The science behind the report:

Unlock faster insights with Azure Databricks

This document describes what we tested, how we tested, and what we found. To learn how these facts translate into real-world benefits, read the report Unlock faster insights with Azure Databricks.

We concluded our hands-on testing on April 24, 2025. During testing, we determined the appropriate hardware and software configurations and applied updates as they became available. The results in this report reflect configurations that we finalized on April 18, 2025 or earlier. Unavoidably, these configurations may not represent the latest versions available when this report appears.

Our results

To learn more about how we have calculated the wins in this report, go to http://facts.pt/calculating-and-highlighting-wins. Unless we state otherwise, we have followed the rules and principles we outline in that document.

Table 1: Results, in seconds, of our testing.

AutoScale	Test phase	Azure Databricks cluster	Databricks cluster on Amazon Web Services (AWS™)	% less time, Azure vs AWS
Enabled	Power	2,661.53	3,073.37	13.40%
	Throughput	5,884.16	6,349.02	7.32%
Disabled	Power	1,972.43	2,500.76	21.13%
	Throughput	5,321.74	5,875.86	9.43%

System configuration information

Table 2: Detailed information on the systems we tested.

System configuration information	Azure cluster with Azure Databricks	AWS cluster with Databricks for AWS
General information		
Date testing ended	4/24/2025	4/24/2025
Cloud service provider (CSP)	Azure	AWS
Region	East US	us-east-1
Workload information		
Workload name and version	Databricks Spark SQL Performance TPC-DS-like	Databricks Spark SQL Performance TPC-DS-like
Workload- or software-specific parameters	Spark broadcast timeout - 100000	Spark broadcast timeout - 100000
Iterations and result choice	3 test runs, median reported	3 test runs, median reported
Databricks cluster information		
Databricks runtime engine	15.4 LTS	15.4 LTS
Photon enabled (Y/N)	Y	Y
Spark version	3.5	3.5
Scala version	2.12	2.12
Cloud VM or instance details		
Number of driver VMs	1	1
VM or instance size	Standard_E16ds_v5	r6id.4xlarge
vCPU	16	16
Number of cores/threads	8/2	8/2
Memory (GB)	128	128
Underlying processor model (if attainable)	Intel® Xeon® Platinum 8370C	Intel Xeon Platinum 8375C
Number of worker VMs	20	20
VM or instance size	Standard_E8ds_v5	r6id.2xlarge
vCPU	8	8
Number of cores/threads	8/1	8/1
Memory (GB)	64	64
Underlying processor model (if attainable)	Intel Xeon Platinum 8370C	Intel Xeon Platinum 8375C

System configuration information	Azure cluster with Azure Databricks	AWS cluster with Databricks for AWS
Operating system information		
Image or template name and UUID	Publisher: AzureDatabricks Offer: Databricks SKU: DatabricksWorker Version: 20250507.1835.1	databricks-worker-ubuntu-2025-03-03T15-08-35
Operating system name	Ubuntu	Ubuntu
Operating system build number	22.04	22.04
Kernel version	6.8.0-1028-azure	6.8.0-1027-aws
Date patches last applied	N/A	N/A
Changes made from CSP image	No	No
Instance storage (volume type 1)		
Number of volumes	1	1
Volume use in this test	OS	OS
CSP volume type	Premium SSD LRS	Gp3
Volume size (GB)	30	30
Instance storage (volume type 2)		
Number of volumes	1	1
Volume use in this test	Temporary storage	Temporary storage
CSP volume type	Local SSD	NVMe®
Volume size (GB)	300	477
Object Storage		
Type	Azure Data Lake Storage Gen2	S3
Class	Premium	Standard
Size on disk (TB)	2.8	2.8

How we tested

We wanted to show the performance of Azure Databricks and Databricks on Amazon Web Services (AWS). We started on Azure by installing the Azure cli and Terraform. Because Databricks on AWS is a third-party offering, we used the cloud service provider (CSP) GUI to create the Databricks workspace and necessary infrastructure. We then created object storage buckets for AWS. Once all of the infrastructure was in place, we created Databricks clusters in the workspace and connected the object storage to Databricks.

For our test setup and execution, we used a forked version of the tpcds-kit from the Databricks repo. The tpcds-kit relies on the spark-sql-perf library, which we downloaded and compiled from the spark-sql-perf GitHub repo. We loaded the tpcds-kit onto each worker node and the driver node in Databricks by installing the tpcds-install.sh init script (included in the "Scripts we used for testing" section) onto the cluster. We also installed the spark-sql-perf library on the cluster via the Databricks UI. Once all of the necessary components were in place to set up and run the test, we created Jupyter notebooks in the Databricks workspace to generate the 10TB dataset, put the data on the object storage, and created external tables, which we then turned into delta tables.

Our TPC-DS-like test consisted of two parts, a power test and a throughput test. The power test ran through a series of 99 complex queries with a single stream or user. The throughput test ran through the same 99 queries with four streams, each stream running in a random order. For the power test, we report the time to complete all 99 queries. For the throughput test, we report the time of the longest running stream. We ran each test three times and report the median.

We tested each cluster with AutoScale enabled and disabled.

Creating the Azure environment

We used the Terraform scripts in the "Scripts we used for testing" section to create an Azure resource group, storage account, container, Databricks workspace, and Databricks cluster automatically.

Creating the Databricks cluster on AWS

1. Log into the Databricks workspace.
2. In the left pane, click Compute.
3. Under All-purpose compute, click Create compute.
4. Change the Access mode to No isolation shared.
5. For the Databricks runtime version, select Runtime: 15.4 LTS (Scala 2.12, Spark 3.5.0).
6. Click the checkbox next to Use Photon Acceleration.
7. For the Worker type, select r6id.2xlarge, and set the Max workers to 20.
8. For the Driver type, select r6id.4xlarge.
9. Click the Advanced Options drop-down, and slide the On-demand slider all the way to the right to All On-demand.
10. Click Create Compute.

Creating AWS S3 storage

1. Log into AWS.
2. Navigate to the Amazon S3 page.
3. Click Create Bucket.
4. Enter a name for the bucket.
5. Click Create bucket.

Once you create the AWS S3 storage, you must follow additional steps to set up IAM profiles so that Databricks can talk to the storage. We followed these instructions from Databricks.

Mounting Azure Data Lake Storage (ADLS) storage in Databricks

Before you can mount the ADLS storage in Databricks, you must follow additional steps as with the other CSPs to connect Databricks. We followed the steps in the tutorial at https://learn.microsoft.com/en-us/azure/databricks/connect/storage/tutorial-azure-storage.

1. In the Databricks workspace, right-click → Create → Notebook.
2. Type the following into the notebook, and run the cell:

```
configs = {"fs.azure.account.auth.type": "OAuth",
          "fs.azure.account.oauth.provider.type": "org.apache.hadoop.fs.azurebfs.oauth2.
ClientCredsTokenProvider",
          "fs.azure.account.oauth2.client.id": "<Client Id>",
          "fs.azure.account.oauth2.client.secret": dbutils.secrets.get(scope="<Scope
name>",key="<Secret name>"),
          "fs.azure.account.oauth2.client.endpoint": "https://login.microsoftonline.com/<Key vault
Directory ID>/oauth2/token"}


dbutils.fs.mount(
  source = "abfss://<storage blob name>@<storage account name>.dfs.core.windows.net/",
  mount_point = "/mnt/databricks",
  extra_configs = configs)
```

Mounting AWS S3 storage in Databricks

1. In the Databricks workspace, right-click → Create → Notebook.
2. Type the following into the notebook, and run the cell:

```
aws_bucket_name = "<AWS S3 bucket name>"
mount_name = "databricks"
dbutils.fs.unmount(f"/mnt/{mount_name}")
dbutils.fs.mount(f"s3a://{aws_bucket_name}", f"/mnt/{mount_name}")
display(dbutils.fs.ls(f"/mnt/{mount_name}"))
```

Compiling the spark-sql-perf library

1. On a Linux machine (we used Ubuntu 22.04), use git to clone the spark-sql-perf repository to your local directory:

```
git clone https://github.com/databricks/spark-sql-perf.git
```

2. Navigate to the spark-sql-perf directory:

```
cd spark-sql-perf/
```

3. Edit the build.sbt file, and change the Scala version to 2.12 and the Spark version to 3.5.
4. Build the library:

```
build/sbt +package
```

5. Copy the JAR file to your local machine for upload in the next section.

Uploading spark-sql-perf library and tpcds-install.sh init script to the Databricks cluster

1. Create a folder in the Databricks workspace.
2. Upload the spark-sql-perf library and the tpcds-install.sh init script to that folder.
3. Click the compute tab, and then the cluster.
4. Click Edit.
5. Click the drop-down menu next to Advanced options.

6. Click the Init scripts tab, navigate to the folder path you uploaded the init script, and click Add.
7. Click Confirm.
8. Click the Libraries tab.
9. Click Install new.
10. Navigate to the workspace folder with the library, and click Install.

Note: For AWS, we also had to edit the security group for Databricks to allow egress and ingress traffic so that the tpcds-install.sh script could download the tpcds-kit from GitHub and compile the data generation program dsdgen. Without this step, the cluster will fail to start.

Creating the 10TB dataset and tpc-ds database

1. In the Databricks workspace, right-click → Create → Notebook.
2. Type the following into the notebook, and run the cell:

```
// Databricks notebook source
// MAGIC %md
// MAGIC This notebook generates the TPCDS data using the spark-sql-perf library.

// COMMAND ----------

// IMPORTANT: SET PARAMETERS!!!
// TPCDS Scale factor
val scaleFactor = "10000"

// data format.
val format = "parquet"
// If false, float type will be used instead of decimal.
val useDecimal = true
// If false, string type will be used instead of date.
val useDate = true
// If true, rows with nulls in partition key will be thrown away.
val filterNull = false
// If true, partitions will be coalesced into a single file during generation.
val shuffle = true

// dbfs path to generate the data to.
val rootDir = s"/mnt/databricks/sf$scaleFactor-$format/useDecimal=$useDecimal,useDate=$useDate,filter
Null=$filterNull"
// name of database to be created.
val databaseName = s"tpcds_sf${scaleFactor}" +
  s"""_${if (useDecimal) "with" else "no"}decimal""" +
  s"""_${if (useDate) "with" else "no"}date""" +
  s"""_${if (filterNull) "no" else "with"}nulls"""

// COMMAND ----------

// Create the table schema with the specified parameters.
import com.databricks.spark.sql.perf.tpcds.TPCDSTables
val tables = new TPCDSTables(sqlContext, dsdgenDir = "/usr/local/bin/tpcds-kit/tools", scaleFactor =
scaleFactor, useDoubleForDecimal = !useDecimal, useStringForDate = !useDate)

// COMMAND ----------

// Data generation tuning:

import org.apache.spark.deploy.SparkHadoopUtil
// Limit the memory used by parquet writer
SparkHadoopUtil.get.conf.set("parquet.memory.pool.ratio", "0.1")
// Compress with snappy:
sqlContext.setConf("spark.sql.parquet.compression.codec", "snappy")
// TPCDS has around 2000 dates.
spark.conf.set("spark.sql.shuffle.partitions", "2000")
// Don't write too huge files.
sqlContext.setConf("spark.sql.files.maxRecordsPerFile", "20000000")

val dsdgen_partitioned=10000 // recommended for SF10000+.
val dsdgen_nonpartitioned=10 // small tables do not need much parallelism in generation.

// COMMAND ----------
```

```
// val tableNames = Array("") // Array("") = generate all.
//val tableNames = Array("call_center", "catalog_page", "catalog_returns", "catalog_sales",
"customer", "customer_address", "customer_demographics", "date_dim", "household_demographics",
"income_band", "inventory", "item", "promotion", "reason", "ship_mode", "store", "store_
returns", "store_sales", "time_dim", "warehouse", "web_page", "web_returns", "web_sales", "web_
site") // all tables

// generate all the small dimension tables
val nonPartitionedTables = Array("call_center", "catalog_page", "customer", "customer_address",
"customer_demographics", "date_dim", "household_demographics", "income_band", "item", "promotion",
"reason", "ship_mode", "store",  "time_dim", "warehouse", "web_page", "web_site")
nonPartitionedTables.foreach { t => {
  tables.genData(
      location = rootDir,
      format = format,
      overwrite = true,
      partitionTables = true,
      clusterByPartitionColumns = shuffle,
      filterOutNullPartitionValues = filterNull,
      tableFilter = t,
      numPartitions = dsdgen_nonpartitioned)
}}
println("Done generating non partitioned tables.")

// leave the biggest/potentially hardest tables to be generated last.
val partitionedTables = Array("inventory", "web_returns", "catalog_returns", "store_returns", "web_
sales", "catalog_sales", "store_sales")
partitionedTables.foreach { t => {
  tables.genData(
      location = rootDir,
      format = format,
      overwrite = true,
      partitionTables = true,
      clusterByPartitionColumns = shuffle,
      filterOutNullPartitionValues = filterNull,
      tableFilter = t,
      numPartitions = dsdgen_partitioned)
}}
println("Done generating partitioned tables.")

// COMMAND ----------

// MAGIC %md
// MAGIC Create database

// COMMAND ----------

sql(s"drop database if exists $databaseName cascade")
sql(s"create database $databaseName")

// COMMAND ----------

sql(s"use $databaseName")

// COMMAND ----------

tables.createExternalTables(rootDir, format, databaseName, overwrite = true,
discoverPartitions = true)

// COMMAND ----------

// MAGIC %md
// MAGIC Analyzing tables is needed only if cbo is to be used.

// COMMAND ----------

tables.analyzeTables(databaseName, analyzeColumns = true)
Verify the database has been created and is selected as the current schema:
%sql
SHOW DATABASES;
SELECT CURRENT_SCHEMA();
```

```
Get a list of all the tables in the database to convert to delta tables:
%sql
USE tpcds_sf10000_withdecimal_withdate_withnulls;
SHOW TABLES IN tpcds_sf10000_withdecimal_withdate_withnulls;
Convert the tables to delta tables:
%sql
CONVERT TO DELTA tpcds_sf10000_withdecimal_withdate_withnulls.call_center;
CONVERT TO DELTA tpcds_sf10000_withdecimal_withdate_withnulls.catalog_page;
CONVERT TO DELTA tpcds_sf10000_withdecimal_withdate_withnulls.catalog_returns;
CONVERT TO DELTA tpcds_sf10000_withdecimal_withdate_withnulls.catalog_sales;
CONVERT TO DELTA tpcds_sf10000_withdecimal_withdate_withnulls.customer;
CONVERT TO DELTA tpcds_sf10000_withdecimal_withdate_withnulls.customer_address;
CONVERT TO DELTA tpcds_sf10000_withdecimal_withdate_withnulls.customer_demographics;
CONVERT TO DELTA tpcds_sf10000_withdecimal_withdate_withnulls.date_dim;
CONVERT TO DELTA tpcds_sf10000_withdecimal_withdate_withnulls.household_demographics;
CONVERT TO DELTA tpcds_sf10000_withdecimal_withdate_withnulls.income_band;
CONVERT TO DELTA tpcds_sf10000_withdecimal_withdate_withnulls.inventory;
CONVERT TO DELTA tpcds_sf10000_withdecimal_withdate_withnulls.item;
CONVERT TO DELTA tpcds_sf10000_withdecimal_withdate_withnulls.promotion;
CONVERT TO DELTA tpcds_sf10000_withdecimal_withdate_withnulls.reason;
CONVERT TO DELTA tpcds_sf10000_withdecimal_withdate_withnulls.ship_mode;
CONVERT TO DELTA tpcds_sf10000_withdecimal_withdate_withnulls.store;
CONVERT TO DELTA tpcds_sf10000_withdecimal_withdate_withnulls.store_returns;
CONVERT TO DELTA tpcds_sf10000_withdecimal_withdate_withnulls.store_sales;
CONVERT TO DELTA tpcds_sf10000_withdecimal_withdate_withnulls.time_dim;
CONVERT TO DELTA tpcds_sf10000_withdecimal_withdate_withnulls.warehouse;
CONVERT TO DELTA tpcds_sf10000_withdecimal_withdate_withnulls.web_page;
CONVERT TO DELTA tpcds_sf10000_withdecimal_withdate_withnulls.web_returns;
CONVERT TO DELTA tpcds_sf10000_withdecimal_withdate_withnulls.web_sales;
CONVERT TO DELTA tpcds_sf10000_withdecimal_withdate_withnulls.web_site;
```

3. Rerun statistics on the tables:

```
%sql
USE tpcds_sf10000_withdecimal_withdate_withnulls;
analyze table call_center COMPUTE STATISTICS  for all columns;
analyze table catalog_page COMPUTE STATISTICS  for all COLUMNS;
analyze table catalog_returns COMPUTE STATISTICS  for all COLUMNS;
analyze table catalog_sales COMPUTE STATISTICS  for all COLUMNS;
analyze table customer COMPUTE STATISTICS  for all COLUMNS;
analyze table customer_address COMPUTE STATISTICS  for all COLUMNS;
analyze table customer_demographics COMPUTE STATISTICS  for all COLUMNS;
analyze table date_dim COMPUTE STATISTICS  for all COLUMNS ;
analyze table household_demographics COMPUTE STATISTICS  for all COLUMNS;
analyze table income_band COMPUTE STATISTICS  for all COLUMNS;
analyze table inventory COMPUTE STATISTICS  for all COLUMNS;
analyze table item COMPUTE STATISTICS  for all COLUMNS;
analyze table promotion COMPUTE STATISTICS  for all COLUMNS;
analyze table reason COMPUTE STATISTICS  for all COLUMNS;
analyze table ship_mode COMPUTE STATISTICS  for all COLUMNS;
analyze table store COMPUTE STATISTICS  for all COLUMNS;
analyze table store_returns COMPUTE STATISTICS  for all COLUMNS;
analyze table store_sales COMPUTE STATISTICS  for all COLUMNS;
analyze table time_dim COMPUTE STATISTICS  for  all COLUMNS ;
analyze table warehouse COMPUTE STATISTICS  for all COLUMNS;
analyze table web_page COMPUTE STATISTICS  for all COLUMNS;
analyze table web_returns COMPUTE STATISTICS  for all COLUMNS;
analyze table web_sales COMPUTE STATISTICS  for all COLUMNS;
analyze table web_site COMPUTE STATISTICS  for all COLUMNS;
```

Running the power test

1. Power on the cluster, and let it sit idle for 20 minutes.
2. In the Databricks workspace, right-click → Create → Notebook.
3. Type the following into the notebook, and run the cell:

```
// Databricks notebook source
// MAGIC %md
// MAGIC This notebook runs spark-sql-perf TPCDS benchmark on and saves the result.

// COMMAND ----------

// Database to be used:
// TPCDS Scale factor
val scaleFactor = "10000"
// If false, float type will be used instead of decimal.
val useDecimal = true
// If false, string type will be used instead of date.
val useDate = true
// If true, rows with nulls in partition key will be thrown away.
val filterNull = false
// name of database to be used.
val databaseName = s"tpcds_sf${scaleFactor}" +
  s"""_${if (useDecimal) "with" else "no"}decimal""" +
  s"""_${if (useDate) "with" else "no"}date""" +
  s"""_${if (filterNull) "no" else "with"}nulls"""

val iterations = 1 // how many times to run the whole set of queries.

val timeout = 60 // timeout in hours

val query_filter = Seq() // Seq() == all queries
//val query_filter = Seq("q1-v2.4", "q2-v2.4") // run subset of queries
val randomizeQueries = false // run queries in a random order. Recommended for parallel runs.

// detailed results will be written as JSON to this location.
val resultLocation = "/mnt/databricks/results/power/10TB/json"

// COMMAND ----------
val timestamp = System.currentTimeMillis()
val powerResultPath = s"/mnt/databricks/results/power/10TB/csv/timestamp=$timestamp"

// Spark configuration
spark.conf.set("spark.sql.broadcastTimeout", "100000") // good idea for Q14, Q88.

// ... + any other configuration tuning

// COMMAND ----------

sql(s"use $databaseName")

// COMMAND ----------

import com.databricks.spark.sql.perf.tpcds.TPCDS

val tpcds = new TPCDS (sqlContext = sqlContext)
def queries = {
  val filtered_queries = query_filter match {
    case Seq() => tpcds.tpcds2_4Queries
    case _ => tpcds.tpcds2_4Queries.filter(q => query_filter.contains(q.name))
  }
  if (randomizeQueries) scala.util.Random.shuffle(filtered_queries) else filtered_queries
}
val experiment = tpcds.runExperiment(
  queries,
  iterations = iterations,
  resultLocation = resultLocation,
  tags = Map("runtype" -> "benchmark", "database" -> databaseName, "scale_factor" -> scaleFactor),
  forkThread = true)

println(experiment.toString)
```

```
experiment.waitForFinish(timeout*60*60)

// COMMAND ----------

displayHTML(experiment.html)

// COMMAND ----------

import org.apache.spark.sql.functions.{col, lit, substring}
val summary = experiment.getCurrentResults
  .withColumn("Name", substring(col("name"), 2, 100))
  .withColumn("Runtime", (col("parsingTime") + col("analysisTime") + col("optimizationTime") +
col("planningTime") + col("executionTime")) / 1000.0)
  .select('Name, 'Runtime)

summary.coalesce(1).write.format("csv").save(powerResultPath)
display(summary)
```

4. Once the test is complete, record the time to complete all 99 queries.

Setting up the throughput test

1. In the Databricks workspace, right-click → Create → Notebook.
2. Type the following into the notebook:

```
// Databricks notebook source
// MAGIC %md
// MAGIC This notebook runs spark-sql-perf TPCDS benchmark on and saves the result.

// COMMAND ----------

// Database to be used:
// TPCDS Scale factor
val scaleFactor = "10000"
// If false, float type will be used instead of decimal.
val useDecimal = true
// If false, string type will be used instead of date.
val useDate = true
// If true, rows with nulls in partition key will be thrown away.
val filterNull = false
// name of database to be used.
val databaseName = s"tpcds_sf${scaleFactor}" +
  s"""_${if (useDecimal) "with" else "no"}decimal""" +
  s"""_${if (useDate) "with" else "no"}date""" +
  s"""_${if (filterNull) "no" else "with"}nulls"""

//val databaseName = "tpcds_sf10_withdecimal_withdate_withnulls"

val iterations = 1 // how many times to run the whole set of queries.

val timeout = 60 // timeout in hours

val query_filter = Seq(
   "q1-v2.4", "q2-v2.4", "q3-v2.4", "q4-v2.4", "q5-v2.4", "q6-v2.4", "q7-v2.4", "q8-v2.4", "q9-
v2.4", "q10-v2.4",
   "q11-v2.4", "q12-v2.4", "q13-v2.4", "q14a-v2.4", "q14b-v2.4", "q15-v2.4", "q16-v2.4", "q17-v2.4",
"q18-v2.4", "q19-v2.4",
    "q20-v2.4", "q21-v2.4", "q22-v2.4", "q23a-v2.4", "q23b-v2.4", "q24a-v2.4", "q24b-v2.4",
"q25-v2.4", "q26-v2.4", "q27-v2.4",
    "q28-v2.4", "q29-v2.4", "q30-v2.4", "q31-v2.4", "q32-v2.4", "q33-v2.4", "q34-v2.4", "q35-v2.4",
"q36-v2.4", "q37-v2.4",
    "q38-v2.4", "q39a-v2.4", "q39b-v2.4", "q40-v2.4", "q41-v2.4", "q42-v2.4", "q43-v2.4",
"q44-v2.4", "q45-v2.4", "q46-v2.4", "q47-v2.4",
    "q48-v2.4", "q49-v2.4", "q50-v2.4", "q51-v2.4", "q52-v2.4", "q53-v2.4", "q54-v2.4", "q55-v2.4",
"q56-v2.4", "q57-v2.4", "q58-v2.4",
    "q59-v2.4", "q60-v2.4", "q61-v2.4", "q62-v2.4", "q63-v2.4", "q64-v2.4", "q65-v2.4", "q66-v2.4",
"q67-v2.4", "q68-v2.4", "q69-v2.4",
    "q70-v2.4", "q71-v2.4", "q72-v2.4", "q73-v2.4", "q74-v2.4", "q75-v2.4", "q76-v2.4", "q77-v2.4",
```

```
"q78-v2.4", "q79-v2.4",
    "q80-v2.4", "q81-v2.4", "q82-v2.4", "q83-v2.4", "q84-v2.4", "q85-v2.4", "q86-v2.4", "q87-v2.4",
"q88-v2.4", "q89-v2.4",
    "q90-v2.4", "q91-v2.4", "q92-v2.4", "q93-v2.4", "q94-v2.4", "q95-v2.4", "q96-v2.4", "q97-v2.4",
"q98-v2.4", "q99-v2.4") // Seq() == all queries
//val query_filter = Seq("q1-v2.4", "q2-v2.4") // run subset of queries
val randomizeQueries = true // run queries in a random order. Recommended for parallel runs.

// detailed results will be written as JSON to this location.
val resultLocation = "/mnt/databricks/results/throughput/10TB/json/stream_{1..4}"

// COMMAND ----------
val timestamp = System.currentTimeMillis()
val throughputResultPath = s"/mnt/databricks/results/throughput/10TB/csv/stream_{1..4}/
timestamp=$timestamp"

// Spark configuration
spark.conf.set("spark.sql.broadcastTimeout", "100000") // good idea for Q14, Q88.

// ... + any other configuration tuning

// COMMAND ----------

sql(s"use $databaseName")

// COMMAND ----------

import com.databricks.spark.sql.perf.tpcds.TPCDS

val tpcds = new TPCDS (sqlContext = sqlContext)
def queries = {
  val filtered_queries = query_filter match {
    case Seq() => tpcds.tpcds2_4Queries
    case _ => tpcds.tpcds2_4Queries.filter(q => query_filter.contains(q.name))
  }
  if (randomizeQueries) scala.util.Random.shuffle(filtered_queries) else filtered_queries
}
val experiment = tpcds.runExperiment(
  queries,
  iterations = iterations,
  resultLocation = resultLocation,
  tags = Map("runtype" -> "benchmark", "database" -> databaseName, "scale_factor" -> scaleFactor),
  forkThread = true)

println(experiment.toString)
experiment.waitForFinish(timeout*60*60)

// COMMAND ----------

displayHTML(experiment.html)

// COMMAND ----------

import org.apache.spark.sql.functions.{col, lit, substring}
val summary = experiment.getCurrentResults
  .withColumn("Name", substring(col("name"), 2, 100))
  .withColumn("Runtime", (col("parsingTime") + col("analysisTime") + col("optimizationTime") +
col("planningTime") + col("executionTime")) / 1000.0)
  .select('Name, 'Runtime)

summary.coalesce(1).write.format("csv").save(throughputResultPath)
display(summary)
```

3. Complete step 2 three more times for a total of four notebooks, changing the result path to match the stream number for each.
4. In the navigation pane, click Workflows.
5. Click Create → Job.
6. Enter a name for the job and the task the job will be running. We used the name of the stream that would be running the corresponding notebook.
7. Select the path to the first stream notebook you created in step 2.

8. For the compute, select the cluster you created previously.
9. Click Create Task.
10. Complete steps 5 through 9 three more times for a total of four jobs, each tied to a task running a separate notebook you created in steps 2 and 3.

Running the throughput test

1. Power on the cluster, and let it sit idle for 20 minutes.
2. In the Databricks workspace, click Workflows in the navigation pane.
3. Click Jobs & Pipelines.
4. After the idle period, click Start next to each job to kick off the throughput test with four simultaneous streams.

Scripts we used for testing

Variables.tf

```
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Azure Region and RG
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
variable "location" {
  description = "Azure region for deployment"
  type        = string
  default     = "East US"
}

variable "resource_group_name" {
  description = "Resource group name"
  type        = string
  default     = "<Azure resource group name>"
}

# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Databricks items
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
variable "databricks_workspace_name" {
  description = "Databricks workspace name"
  type        = string
  default     = "<Databricks workspace name>"
}

variable "databricks_workspace_sku" {
  description = "SKU for the Databricks workspace (standard or premium)"
  type        = string
  default     = "premium"
}

variable "databricks_cluster_name" {
  description = "Databricks cluster name"
  type        = string
  default     = "<Databricks cluster name>"
}

variable "spark_version" {
  description = "Databricks Spark version"
  type        = string
  default     = "15.4.x-scala2.12"
}

variable "node_type_id" {
  description = "VM instance type for Databricks nodes"
  type        = string
  default     = "Standard_E8ds_v5"
}

variable "autotermination_minutes" {
```

```
  description = "Minutes before auto-termination of the cluster"
  type        = number
  default     = 15
}

variable "num_workers" {
  description = "Number of worker nodes in the Databricks cluster"
  type        = number
  default     = 2
}

variable "min_workers" {
  description = "Minimum number of worker nodes for auto-scaling"
  type        = number
  default     = 2
}

variable "max_workers" {
  description = "Maximum number of worker nodes for auto-scaling"
  type        = number
  default     = 20
}

variable "driver_node_type_id" {
  description = "VM instance type for the Databricks driver node (control node)"
  type        = string
  default     = "Standard_E16ds_v5"  # Choose a different instance type
}

# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Azure Storage items
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
variable "storage_account_name" {
  description = "Azure Data Lake Storage account name"
  type        = string
  default     = "<Azure storage account name>"
}

variable "storage_account_tier" {
  description = "Storage account tier (Standard or Premium)"
  type        = string
  default     = "Premium"
}

variable "container_name" {
  description = "Storage container name"
  type        = string
  default     = "<Azure container name>"
}

# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Providers.tf

```
terraform {
  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "~> 3.0"
    }
    databricks = {
      source  = "databricks/databricks"
      version = "~> 1.10.0"
    }
  }
}

provider "azurerm" {
  features {}
}

provider "databricks" {
  host = azurerm_databricks_workspace.databricks_ws.workspace_url
}
```

Main.tf

```
# Create Resource Group
resource "azurerm_resource_group" "databricks_rg" {
  name     = var.resource_group_name
  location = var.location
}

# Create Azure Databricks Workspace
resource "azurerm_databricks_workspace" "databricks_ws" {
  name                = var.databricks_workspace_name
  resource_group_name = azurerm_resource_group.databricks_rg.name
  location            = azurerm_resource_group.databricks_rg.location
  sku                 = var.databricks_workspace_sku
}

# Create Azure Data Lake Storage (ADLS) Account
resource "azurerm_storage_account" "adls" {
 name                     = var.storage_account_name
 resource_group_name      = azurerm_resource_group.databricks_rg.name
 location                 = azurerm_resource_group.databricks_rg.location
 account_tier             = var.storage_account_tier
 account_replication_type = "LRS"
 is_hns_enabled                = true
 account_kind             = "BlockBlobStorage"
# add is_hns_enabled for ADLS gen2
}

# Create Storage Container in ADLS
resource "azurerm_storage_container" "adls_container" {
  name                  = var.container_name
  storage_account_name  = azurerm_storage_account.adls.name
  container_access_type = "private"
}

# Create Databricks Cluster
resource "databricks_cluster" "tpcds_cluster" {
  cluster_name            = var.databricks_cluster_name
  spark_version           = var.spark_version
  node_type_id            = var.node_type_id
  driver_node_type_id     = var.driver_node_type_id  # Separate control node type
  autotermination_minutes = var.autotermination_minutes
  # num_workers             = var.num_workers

  autoscale {
    min_workers = var.min_workers
    max_workers = var.max_workers
  }
  runtime_engine = "PHOTON"
  depends_on = [azurerm_databricks_workspace.databricks_ws]
}
```

Terraform.tfvars

```
location = "East US"
resource_group_name = "<Resource group>"
databricks_workspace_name = "<Databricks workspace name>"
databricks_workspace_sku = "premium"
storage_account_name = "<Azure storage account>"
storage_account_tier = "Premium"
container_name = "<Container name>"
databricks_cluster_name = "<Databricks cluster name>"
spark_version = "15.4.x-scala2.12"
node_type_id = "Standard_E8ds_v5"
driver_node_type_id = "Standard_E16ds_v5"
autotermination_minutes = 15
num_workers = 2
```

tpcds-install.sh

This script is used for loading the tpc-ds script onto each worker node and the driver node.

```
#!/bin/bash

sudo apt update -y
sudo apt -y install gcc make flex bison byacc git

cd /usr/local/bin
git clone https://github.com/databricks/tpcds-kit.git
cd tpcds-kit/tools
make OS=LINUX
```

Read the report at https://facts.pt/L0CNx3e ▶

This project was commissioned by Microsoft.