Data 603 – Big Data Platforms

Lecture 6
Structured APIs (Part 2)

Spark SQL Intro

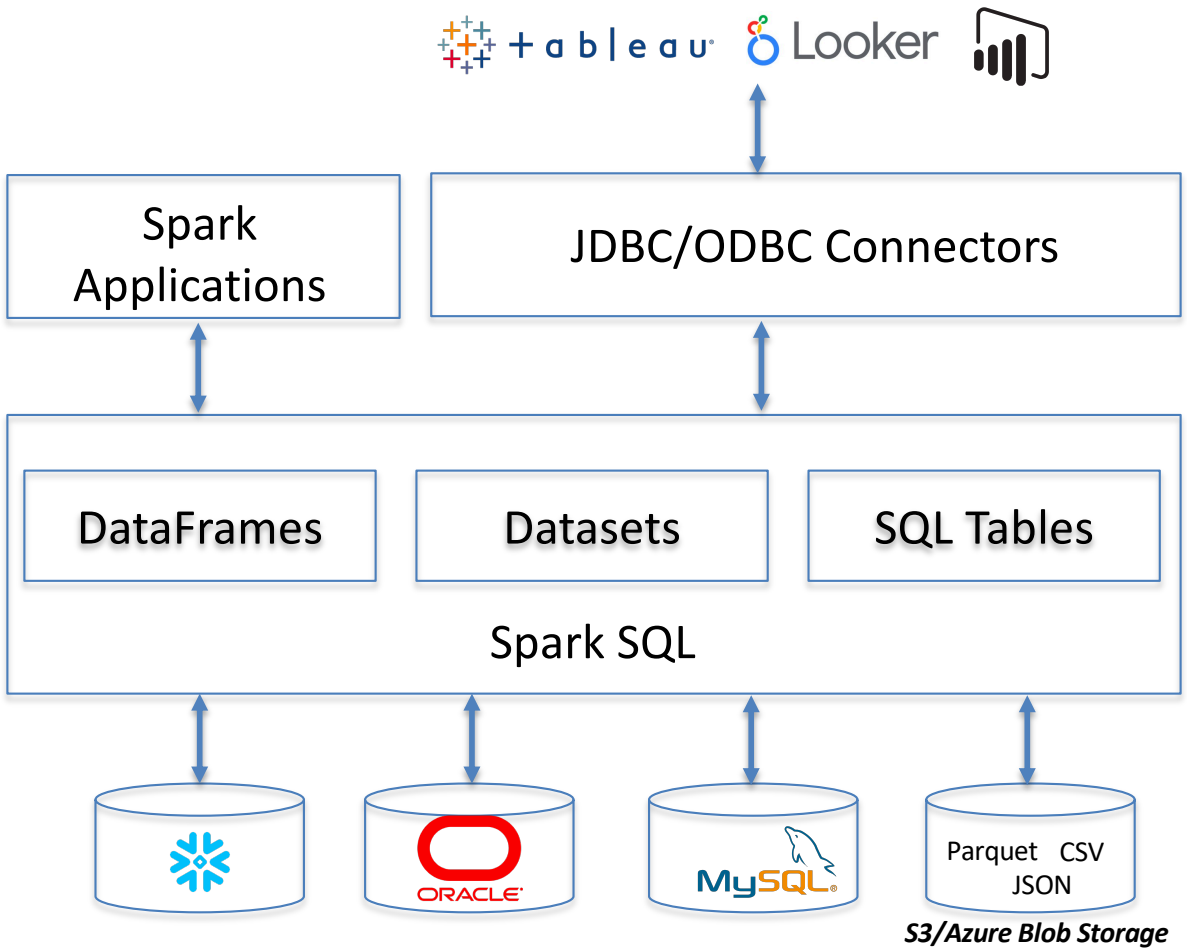
Spark SQL

- Allows developers to work with standard SQL (ANSI SQL:2003) to make queries on structured data with a schema
  - Introduced in Spark 1.3
  - Provides high-level structured functionalities for DataFrame and Dataset.
- Unifies Spark components and simplifies working with structured data sets by providing abstractions for DataFrames/Datasets.
- Connects to Apache Hive metastore and tables

Spark SQL

- Reads and writes structured data with a specific schema from structured file formats (JSON, CSV, Text, Avro, Parquet, ORC, etc.) and converts data into temporary tables.
- Connecting with external tools, including BI tools (Power BI, Tableau, SAS), via JDBC and ODBC
- Optimization via Catalyst Optimizer and Project Tungsten.
  - These support high-level DataFrame, Dataset APIs and SQL queries.

Spark SQL


+ a b l e a u
Looker
Spark Applications
JDBC/ODBC Connectors
DataFrames
Datasets
SQL Tables
Spark SQL
ORACLE
MySQL
Parquet CSV JSON
S3/Azure Blob Storage

Catalyst Optimizer

- Takes a computation query and converts it into an execution plan.
- Four Transformational phases:
  - **Phase 1: Analysis**
    - Abstract Syntax Tree (AST) is generated for the SQL/DataFrame query
    - Any columns or table names will be resolved by consulting an internal Catalog which holds a list of names of columns, data types, functions, tables and databases
  - **Phase 2: Logical Optimization**
    - Construct a set of multiple plans and then, using its cost-based optimizer (CBO), assign costs to each plan which are laid out as operator trees
    - Logical plan becomes the input for the physical plan
  - **Phase 3: Physical Planning**
    - Selection of the optimal physical plan based on the logical plan
  - **Phase 4: Code Generation**
    - Project Tungsten facilitates whole-stage code generation to generate efficient Java bytecode.

Catalyst Optimizer

```
# In Python

count_mnm_df = (mnm_df
  .select("State", "Color", "Count")
  .groupBy("State", "Color")
  .agg(sum("Count")
  .alias("Total"))
  .orderBy("Total", ascending=False))
```

```
-- In SQL

SELECT State, Color, sum(Count) AS Total
FROM MNM_TABLE_NAME
GROUP BY State, Color
ORDER BY Total DESC
```

The resulting byte code to compute above codes are likely the same.

Catalyst Optimizer

- In Python, use *explain(True)* method on DataFrames to see the different stages the code goes through.
- In Scala, call *df.queryExecution.logical*, or *df.queryExecution.optimizedPlan*

Tungsten Whole-stage Code Generation

- Introduced in Spark 2.0
- Physical query optimization phase
- Collapses the whole query into a single function
- Gets rid of virtual function calls and leverages CPU registers for intermediate data
- Generates compact RDD code for final execution
- Significantly improves CPU efficiency and performance

Spark SQL and DataFrames

- *SparkSesssion* – unified entry point for programming Spark with the Structured APIs.
  - Provides *sql()* method for SQL queries
    - All SQL queries return a DataFrame
    - ANSI:2003-compliance SQL
- Any DataFrame can be registered as a table or view (a temporary table) and query it using SQL.
  - No performance difference between using SQL or DataFrame API
  - Both compile to the same underlying plan specified in DataFrame code.

Spark Metastore

- Stores metadata of tables in Spark
  - Information about the table and its data
  - The schema, description, table name, database name, column names, partitions, physical location where the actual data resides
- Spark by default uses Apache Hive metastore
  - By default Located at /usr/hive/warehouse
  - Persist all the metadata about the tables
  - The default location can be changed using config variable *spark.sql.warehouse.dir*

SparkSession.catalog

- Metadata is captured in the Catalog
  - A high-level abstraction in Spark SQL for storing metadata
  - Functionality was expanded in Spark 2.x with new public methods providing ability to examine the metadata associated with the databases, tables and views.
  - Spark 3.0 extends it to use external catalogs.
  - *spark.catalog.listDatabases()*
  - *spark.catalog.listTables()*
  - *spark.catalog.listColumns("us_delay_flights_tbl")*
- For the list of available methods, refer here

Managed vs. Unmanaged Tables

Two types of tables can be created:

- Managed
  - Spark manages both the metadata and the data in the file store (local filesystem, HDFS, an object store such as AWS S3, or Azure Blob Storage)
  - DROP TABLE command deletes both the metadata and the data
  - *CREATE TABLE <example-table>(id STRING, value STRING)*
- Unmanaged
  - Spark only manages the metadata
  - External data is managed outside of Spark.
  - DROP TABLE command only deletes the metadata.
  - *CREATE TABLE <example-table>(id STRING, value STRING) USING org.apache.spark.sql.parquet OPTIONS (PATH "<your-storage-path>")*

Creating SQL Databases and Tables

```
# In Scala/Python
spark.sql("CREATE DATABASE database_name")
spark.sql("USE database_name")

# Creating managed table
spark.sql("CREATE TABLE managed_us_delay_flights_tbl (date STRING, delay INT,
distance INT, origin STRING, destination STRING)")

# Or using the DataFrame API
csv_file = "/databricks-datasets/learning-spark-v2/flights/departuredelays.csv"
schema="date STRING, delay INT, distance INT, origin STRING, destination STRING"
flights_df = spark.read.csv(csv_file, schema=schema)
flights_df.write.saveAsTable("managed_us_delay_flights_tbl")
```

- Tables reside within a database
- By default Spark creates tables under the default database.

Creating SQL Databases and Tables

```
# Creating an unmanaged table

spark.sql("""CREATE TABLE us_delay_flights_tbl(date STRING, delay INT,
     distance INT, origin STRING, destination STRING)
     USING csv OPTIONS (PATH
     '/databricks-datasets/learning-spark-v2/flights/departuredelays.csv')""")
```

- Unmanaged tables are created from existing data sources

Views

- Session Scoped (Temporary View)
  - *df.createOrReplaceTempView("temp_view_name")*
  - Visible only to a single SparkSession
  - Disappears if the session that creates it terminates.
- Global Views (Global Temporary View)
  - *df.createGlobalTempView("global_temp_view_name")*
  - Shared among all sessions and keep alive until the Spark application terminates
  - Tied to a system preserved database global_temp
  - Must use the qualified name to refer it: e.g. SELECT * FROM global_temp.*global_temp_view_name*

Views

- Views are temporary (both session-scoped and global). They disappear when the Spark application terminates
- Built on top of tables. Views don't hold data
- When accessing a global temporary view, prefix *global_temp.<view_name>* must be used
  - Spark creates global temporary views in a global temporary database called *global_temp*
  - Normal temporary view (session scoped) can be accessed without the *global_temp* prefix.

Creating Views

```
-- In SQL
CREATE OR REPLACE GLOBAL TEMP VIEW us_origin_airport_SFO_global_tmp_view AS
     SELECT date, delay, origin, destination from us_delay_flights_tbl WHERE
     origin = 'SFO';

CREATE OR REPLACE TEMP VIEW us_origin_airport_JFK_tmp_view AS
     SELECT date, delay, origin, destination from us_delay_flights_tbl WHERE
     origin = 'JFK';
```

```
# In Python
df_sfo = spark.sql("SELECT date, delay, origin, destination FROM us_delay_flights_tbl
     WHERE origin = 'SFO'")
df_jfk = spark.sql("SELECT date, delay, origin, destination FROM us_delay_flights_tbl
     WHERE origin = 'JFK'")

Spark.sql("select * from df_sfo

# Create a temporary and global temporary view
df_sfo.createOrReplaceGlobalTempView("us_origin_airport_SFO_global_tmp_view")
df_jfk.createOrReplaceTempView("us_origin_airport_JFK_tmp_view")
```

Dropping Views

```
-- In SQL
DROP VIEW IF EXISTS us_origin_airport_SFO_global_tmp_view;

DROP VIEW IF EXISTS us_origin_airport_JFK_tmp_view
```

```
# In Scala/Python
spark.catalog.dropGlobalTempView("us_origin_airport_SFO_global_tmp_view")

spark.catalog.dropTempView("us_origin_airport_JFK_tmp_view")
```

Caching SQL Tables

- SQL tables and views can be cached and uncached.
- In Spark 3.0, a table can be specified as LAZY - it should only be cached when it is first used instead of immediately.
- *CACHE [LAZY] TABLE <table-name>*
- *UNCACHE TABLE <table-name>*

Reading Tables in DataFrames

```
# In Python
us_flights_df = spark.sql("SELECT * FROM us_delay_flights_tbl")

us_flights_df2 = spark.table("us_delay_flights_tbl")
```

Data Sources for DataFrames and SQL Tables

DataFrameReader

- Core construct for reading data from a data source into a DataFrame
- Defined format and recommended pattern for usage:
  - *DataFrameReader.format(args).option("key", "value").schema(args).load()*
- DataFrameReader can only be accessed through a SparkSession instance.
  - DataFrameReader instances cannot be created.
  - Getting an instance: *SparkSession.read*
- No schema is required when reading from a static Parquet data source
  - The Parquet metadata usually contains the schema, so it's inferred.
  - For streaming data source it is required to have the schema provided.

DataFrameReader

Table 4-1. DataFrameReader methods, arguments, and options

Method	Arguments	Description
`format()`	`"parquet"`, `"csv"`, `"txt"`, `"json"`, `"jdbc"`, `"orc"`, `"avro"`, etc.	If you don't specify this method, then the default is Parquet or whatever is set in `spark.sql.sources.default`.
`option()`	`("mode", {PERMISSIVE | FAILFAST | DROPMALFORMED } )` `("inferSchema", {true | false})` `("path", "path_file_data_source")`	A series of key/value pairs and options. The Spark documentation shows some examples and explains the different modes and their actions. The default mode is `PERMISSIVE`. The `"inferSchema"` and `"mode"` options are specific to the JSON and CSV file formats.
`schema()`	DDL `String` or `StructType`, e.g., `'A INT, B STRING'` or `StructType(...)`	For JSON or CSV format, you can specify to infer the schema in the `option()` method. Generally, providing a schema for any format makes loading faster and ensures your data conforms to the expected schema.
`load()`	`"/path/to/data/source"`	The path to the data source. This can be empty if specified in `option("path", "...")`.

From: Learning Spark, 2nd Ed., Damji

DataFrameWriter

- Unlike with DataFrameReader, an instance of DataFrameWriter is accessed from the DataFrame to be saved - not from a SparkSession.

```
DataFrameWriter.format(args)
     .option(args)
     .bucketBy(args)
     .partitionBy(args)
     .save(path)

# Getting an instance
DataFrame.write
```

DataFrameWriter

Table 4-2. DataFrameWriter methods, arguments, and options

Method	Arguments	Description
`format()`	`"parquet"`, `"csv"`, `"txt"`, `"json"`, `"jdbc"`, `"orc"`, `"avro"`, etc.	If you don't specify this method, then the default is Parquet or whatever is set in `spark.sql.sources.default`.

`option()`	`("mode", {append | overwrite | ignore | error or errorifexists} )`
	`("mode", {SaveMode.Overwrite | SaveMode.Append, SaveMode.Ignore, SaveMode.ErrorIfExists})`
	`("path", "path_to_write_to")`
		A series of key/value pairs and options. The Spark documentation shows some examples. This is an overloaded method. The default mode options are `error or errorifexists` and `SaveMode.ErrorIfExists`; they throw an exception at runtime if the data already exists.

`format()`	`"parquet"`, `"csv"`, `"txt"`, `"json"`, `"jdbc"`, `"orc"`, `"avro"`, etc.	If you don't specify this method, then the default is Parquet or whatever is set in `spark.sql.sources.default`.

`option()`	`("mode", {append | overwrite | ignore | error or errorifexists} )`
	`("mode", {SaveMode.Overwrite | SaveMode.Append, SaveMode.Ignore, SaveMode.ErrorIfExists})`
	`("path", "path_to_write_to")`
		A series of key/value pairs and options. The Spark documentation shows some examples. This is an overloaded method. The default mode options are `error or errorifexists` and `SaveMode.ErrorIfExists`; they throw an exception at runtime if the data already exists.

From: Learning Spark, 2nd Ed., Damji

Parquet

- Default data source in Spark
- Supported and widely used by many big data processing frameworks
- Open source columnar file format that offers many I/O optimizations such as compression
- Stored in a directory structure containing the data files, metadata, a number of compressed files and status files

Parquet

```
-- In SQL
CREATE OR REPLACE TEMPORARY VIEW us_delay_flights_tbl
     USING parquet
     OPTIONS (
          path "/databricks-datasets/learning-spark-v2/flights/summary-data/parquet/
          2010-summary.parquet/" )
```

```
# Reading from a Parquet file in Python
spark.sql("SELECT * FROM us_delay_flights_tbl").show()
```

```
# Writing out to a Parquet file in Python
(df.write.format("parquet")
     .mode("overwrite")
     .option("compression", "snappy")
     .save("/tmp/data/parquet/df_parquet"))
```

```
# Writing out to a table in Python
(df.write
.mode("overwrite")
.saveAsTable("us_delay_flights_tbl"))
```

JSON

- Popular file format due to its ease of use.
- Two representational formats:
  - Single-line mode
    - Each line denotes a single JSON object
    - *multiline* option set to False
  - multi-line mode
    - Entire multiline object constitutes a single JSON object
    - *multiline* option set to True

JSON

```
# In Python
file = "/databricks-datasets/learning-spark-v2/flights/summary-data/json/*"
df = spark.read.format("json").load(file)
```

```
-- In SQL
CREATE OR REPLACE TEMPORARY VIEW us_delay_flights_tbl
     USING json
     OPTIONS (
          path "/databricks-datasets/learning-spark-v2/flights/summary-data/json/*"
     )
```

```
# In Python
(df.write.format("json")
     .mode("overwrite")
     .option("compression", "snappy")
     .save("/tmp/data/json/df_json"))
```

User Defined Functions (UDF)

https://docs.databricks.com/spark/latest/spark-sql/udf-python.html

User Defined Function (UDF)

- Spark allows for data engineers and data scientists to define their own functions called user-defined functions (UDFs).
  - Data scientist can wrap an ML model within a UDF so that a data analyst can query its predictions in Spark SQL.
- UDFs operate per session and they will not be persisted in the underlying metastore.

User Defined Function (UDF)

- Issues with PySpark UDFs: they were slower than Scala UDFs.
  - Data had to be moved between JVM and Python.
- PySpark UDF vs. Pandas (vectorized) UDF
  - A pandas user-defined function (UDF)—also known as vectorized UDF—is a user-defined function that uses Apache Arrow to transfer data and pandas to work with the data.
  - Pandas UDFs allow vectorized operations that can increase performance up to 100x compared to row-at-a-time Python UDFs.
  - References
    - https://docs.databricks.com/spark/latest/spark-sql/udf-python-pandas.html
    - https://databricks.com/blog/2020/05/20/new-pandas-udfs-and-python-type-hints-in-the-upcoming-release-of-apache-spark-3-0.html
    - https://docs.databricks.com/spark/latest/spark-sql/spark-pandas.html

User Defined Function (UDF)

- Pandas UDFs (vectorized UDFs) were introduced in Apache Spark 2.3.
  - https://databricks.com/blog/2017/10/30/introducing-vectorized-udfs-for-pyspark.html
  - A Pandas UDF is defined using the keyword **pandas_udf** as the decorator, or to wrap the function itself.
  - Once the data is in Apache Arrow format, there is no longer the need to serialize/pickle the data as it is already in a format consumable by the Python process. Instead of operating on individual inputs row by row, you are operating on a Pandas Series or DataFrame (i.e., vectorized execution).

User Defined Function (UDF)

- Apache Spark 3.0 with Python 3.6 and above, Pandas UDFs are split into two API categories:
  - Pandas UDFs
    - Pandas UDFs infer the Pandas UDF type from Python type hints in Pandas UDFs such as pandas.Series, pandas.DataFrame, Tuple, and Iterator.
  - Pandas Function APIs
    - Allows users to directly apply a local Python function to a PySpark DataFrame where both the input and output are Pandas instances.

External Data Sources

- JDBC and SQL Databases - Using the data source API, the tables from the remote database can be loaded as a DataFrame or Spark SQL temporary view.
- Partitioning of the data source
  - Required for large amount of data between Spark SQL and a JDBC external source
    - All data goes through one driver connection
  - numPartitions, use a multiple of the number of Spark workers.
  - Need to consider how well the source system can handle the read requests. It is possible to saturate the source system.

External Data Sources

`partitionColumn, lowerBound, upperBound`	These options must all be specified if any of them is specified. In addition, `numPartitions` must be specified. They describe how to partition the table when reading in parallel from multiple workers. `partitionColumn` must be a numeric, date, or timestamp column from the table in question. Notice that `lowerBound` and `upperBound` are just used to decide the partition stride, not for filtering the rows in table. So all rows in the table will be partitioned and returned. This option applies only to reading.
`numPartitions`	The maximum number of partitions that can be used for parallelism in table reading and writing. This also determines the maximum number of concurrent JDBC connections. If the number of partitions to write exceeds this limit, we decrease it to this limit by calling `coalesce(numPartitions)` before writing.

https://spark.apache.org/docs/latest/sql-data-sources-jdbc.html
https://docs.databricks.com/data/data-sources/sql-databases.html#manage-parallelism

Higher-Order Functions

Complex Data Types

Before Spark 2.4, two typical solutions for manipulating complex data types:

- Exploding the nested structure into individual rows, applying some function, and then re-creating the nested structure (using EXPLODE() and collect_list())
  - As values could be any number of dimensions (a really wide and/or really long array) and we're doing a GROUP BY, (shuffle operation) this approach could be very expensive.
  - collect_list() may cause executors to experience out-of-memory issues for large data sets.
  - Utility functions: get_json_object(), from_json(), to_json(), explode(), and selectExpr().
- Building a user-defined function
  - The serialization and deserialization process may be expensive.
  - UDFs do not have the out-of-memory issues that collect_list() may run into.

Better Approach: Use built-in functions for Complex Data Types

* Complex data types are nested data structures composed of simple data types (primitives)

The Best Approach: Higher-Order Functions

- Higher-order functions take lambda functions as arguments.
- They allow users to efficiently create functions, in SQL, to manipulate array based data.
  - A simple extension to SQL to manipulate nested data such as arrays

Lab: Higher-Order Functions

- https://docs.databricks.com/_static/notebooks/higher-order-functions.html

Common DataFrames and Spark SQL Operations

- Aggregate functions
- Collection functions
- Datetime functions
- Math functions
- Miscellaneous functions
- Non-aggregate functions
- Sorting functions
- String functions
- UDF functions
- Window functions

https://spark.apache.org/docs/latest/sql-ref-functions-builtin.html

Window Functions

- Aggregate Function vs. Window Functions
  - Aggregate functions take a group of rows and output a single value for them
  - Window functions allow keeping the individual rows as well as gaining a summarized value.
  - All aggregate functions can be used as window functions

Window Functions

```
SELECT columns,
window_function OVER (PARTITION BY partition_key ORDER BY order_key)
FROM table
```

- PARTITION BY works like GROUP BY. It divides the dataset into multiple groups
  - For each group a window is created
- When ORDER BY is not specified, the window is the entire group
- When ORDER BY is specified, **the rows in the group are ordered** according to it, and a window is created for each row over which a function is applied.
- When window is not specified, by default a window is created from every rows.
- PARTITION BY groups, ORDER BY creates windows

Window Functions

	SQL	DataFrame API
Ranking functions	rank	rank
	dense_rank	denseRank
	percent_rank	percentRank
	ntile	ntile
	row_number	rowNumber
Analytic functions	cume_dist	cumeDist
	first_value	firstValue
	last_value	lastValue
	lag	lag
	lead	lead

Window Functions

Additional references:

- https://jaceklaskowski.gitbooks.io/mastering-spark-sql/content/spark-sql-functions-windows.html
- https://medium.com/expedia-group-tech/deep-dive-into-apache-spark-window-functions-7b4e39ad3c86

Questions

Homework

https://docs.databricks.com/getting-started/spark/dataframes.html