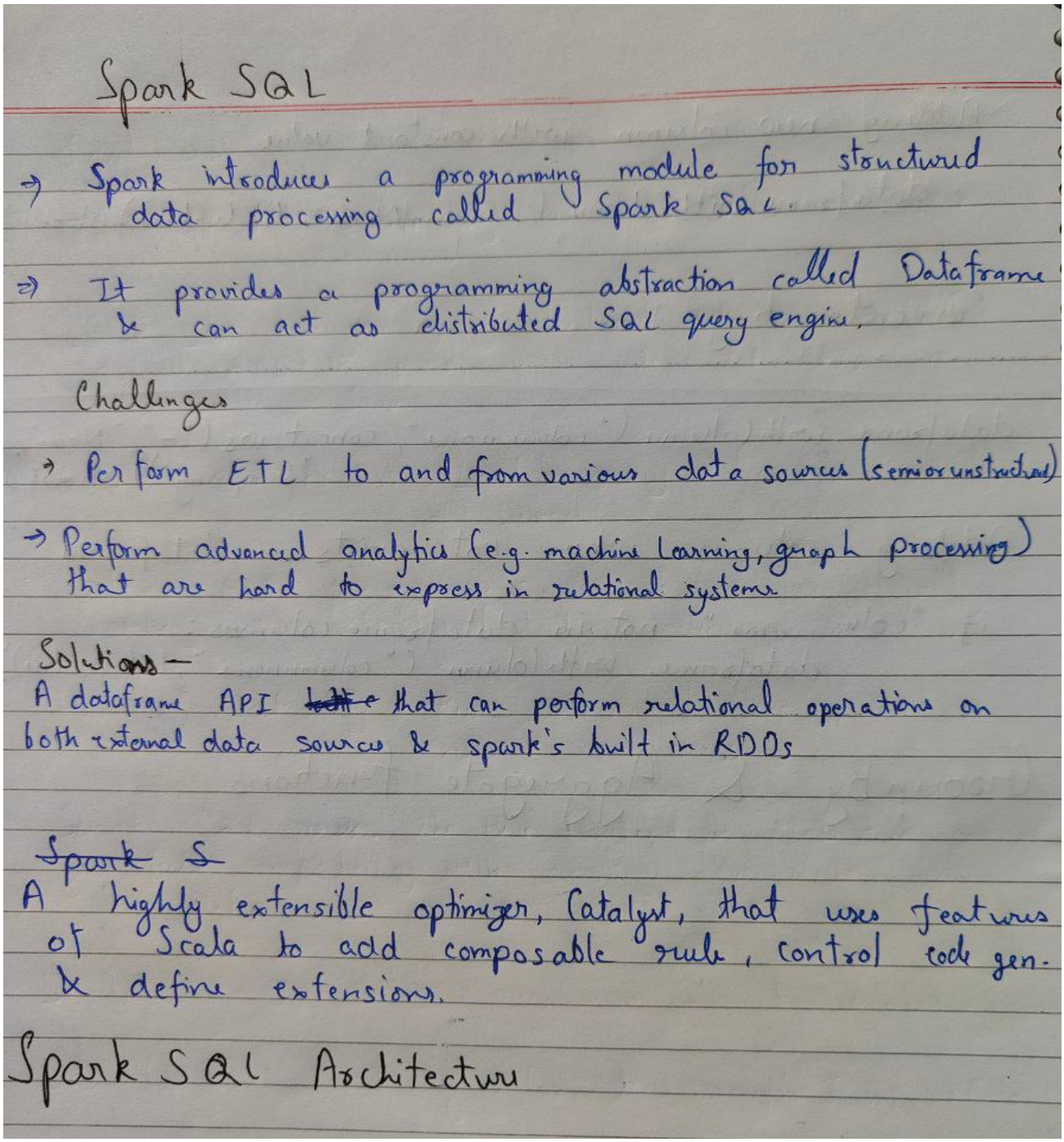
Assignment-15

Name- Ajay Chaudhary
Batch-Data Engineering (Batch-1)

Handwritten notes-



Spark SQL

⇒ Spark introduces a programming module for structured data processing called Spark SQL.

⇒ It provides a programming abstraction called Dataframe & can act as distributed SQL query engine.

Challenges

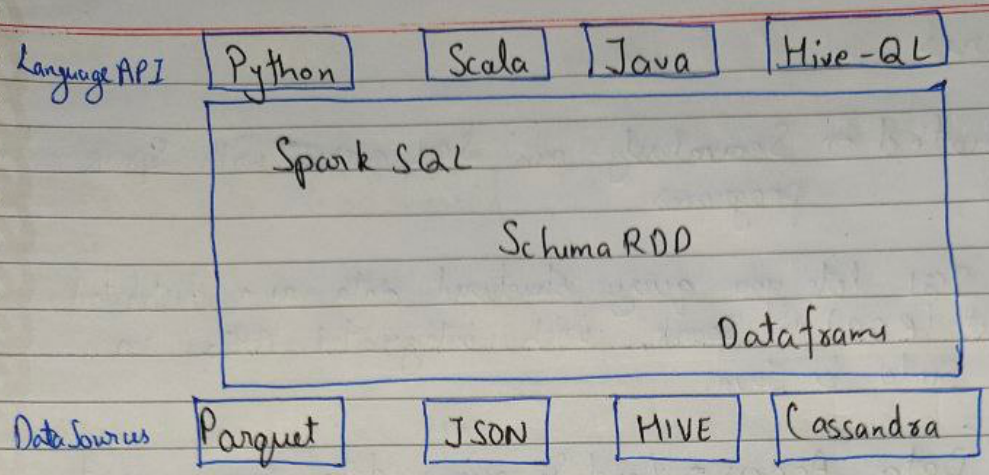
→ Perform ETL to and from various data sources (semi or unstructured)

→ Perform advanced analytics (e.g. machine learning, graph processing) that are hard to express in relational systems

Solutions –
A dataframe API ~~that~~ that can perform relational operations on both external data sources & spark's built in RDDs

~~Spark~~ S
A highly extensible optimizer, Catalyst, that uses features of Scala to add composable rule, control code gen. & define extensions.

Spark SQL Architecture



Language API ⇒ Spark is compatible with different languages & spark SQL.
It is also, supported by these languages - API (python, scala, java, HiveQL)

Schema RDD ⇒ Spark core is designed with special data structure called RDD.
Generally, Spark SQL works on schemas, tables & records.
Therefore, we can use the schema RDD as temporary table.
We can call this Schema RDD as data frame.

Data Sources ⇒ Usually data source are txt file, Avro file.
However data sources for Spark SQL is different.
Those are Parquet file, JSON document, HIVE tables & Cassandra database.

Features

⇒ Integrated ⇒ Seamlessly mix SQL queries with Spark programs.

Spark SQL lets you query structured data as a distributed dataset (RDD) in Spark, with integrated APIs in Python Scala & Java.

Unified Data Access ⇒ Load & query data from a variety of sources.

- Schema - RDDs provide a single interface for efficiently working with structured data, including Hive tables.

Hive Compatibility ⇒ Run unmodified Hive queries on existing warehouses.

Spark SQL reuses the Hive frontend & MetaStore, giving you full compatibility with existing Hive data, queries & UDFs

Scalibility - Use the same engine for both interactive & long queries.

Standard Connectivity ⇒ Connect through JDBC or ODBC

Spark RDD

- RDD is a fundamental data structure of Spark.
- It is an immutable distributed collection of objects that can be stored in memory or disk across a cluster.
- Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster.
- Parallel functional transformations (map, filter, ...)
- Automatically rebuilt on failure.
- RDDs can contain any type of Python, Java or Scala Objects, including user-defined classes.
- RDD is a fault-tolerant collection of elements that can be operated on in parallel.
- Two ways to create RDDs —
  - parallelizing an existing collection in your driver program.
  - referencing a dataset in an external storage system, such as shared file system, HDFS, HBase.
- Spark makes use of concept of RDD to achieve faster & efficient map reduce.

Dataset & Dataframe

- a distributed collection of data, which is organized into named columns.
- Equivalent to relational tables with good optimization

Dataframe ⇒ Data is organized into named columns, like a table in a relational database.

Dataset ⇒ a distributed collection of data
- a new interface added in Spark 1.6
- static-typing & runtime-safety

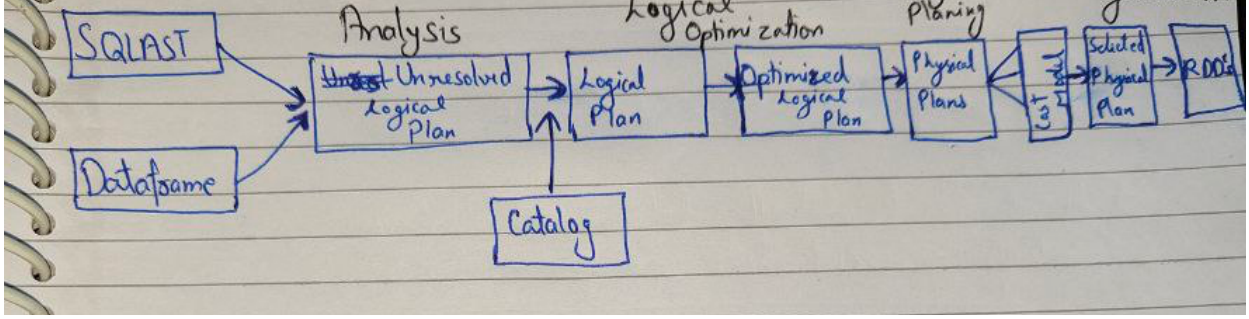
Features of Dataframe —

⇒ Ability to process the data in the size of KB to Petabytes on a single node cluster to large cluster.

State of art optimization & code generation through the Spark SQL catalyst optimizer.

Provides API for Python, Java, Scala & R programming.

Plan Optimization & Execution



The entry point into all functionality in Spark is the `SparkSession` class. To create a basic `SparkSession`, just use `SparkSession.builder`:

[2]:
```
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("Python Spark SQL basic example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

```
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
```

Creating DataFrames

With a `SparkSession`, applications can create DataFrames from an existing `RDD`, from a Hive table, or from Spark data sources.

As an example, the following creates a DataFrame based on the content of a JSON file:

```
df = spark.read.json("/Users/ajaychaudhary/jupyter/PySpark/people.json", multiLine=True)
# Displays the content of the DataFrame to stdout
df
df.show()
```

```
+----+-------+
| age|   name|
+----+-------+
|NULL|Michael|
|  30|   Andy|
|  19| Justin|
+----+-------+
```

Untyped Dataset Operations (aka DataFrame Operations)

DataFrames provide a domain-specific language for structured data manipulation in Scala, Java, Python and R.

As mentioned above, in Spark 2.0, DataFrames are just Dataset of `Row`s in Scala and Java API. These operations are also referred as “untyped transformations” in contrast to “typed transformations” come with strongly typed Scala/Java Datasets.

Here we include some basic examples of structured data processing using Datasets:

```
# spark, df are from the previous example
# Print the schema in a tree format
df.printSchema()

# Select only the "name" column
df.select("name").show()
```

```
root
 |-- age: long (nullable = true)
 |-- name: string (nullable = true)

+-------+
|   name|
+-------+
|Michael|
|   Andy|
| Justin|
+-------+
```

[6]:
```
# Select everybody, but increment the age by 1
df.select(df['name'], df['age'] + 1).show()
```

```
+-------+---------+
|   name|(age + 1)|
+-------+---------+
|Michael|     NULL|
|   Andy|       31|
| Justin|       20|
+-------+---------+
```

[7]:
```
# Select people older than 21
df.filter(df['age'] > 21).show()
```

```
+---+----+
|age|name|
+---+----+
| 30|Andy|
+---+----+
```

[8]:
```
# Count people by age
df.groupBy("age").count().show()
```

```
+----+-----+
| age|count|
+----+-----+
|  19|    1|
|NULL|    1|
|  30|    1|
+----+-----+
```

Running SQL Queries Programmatically

The `sql` function on a `SparkSession` enables applications to run SQL queries programmatically and returns the result as a `DataFrame`.

[9]:
```
# Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView("people")

sqlDF = spark.sql("SELECT * FROM people")
sqlDF.show()
```

```
+----+-------+
| age|   name|
+----+-------+
|NULL|Michael|
|  30|   Andy|
|  19| Justin|
+----+-------+
```

Global Temporary View

Temporary views in Spark SQL are session-scoped and will disappear if the session that creates it terminates. If you want to have a temporary view that is shared among all sessions and keep alive until the Spark application terminates, you can create a global temporary view. Global temporary view is tied to a system preserved database `global_temp`, and we must use the qualified name to refer it, e.g. `SELECT * FROM global_temp.view1`.

[10]:
```
# Register the DataFrame as a global temporary view
df.createGlobalTempView("people")

# Global temporary view is tied to a system preserved database `global_temp`
spark.sql("SELECT * FROM global_temp.people").show()

# Global temporary view is cross-session
spark.newSession().sql("SELECT * FROM global_temp.people").show()
```

```
+----+-------+
| age|   name|
+----+-------+
|NULL|Michael|
|  30|   Andy|
|  19| Justin|
+----+-------+

+----+-------+
| age|   name|
+----+-------+
|NULL|Michael|
|  30|   Andy|
|  19| Justin|
+----+-------+
```