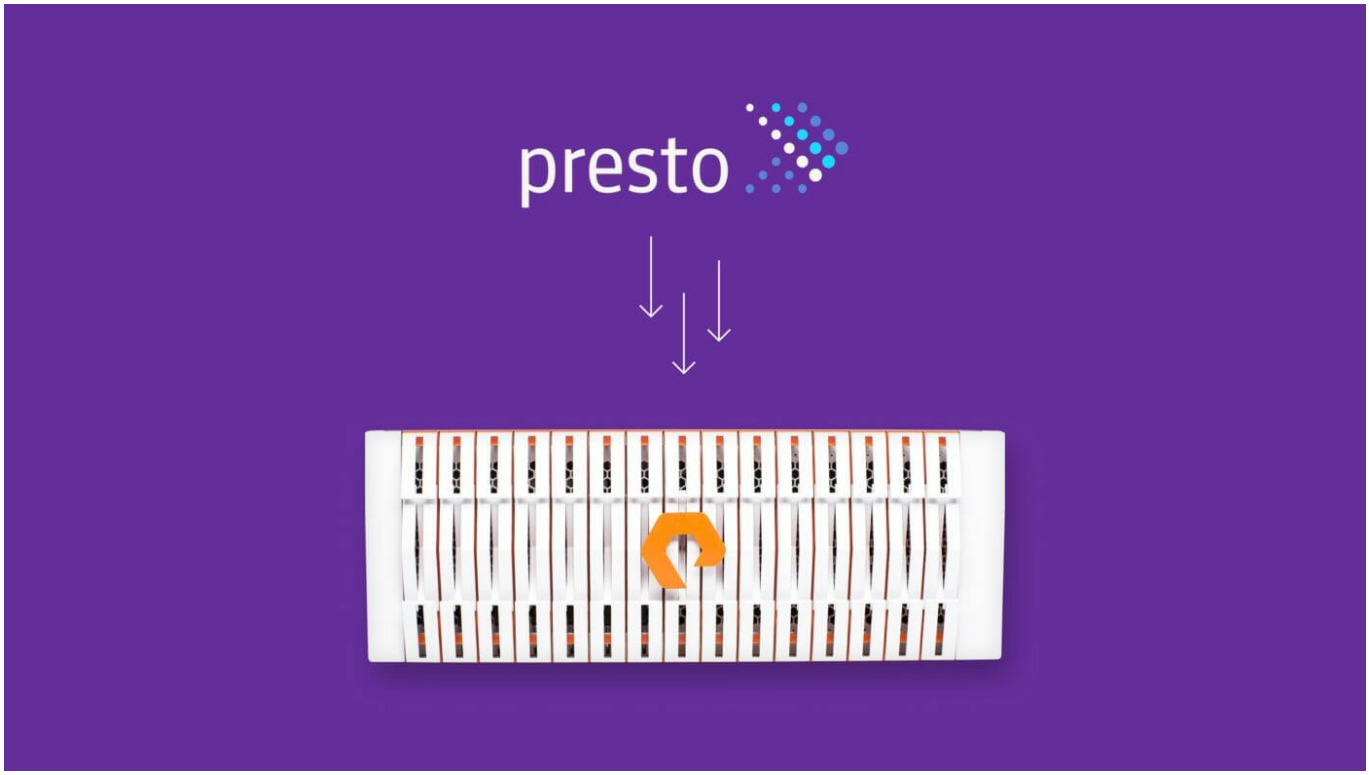
How to Configure Presto on FlashBlade S3



This article originally appeared on Medium.com and is republished with permission from the author.

New-generation data warehouse tools expand queries to unstructured data lakes. These warehouses avoid siloing data in order to take advantage of highly scalable storage tiers such as an S3 object store and widely supported file formats like Parquet and ORC. Now, the data warehouse and the data lake are merging; instead of silos and multiple copies of data, a single data set can be queried with SQL, as well as other advanced tools (e.g., Spark, TensorFlow, and others).

Presto is an open source, scale-out SQL engine for interactively querying large data sets, originally created inside Facebook. There are a variety of connectors to different data sources; in order to query data from S3, Presto leverages the Hive connector and the Hive Metastore. Presto operates at impressively large scales and shows promising performance results against Hive on S3 in previous testing.

What are the advantages of Presto on FlashBlade S3?

Presto on FlashBlade S3 has the advantages of

1. high performance for demanding query and ingest workloads
2. the ability to scale compute and storage independently, and
3. the simplicity, reliability, non-disruptive upgrades, and scalability of FlashBlade®.

This means storage that always provides high performance with no tuning and the occasional one-click

process of adding blades to scale out the system. Simplifying infrastructure allows end users to work more productively with faster data and more agile applications.

How to Configure Presto on S3

In order for Presto to query data on S3, it relies on the Hive Metastore. Therefore, we first configure a Hive standalone Metastore and then separately the Presto servers. If there is already an external Hive Metastore service you will use, then skip setup of the first two components.

I have built two different Docker images: one that contains the necessary dependencies for running the Hive Metastore and the other for the Presto servers. The next section describes these Docker images and how to use them.

Component 1: RDBMS for Metastore

For this example, the metastore is backed by a MariaDB Docker container with a FlashArray™ volume for persistence:

```
docker volume create --driver=pure -o size=1TiB \

-o volume_label_selector=”purestorage.com/backend=block” \

   metastore-vol

docker run --name metastoremaria -d --rm \

   -v metastore-vol:/var/lib/mysql \

   -e MYSQL_ROOT_PASSWORD=mypass \

   -p 13306:3306 \

   mariadb/server:latest
```

If you do not have a shared storage Docker volume plugin, a simple local volume still provides local persistence, though without the benefits of being able to restart the instance on a different server or other enterprise features like snapshots and replication:

```
docker volume create metastore-vol
```

…

Note that this configuration is not properly secured for a production use case.

Component 2: Hive Metastore Service

For this test, I used the standalone metastore server with a straightforward configuration for using MySQL and S3A. The Dockerfile I used can be found here and relies on two configuration files described next: metastore-site.xml and core-site.xml.

First, configure metastore-site.xml with the necessary jdbc connectivity information to reach the MariaDB instance:

- javax.jdo.option.ConnectionDriverName
- javax.jdo.option.ConnectionURL
- javax.jdo.option.ConnectionUserName
- javax.jdo.option.ConnectionPassword

My version of the file can be found here.

Note that my Dockerfile downloads the necessary connector jar for the RDBMS in use, in this case MySQL, and places it in the Hive classpath. If using a different RDBMS, a different jar file will be needed.

Then, for the metastore to access FlashBlade via S3, set the following S3A configuration properties in core-site.xml:

- fs.s3a.access.key
- fs.s3a.secret.key
- fs.s3a.connection.ssl.enabled = false (recommended)
- fs.s3a.endpoint = {FlashBlade data VIP}

With the 'hivemetastore' Docker image configured and built, use the Hive schematool to add the necessary metastore tables to the MariaDB instance at HOSTNAME:

```
docker run -it --rm --name initschema hivemetastore \

 /opt/hive-metastore/bin/schematool --verbose -initSchema \

   -dbType mysql -userName root -passWord mypass \

   -url jdbc:mysql://$HOSTNAME:13306/metastore_db?createDatabaseIfNotExist=true
```

The initschema command only needs to be run once during initial setup of the backing database. The container should run to completion and stop automatically.

Finally, start the standalone metastore service:

```
docker run -d --rm --name=hivemeta -p 9083:9083 hivemetastore \

   /opt/hive-metastore/bin/start-metastore
```

The above steps create a standalone Hive metastore service which can be used independently of Presto or used together with Presto and other tools, like Apache Spark. In other words, the setup so far is not tied to Presto in any way.

Component 3: Presto Servers

I next configure Presto using the following Docker image, which has been tested with versions 0.219, 0.220, and 0.221. The Presto architecture has two types of servers: a coordinator node and multiple worker nodes, which have small configuration differences.

We need to configure jvm.config, config.properties, node.properties, and hive.properties. Of these, only hive.properties contains FlashBlade/S3 specific configuration.

The `jvm.config` is common across all Presto nodes:

```
-server

-Xmx16G

-XX:+UseG1GC

-XX:G1HeapRegionSize=32M

-XX:+UseGCOverheadLimit

-XX:+ExplicitGCInvokesConcurrent

-XX:+HeapDumpOnOutOfMemoryError

-XX:+ExitOnOutOfMemoryError
```

Config.properties is different between coordinator and worker nodes. Set the value of MASTERHOST to connect to the coordinator node:

```
Coordinator config:

coordinator=true

node-scheduler.include-coordinator=false

http-server.http.port=8080

query.max-memory=50GB

query.max-memory-per-node=1GB

query.max-total-memory-per-node=2GB

discovery-server.enabled=true

discovery.uri=http://$MASTERHOST:8080

Worker config:

coordinator=false

http-server.http.port=8080

query.max-memory=50GB

query.max-memory-per-node=1GB

query.max-total-memory-per-node=2GB

discovery.uri=http://$MASTERHOST:8080
```

And node.properties is different for each node in order to uniquely identify the node:

```
node.environment=test

node.id=nodeXX
```

```
node.data-dir=/var/presto/data
```

The data-dir location is used only for storing log file and operational data. As long as ‘hive’ is used as the default catalog, all new tables will be automatically stored on FlashBlade S3.

I used a small script to automate the process of creating a unique node.id value and node.properties file per node:

```
for i in {01..20}; do

   NODEIP=$(dig +short myhost$i.dev.mydomain.com)

   cp node.properties.template node.properties.$NODEIP

   echo “node.id=node$i” >> node.properties.$NODEIP

done
```

Finally, the hive.properties file controls S3 connectivity to FlashBlade:

```
connector.name=hive-hadoop2

hive.metastore.uri=thrift://$METASTORE:9083

hive.s3.aws-access-key=XXXXXXXXX

hive.s3.aws-secret-key=YYYYYYYYYYYYY

hive.s3.endpoint=$DATAVIP

hive.s3.path-style-access=true

hive.s3.ssl.enabled=false
```

Note in the above configuration, SSL access for traffic is disabled for expediency, but it is also possible to import an SSL certificate into the FlashBlade to enable encrypted traffic. There is also further configuration available for S3 access.

The Presto servers are then started on each node with the following command-line, which injects the node-specific and security-sensitive configuration into the container at runtime:

```
docker run -d --rm --net=host \

 --name=presto \

 -v ${PWD}/node.properties.$(hostname -i):/opt/presto-server/etc/node.properties
\

 -v ${PWD}/config.properties.coordinator:/opt/presto-server/etc/config.properties
\

 $IMGNAME \

 /opt/presto-server/bin/launcher run
```

The above command assumes $IMGNAME matches the Presto Dockerfile built previously and that the properties files are in the local working directory.

Finally, download the presto-cli client to connect to the Presto service using hive as the default catalog:

```
presto-cli --server localhost:8080 --catalog hive --schema default
```

Experimental Setup

The data used is a 35GB text table of Wikipedia pageview counts that was previously loaded onto a FlashBlade S3 bucket. The test methodology is to create an external table from the Wikipedia pageviews data set and then run a simple COUNT(*) query on the data set to check IO performance.

> CREATE TABLE pagecounts (code varchar, title varchar, count bigint, pgsize bigint) WITH (format = 'textfile', external_location = 's3a://BUCKETNAME/wiki_pagecounts/2016-08/');

When loading data from textfiles, Presto requires columns to be delimited by the character '\x01' so I found it necessary to transform the spaces in each file before loading the data to S3:

```
for i in pagecounts*; do sed -i 's/ /\x01/g' $i; done
```

My workflow involved manually uploading the '\x01' transformed text files to S3 using s5cmd.

Finally, you can view a system table in order to examine the cluster topology and verify the number of active workers:

```
presto:default> SELECT * FROM system.runtime.nodes;

node_id | http_uri | node_version | coordinator | state

– – – – -+ – – – – – – – – – – – – – -+ – – – – – – – + – – – – – – -+ – – – – –

node23 | http://10.62.185.120:8080 | 0.219 | false | active

node24 | http://10.62.185.121:8080 | 0.219 | false | active

node1  | http://10.62.205.205:8080 | 0.219 | true | active

node25 | http://10.62.185.122:8080 | 0.219 | false | active

(4 rows)
```

Testbed Configurations

The multi-node Presto configuration uses one dedicated coordinator node and varies the number of worker nodes. The goal of these tests is to understand when there is a bottleneck imposed by the storage tier hosting the data.

I tested three different configurations, summarized in the table below: one on-premises with FlashBlade S3 as storage and two on AWS-EMR using AWS-S3 as storage. For AWS-EMR, I tested with two different instances, m3.2xlarge and c3.8xlarge. The on-prem experiment with FlashBlade uses approximately equivalent compute to the c3.8xlarge config, whereas the m3.2xlarge is a smaller instance profile. The larger AWS config is expected to closely, though not exactly, match the worker performance of the on-

prem nodes in terms of compute and networking, isolating any differences to the storage tier.

```
+-------------------------------------------------------------+
|            | Environment  | Compute Node       | Storage      |
+-------------------------------------------------------------+
| on-prem/FB | on-premises  | 32 core, 60GiB mem | FlashBlade S3|
| c3.8xlarge | public cloud | 32 core, 60GiB mem | AWS S3       |
| m3.2xlarge | public cloud | 16 core, 30GiB mem | AWS S3       |
+-------------------------------------------------------------+
```

For the comparison, I ran the on-prem servers with an older Presto release, 0.219, in order to match the latest version supported by AWS-EMR.

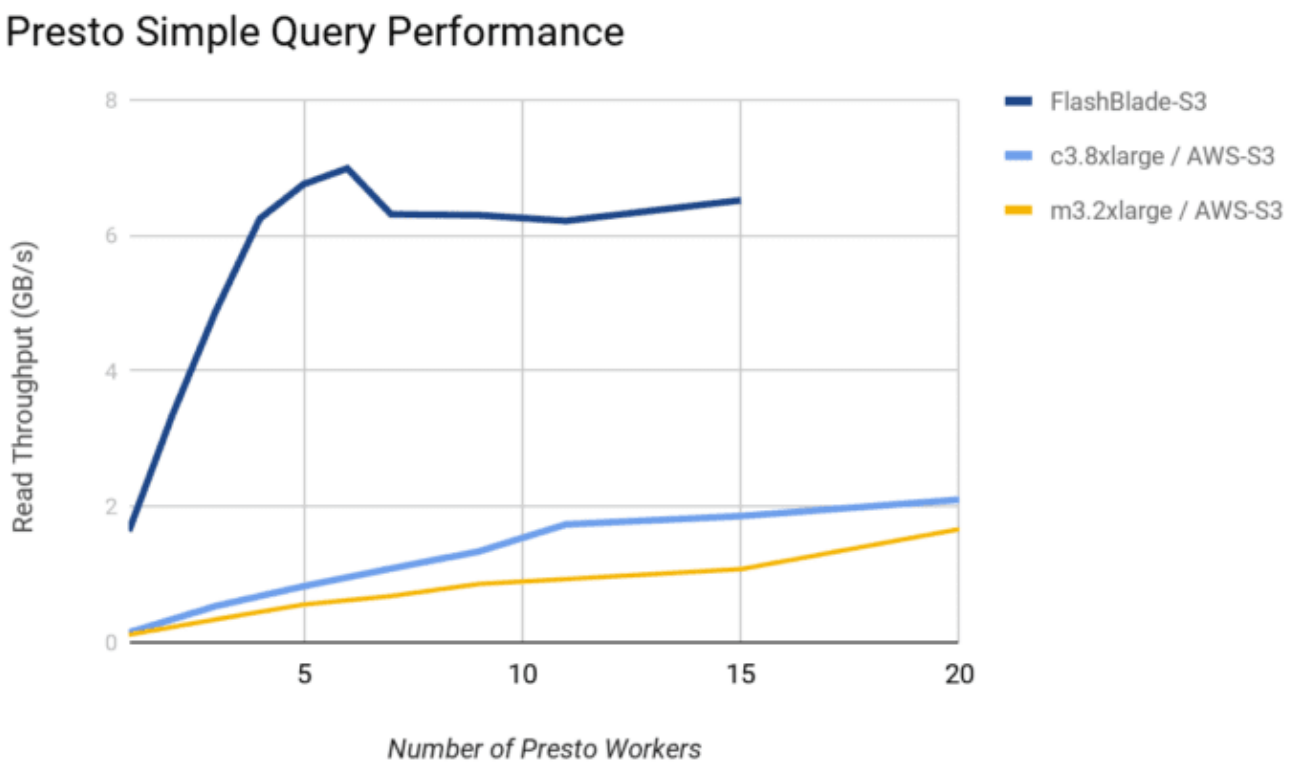
I tested query performance with a simple counting query designed to maximize the IO requirements from the storage:

```
presto:default> SELECT COUNT(*) from pagecounts;
```

Because this table is backed by external text files, the query requires reading all data to count the number of rows. These results provide an upper bound for what performance the storage can deliver, while better table encodings and a more realistic mix of queries will not stress storage at this level.

Performance Results

The graph below shows significantly higher performance on FlashBlade, even with a much smaller number of Presto workers. Specifically, six workers are able to reach the peak throughput of the 15-blade FlashBlade (~10GB/s reads). In contrast, even with 20 similarly sized Presto workers in AWS, the maximum query performance only reaches 2GB/s. This suggests that the storage tier presents a bottleneck at a much lower throughput point and scaling the number of workers does not help significantly.



The cluster with c3.8xlarge instances reached slightly over 2GB/s of query performance with 20 workers. To understand where the limitation arises, I leveraged S3 performance testing on a single c3.8xlarge instance, replacing Presto with an idealized load generator. The results showed maximum single instance S3 read performance of 450MB/s with large object sizes. With a single Presto worker, the query performance (145MB/s) was 1/3 lower than this maximum measured performance (450MB/s) from that same instance type to AWS S3. In contrast, the on-prem results demonstrate that the Presto worker is able to take better advantage of a more performant S3, with a single worker reaching ~1.7GB/s, i.e., close to the maximum dictated by the 20Gbps networking connectivity of each on-prem node.

Looking closer at the peak read performance supported by FlashBlade S3 in the screenshots below, we can see that a single worker can read at 1.69GB/s and six workers reach an aggregate throughput of 9.34GB/s. For this query then, the effective throughput is significantly smaller than the observed peak throughput, indicating that the per-query overhead is a large portion of total runtime for this size of data set. Indeed, by querying larger data sets, the effective performance increases further.

The peak performance of the on-prem environment at six Presto workers is consistent across multiple test runs, suggesting larger Presto clusters have additional overhead due to how work is divided. This peak changes as the queried data set grows.

While this test is not a perfect apples-to-apples comparison, the performance limits of AWS-S3 are clear. As data warehouse queries become critical for the business, a high-performance S3 tier like FlashBlade is necessary to improve responsiveness.

In-Memory Tables

When even higher performance is needed, the Presto in-memory connector enables queries with near real-time responses by creating tables that remain entirely in-memory. This enables Presto to support workloads that have a mix of query types: analytics with multi-second and multi-minute latencies that leverage the persistent data hub, as well as real-time queries with sub-second latency.

To configure the in-memory connector, add the file memory.properties to etc/catalog/ with the following contents. Select the max memory dedicated per node based on your needs.

```
connector.name=memory

memory.max-data-per-node=8GB
```

```
A simple CTAS easily creates an in-memory table:

> CREATE TABLE memory.default.pagecounts AS SELECT * FROM pagecounts;

> SELECT AVG(pgsize) FROM memory.default.pagecounts;
```

The query operating on the in-memory table completes significantly faster because the entire table is readily available in DRAM. The limitation, of course, is that the table must fit within the configured DRAM budget.

Summary

Presto provides an easy and performant SQL interface to unlock the value in a data lake. The result is a single tool that can query multiple different data sources, including unstructured data on an object store stored as text files, Parquet, ORC, or other formats. And these same data sets remain useful for ad-hoc analysis with other tools, like Dask, Spark, or Keras. Unifying data warehouses and data lakes with a high-performance FlashBlade object store creates a true data hub, an agile and flexible platform for data analytics without the rigid and inefficient silos of separate systems.

Color orange–gradient

Color orange–gradient

Color orange–gradient

Color orange–gradient

Color orange–gradient