  - authorization, authentication, lineage tracing, auditing, wire and rest encryption
- Scalable
  - >1400 customers, >97000 machines
  - Large clusters with 500+ nodes

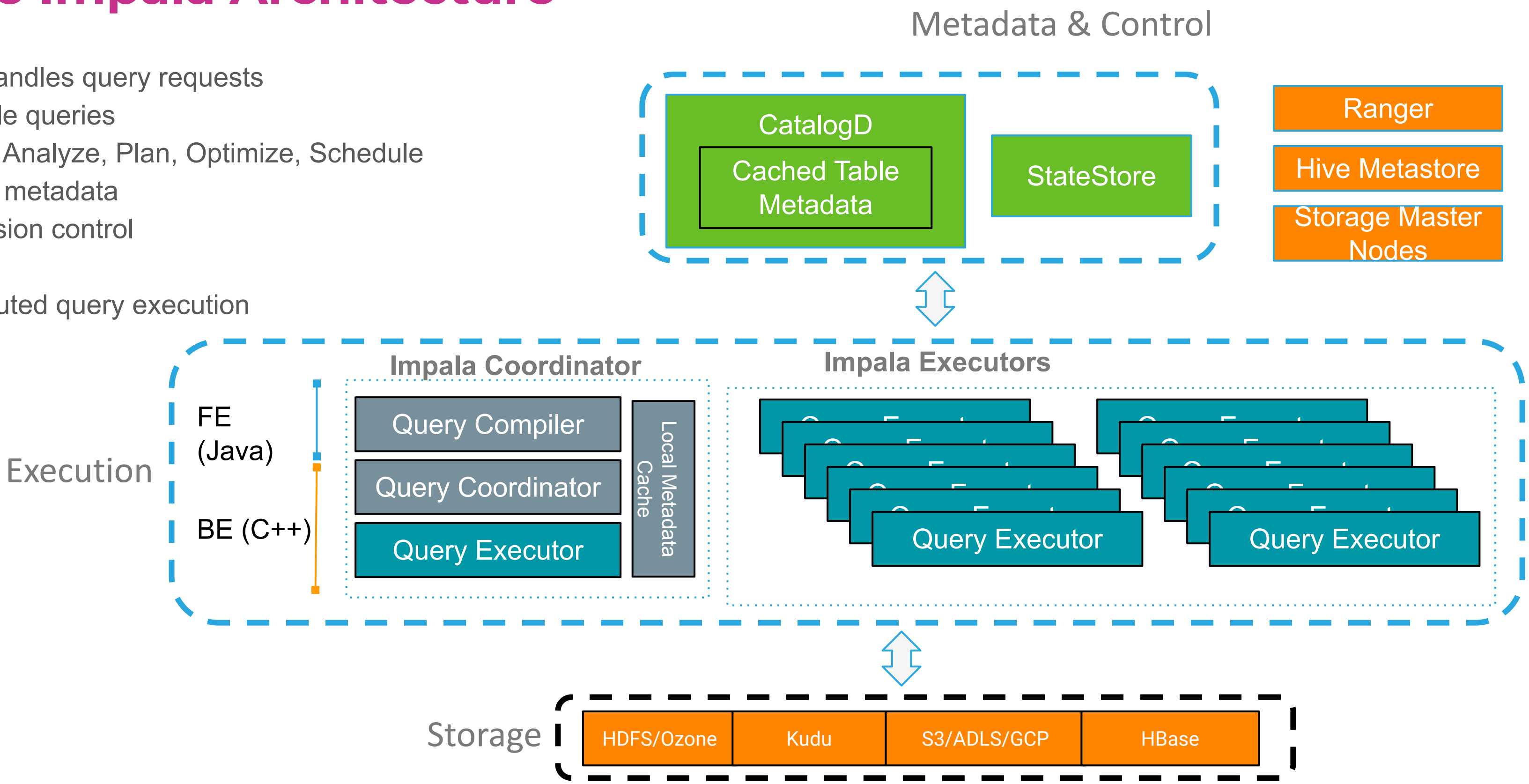
# Apache Impala Architecture

Coordinator handles query requests

- Compile queries
  Parse, Analyze, Plan, Optimize, Schedule
- Cache metadata
- Admission control

Executor

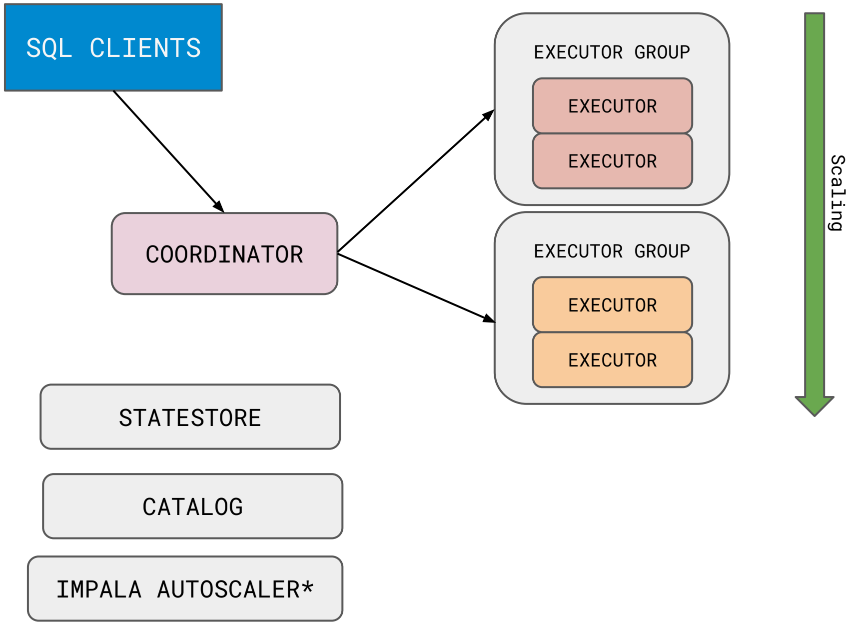
- Distributed query execution



# Scaling Impala

# Scaling the Impala Compute Cluster

## Current implementation

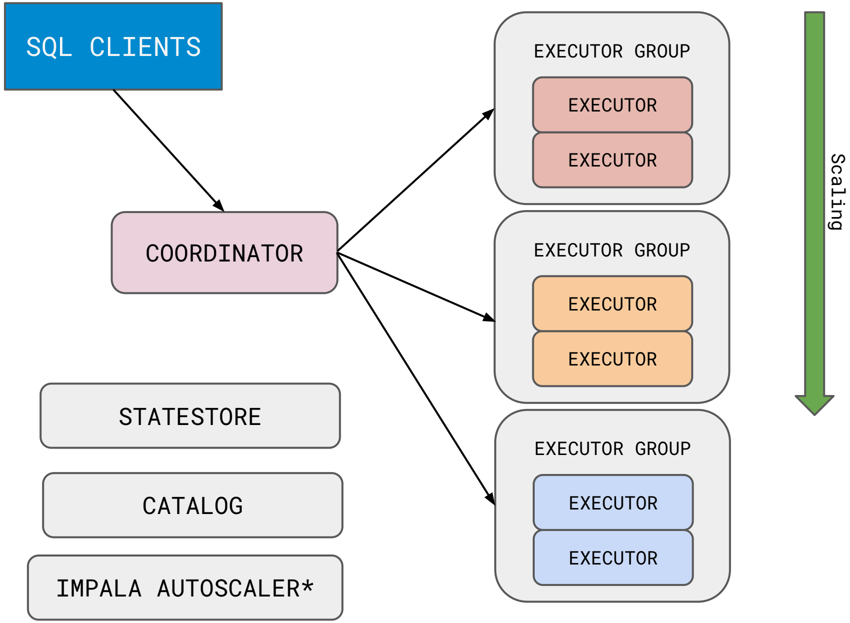



- Cloud enables on-demand compute provisioning
- Executor Groups are the unit of scaling of Impala in an on-demand environment
- Sized large enough to run most queries
- Queries cannot span between Executor Groups
- Each query runs on the first Executor Group with available capacity

*Impala Autoscaler is an external component which scales Impala based on certain metrics

# Scaling the Impala Compute Cluster

## Current implementation




- Queries are queued when all Executor Group is “full” in terms of Memory or CPU
- Executor Groups are added when queries are queued
- Idle Executor Groups are deleted (after a configurable delay)

*Impala Autoscaler is an external component which scales Impala based on certain metrics

# Problem: Handling Mixed Workloads

Most workloads are mixed of small & large queries. Some are more mixed than others.

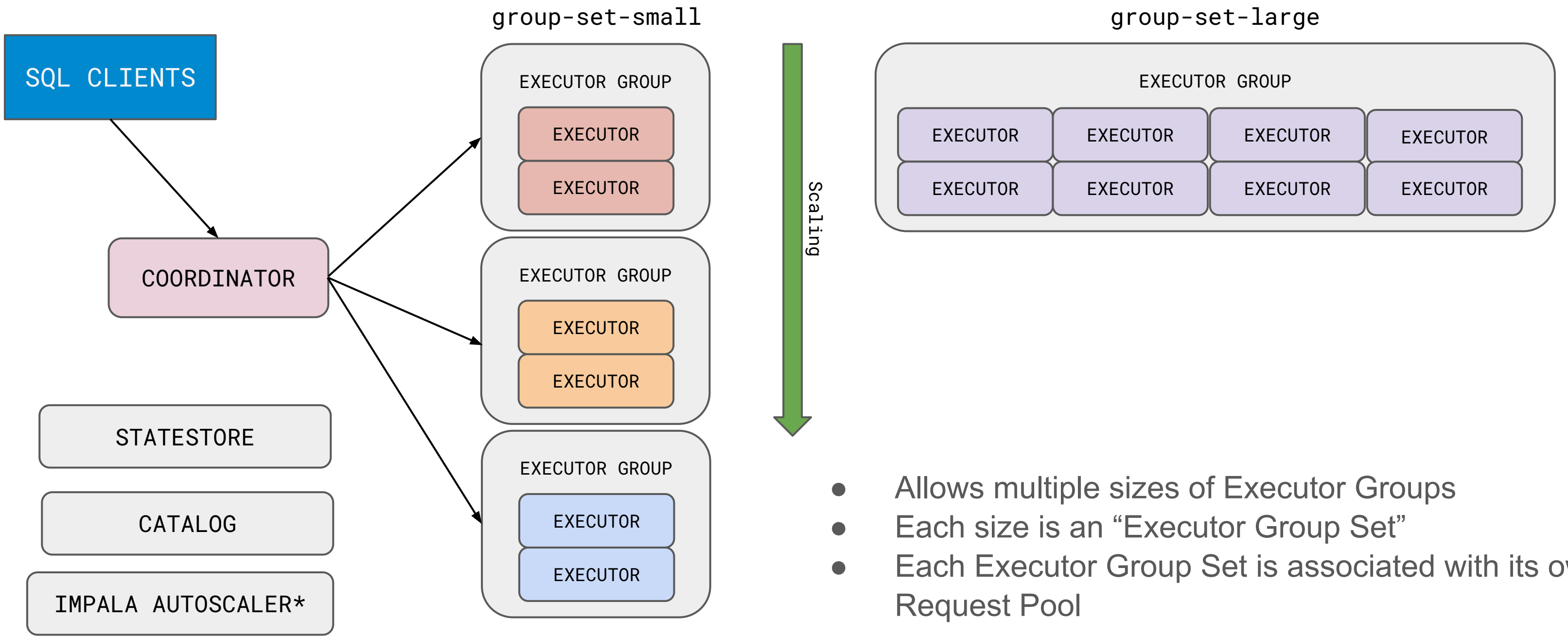
- Use separate Compute Clusters for different query sizes
  - Incurs multiple cluster cost and management overhead
  - Shift the burden of responsibility to end users
  - Not ideal and could lead to poor performance and/or low utilization
- Use a large enough Compute Cluster to handle the largest query
  - Low utilization and increased cost
  - Prone to noisy neighbor problems

*Impala should measure the expected utilization of incoming query and scale the size of Compute Cluster accordingly*

# New: Utilization Aware AutoScaling

# New: Utilization Aware Autoscaling

## Multiple executor group sets

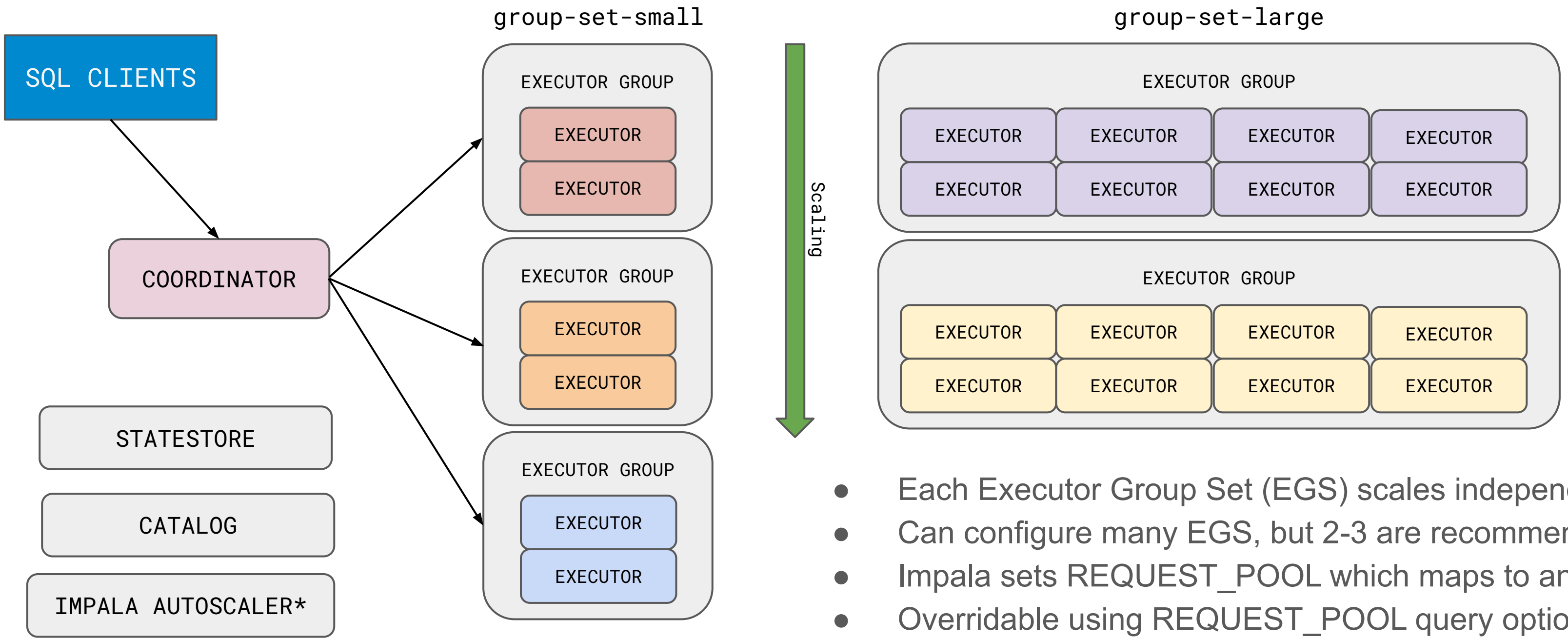



- Allows multiple sizes of Executor Groups
- Each size is an “Executor Group Set”
- Each Executor Group Set is associated with its own Request Pool

*Impala Autoscaler is an external component which scales Impala based on certain metrics

# New: Utilization Aware Autoscaling

## Multiple executor group sets




- Each Executor Group Set (EGS) scales independently
- Can configure many EGS, but 2-3 are recommended.
- Impala sets REQUEST_POOL which maps to an EGS
- Overridable using REQUEST_POOL query option, ie.

`set REQUEST_POOL=”root.group-set-large”;`

*Impala Autoscaler is an external component which scales Impala based on certain metrics

# New: Utilization Aware Autoscaling

## The benefits

1. Maximizes utilization, reduce cloud spend, and retain performance
    a. Node allocation follows incoming workload
    b. Enables multiple groups sizes - no longer a single step function
    c. More flexibility for tuning cluster capacity
2. Preserves performance of queries using transient resources
    a. Can pin a few smaller groups for low-latency response
    b. Larger groups can spin up on-demand only as large queries arrive.
3. Simplify sizing and planning from the user perspective
    a. User only see 1 cluster handling all sizes of queries

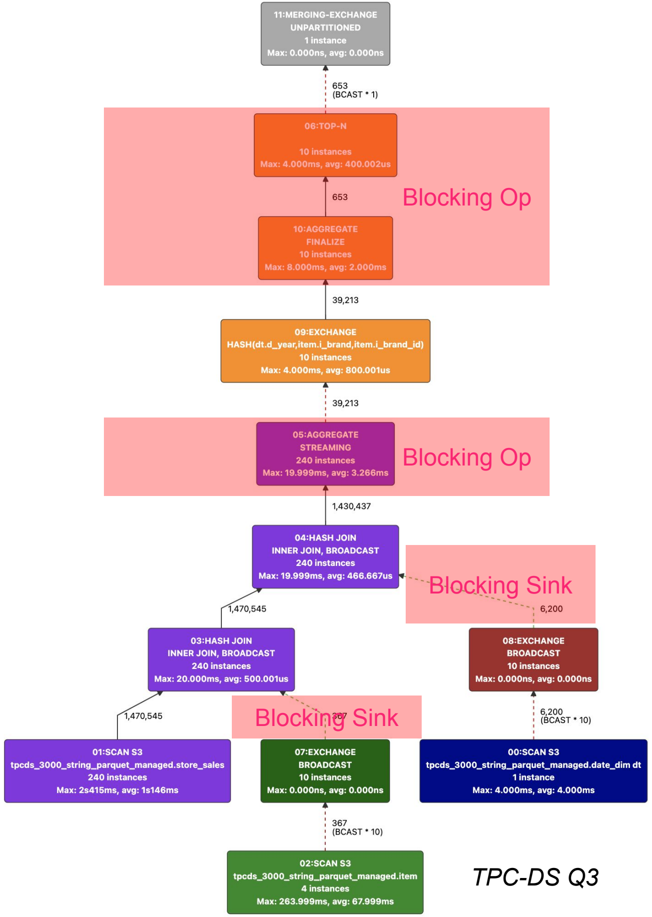
# Next to solve: Utilization Aware Scheduling

1. Given an Executor Group Sets, what is the best way to schedule the query operators?
2. How can Impala decide which queries should go to which Executor Group Sets?

# Impala Threading Model Review

# Impala Query Execution

- Row-based, Volcano-style (iterator-based with batches) with Exchange operators
- Query fragment (unit of work):
  - Portion of the plan tree that operates on the same data partition on a single machine (coded in same color)
  - Each fragment is executed in one or more impalads
- Row batches stream from leaf fragments towards the root, with “stop-and-go” transformation at *blocking* operators.




*TPC-DS Q3*

https://www.cidrdb.org/cidr2015/Papers/CIDR15_Paper28.pdf

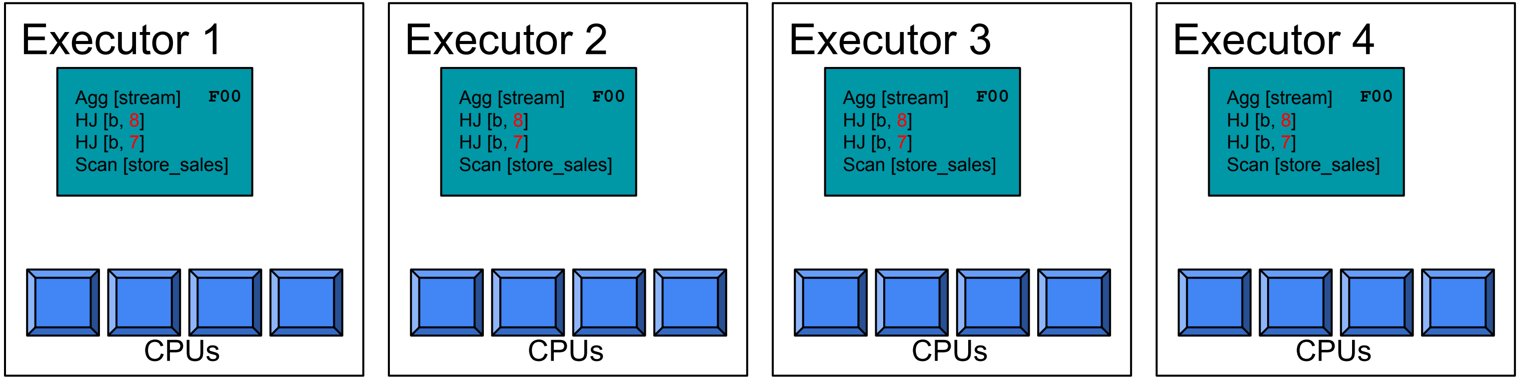
# Classic Impala Threading Model (scale out)

## Characteristic

- Single "main" thread per fragment per host
- Dynamic multithreading within scan (based on available "thread tokens")
- Dynamic multithreading for join builds (branches of plan tree run in parallel)

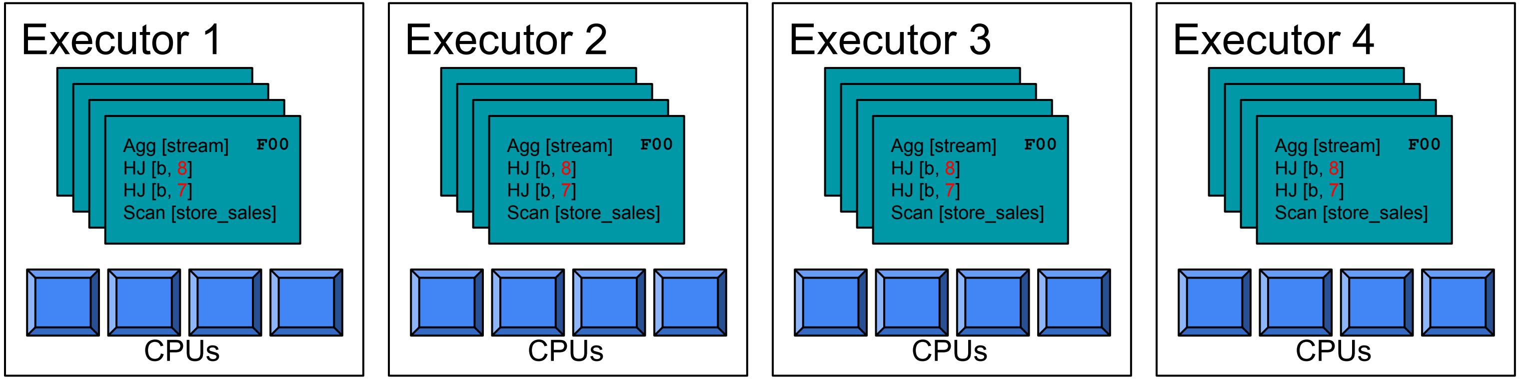
## Challenges

- Single "main" thread causes expensive joins, aggs, sorts, etc.
- Poor resource utilisation (1 busy core on a 40 core server is bad)
- Hard resource management - how many cores does a query want?
- Higher latency for users



# Multi-Threaded Execution (scale up)

- Impala 4.0: MultiThreading in all query operators (Scan, Aggregation, HashJoin, Sort, Analytic, etc)
- set **MT_DOP** = **N** (**M**ulti**T**hreading **D**egree **O**f **P**arallelism)
- Each Fragment can launch up to **N** copies of fragment instances per host
- Linear speedup for most operations (read more in this blog)
- Tradeoff:
  - Parent fragments can over parallelize, because they match up parallelism of children
  - Underutilize memory and oversubscribed **N** CPU per host for the whole query



# New: CPU Costing Model

# Accurate Sizing of Memory & CPU requirement

- Sizing Memory

  Simple addition of memory estimates for all fragment instances scheduled in a single host.

- But how to size CPU requirements?

  Unlike memory, it is OK to oversubscribe CPU a little bit.
  Fragments must scale independently based on their amount of work.

- Improve MT_DOP model by adding following steps

  a. Create a *ProcessingCost* model for each fragment
  b. Determine effective parallelism of each fragment
  c. Match up parallelism between producer vs consumer fragments
  d. Sum-and-overlap the CPU count

# What is ProcessingCost?

- Weighted amount of data to process by a query operator. (IMPALA-11604 part1, IMPALA-12657).
- Describes how compute-intensive a certain query operator is.
- Each kind of query operator has its own cost model.
- Based on the benchmark data.
  1 unit of cost corresponds to 100 nanoseconds of expected CPU time on a single core for a given operator.

# Analyze ProcessingCosts of a Fragment

- Begin with calculating *ProcessingCost* for individual query operators.
- Split a fragment into *Segments* with a *blocking* operator at boundary.
  Adjacent *Segment* execute serially. Therefore, CPUs from the previous *Segment* are reusable by the next *Segment*.
- Sum *ProcessingCost* for all operators in one *Segment* into a *SegmentCost*.

For example, given the following fragment plan:

```
Seg 3
F03:PLAN FRAGMENT [HASH(i_class)] hosts=3
instances=3
segment-costs=[34550429, 2159270, 23752870, 1]
08:TOP-N [LIMIT=100]
Seg 2
| cost=900
|
07:ANALYTIC
| cost=23751970
|
Seg 1
06:SORT
| cost=2159270
|
12:AGGREGATE [FINALIZE]
Seg 0
| cost=34548320
|
11:EXCHANGE [HASH(i_class)]
cost=2109
```

The post-order traversal of rootSegment_ tree show processing cost detail of [(2109+34548320), 2159270, (23751970+900), 1].
The DataSink with cost 1 is a separate segment since the last PlanNode (TOP-N) is a blocking node.

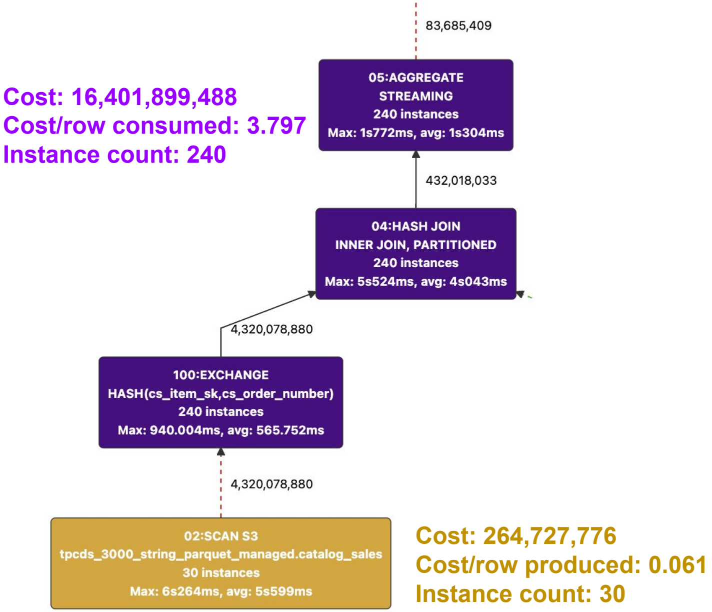
# Determine Effective Parallelism of each Fragment

- Given a fragment, how many copies of fragment instances to schedule such that they complete within reasonable time?
- The *SegmentCost* list provides estimated CPU costs.
  1 unit of cost is roughly 100 nanosecs on a single core.
- So the *SegmentCosts* can be translated into a target num CPU by dividing max *SegmentCosts* with a desired constant (`--min_processing_per_thread=10M`).
- This results in a target core count (parallelism) that attempts to allocate ~10M cost units (1 second of CPU time) on each core.

```
F03:PLAN FRAGMENT [HASH(i_class)] hosts=3
instances=3
segment-costs=[34550429, 2159270, 23752870, 1]
```

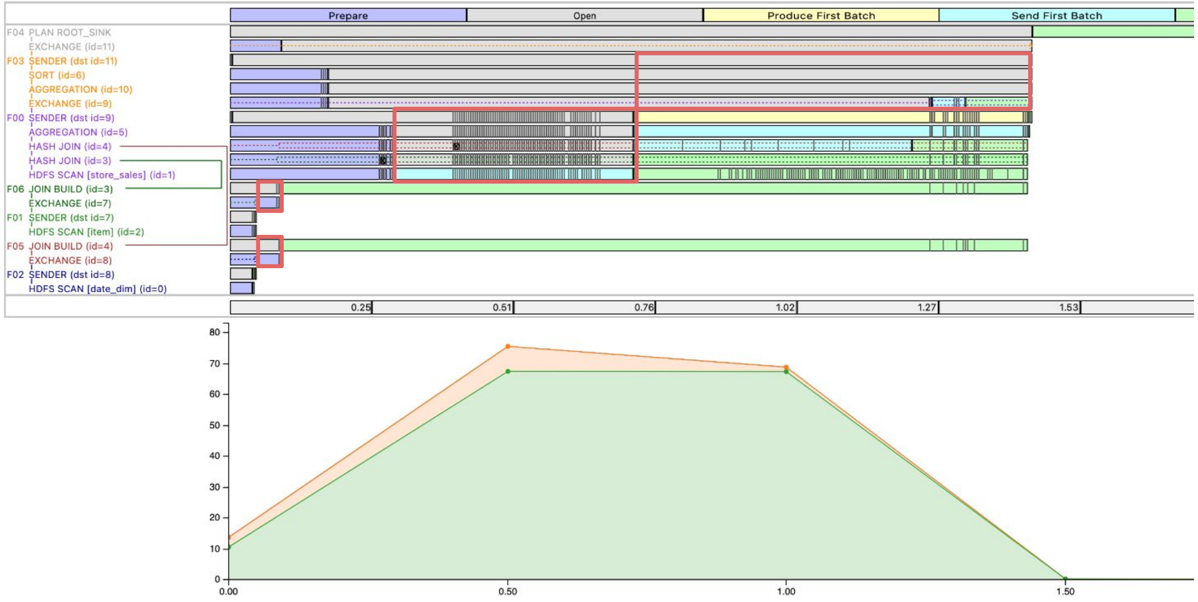
# Match Parallelism of Producer vs Consumer

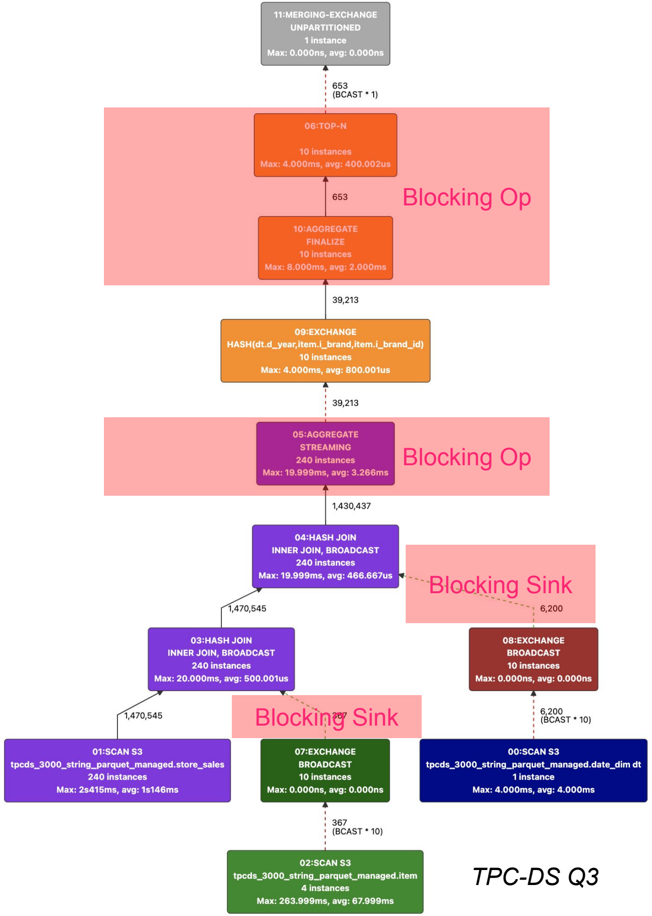
- Fragments produce and consume rows at different rates. Need to avoid resource waste if one fragment can't keep up with the other fragment.
- Scale adjacent fragments so that *row production rate* and *row consumption* rate between them are roughly equal.
- Parallelism follows the ratio between *per-row production* cost of child vs *per-row consumption* cost of parent.
- Enforces min and max parallelism bounding from query options or Executor Group Set configuration.
- Adjusted bottom-up from scanners up to plan root.



# Overlap CPU between Blocking Subtrees

Fragments don’t get busy at the same time



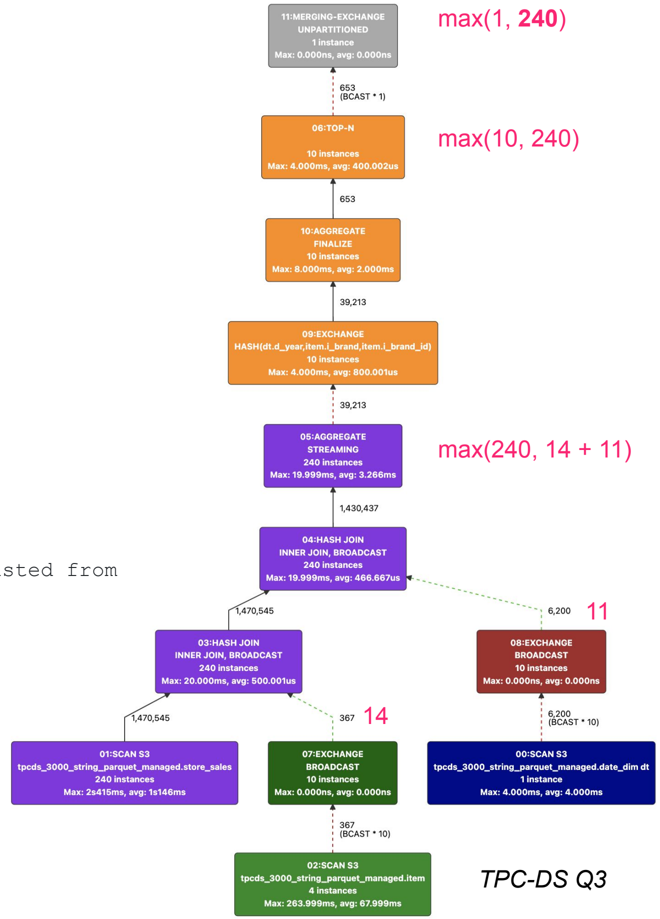




*TPC-DS Q3*

# Sum-and-Overlap CPU Count

- Identify all blocking points in the plan tree and overlap CPU requirements between plan subtrees.
- At blocking fragment, take the max between current subtree's total CPU vs total CPU of child subtrees.

```
F03:PLAN FRAGMENT [HASH(dt.d_year,item.i_brand,item.i_brand_id)] hosts=10 instances=10 (adjusted from
240)
Per-Instance Resources: mem-estimate=78.20MB mem-reservation=34.00MB thread-reservation=1
  max-parallelism=10 segment-costs=[36475081, 300, 6]
  cpu-comparison-result=240 [max(10 (self) vs 240 (sum children))]
```




*TPC-DS Q3*

# Recap on CPU Costing Model

1. Create a *ProcessingCost* model for each fragment

   *ProcessingCost* for individual operator

   *SegmentCost(s)* for individual fragment

2. Determine effective parallelism of each fragment

   `max(SegmentCost) / min_processing_per_thread`

3. Match up parallelism of producer vs consumer fragment

   Compare per-row production cost vs per-row consumption cost

4. Sum-and-overlap the CPU count

   Overlap CPU between Blocking Fragments

# Query to Executor Group Set assignment

- First, compile the query against the smallest Executor Group Set.
- Compare the requested resources against configured resources.
    - If `MemoryAsk <= MemoryMax` AND `CpuAsk <= CpuMax`, then assign to the current Executor Group Set. Otherwise, step up to the next larger Executor Group Set and recompile query.
- Largest Executor Group Set is a “catch all” group.

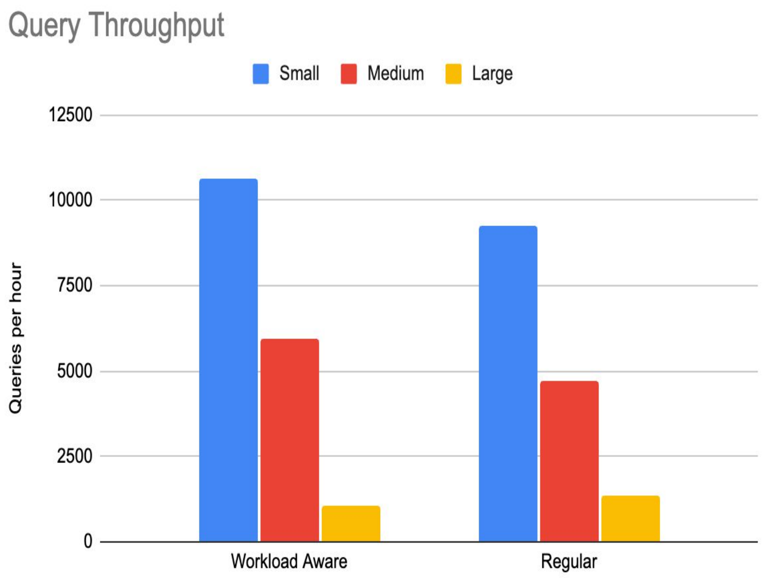
```
Frontend:
...
   - ExecutorGroupsConsidered: 2 (2)
  Executor group 1 (root.group-set-small):
    Verdict: not enough cpu cores
     - CpuAsk: 240 (240)
     - CpuMax: 48 (48)
     - EffectiveParallelism: 240 (240)
     - MemoryAsk: 7.91 GB (8489792424)
     - MemoryMax: 100.00 GB (107374182400)
  Executor group 2 (root.group-set-large):
    Verdict: Match
     - CpuAsk: 240 (240)
     - CpuMax: 240 (240)
     - EffectiveParallelism: 240 (240)
     - MemoryAsk: 8.64 GB (9272936930)
     - MemoryMax: 500.00 GB (536870912000)
```

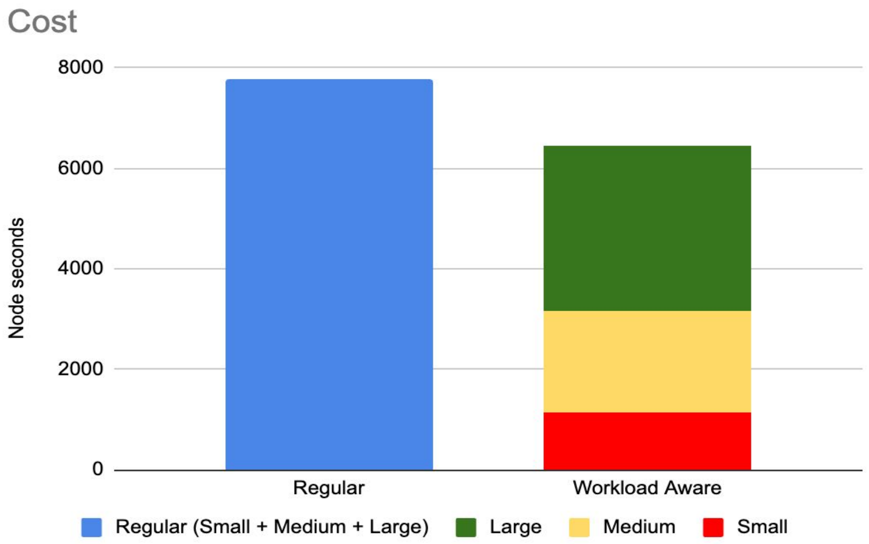
# Evaluation

# Workload Characteristics

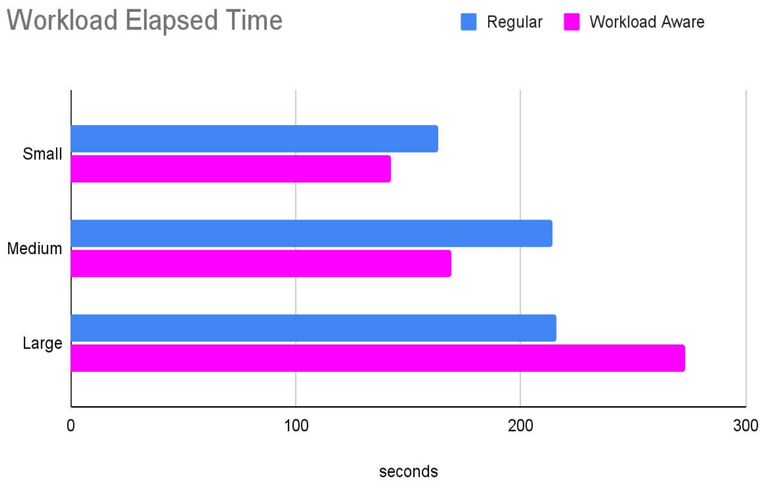
- Concurrent Workload
  - Subset of TPC-DS 3TB scale
    - Small (14 queries), Medium (14 queries) and Large (8 queries)
  - 60 concurrent users
    - 30 running Small, 20 running Medium and 10 running Large queries
    - No think time
- Regular Impala
  - 36 nodes of r5d.4xlarge, MT_DOP model, fixed Executor Group size
    - 4 executor groups, each with 9 nodes
- Workload Aware Impala
  - 36 nodes of r5d.4xlarge, CPU Costing model, optimized for interactive queries
    - 6 small executor groups, each with 2 nodes
    - 2 medium executor groups, each with 6 nodes
    - 1 large executor group with 12 nodes

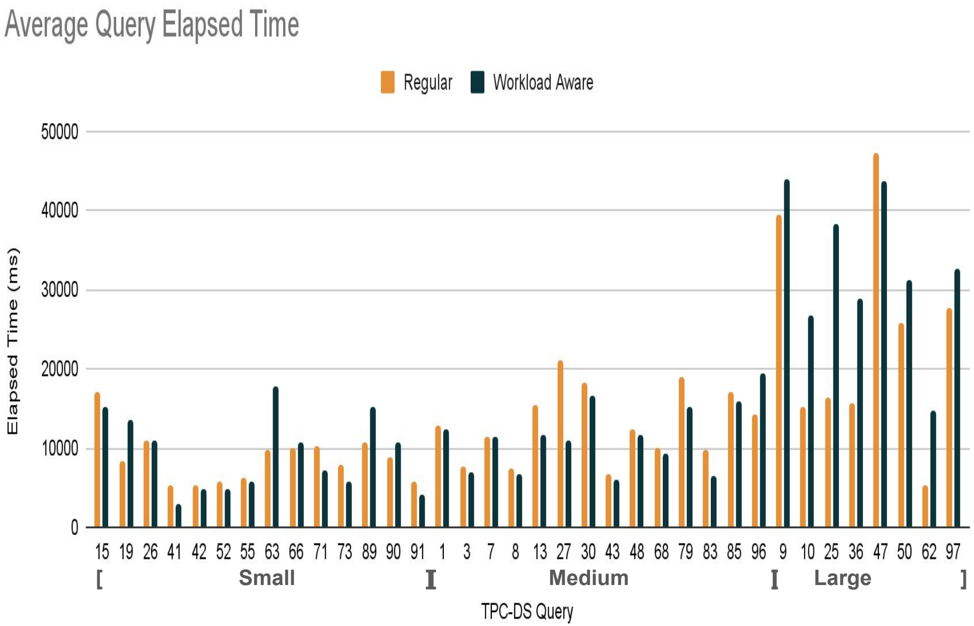
# Results


Query Throughput
Small
Medium
Large
12500
10000
7500
5000
2500
0
Queries per hour
Workload Aware
Regular


Cost
8000
6000
4000
2000
0
Node seconds
Regular
Workload Aware
Regular (Small + Medium + Large)
Large
Medium
Small

# Results


Workload Elapsed Time
Regular
Workload Aware
Small
Medium
Large
0
100
200
300
seconds


Average Query Elapsed Time
Regular
Workload Aware
Elapsed Time (ms)
50000
40000
30000
20000
10000
0
15 19 26 41 42 52 55 63 66 71 73 89 90 91 1 3 7 8 13 27 30 43 48 68 79 83 85 96 9 10 25 36 47 50 62 97
[ Small ] [ Medium ] [ Large ]
TPC-DS Query

# Future work

- Performance tuning.
- Consider other resources for planning queries.
  - Local disk capacity for spilling & caching, network bandwidth, file handles, etc.
- More flexible Auto Sizing
  - Dynamically update executor group size based on workload history
  - More elastic executor group based scaling model (nodes to EG assignment)
  - SLA aware planning and scheduling of queries

# Contributing to Apache Impala

Mailing lists:

- user@impala.apache.org (users), subscribe by mailing user-subscribe@impala.apache.org
- dev@impala.apache.org (developers), subscribe by mailing dev-subscribe@impala.apache.org

Issues: https://issues.apache.org/jira/browse/IMPALA

Twitter: @ApacheImpala

Slack: apache-impala.slack.com

# Thank you!

Questions?

# llama config for Utilization Aware Scheduling

```
<property>
  <name>impala.admission-control.max-query-mem-limit.root.small</name>
 <!-- 90 MB -->
  <value>94371840</value>
</property>

<property>
  <name>impala.admission-control.min-query-mem-limit.root.small</name>
 <!-- 0MB -->
  <value>0</value>
</property>

<property>
  <name>impala.admission-control.max-query-cpu-core-per-node-limit.root.small</name>
  <value>8</value>
</property>
```

# Tuneable knobs

- Query options
  - `COMPUTE_PROCESSING_COST`
  - `PROCESSING_COST_MIN_THREADS`
  - `MAX_FRAGMENT_INSTANCES_PER_NODE`
  - `QUERY_CPU_COUNT_DIVISOR`

- Flags
  - `--min_processing_per_thread`
  - `--skip_resource_checking_on_last_executor_group_set`
  - `--query_cpu_root_factor`
  - `--processing_cost_use_equal_expr_weight`