“Azure - Databricks - Cheat Sheet”

12 3 2022

Introduction

Apache Spark is a unified analytics engine for large-scale data processing and machine learning.

The Three V's of Big Data: Volume, Velocity, and Variety.

Understanding the architecture of spark job

- Spark is distributed computing environment.
- The unit of distribution is a Spark Cluster.
- Every Cluster has a Driver and one or more executors.
- Work submitted to the Cluster is split into as many independent Jobs as needed - this is how work is distributed across the Cluster's nodes.
- Jobs are further subdivided into tasks.
- The input to a job is partitioned into one or more partitions.

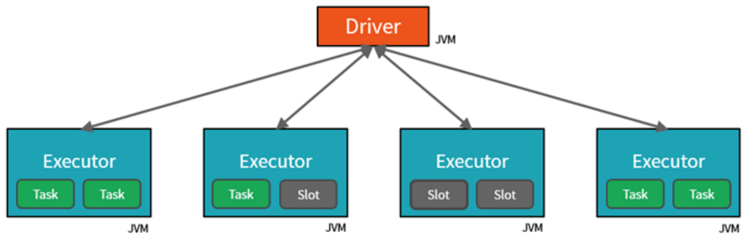

Figure 1: Spark - Cluster - Tasks

Jobs & stages

- Each parallelized action is referred to as a **Job**.
- The results of each Job (parallelized/distributed action) is returned to the Driver.
- Depending on the work required, multiple Jobs will be required.
- Each Job is broken down into **Stages**.

# Reading Data

```
fileName = "dbfs:/mnt/training/wikipedia/clickstream/2015_02_clickstream.tsv"
csvSchema = StructType([
  StructField("prev_id", IntegerType(), False),
  StructField("curr_id", IntegerType(), False),
  StructField("n", IntegerType(), False),
  StructField("prev_title", StringType(), False),
  StructField("curr_title", StringType(), False),
  StructField("type", StringType(), False)
])

testDF = (spark.read          #The DataFrameReader
  .option('header', 'true')   #Ignore line #1 - it's a header
  .option('sep', "\t")        #Use tab delimiter (default is comma-separator)
  .schema(csvSchema)          #Use the specified schema
  .csv(fileName)              #Creates a DataFrame from CSV after reading in the file
)

#Display data
testDF.printSchema()
```

DataFrames vs SQL & Temporary Views

```
dataDF.createOrReplaceTempView("name_of_db")

#Use simple sql

%sql
SELECT * FROM pagecounts
```

Convert from SQL back to a `DataFrame`

```
tableDF = spark.sql("SELECT DISTINCT project FROM pagecounts ORDER BY project")
display(tableDF)
```

Exercise

```
(source, sasEntity, sasToken) = getAzureDataSource()
spark.conf.set(sasEntity, sasToken)

path = source + "/wikipedia/pagecounts/staging_parquet_en_only_clean/"

# 1. Define data frame
#
df = (spark                     # Our SparkSession & Entry Point
  .read                         # Our DataFrameReader
  .parquet(path)                # Read in the parquet files
  .select("article")            # Reduce the columns to just the one
  .distinct()                   # Produce a unique set of values
)
totalArticles = df.count() # Identify the total number of records remaining.

print("Distinct Articles: {0:,}".format(totalArticles))
```

Describe the difference between eager and lazy execution

Fundamental to Apache Spark are the notions that

- Transformations are LAZY - like creating data - They eventually return another `DataFrame`.
- Actions are EAGER - display data (*touch data*)

Transformations applied to `DataFrames` are lazy, meaning they will not trigger any jobs. If you pass the `DataFrame` to a display function, a job will be triggered because display is an action.

Types of Transformations

- A transformation may be wide or narrow.
- A wide transformation requires sharing data across workers.
- A narrow transformation can be applied per partition/worker with no need to share or shuffle data to other workers.

Narrow Transformations

The data required to compute the records in a single partition reside in at most one partition of the parent Dataframe.

```
from pyspark.sql.functions import col
display(countsDF.filter(col("NAME").like("%TX%")))
```

Wide Transformations

The data required to compute the records in a single partition may reside in many partitions of the parent Dataframe. These operations require that data is shuffled between executors. Wide transformation shares data across workers by shuffling data between executors.

```
from pyspark.sql.functions import col
display(countsDF.groupBy("UNIT").sum("counts"))
```

Catalyst Optimizer

Among the most powerful components of Spark are Spark SQL. At its core lies the Catalyst optimizer. This extensible query optimizer supports both rule-based and cost-based optimization. Spark SQL uses Catalyst's general tree transformation framework in four phases - Analysis, Logical Optimization, Physical Planning, and Code Generation. Our code is evaluated and optimized by the Catalyst Optimizer.

UnsafeRow (also known as Tungsten Binary Format)

- Sharing data from one worker to another can be a costly operation.
- Spark has optimized this operation by using a format called Tungsten.
- Tungsten prevents the need for expensive serialization and de-serialization of objects in order to get data from one JVM to another.

DataFrame Column Expressions

filter(..) & where(..) w/Column

```
filteredDF = (sortedDescDF
  .filter( col("project") == "en")
)
filteredDF.show(10, False)
```

1.DateTime-Manipulation

Data conversion:

https://docs.oracle.com/javase/tutorial/i18n/format/simpleDateFormat.html

`string to timestamp`

```
unix_timestamp(..)
```

```
tempC = (initialDF
  .withColumnRenamed("timestamp", "capturedAt")
  .select( col("*"), unix_timestamp( col("capturedAt"), "yyyy-MM-dd'T'HH:mm:ss").cast("timestamp").alias
)
tempC.printSchema()


pageviewsDF = (initialDF
  .withColumnRenamed("timestamp", "capturedAt")
  .withColumn("capturedAt", unix_timestamp( col("capturedAt"), "yyyy-MM-dd'T'HH:mm:ss").cast("timestamp
)
```

year(..), month(..), dayofyear(..)

```
display(
  pageviewsDF
    .select( month( col("capturedAt")) ) # Every record converted to a single column - the month captur
    .distinct()                          # Reduce all months to the list of distinct months
)

(pageviewsDF
  .select( month(col("capturedAt")).alias("month"), year(col("capturedAt")).alias("year"))
  .distinct()
  .show()
)
```

`groupBy()`

Doen't return dataframe. Its $\frac{1}{2}$ of transformation.

Common agregatons:

- `avg()`
- `count()`

- `sum()`
- `min()`
- `max()`
- `mean()`
- `agg()`
- `pivot()`: Pivots a column of the current `DataFrame` and perform the specified aggregation.

```
# rename and aggregate

display(
  pageviewsDF
    .groupBy( col("site") )
    .sum("requests")
    .withColumnRenamed("sum(requests)", "totalRequests")
)
```

Summary statistics

```
(pageviewsDF
  .filter("site = 'mobile'")
  .select( sum( col("requests")), count(col("requests")), avg(col("requests")), min(col("requests")), ma
  .show()
)

(pageviewsDF
  .filter("site = 'desktop'")
  .select( sum( col("requests")), count(col("requests")), avg(col("requests")), min(col("requests")), ma
  .show()
)


(pageviewsDF
  .filter("site = 'mobile'")
  .select(
    format_number(sum(col("requests")), 0).alias("sum"),
    format_number(count(col("requests")), 0).alias("count"),
    format_number(avg(col("requests")), 2).alias("avg"),
    format_number(min(col("requests")), 0).alias("min"),
    format_number(max(col("requests")), 0).alias("max")
  )
  .show()
)
```

Exercie

```
# TODO
from pyspark.sql.functions import upper, col
from pyspark.sql.functions import *

(source, sasEntity, sasToken) = getAzureDataSource()
spark.conf.set(sasEntity, sasToken)

sourceFile = source + "/dataframes/people-with-dups.txt"
destFile = userhome + "/people.parquet"
```

```
# In case it already exists
#dbutils.fs.rm(destFile, True)

partitions = 8
df = spark.read.csv(sourceFile, sep=':', inferSchema=True, header=True)

dedupedDF = (df
  .select(col("*"),
      lower(col("firstName")).alias("lcFirstName"),
      lower(col("lastName")).alias("lcLastName"),
      lower(col("middleName")).alias("lcMiddleName"),
      translate(col("ssn"), "-", "").alias("ssnNums")
      # regexp_replace(col("ssn"), "-", "").alias("ssnNums")
      # regexp_replace(col("ssn"), """^(\d{3})(\d{2})(\d{4})$""", "$1-$2-$3").alias("ssnNums")
   )
  .dropDuplicates(["lcFirstName", "lcMiddleName", "lcLastName", "ssnNums", "gender", "birthDate", "sala
  .drop("lcFirstName", "lcMiddleName", "lcLastName", "ssnNums")
)
display(dedupedDF)

# ANSWER

# Now we can save the results. We'll also re-read them and count them, just as a final check.
# Just for fun, we'll use the Snappy compression codec. It's not as compact as Gzip, but it's much faste
(dedupedDF.write
   .mode("overwrite")
   .option("compression", "snappy")
   .parquet(destFile)
)
dedupedDF = spark.read.parquet(destFile)
print("Total Records: {0:,}".format( dedupedDF.count() ))
```

Useful functions

- `initialDF.printSchema()` - show structure
- `df.select( col("a").alias("b")` - rename
- `df.withColumnRenamed("name", "newName")` - rename
- `df.toDF("col1", "col1", "col3")` - rename all following columns.

Describe platform architecture, security, and data protection in Azure Databricks

Access control - Folders

- no permissions
- read
- run
- edit
- manage

Access control - Notebooks

- no permissions
- read
- run
- edit
- manage

Access control - Clusters

- can attach to
- can restart
- can manage

Access control - Jobs

- no permissions
- can view
- can manage tun
- is owner
- can manage (admin)

Access control - Tables

- by default all users have access to all data stored in cluster's managed tables, unless administrator enables table access control from that cluster.

View-based access control model defines following privileges:

- select, create, modify, read_metadata, create_named_function, all_privilages.

The privileges can apply to the following classes of objects:

- catalog, database, table, view, function, anonymous_function, any file.

Secrets

Azure Databricks has two types of secret scopes: Key Vault-backed and Databricks-backed.

As a best practice, instead of directly entering your credentials into a notebook, use Azure Databricks secrets to store your credentials and reference them in notebooks and jobs.

Azure Blob

- add files to blob,
- add access SAS token for container
- Your SAS Token in a production environment should be stored in Secrets/KeyVault to prevent it from being displayed in plain text inside a notebook.

Azure Key Vault is used to Securely store and tightly control access to tokens, passwords, certificates, API keys, and other secrets.

Azure Key Vault

Provides us with a number of options for storing and sharing secrets and keys between Azure applications, and has direct integration with Azure Databricks.

Access Azure Databricks Secrets UI

- get your databricks url, and add `#secrets/createScope` it will look like this:

`https://adb-2191932909012589.9.azuredatabricks.net/?o=2191932909012589#secrets/createScope`

Link Azure Databricks to Key Vault

In the Azure Portal on your Key Vault tab: 1. Go to properties 2. Copy and paste the DNS Name 3. Copy and paste the Resource ID 4. Paste to secret scope.

List Secret Scopes

To list the existing secret scopes the `dbutils.secrets` utility can be used.

You can list all scopes currently available in your workspace with:

`dbutils.secrets.listScopes()`

- Secrets are not displayed in clear text
- Notice that the value when printed out is [REDACTED]. This is to prevent your secrets from being exposed.

Facts check

- **The Data Plane** is hosted within the client subscription and is where all data is processed and stored. All data is processed by clusters hosted within the client Azure subscription and data is stored within Azure Blob storage and any connected Azure services within this portion of the platform architecture.
- Data stored in Azure Storage is encrypted using server-side encryption that is seamlessly accessed by Azure Databricks. All data transmitted between the Data Plane and the Control Plane is always encrypted in-flight via TLS.**At-rest and in-transit**
- Commands running on a configured cluster can read and write data in ADLS without configuring service principal credentials. In addition, authentication to ADLS from Azure Databricks clusters is automatic, using the same Azure AD identity one uses to log into Azure Databricks. In addition, authentication to ADLS from Azure Databricks clusters is automatic, using the same Azure AD identity one uses to log into Azure Databricks.
- **Azure Key Vault-backed secret scope**: A secret scope is provided by Azure Databricks and can be backed by either Databricks or Azure Key Vault.

Data Lakes and Delta Lakes

A data lake is a storage repository that inexpensively stores a vast amount of raw data, both current and historical, in native formats such as XML, JSON, CSV, and Parquet

Delta Lake is a file format that can help you build a data lake comprised of one or many tables in Delta Lake format.

Delta Lake makes data ready for analytics.

ACID Transactions:

Key Concepts: Delta Lake Architecture

We'll touch on this further in future notebooks.

Throughout our Delta Lake discussions, we'll often refer to the concept of Bronze/Silver/Gold tables. These levels refer to the state of data refinement as data flows through a processing pipeline.

These levels are conceptual guidelines, and implemented architectures may have any number of layers with various levels of enrichment. Below are some general ideas about the state of data in each level.

- **Bronze tables**
  - Raw data (or very little processing)
  - Data will be stored in the Delta format (can encode raw bytes as a column)
- **Silver tables**
  - Data that is directly queryable and ready for insights
  - Bad records have been handled, types have been enforced
- **Gold tables**
  - Highly refined views of the data
  - Aggregate tables for BI
  - Feature tables for data scientists

Delta Lake Batch Operations

- read csv -> change to delta

```
df
  .write
  .format("delta")
  .save("/data"`)
```

CREATE A Table Using Delta Lake

```
spark.sql("""
```

Metadata

```
DESCRIBE DETAIL customer_data_delta
```

Delta Lake Batch Operations - Append

```
(newDataDF
  .write
  .format("delta")
  .partitionBy("Country")
  .mode("append")
  .save(DataPath)
)
```

Delta Lake Batch Operations - Upsert: Insert and Update

```
MERGE INTO customer_data_delta
USING upsert_data
ON customer_data_delta.InvoiceNo = upsert_data.InvoiceNo
  AND customer_data_delta.StockCode = upsert_data.StockCode
WHEN MATCHED THEN
  UPDATE SET *
WHEN NOT MATCHED
  THEN INSERT *
```

Exercise: Because we stored our data in Delta, our schema and partions are preserved. All we'll need to do is specify the format and the path.

```
deltaDF = (spark.read
           .format("delta")
           .load(DataPath)
           )
```

Register this Delta table as a Spark SQL table.

```
spark.sql("""
```

READ updated CSV data

VACUUM, optimization zeroorder.

```
customer_data_delta RETAIN 0 HOURS;
```

Knowledge check

1. What is the Databricks Delta command to display metadata? `DESCRIBE DETAIL tablename`

2. How do you perform UPSERT in a Delta dataset? Use `MERGE INTO my-table USING data-to-upsert`

3. What optimization does the following command perform: OPTIMIZE Students ZORDER BY Grade?

> Ensures that all data backing, for example, Grade=8 is colocated, then rewrites the sorted data into new Parquet files

4. What size does OPTIMIZE compact small files to?

Around 1GB