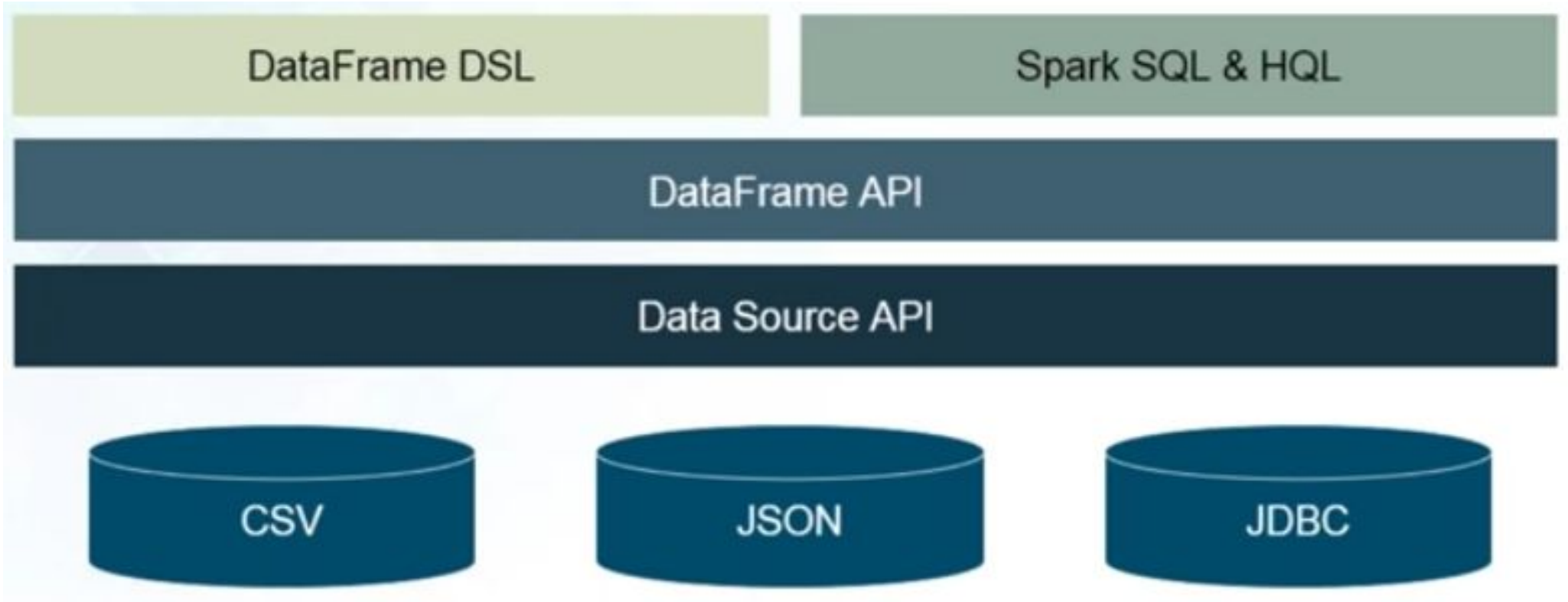
Spark SQL

Introduction

- Spark SQL is Spark module for structured data processing.
- It runs on top of Spark Core.
- It offers much tighter integration between relational and procedural processing, through declarative **DataFrame** and **Datasets API**.
- These are the ways which enable users to run SQL queries over Spark.
- **Apache Spark SQL** integrates relational processing with Sparks functional programming.
- Spark SQL blurs the line between **RDD** and relational table.

Spark SQL Architecture



What is Apache Spark SQL?

- **DataFrame API** and **Datasets API** are the ways to interact with Spark SQL. As a result, Apache Spark is accessible to more users and improves optimization for current ones.

- It allows developers to import relational data from **Hive** tables and parquet files, run SQL queries over imported data and existing RDDs and easily write RDDs out to Hive tables or Parquet files.

- Spark SQL introduces extensible optimizer called **Catalyst** as it helps in supporting a wide range of data sources and algorithms in **Bigdata.**

Catalyst Optimizer

- It is the newest and most technical component of Spark SQL.
- A **catalyst** is a query plan optimizer. It provides a general framework for transforming *trees*, which performs analysis/evaluation, optimization, planning, and runtime code spawning.
- Catalyst supports **cost based optimization** and **rule-based optimization**.
- It makes queries run much faster than their RDD counterparts.
- A catalyst is a rule-based modular library. Each rule in framework focuses on the distinct optimization.

- The **Spark SQL Catalyst** Optimizer improves developer productivity and the performance of their written queries.

DataFrames – Introduced in Spark 1.3

- A DataFrame is similar/identical to a table in a relational database but with richer optimization
- Spark DataFrames evaluates lazily like **RDD Transformations** in Apache Spark.
- It is a data abstraction and **domain-specific language (DSL)** applicable on the structure and semi-structured data.
- It is a distributed collection of data in the form of named column and row.
- A DataFrame is a *Dataset* organized into named columns.
- Any operation on DataFrame results in DataFrame only.
- Like RDD, DataFrames are also immutable.
- Available in Java, Python, Scala and R.

Datasets – Introduced in Spark 1.6

- Extension of DataFrames
- Provides the benefits of RDDs (strong typing, ability to use powerful lambda functions) with the benefits of Spark SQL's optimized execution engine.
- A Dataset can be constructed from JVM objects and then manipulated using functional transformations (map, flatMap, filter, etc.).
- The Dataset API is available in Scala and Java. Python does not have the support for the Dataset API. But due to Python's dynamic nature, many of the benefits of the Dataset API are already available (i.e. you can access the field of a row by name naturally row.columnName). The case for R is similar.
- Offers compile-time safety, which is not available in DataFrames

Why DataFrames?

- DataFrame is one step ahead of **RDD s**ince it provides memory management and optimized execution plan.

a. Custom Memory Management: This is also known as Project **Tungsten.** A lot of memory is saved as the data is stored in off-heap memory in binary format. Apart from this, there is no Garbage Collection overhead. Expensive Java serialization is also avoided.

b. Optimized Execution plan: This is also known as the **query optimizer**. Using this, an optimized execution plan is created for the execution of a query. Once the optimized plan is created final execution takes place on RDDs of Spark.

SQLContext, Hive Context and JDBC Datasource(Version 1(Version 2-SparkSession spark))

- SQLContext is the entry point for working with structured data (rows and columns) in Apache Spark. It Allows the creation of DataFrame objects as well as the execution of SQL queries.

- Working with Hive tables, Hive Context is a descendant of SQLContext. Hive Context is more battle-tested and provides a richer functionality than SQLContext.

- In Apache Spark, JDBC data source can read data from relational databases using **JDBC API**. It has preference over the *RDD* because the data source returns the results as a DataFrame, can be handled in Spark SQL or joined beside other data sources.


Data Flair
01
Integrated
02
Uniform Data Access
Features of Spark SQL
05
Hive Compatibility
03
Standard Connectivity
04
Performance & Scalability

i. Integrated

Logically mix SQL queries with Spark programs. Apache Spark SQL allows to query structured data inside Spark programs, using SQL or DataFrame API.

ii. Uniform Data Access

Spark DataFrames and SQL supports a common way to access a variety of data sources, like *Hive, Avro, Parquet, ORC, JSON, and JDBC.* Hence Spark SQL can join data across these sources.

iii. Hive Compatibility

Runs unmodified Hive queries on current data. Spark SQL rewrites the Hive frontend and meta store, allowing full compatibility with current Hive data, queries, and UDFs.

iv. Standard Connectivity

Connect through JDBC or ODBC. A server that supports industry norms JDBC and ODBC connectivity for business intelligence tools.

v. Performance & Scalability

Apache Spark SQL incorporates a cost-based optimizer, code generation, and columnar storage to make queries agile alongside computing thousands of nodes using the Spark engine, which provides full mid-query fault tolerance.

HandsOn

Demonstration in local mode

- Start spark in local mode using ‘pyspark’ command.
- from pyspark.sql import SparkSession //May not be needed if Sparksession object is already created.
- spark=SparkSession.builder.appName("Basics").getOrCreate()
- df=spark.read.csv('file:///home/pinkal/Downloads/booklist.csv')
- df.show()
- df.columns
- df.describe()
- df.printSchema()

Demonstration in Standalone mode with HDFS as storage

- Start HDFS deamons using ‘start-dfs.sh’
- Start spark master using ‘start-master.sh’
- Start worker using ‘start-slave.sh spark://hadoop-master:7077’
- Start python shell using ‘pyspark –master spark://hadoop-master:7077’
- from pyspark.sql import SparkSession //May not be needed if Sparksession object is already created.
- spark=SparkSession.builder.appName("Basics").getOrCreate()
- df=spark.read.csv(‘/PartitionerDemo/input/partitioner_input.csv')
- df.show() //Shows first 20 lines
- df.columns
- df.describe()
- df.printSchema()
- df.head(2) //Returns first two row object

Customized Schema

- By Default, Spark automatically tries to infer schema from the file.
- But if there is a need to have specific schema, then we will have to specify schema explicitly.
- In this case, spark will not automatically infer schema, instead it will only read the created schema and will try to fit data into that schema.

Using StructType

- from pyspark.sql.types import StructField,StringType,IntegerType,StructType
- data_schema = [StructField("id",IntegerType(),True),StructField("title",StringType(),True),StructField("author",StringType(),True),StructField("format",StringType(),True),StructField("price",IntegerType(),True)]
- final_struct=StructType(fields=data_schema)
- df=spark.read.csv("file:///home/pinkal/Downloads/booklist.csv",schema=final_struct)
- df.printSchema()
- df.columns

Some more operations

- df[“id”]
- type(df[‘id’])
- df.select(‘id’).show()
- df.withColumn("newid",df["id"]).show()
- df.withColumn("doubleid",df["id"]*2).show()
- df.show() //Note original dataframe is not changing
- df.withColumnRenamed('id','uniqueid').show()

Working with SQL

- df.createOrReplaceTempView("book")
- sql_results = spark.sql("Select * from book")
- sql_results.show()

References

- https://data-flair.training/blogs/apache-spark-sql/
- https://blog.bi-geek.com/en/spark-sql-optimizador-catalyst/
- https://www.edureka.co/community/53384/spark-dataframe-vs-data set