Configuring Databricks as MPP and Cache

Revision 20200918

NOTE
This document is confidential and proprietary of **Denodo Technologies**.
No part of this document may be reproduced in any form by any means without prior written authorization of **Denodo Technologies**.

Copyright © 2020
Denodo Technologies Proprietary and Confidential

Denodo North America & APAC: 525 University Avenue, Suite 31, Palo Alto, CA 94301. USA
Denodo Iberia & Latino América: Montalbán 5, 28014 Madrid, Spain
Denodo EMEA: 21st Floor, Portland House, Bressenden Place, London SW1E 5RS. UK
Denodo DACH: Karlstraße 10, 80333 München. Germany
www.denodo.com

1 INTRODUCTION

This document shows how to configure Databricks as MPP and Cache database. Denodo 7.0 Update 20181011 or later is required for this integration.

2 CONFIGURE DATABRICKS

2.1 CREATE A DATABRICKS INSTANCE

- ***In Azure Only:*** Create an Azure Databricks instance using Premium (in other case there will be no JDBC access).
- Create a cluster.
- Create a JDBC notebook to execute queries.

2.2 CONFIGURE YOUR STORAGE IN AZURE/AWS

Since Databricks runs on AWS/Azure, it will use their storage systems. Therefore, instead of HDFS, Databricks File System (DBFS) will use S3 in AWS and Azure Data Lake (ADL) or Azure Blob Storage (WABS) in Azure.

To enable the interaction with those systems, we will need to perform two steps:

- Configure the storage folder.
- Configure authentication, in the case of Azure.

2.2.1 Design a folder for storage and mount it in the DataBricks file system

We need a path that matches both on DBFS and in the direct S3/ADL/WABS access that Denodo will use to upload the content. Since DBFS will only allow to mount folders in the /mnt folder, we need to perform the following steps:

1. In your bucket (S3) or blob container (Azure), create the folder /mnt/denodo_mppcache.
2. In Databricks, mount the folder /mnt/denodo_mppcache from AWS S3 or Azure ALD/WABS into /mnt/denodo_mppcache in DBFS.
    a. For AWS (more details here for S3)
        - create a Scala notebook and mount S3 with

```
val AccessKey = "<aws-access-key>"
// Encode the Secret Key as that can contain "/"
val SecretKey = "<aws-secret-key>"
val EncodedSecretKey = SecretKey.replace("/", "%2F")
val AwsBucketName = "<aws-bucket-name>"
val MountName = "<destination-mount-folder-name-in-dbfs>"

dbutils.fs.mount(s"s3a://$AccessKey:$EncodedSecretKey@$AwsBucketName",
s"/mnt/$MountName")
display(dbutils.fs.ls(s"/mnt/$MountName"))
```

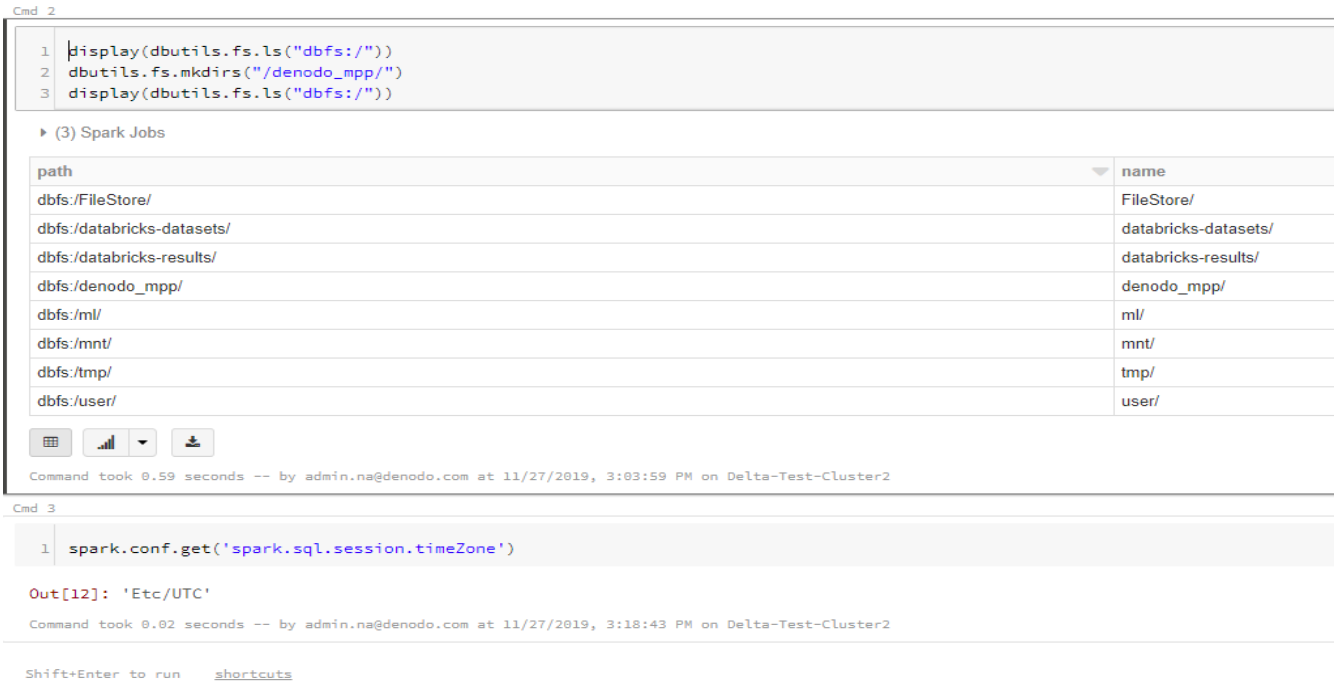
    b. For Azure (more details here for WASB)
        - create a Scala notebook and mount WASB with

```
dbutils.fs.mount(
```

```
source = "wasbs://<blob container>@<storage account>.blob.core.windows.net/mnt/denodo_mppcache",
  mountPoint = "/mnt/denodo_mppcache",
                                    extraConfigs = Map("fs.azure.account.key.denodosecache.blob.core.windows.net" -> "<your key>"))
display(dbutils.fs.ls(s"/mnt/$MountName"))
```

c. For DBFS
   - Create a directory to store the data e.g. denodo_mpp

```
display(dbutils.fs.ls("dbfs:/"))
dbutils.fs.mkdirs("/denodo_mpp/")
display(dbutils.fs.ls("dbfs:/"))
```



Cmd 2

```
1 display(dbutils.fs.ls("dbfs:/"))
2 dbutils.fs.mkdirs("/denodo_mpp/")
3 display(dbutils.fs.ls("dbfs:/"))
```

▸ (3) Spark Jobs

path	name
dbfs:/FileStore/	FileStore/
dbfs:/databricks-datasets/	databricks-datasets/
dbfs:/databricks-results/	databricks-results/
dbfs:/denodo_mpp/	denodo_mpp/
dbfs:/ml/	ml/
dbfs:/mnt/	mnt/
dbfs:/tmp/	tmp/
dbfs:/user/	user/

Command took 0.59 seconds -- by admin.na@denodo.com at 11/27/2019, 3:03:59 PM on Delta-Test-Cluster2

Cmd 3

```
1 spark.conf.get('spark.sql.session.timeZone')
```

Out[12]: 'Etc/UTC'

Command took 0.02 seconds -- by admin.na@denodo.com at 11/27/2019, 3:18:43 PM on Delta-Test-Cluster2

Shift+Enter to run shortcuts

Depending on the installation it may also be needed to add the following configuration property:

```
spark.conf.set("Fs.azure.account.key.denodosecache.blob.core.windows.net", "Your key")
```

2.2.2 Obtain the Authentication Tokens

2.2.2.1 Azure Authentication Tokens

Since Azure uses OAuth 2.0 for authentication, you will need to provide some details in the core-site.xml file of the Hadoop client. Follow the instructions in this link

2.2.2.2 AWS Access Key/Secret Key

In order to get your Access Key ID and Secret Access Key follow next steps:

1. Open the IAM console.
2. From the navigation menu, click Users.
3. Select your IAM user name.
4. Click User Actions, and then click Manage Access Keys.
5. Click Create Access Key.
6. Your keys will look something like this:
    a. Access key ID example: AKIAIOSFODNN7EXAMPLE
    b. Secret access key example: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
7. Click Download Credentials, and store the keys in a secure location.

3 CONFIGURE THE HADOOP CLIENT IN THE DENODO SERVER

IMPORTANT: After update Denodo 20190903 DataBricks client is supported so it simplifies the connectivity. if you are using Denodo update 21090903 or later then skip this section and continue with the next one.

1. Configure your `JAVA_HOME`
    a. IMPORTANT: You must use a path with **no blank spaces**
2. Download Hadoop 3.0.0 and unzip it into a path that will be our `HADOOP_HOME` (for example, `<DENODO_HOME>/lib-external/hadoop-client/`)
    a. IMPORTANT: Use a path with **no blank spaces**
3. If you are in Windows, you will also need the Hadoop Windows binaries. You can download them from here:
    a. Use the "Clone or download" link in GitHub to download the whole project, and then use the content of the folder hadoop-3.0.0/bin to overwrite the bin folder obtained in the step above.
    b. If you are in Linux, ignore this step.
4. Configure the environment variable `<HADOOP_HOME>` using that path
5. Add the following libraries to `<HADOOP_HOME>/share/hadoop/common` depending on your system
    a. For AWS
        i. aws-java-sdk-bundle-1.11.416.jar
        ii. hadoop-aws-3.1.1.jar
        iii. hadoop-common-3.1.1.jar
    b. For Azure
        i. azure-data-lake-store-sdk-2.3.1.jar
        ii. azure-storage-3.1.0.jar
        iii. hadoop-azure-3.1.0.jar
        iv. hadoop-azure-datalake-3.1.0.jar
6. Create a core-site.xml file and put it into `<HADOOP_HOME>/etc/hadoop` with the credentials information
    a. For AWS (other configurations are possible e.g. use IAM roles as described)

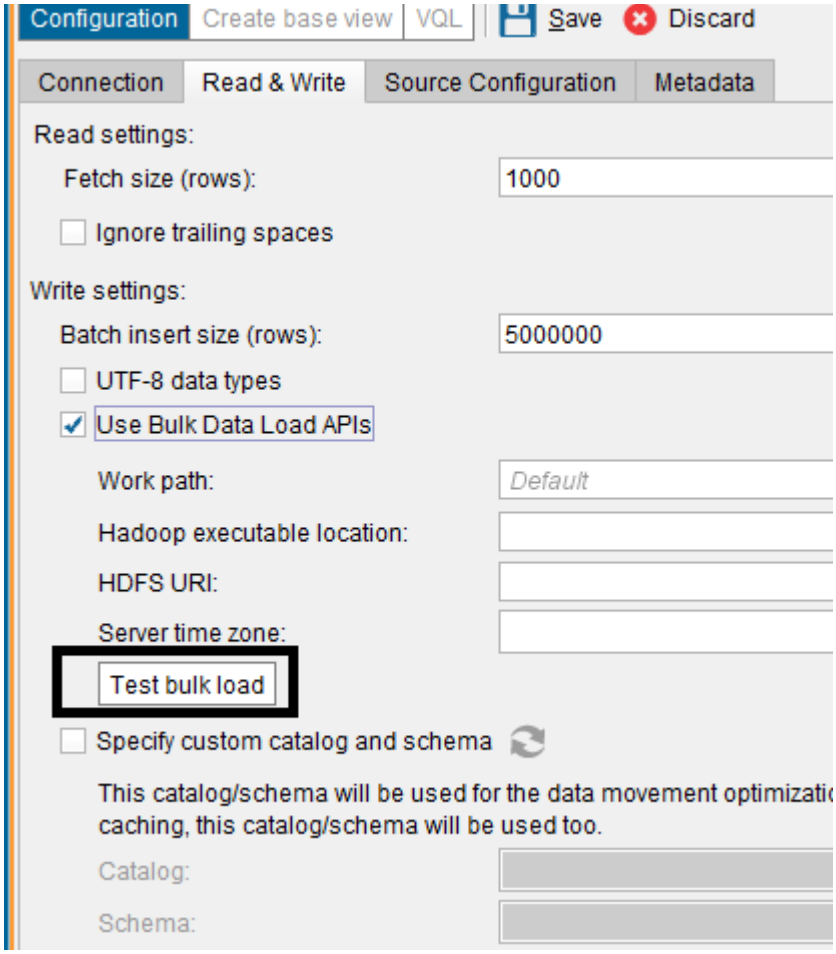
```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
  <property>
    <name>fs.defaultFS</name>
    <value>s3a://YOUR_BUCKET_NAME</value>
  </property>
  <property>
    <name>fs.s3a.access.key</name>
    <value>YOUR_AWS_ACCESS_KEY</value>
  </property>
  <property>
```

```
    <name>fs.s3a.secret.key</name>
    <value>YOUR_AWS_ACCESS_SECRET</value>
  </property>
  <property>
    <name>fs.AbstractFileSystem.s3a.imp</name>
    <value>org.apache.hadoop.fs.s3a.S3A</value>
  </property>
</configuration>
```

b. For Azure (other configurations are possible e.g. use IAM roles as described)

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
  <property>
    <name>fs.defaultFS</name>
<value>wasb://CONTAINER_NAME@STORAGE_ACCOUNT.blob.core.windows.net</value>
  </property>
  <property>
        <name>fs.azure.account.key.STORAGE_ACCOUNT.blob.core.windows.n
et</name>
  <value>KEY==</value>
  </property>
  <property>
    <name>fs.AbstractFileSystem.adl.impl</name>
    <value>org.apache.hadoop.fs.azure.Wasb</value>
   </property>
</configuration>
```

7. Test that it works executing in a cmd, for example
   a. For AWS on Windows
      i. `<HADOOP_HOME>/bin/hadoop fs -put c:/tmp/file.csv s3a://<bucket name>/mnt/denodo_mppcache`
   b. For Azure on Windows
      i. `<HADOOP_HOME>/bin/hadoop fs -put c:/tmp/file.csv wasb://<blob container>@<storage account>.blob.core.windows.net/mnt/denodo_mppcache`
   c. You can also use the Test Bulk Load capability of your MPP data source available in the Denodo Administration Tool


Configuration
Create base view
VQL
Save
Discard
Connection
Read & Write
Source Configuration
Metadata
Read settings:
Fetch size (rows):
1000
Ignore trailing spaces
Write settings:
Batch insert size (rows):
5000000
UTF-8 data types
Use Bulk Data Load APIs
Work path:
Default
Hadoop executable location:
HDFS URI:
Server time zone:
Test bulk load
Specify custom catalog and schema
This catalog/schema will be used for the data movement optimizati
caching, this catalog/schema will be used too.
Catalog:
Schema:

4 CONFIGURE THE DATABRICKS CLIENT IN THE DENODO SERVER (DENODO UPDATE 20190903 OR LATER)

If you are using Denodo update 20190903 or later it is possible to use Databricks client instead of Hadoop's.

4.1 INSTALL

```
$ pip install databricks-cli
```

4.2 CONFIGURE

```
$ databricks configure --token
Databricks Host (should begin with https://): https://dbc-b1b8888e-
0fc5.cloud.databricks.com
Token: dapie5709eb881cb03e84d14d66cebc78888
```

5 CONFIGURE THE DENODO SERVER

1. Download the Databricks JDBC driver linked in this page. Copy the driver to the `<DENODO_HOME>/lib-external/jdbc/databricks` folder.
2. In Denodo, create a new JDBC connection and select your Spark version as your adapter
    a. In the classpath, select the driver downloaded.
    b. In the URI, use something like `jdbc:hive2://<region>.azuredatabricks.net:443/default;transportMode=http;ssl=true;httpPath=<your http path>`
        i. In the link above you can find more details on how to find your HTTP Path
    c. In the user, use the word “token”.
    d. For the password, use your actual token. You can find more details in the link above
3. Test that the connection is working.
4. Go to the `Read & Write` tab to configure the bulk data loading options.
5. In the Hadoop executable location, use the path of your Hadoop 3.x libraries (remember that you need the core-site.xml with your auth), for example
    a. `<DENODO_HOME>/lib-external/HadoopClient/hadoop-3.0.0/bin/hadoop.cmd`
    b. When using Databricks-cli if it is in the path you only need to use:
        i. dbfs
6. In the HDFS uri, use your S3 or WABS route, with the folder previously created at the end, for example:
    a. `wasb://<blob container>@<storage account>.blob.core.windows.net/mnt/denodo_mppcache`
    b. `dbfs://denodo_mpp`
7. In the server time zone, select the timezone of the server (Etc/GMT+8 for example)
    a. To obtain this value you can execute in Databricks
        i. `spark.conf.get('spark.sql.session.timeZone')`

Now you are ready to use this data source as your cache. To enable the use of Spark as MPP accelerator, enable the corresponding options in the data source and in the Optimizer settings.

6 PERFORMANCE CONSIDERATIONS USING DATABRICKS AS CACHE

- Use Databricks Simba JDBC drivers.
- Use the Hadoop client to load the data to S3.
- Databricks and the S3 bucket should be in the same availability zone.
- Enable the compression of Parquet files: SET `'com.denodo.vdb.util.tablemanagement.sql.insertion.HdfsInsertWorker.parquet.compression'='snappy'`
- Change the batch insert size to take advantage of the parallelism generating and uploading 10-11 files.

7 KNOWN LIMITATIONS

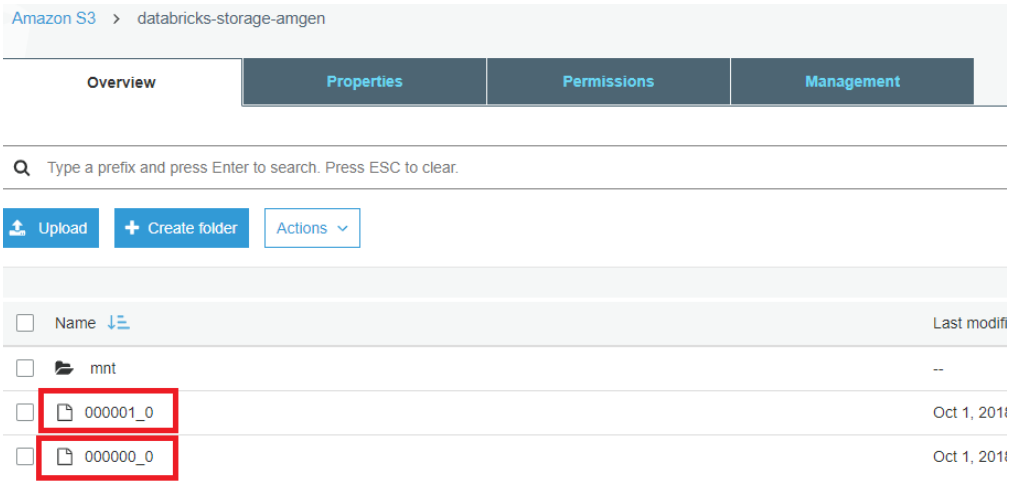
Due to the limitation of Spark external tables, the processes that invalidate the cache are not transactional. Therefore, the following limitations apply:

- “matching_rows” invalidation does not work. Only adding increments or full refresh (“all_rows”).
- Do not query the tables during the invalidation process, as they may through an error.
- Do not attempt concurrent executions of queries that load and invalidate the cache in the same command.

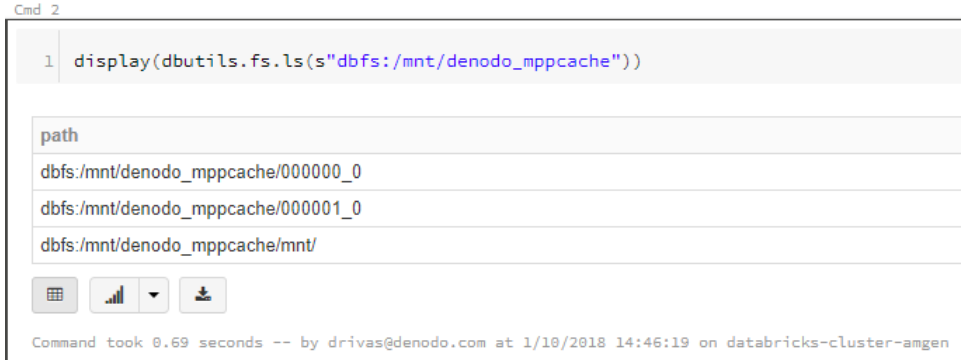
8 HOW TO READ PARQUET DATA

8.1 FOR AWS

1. Upload a parquet file in your S3 bucket



2. On a Scala Notebook, mount the bucket following the previous section **Design a folder for storage and mount it in the DataBricks file system**.
3. Search for the Files in dbfs with the display command



4. Read and show the parquet files with the command:

```
val data =
sqlContext.read.parquet(s"dbfs:/mnt/denodo_mppcache/mnt/denodo_mppcache")
display(data)
```

```
3 val data = sqlContext.read.parquet(s"dbfs:/mnt/denodo_mppcache/mnt/denodo_mppcache")
4 display(data)
```

▸ (2) Spark Jobs

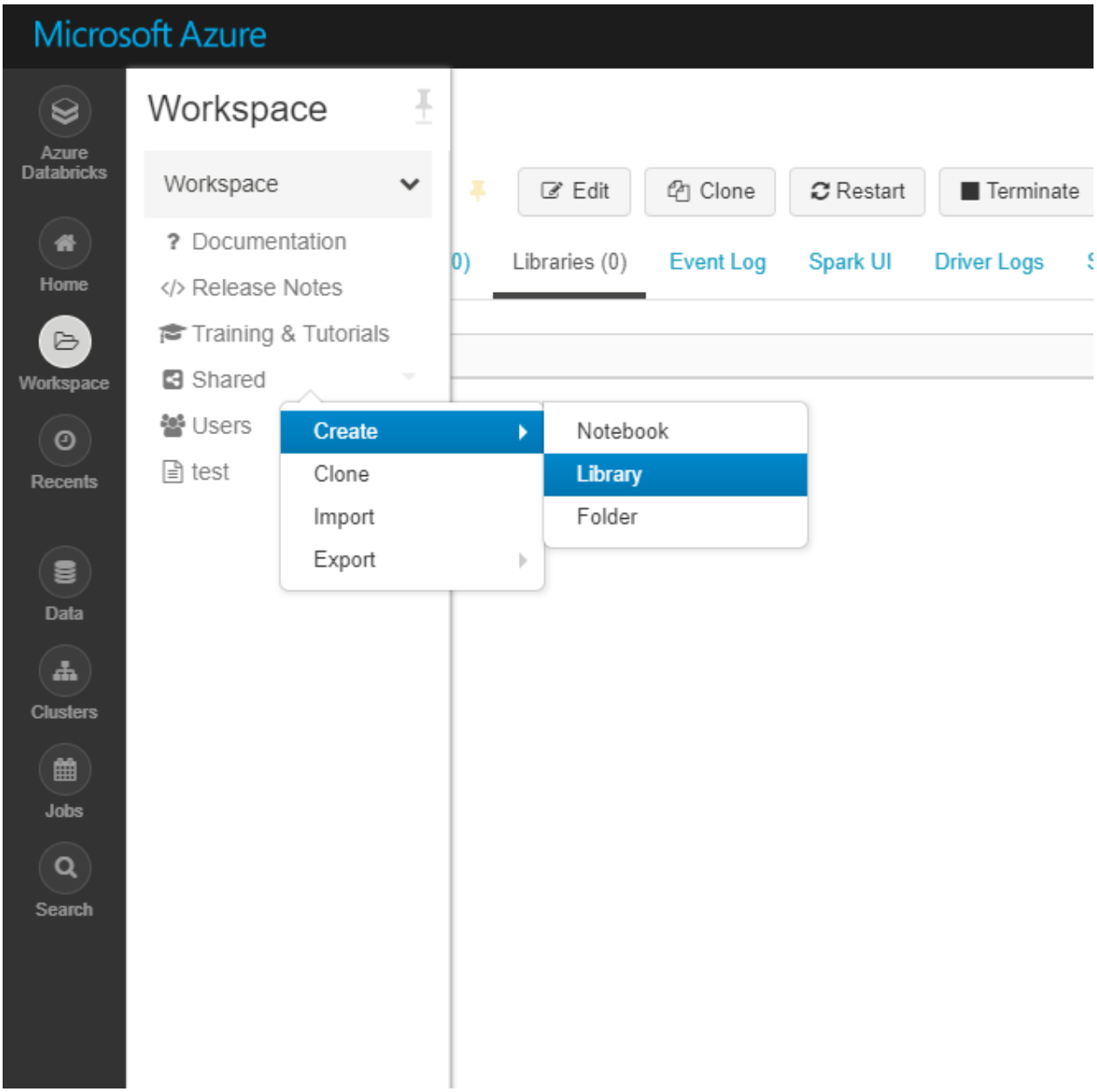
▸ data: org.apache.spark.sql.DataFrame = [c_customer_sk: integer, c_customer_id: string ... 16 more fields]

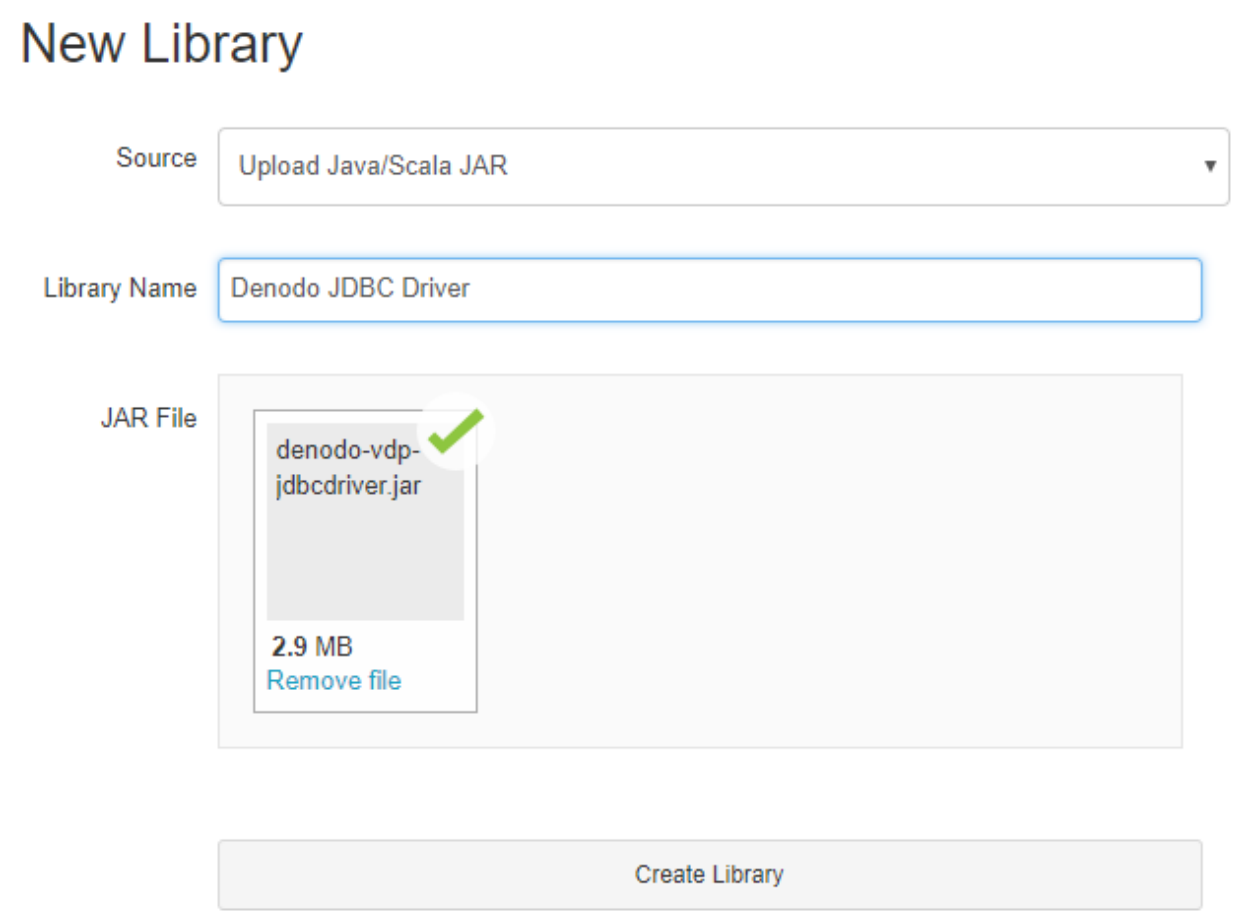
c_customer_sk	c_customer_id	c_current_cdemo_sk	c_current_hdemo_sk	c_current_addr_sk
978420	AAAAAAAAEPNOOAAA	1708329	6447	48565
978421	AAAAAAAAFPNOOAAA	1864494	5373	867052
978422	AAAAAAAAGPNOOAAA	1418997	464	273931
978423	AAAAAAAAHPNOOAAA	1359861	248	307835
978424	AAAAAAAAIPNOOAAA	1612802	5303	465804
978425	AAAAAAAAJPNOOAAA	1734013	1751	181842
978426	AAAAAAAAKPNOOAAA	1638319	4427	204582
978427	AAAAAAAALPNOOAAA	196154	4647	761940

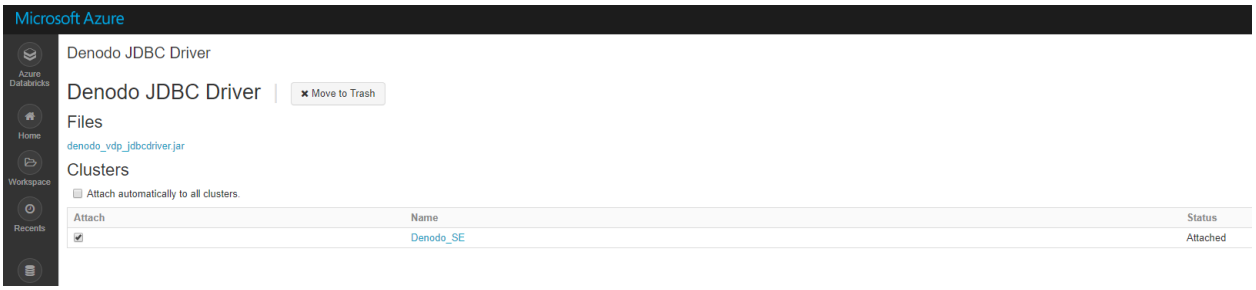
Showing the first 1000 rows.

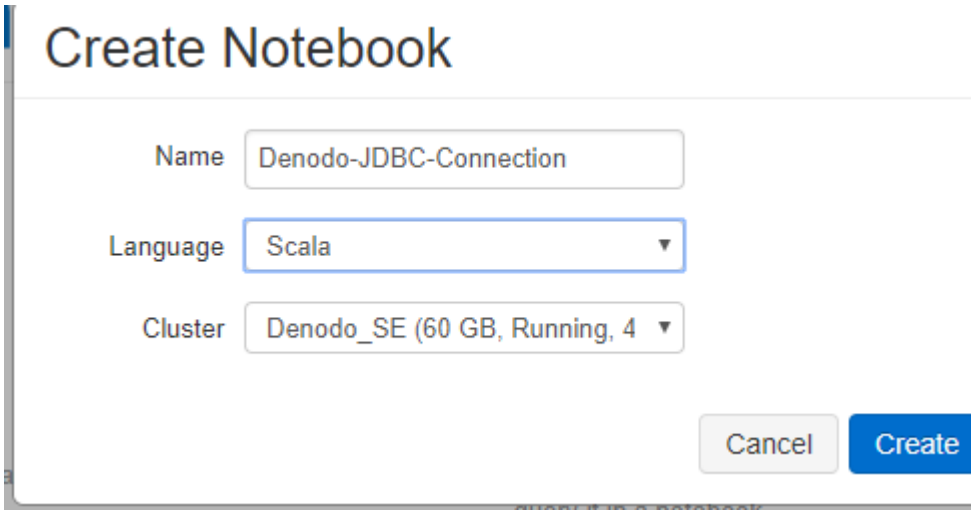
9 DENODO AS A JDBC SOURCE IN DATABRICKS

1. Load the Denodo JDBC driver from <DENODO_HOME>/tools/client-drivers/jdbc/ following Create a Workspace library.




New Library
Source
Upload Java/Scala JAR
Library Name
Denodo JDBC Driver
JAR File
denodo-vdp-jdbcdriver.jar
2.9 MB
Remove file
Create Library


Microsoft Azure
Azure Databricks
Home
Workspace
Recents
Denodo JDBC Driver
Denodo JDBC Driver
Move to Trash
Files
denodo_vdp_jdbcdriver.jar
Clusters
Attach automatically to all clusters.
Attach
Name
Status
Denodo_SE
Attached



2. Create a connection and execute a query in Notebook following these steps (more info can be found at SQL Databases using JDBC),

```
Class.forName("com.denodo.vdp.jdbc.Driver")

val jdbcHostname = "myhost"
val jdbcPort = 9999
val jdbcDatabase = "big_data"

// Create the JDBC URL without passing in the user and password parameters.
val jdbcUrl = s"jdbc:vdb://${jdbcHostname}:${jdbcPort}/${jdbcDatabase}"

// Create a Properties() object to hold the parameters.
import java.util.Properties
val connectionProperties = new Properties()

connectionProperties.put("user", "admin")
connectionProperties.put("password", "mypass")

val pushdown_query = "(select * from oracle_customers"
val     df     =     spark.read.jdbc(url=jdbcUrl,     table=pushdown_query,
properties=connectionProperties)
display(df)
```

Depending on the Databricks version when following these steps you might get a "`No suitable driver`" error. In such scenarios, the postgresql driver can be used as a workaround, this driver is loaded by default.

```
Class.forName("org.postgresql.Driver")
```

```
val jdbcHostname = "myhost"
val jdbcPort = 9996
val jdbcDatabase = "big_data"

// Create the JDBC URL without passing in the user and password parameters.
val jdbcUrl = s"jdbc:postgresql://${jdbcHostname}:${jdbcPort}/$
{jdbcDatabase}"

// Create a Properties() object to hold the parameters.
import java.util.Properties
val connectionProperties = new Properties()

connectionProperties.put("user", "admin")
connectionProperties.put("password", "mypass")

val pushdown_query = "(select * from oracle_customer) oracle_customer"
val df = spark.read.jdbc(url=jdbcUrl, table=pushdown_query,
properties=connectionProperties)
Console.println(pushdown_query)
display(df)
```