# Comprehensive Performance Benchmarking, Monitoring, and Reporting Infrastructure for Presto and Prestissimo

Ethan Zhang, Ying Su, Linsong Wang, Karteek Murthy, Xin Peng
OSS Presto Team

## Introduction

In this write-up, we discuss the details of a full suite of infrastructure we developed to bring an efficient and convenient way of performance benchmarking, monitoring, and reporting for OSS Presto and Prestissimo clusters to both developers and executive leaders. For developers, this means providing tools for running the benchmarks, collecting metrics data, deep performance analysis, enabling bottleneck identification and query efficiency optimization. For executives, this offers insights into performance trends over time and an intuitive view into the team's progress on optimizer and engine optimization.

The goals of our performance infrastructure are:

- Fully AUTOMATED runs
- Easy ad-hoc run submissions
- Comprehensive coverage
    - Multiple workloads
        - TPC-H, TPC-DS and other formal benchmarks
        - Customer or IBM internal workloads, e.g.: IBM BI day3, Catalina, Nielsen, and AppNexus
        - Self designed primitive benchmarks that target at different areas
            - Execution, operators
            - Optimizer
            - Scheduling
            - Scaling
            - Spilling
    - Multiple Scale factors
        - TPC-H and DS shall have multiple SFs: e.g. 1TB, 10TB, 100TB
        - The actual SF will be decided based on benchmark publication requirements, and customer preferences, marketing decisions, etc.
- Continuous tracking
    - Each workload shall run in its designed cadence
        - TPC benchmarks small SF shall run once per week or per day
        - TPC large SF shall be ran at a less frequent manner due to cluster costs, and the cadence will be decided after the baseline setup is done

    - Dedicated performance team or on-call routine to continuously monitor and debug
- Testing results shall be displayed in a straightforward way with easy to observe insights. Testing people's time should not be wasted on manually querying and interpreting the results.

The tools that help us achieve the above goals include but not limited to:

1. Automated Jenkins Pipelines to build images and deploy clusters
2. A brand new highly flexible and customizable benchmark runner.
3. Comprehensive Performance metrics monitoring, collection, ands storage
    a. Query info stored in relational DB, e.g. MySQL
    b. Timeseries data stored in InfluxDB
    c. Carefully designed reliability dashboards to monitor the cluster health
4. UI: Interactive data visualization for multi-dimension deep analysis via Grafana.
    a. performance dashboards
5. Query lookup tool
6. Cluster management tool to see
    a. the status of all clusters, and
    b. what workloads/benchmarks/tests are running on them,
    c. who is borrowing which clusters,
    d. how long each cluster has been running
    e. Hardware failures
7. Resource group management tool
    a. Queries on each RG
    b. Priorities adjustment

With the aid of the above-mentioned tools, the performance engineering work can be carried out:

- Baseline setup
    - Which benchmarks to run, what scale factors to cover, and in what cadence need to be decided
    - Cluster configurations need to be tuned before the baseline is set up.
    - For all benchmarks we decide to run, a baseline report shall be produced
- Performance tracking
    - Regression criteria need to be set up
    - Workloads to be run at the designed cadence
    - There need to be dedicated people looking into regressions.
- Development work in Presto/Prestissimo engine
    - Cache cleaning to allow cold/warm runs
    - Adding metrics

Below we're going to talk about the tools we have been developing. We hope by showcasing our infrastructure work on the open-source side, together we can push for a unified performance benchmarking, monitoring, and reporting framework for Presto/Prestissimo across various watsonx.data form factors including SaaS, CPD, and BlueRay so developers and executive leaders can have a much easier way to compare performance differences and identify optimization opportunities across all those form factors.

## Automated Jenkins Pipelines

Our Jenkins pipeline is responsible for building images, deploying Presto and Prestissimo clusters, and kicking off regular and ad-hoc benchmark runs:

### Building Images

We automate two types of builds:

- **Regular Image Builds**: Automated construction of new Docker images from the latest Presto code every two days. [Rebase Branch]
    - As part of this process, we also do an auto-rebase to include our customized Dockerfile and Prestissimo metrics endpoint changes which are not yet in the OSS.
- **Customized, Ad-Hoc Builds**: Developers can trigger builds for their development branches, allowing for testing of new features or fixes.

Images we built are pushed to AWS ECR [Java] [Prestissimo].

### Cluster Deployment

For simplicity, we provide 'T-shirt size' configurations for clusters (small, medium, large, etc.), each designed for different benchmark scales. After selecting the cluster size, a cluster is deployed using a Docker Swarm cluster formed by EC2 instances we provision via Pulumi.

See Table 1 T-shirt sizes for cluster configurations for full details:

*Table 1 T-shirt sizes for cluster configurations*

| **Name** | **Instance type** | **# of workers** | **Total vCPU** | **Total memory (GB)** | **Default scale factor** |
|---|---|---|---|---|---|
| **xsmall** | r5.xlarge *(vCPU: 4, Memory: 32)* | 2 | 8 | 64 | Sf 10 |
| **small** | r5.xlarge *(vCPU: 4, Memory: 32)* | 4 | 16 | 128 | Sf 100 |
| **medium** | r5.4xlarge *(vCPU: 16, Memory: 128)* | 8 | 128 | 1024 | Sf 1,000 |
| **large** | r5.8xlarge *(vCPU: 32, Memory: 256)* | 16 | 512 | 4096 | Sf 10,000 |
| **xlarge** | r5.16xlarge *(vCPU: 64, Memory: 512)* | 32 | 2048 | 16384 | Sf 100,000 |
| **xscale** | i3.2xlarge *(vCPU: 8, Memory: 61)* | 256 | 2048 | 15616 | Sf 100,000 |

The cluster sizing need to be decided considering the following factors:

- Data should be mostly in memory for the TPC benchmarks
    - One way is to check the largest `*peakTotalMemoryReservation*` (currently query 95) in the TPC benchmarks at each designated scale factor, and make sure the cluster total

memory is not less than that value. This would make the single stream execution is mostly in memory

    - The throughput phase is usually a fixed number of streams, and we also want to make the data is mostly in memory. To determine the amount of memory needed, we need to do experimental runs and record the largest amount of memory used during the run

- The cost of the cluster

    - To optimize for cost-performance, we need to reduce the total cost of the cluster

- The special goal of each workload, e.g. to test spilling, we need to make sure the spilling amount happens in the designed range.

## PBench: the brand-new benchmark runner

We now not only have a much broader set of benchmarks including standard TPC ones and the ones derived from IBM customer workloads (Catalina, Nielsen, etc.), but also for each benchmark we need to have multiple variants with different scale factors, cluster configurations, and concurrency level, etc. The existing benchmark framework in the Presto OSS community cannot fully satisfy our diversified needs for benchmarking today. Considering this, we developed a brand-new benchmark runner in Go to replace Benchto, aiming to address the following pain points:

1. Lack of support for concurrent query workloads
2. Lack of support for result capturing for correctness verification
3. Lack of query log collection
4. Lack of flexibility in defining the benchmarks
5. No active maintenance of the project

### Graph-Based Benchmark Modeling

To support concurrent query workloads with the ability to customize query sequence and various kinds of intricate execution flows, we break down a benchmark into stages and model them as a single-source directed acyclic graph (DAG) where we represent the benchmark stages as vertices and their execution sequence as edges. In the graph theory, a "source" is commonly defined as "a vertex with no incoming edges" and is the starting point in the graph, from which you can reach other vertices following the directed edges. The term "single-source" here means there can be only one starting point in each graph modeling a benchmark. We call this "single source" the "**main stage**" of the benchmark. This graph is depicted by a set of inter-referencing JSON files (one per stage), which then can be effectively parsed by the runner.

This graph model provides a versatile framework for various testing scenarios:

1. **Sequential Execution**: A single stream of execution follows the stages in sequence, as defined by the graph.
2. **Parallel Execution**: The graph can branch out into multiple parallel streams, enabling simultaneous execution of different stages.
3. Bag Execution: The ordering of the queries do not matter here, but the query concurrency is set
4. **More Complex Flows**: These parallel streams can converge and diverge within the graph, allowing for intricate execution patterns like join-and-split scenarios. Figure 1 shows one example:

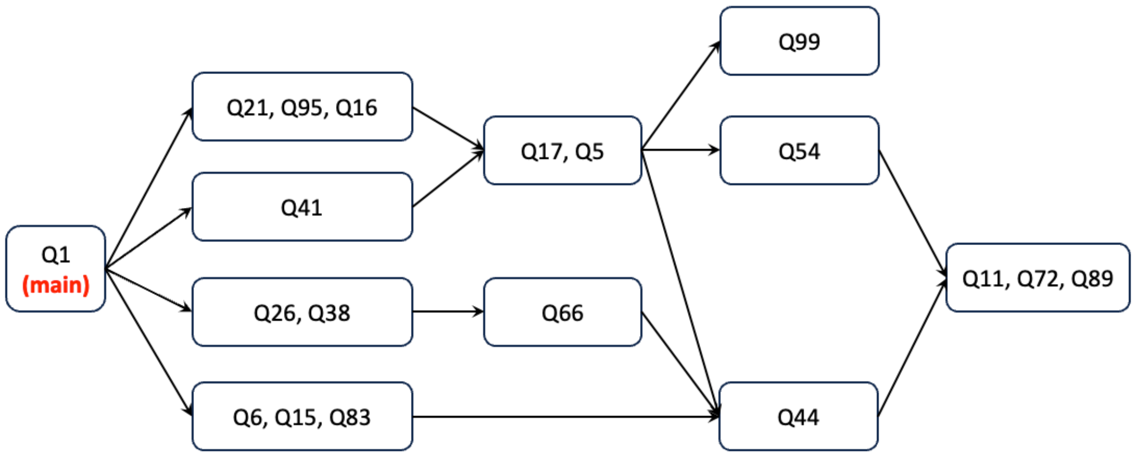


*Figure 1 Example of complex flows modeled as acyclic graph.*

## Rich and Fine-grained Configurations

The benchmark runner provides a rich set of configurations, and they can be customized and applied at the benchmark stage level in the JSON file:

| Field Name | Description | Inherited by children stages |
|---|---|---|
| catalog | Specifies the catalog to use. | Yes |
| schema | Specifies the schema to use. | Yes |
| session_params | Specifies the list of session parameters newly set for this stage. Stages can inherit session parameters from their parent stages if start_on_new_client is not set. | Yes |
| queries | Specifies an array of queries to execute in this stage. | No |
| query_files | Specifies an array of query file names to include for execution in this stage. | No |
| cold_runs | Specifies the number of cold runs. | Yes |
| warm_runs | Specifies the number of warm runs. | Yes |
| start_on_new_client | When this is set to true, we create a new HTTP client instead of inheriting the client from the parent of this stage. (default: false) | No |
| abort_on_error | When this is set to true, we abort the entire benchmark instead of moving on to the next query in the event of an error. (default: false) | Yes |
| save_output | When this is set to true, we save the query result to a file. (default: false) | Yes |
| save_column_metadata | When this is set to true, we save a JSON file for each query documenting the result column names and their data types. (default: false) | Yes |

| Field Name | Description | Inherited by children stages |
|---|---|---|
| save_json | When this is set to true, we save the query JSON log to a file. (default: false) Please note that we always save query JSON log for failed queries regardless of the value of this field. | Yes |
| next | Specifies an array of children stages. Those stages will not start to execute until this current stage finishes. Each child stage will execute in its own thread in parallel. | No |

## Main Stage Configuration Stacking for Composability

In practice, we specify the values for most of the settings in the main stage then let children stages inherit those values. And, those settings can vary based on certain factors, like engine type, scale factor, etc. To make the main stage configuration more composable, users can break down configurations into multiple JSON files, and combine them differently based on the needs, allowing for a broad spectrum of combinatory possibilities, maximum reusability, and fine-tuned control over benchmark settings.

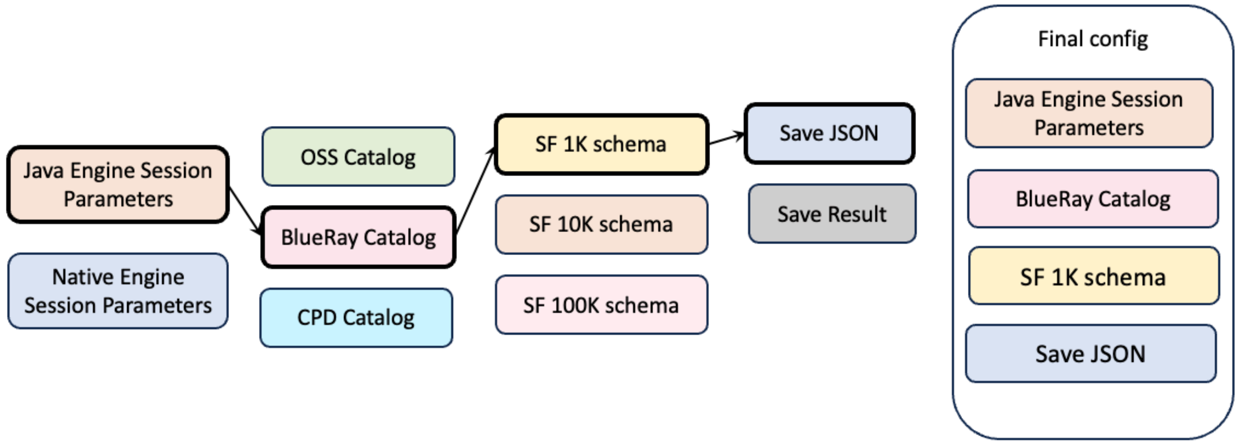


*Figure 2 Main Stage Configuration Stacking Example*

## Data Collection

Based on the configuration, the benchmark runner can capture query results, JSON logs, column metadata, and run-level information like query filename, query IDs, durations, etc. The run-level information is sent to InfluxDB in a bucket called "benchmark_runs" to correlate with the cluster real-time metrics and query logs. The other information is persisted on disk.

In the future we will also support advanced performance data collection:

- CPU/memory profiles for each query
- Custom commands

**Multiple Run Modes**

We will support cold/warm runs for the Power mode(single stream query execution)…..

## Monitoring Mechanisms

During benchmark runs, we need to closely monitor the cluster to obtain all relevant information for further analysis.

There are 3 types of monitoring data we need to collect:

1. Container real-time stats (CPU, memory, I/O, network, etc.)
2. Engine real-time Prometheus metrics
    a. Java Presto: JMX metrics exposed in Prometheus format via the JMX exporter.
    b. Prestissimo: We created a Prometheus endpoint directly inside the server. This change is currently on our auto-rebased branch only, and it is not merged to OSS yet.
3. Query log (query's ID, start and finish time, environment, error, operator level stats, query plans, stage stats, and query statistics, etc.)

We chose InfluxDB for 1 and 2 because it is very good for time-series data, and it has a very pluggable and lightweight solution for push model metrics scraping - Telegraf. We chose MySQL for 3 because it is more structured data and is not time-series.

### Pull Model and Push Model for Metrics Collection

When collecting time-series metrics, there are two options: pull model and push model.

**Pull model:**

We register Prometheus endpoints with a Prometheus instance when new clusters are created, and the Prometheus instance will pull metrics periodically from the endpoints. When we destroy those clusters, we also need to deregestier those endpoints.

**Push model:**

We deploy a local metrics scraper (Telegraf) inside the Presto/Prestissimo containers, and it will push metrics to the data store. No endpoint registration/deregistration is needed.

Considering the number of benchmarks we run and the come-and-go nature of those clusters we use for benchmarking, we chose the push model for simplicity. As a side note, Telegraf has native support for Docker container stats scraping, which further simplified our solution for container monitoring.

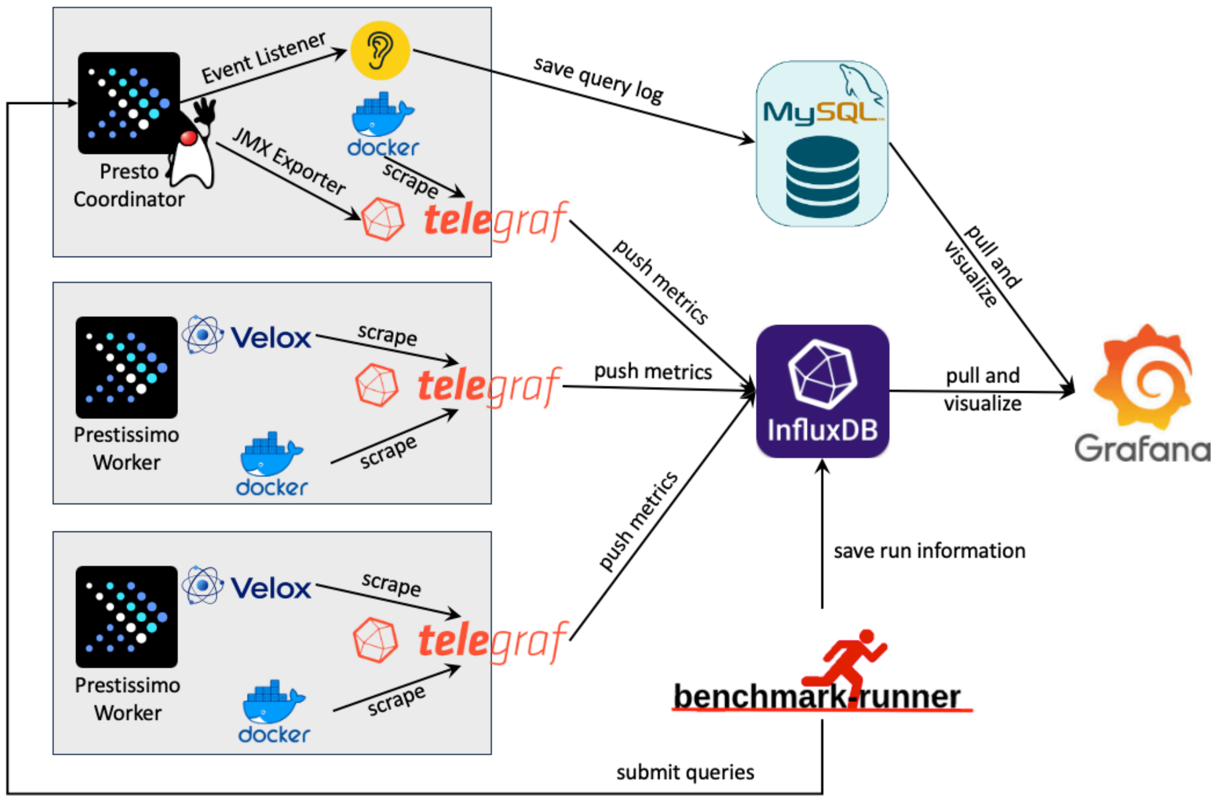
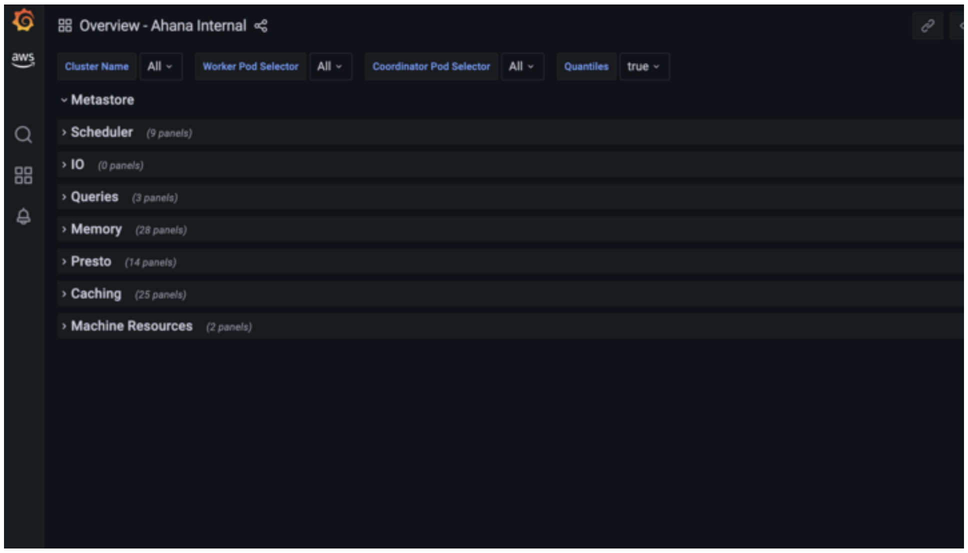


*Figure 3 Architecture diagram for monitoring and visualization.*

## Reliability dashboards

The performance dashboards can be used to monitor the cluster status. This is different than the performance dashboards, which is used to interpret the performance testing results. The performance dashboards, on the other hand, is a monitoring tool that help the cluster owner to check the cluster health. The metrics to monitor are a bit different than the ones used in the performance dashboards.

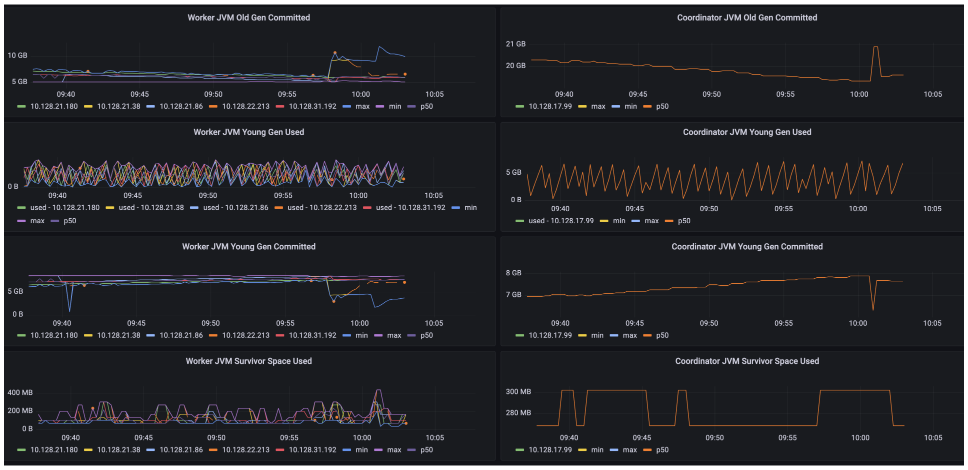
Ahana has built an initial set of Reliability dashboards (click to check it out) for Presto. The Prestissimo ones need to be designed. These dashboards were designed with the goal to help cluster maintainers to quickly find issues. The metrics are listed in this https://ibm.ent.box.com/s/16cyo3jo9wfbgoyl9lyrxkt0wpz0fzxn spreadsheet. Below are some of the screen shots:

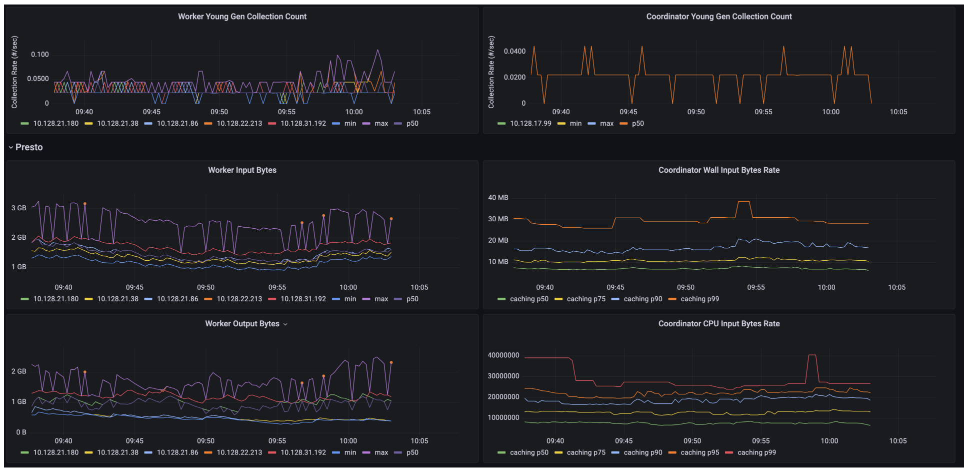


Overview - Ahana Internal

Cluster Name: All · Worker Pod Selector: All · Coordinator Pod Selector: All · Quantiles: true

- Metastore
- Scheduler (9 panels)
- IO (0 panels)
- Queries (3 panels)
- Memory (28 panels)
- Presto (14 panels)
- Caching (25 panels)
- Machine Resources (2 panels)





## Interactive Data Visualization via Grafana

The Grafana dashboard is where data from InfluxDB and MySQL comes together. This integration facilitates real-time and historical analysis of the system's performance. The dashboard is designed to be highly interactive, allowing users to navigate through various levels of data, from high-level overviews of benchmark runs to detailed query analysis and individual execution metrics.

Leveraging the cluster name (via the cluster URL) and the time frame of the query sent by the benchmark runner, users can delve into specific periods of query execution, examining the performance of the cluster that was used to run the query, including CPU and memory usage, during those times. This level of detail in the Grafana dashboard significantly enhances the understanding of each query's impact on the cluster and aids in pinpointing specific performance issues or bottlenecks. Figure 3 shows a holistic view of this architecture.

### Performance Dashboards

- Multi-level dashboards that compare the runs, operators, and queries
- Plan digest and plan comparison tools
- Cluster behavior replay from InfluxDB data

Below are some screenshots showing the current progress of the multi-level performance dashboards on Grafana:

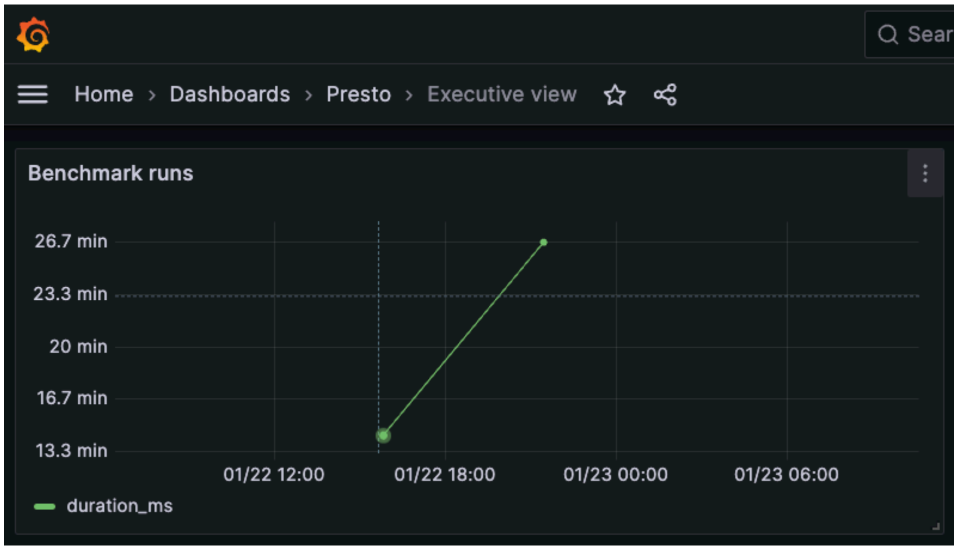
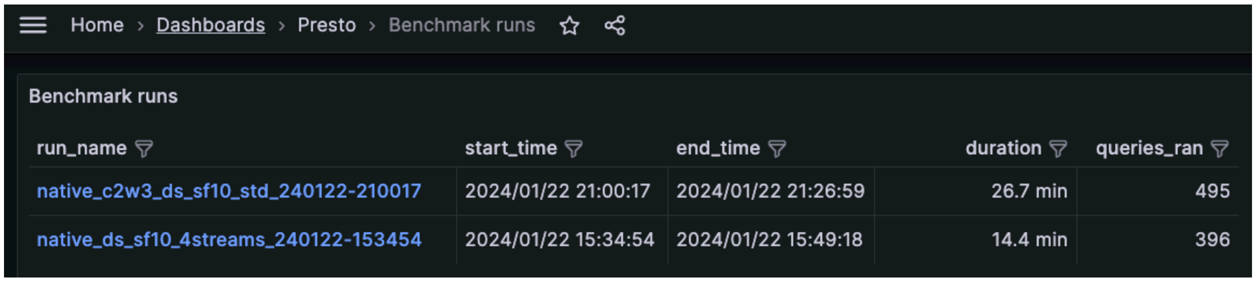


*Figure 4 Executive view for benchmark runs (example)*

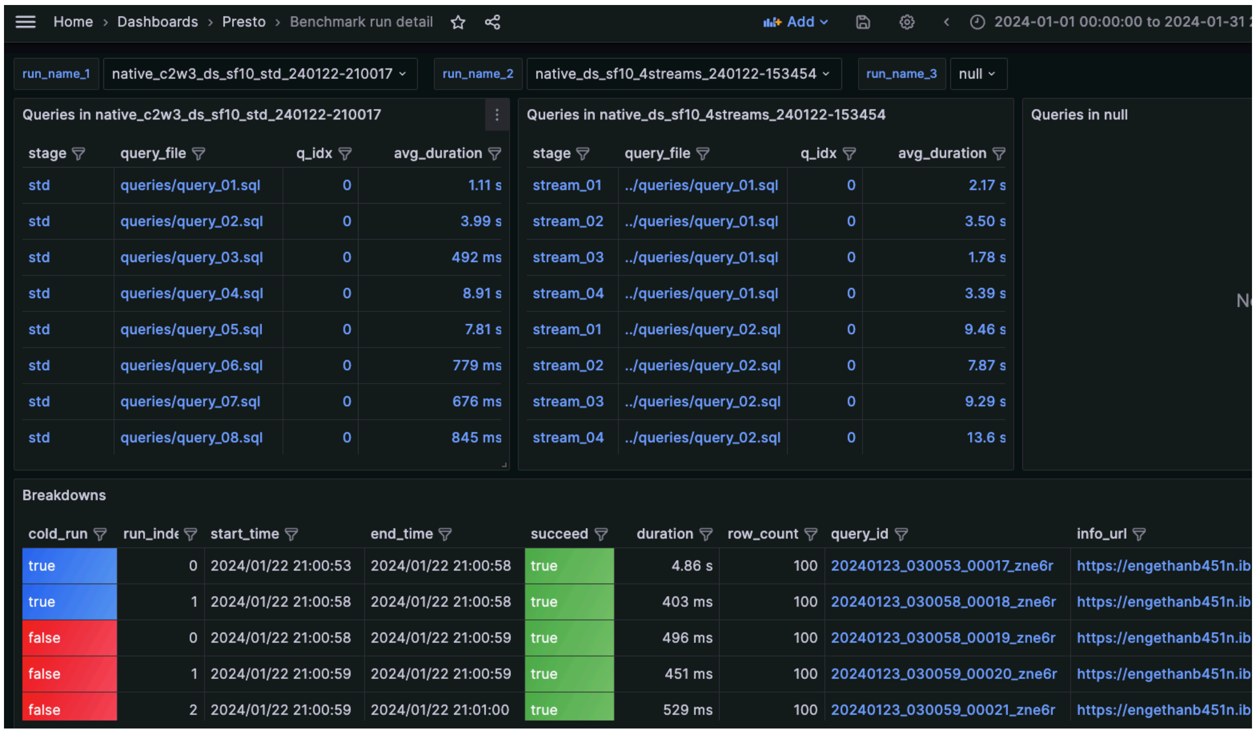


Home › Dashboards › Presto › Benchmark runs

Benchmark runs

| run_name | start_time | end_time | duration | queries_ran |
|---|---|---|---|---|
| native_c2w3_ds_sf10_std_240122-210017 | 2024/01/22 21:00:17 | 2024/01/22 21:26:59 | 26.7 min | 495 |
| native_ds_sf10_4streams_240122-153454 | 2024/01/22 15:34:54 | 2024/01/22 15:49:18 | 14.4 min | 396 |

*Figure 5 Run-level information from the benchmark runner.*


Home › Dashboards › Presto › Benchmark run detail

Add · 2024-01-01 00:00:00 to 2024-01-31

run_name_1: native_c2w3_ds_sf10_std_240122-210017 · run_name_2: native_ds_sf10_4streams_240122-153454 · run_name_3: null

Queries in native_c2w3_ds_sf10_std_240122-210017

| stage | query_file | q_idx | avg_duration |
|---|---|---|---|
| std | queries/query_01.sql | 0 | 1.11 s |
| std | queries/query_02.sql | 0 | 3.99 s |
| std | queries/query_03.sql | 0 | 492 ms |
| std | queries/query_04.sql | 0 | 8.91 s |
| std | queries/query_05.sql | 0 | 7.81 s |
| std | queries/query_06.sql | 0 | 779 ms |
| std | queries/query_07.sql | 0 | 676 ms |
| std | queries/query_08.sql | 0 | 845 ms |

Queries in native_ds_sf10_4streams_240122-153454

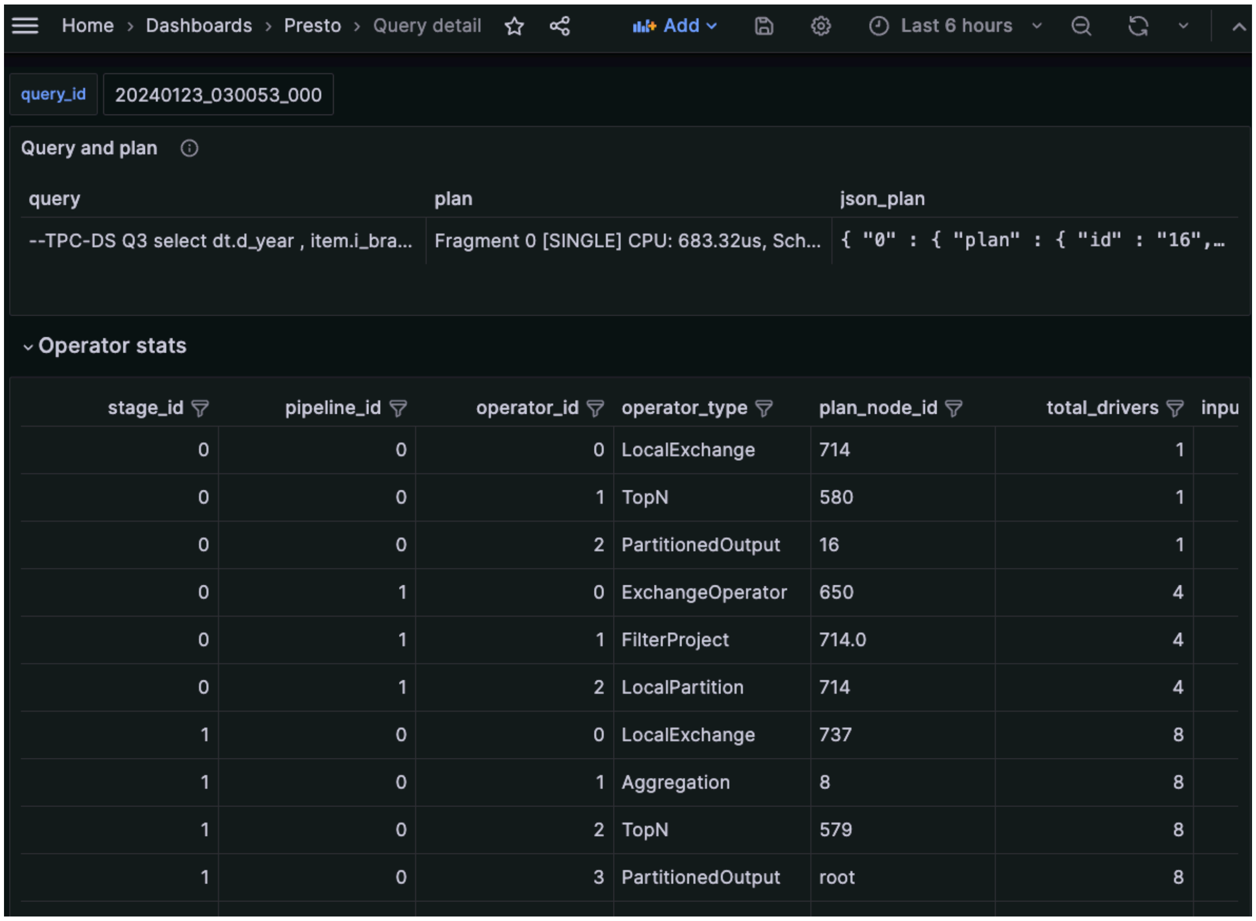
| stage | query_file | q_idx | avg_duration |
|---|---|---|---|
| stream_01 | ../queries/query_01.sql | 0 | 2.17 s |
| stream_02 | ../queries/query_01.sql | 0 | 3.50 s |
| stream_03 | ../queries/query_01.sql | 0 | 1.78 s |
| stream_04 | ../queries/query_01.sql | 0 | 3.39 s |
| stream_01 | ../queries/query_02.sql | 0 | 9.46 s |
| stream_02 | ../queries/query_02.sql | 0 | 7.87 s |
| stream_03 | ../queries/query_02.sql | 0 | 9.29 s |
| stream_04 | ../queries/query_02.sql | 0 | 13.6 s |

Queries in null

Breakdowns

| cold_run | run_inde | start_time | end_time | succeed | duration | row_count | query_id | info_url |
|---|---|---|---|---|---|---|---|---|
| true | 0 | 2024/01/22 21:00:53 | 2024/01/22 21:00:58 | true | 4.86 s | 100 | 20240123_030053_00017_zne6r | https://engethanb451n.ib |
| true | 1 | 2024/01/22 21:00:58 | 2024/01/22 21:00:58 | true | 403 ms | 100 | 20240123_030058_00018_zne6r | https://engethanb451n.ib |
| false | 0 | 2024/01/22 21:00:58 | 2024/01/22 21:00:59 | true | 496 ms | 100 | 20240123_030058_00019_zne6r | https://engethanb451n.ib |
| false | 1 | 2024/01/22 21:00:59 | 2024/01/22 21:00:59 | true | 451 ms | 100 | 20240123_030059_00020_zne6r | https://engethanb451n.ib |
| false | 2 | 2024/01/22 21:00:59 | 2024/01/22 21:01:00 | true | 529 ms | 100 | 20240123_030059_00021_zne6r | https://engethanb451n.ib |

*Figure 6 Run comparison view: query-level.*



query_id 20240123_030053_000

Query and plan

| query | plan | json_plan |
|---|---|---|
| --TPC-DS Q3 select dt.d_year , item.i_bra... | Fragment 0 [SINGLE] CPU: 683.32us, Sch... | { "0" : { "plan" : { "id" : "16",… |

Operator stats

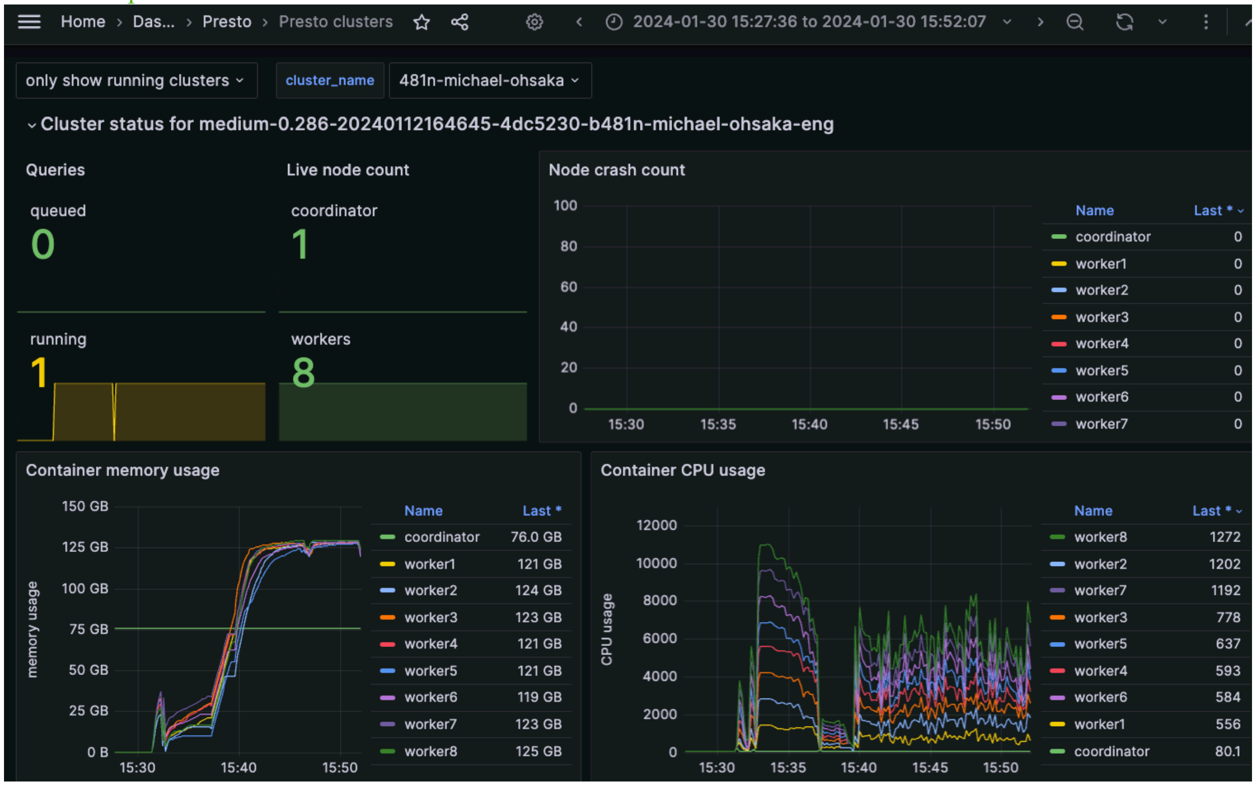
| stage_id | pipeline_id | operator_id | operator_type | plan_node_id | total_drivers | inpu |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | LocalExchange | 714 | 1 | |
| 0 | 0 | 1 | TopN | 580 | 1 | |
| 0 | 0 | 2 | PartitionedOutput | 16 | 1 | |
| 0 | 1 | 0 | ExchangeOperator | 650 | 4 | |
| 0 | 1 | 1 | FilterProject | 714.0 | 4 | |
| 0 | 1 | 2 | LocalPartition | 714 | 4 | |
| 1 | 0 | 0 | LocalExchange | 737 | 8 | |
| 1 | 0 | 1 | Aggregation | 8 | 8 | |
| 1 | 0 | 2 | TopN | 579 | 8 | |
| 1 | 0 | 3 | PartitionedOutput | root | 8 | |

*Figure 7 Operator-level view.*

## Cluster Management

- Test clusters pool overview
- Check out (borrow) and return of the exisiting test clusters
- The configuration and health status of each cluster

- Allow the persistence of the clusters after test


Home > Das... > Presto > Presto clusters
2024-01-30 15:27:36 to 2024-01-30 15:52:07
only show running clusters
cluster_name
481n-michael-ohsaka
Cluster status for medium-0.286-20240112164645-4dc5230-b481n-michael-ohsaka-eng
Queries
queued
0
running
1
Live node count
coordinator
1
workers
8
Node crash count
Container memory usage
memory usage
Container CPU usage
CPU usage

Node crash count

| Name | Last * |
|---|---|
| coordinator | 0 |
| worker1 | 0 |
| worker2 | 0 |
| worker3 | 0 |
| worker4 | 0 |
| worker5 | 0 |
| worker6 | 0 |
| worker7 | 0 |

Container memory usage

| Name | Last * |
|---|---|
| coordinator | 76.0 GB |
| worker1 | 121 GB |
| worker2 | 124 GB |
| worker3 | 123 GB |
| worker4 | 121 GB |
| worker5 | 121 GB |
| worker6 | 119 GB |
| worker7 | 123 GB |
| worker8 | 125 GB |

Container CPU usage

| Name | Last * |
|---|---|
| worker8 | 1272 |
| worker2 | 1202 |
| worker7 | 1192 |
| worker3 | 778 |
| worker5 | 637 |
| worker4 | 593 |
| worker6 | 584 |
| worker1 | 556 |
| coordinator | 80.1 |

*Figure 8 Cluster metrics view.*

## Future Work

To incoroprate performance testing across the form factors into this framework, we need to invest in the following areas:

1. **Dashboard** – the current dashboard for Presto and Prestissimo is not complete yet. We need to finish the interactivity features and add more panels.
2. **InfluxDB and Grafana deployments** – With more form factors plugged in, we need to explore a cost-efficient way to host those internal services. We are looking into IBM Cloud.
3. **Pipelines for convenient image building and cluster deployment** – We need to be able to build image and deploy clusters for testing with the similar flexibily and usability we have in OSS today.