Storing data in HDFS and processing it using Apache Spark has proven to be a high performance data analysis solution when compared to MapReduce.

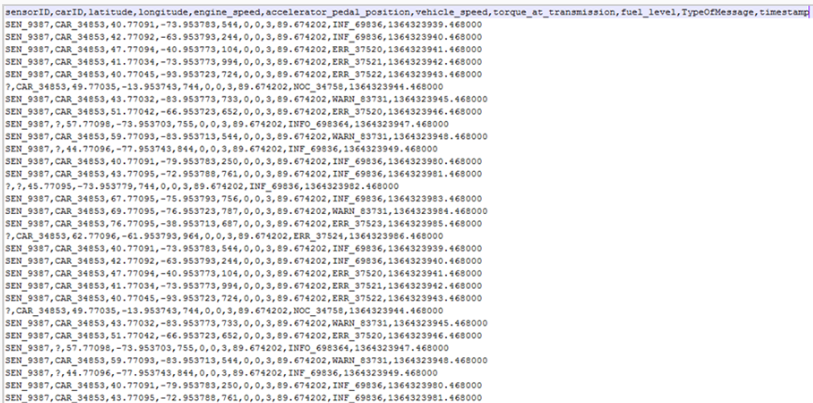
But Spark Core programming produces development complexities in terms of more lines of code to achieve advanced data analysis requirements like aggregations(Computing sum of values grouping by key). There is a need to use a simple querying model to perform these advanced analysis.

This course introduces you to Spark SQL, a querying model on top of Spark Core API, to perform data analysis at very high speed. It is proven to be 100 times faster than MapReduce and querying solution like Hive(Refer here). It supports different file formats and enables developers to write SQL queries against text files, JSON, Parquet and even Hive tables directly.

# User story

Arisconn Cars provides rental car service across the globe. To improve their customer service, the client wants to periodically analyze each car’s sensor data to repair faults and problems in the car.

Sensor data from cars are streamed through Events Hub (data ingestion tool) into Hadoop's HDFS (distributed file system), and analyzed using Spark Core programming to find out cars generating maximum errors. This analysis would help Arisconn send the service team to repair the cars even before they fail.



Arisconn has the below set of requirements to be performed against the dataset:

Filter fields - Sensor id, Car id, Latitude, Longitude, Vehicle Speed, TypeOfMessage

Filter valid records i.e. Discard records containing '?'

Filter records holding only error messages (ignore warnings and info messages)

Apply aggregation to count number of error messages produced by cars

Below is the Spark code to implement the first three requirements.

//Loading a text file in to an RDD

val Car\_Info = sc.textFile(args(0));

//Referring the header of the file

val header = Car\_Info.first();

//Removing header and splitting records with ',' as delimitter and fetching relevant fields

val Car\_temp = Car\_Info.filter(record => record!=header).map(\_.split(",")).map(c =>(c(0),c(1),c(2).toDouble,c(3).toDouble,c(6).toInt,c(9)));

//Filtering only valid records(records not starting with '?'), and \_.\_1 refers to first field (sensorid)

val Car\_Eng\_Specs = Car\_temp.filter(!\_.\_1.startsWith("?"));

//Filtering records holding only error messages and \_.\_6 refers to 6th field (Typeofmessage)

val Car\_Error\_logs = Car\_Eng\_Specs.filter(\_.\_6.startsWith("ERR"));

As per the previous code below are few observations and challenges of Spark Core programming

* No schema or table like structure for performing advanced aggregations
* Use of indices instead of field names. For instance, \_.\_1 is used to refer first field(sensor id)
* Code customization will be highly difficult when the dataset's schema changes

Also, implementation of the fourth aggregate requirement is another challenge as shown below

**C**ode to compute the number of errors produced by every car

1. //Loading a text file in to an RDD
2. val Car\_Info = sc.textFile(args(0));
3. //Splitting records with ',' as delimitter and fetching relevant fields
4. val Car\_temp = Car\_Info.map(\_.split(",")).map(c =>(c(0),c(1),c(2).toDouble,c(3).toDouble,c(6).toInt,c(9)));
5. // Filtering only valid records not starting with '?'
6. val Car\_Eng\_Specs = Car\_temp.filter(!\_.\_1.startsWith("?"));
7. val Car\_Error\_logs = Car\_Eng\_Specs.filter(\_.\_5.startsWith("ERR"));
8. //Filtering records holding only error messages
9. val Car1Errors = Car\_Error\_logs.filter(\_.\_1.startsWith("CAR\_34853")).count();
10. //Filtering car1 records and counting the number of occurences
11. val Car2Errors = Car\_Error\_logs.filter(\_.\_1.startsWith("CAR\_34854")).count();
12. //Filtering car2 records and counting the number of occurences
13. .
14. .
15. //Lines of code would increase as per more number of cars.
16. //Combining and aggregating all the cars errors and printing them would be highly challenging

We find that count() in the above code is applied individually on each car. With more number of cars, more number of statements would be required.

Same applies to other complex aggregate functions such as AVG, SUM, MIN, MAX.

**Spark SQL**provides a simpler solution which provides structure to the underneath dataset and provides SQL like queries for doing data analytics at high speed.

Let us look at how a Spark SQL code looks like for the same set of analysis against Arisconn's dataset.

1. //SQL Context object creation in Spark SQL
2. val sqlContext = new org.apache.spark.sql.SQLContext(sc);
3. //Loading a text file in to an RDD
4. val Car\_Info = sc.textFile("hdfs://vimsmys-42:9000/user/jai\_trng/jaihdfs/ArisconnCars.txt");
5. // Creating a case class mapping the fields in the dataset
6. case class Cars(sensorid:String,carid:String,latitude:Double,longitude:Double,engine\_speed:Int,accelerator\_pedal\_position:Int,vehicle\_speed:Int,torque\_at\_transmission:Int,fuel\_level:Double,TypeOfMessage:String,timestamp:Double);
7. val header = Car\_Info.first();
8. // Creating a Spark SQL DataFrame
9. val DF = Car\_Info.filter(c => c!=header).map(\_.split(",")).map(c => Cars(c(0),c(1),c(2).toDouble,c(3).toDouble,c(4).toInt,c(5).toInt,c(6).toInt,c(7).toInt,c(8).toDouble,c(9),c(10).toDouble)).toDF();
10. //Registering the DataFrame as a temporary table
11. DF.registerTempTable("cars");
12. // A simple query to implement all 4 requirements of Arisconn cars
13. val errors= sqlContext.sql("SELECT sensorid,carid,latitude,longitude,vehicle\_speed,count(TypeOfMessage) FROM cars WHERE sensorID!='sensorID' AND carid!='?' AND TypeOfMessage Like 'ERR.\*' group by carid");

**Advantages of Spark SQL**

As you see above,

* The code contains a single elegant Spark SQL query for all 4 requirements decreasing the number of lines of code
* For future client analysis requirements, new query statements shall be added making code maintainability easier

**How does the code work?**

* In Spark SQL, data is loaded in to a special component called **DataFrame**which is a programmatic component meant for loading and applying schema to datasets
* Temporary table 'cars' is created on DataFrame
* SQL Query is run against it

In the above code, 'DF'(DataFrame) applies schema to the dataset via the 'Cars' case class

# Uses of Spark \_SQL

Spark SQL is a Spark ecosystem component used for structured data processing. It provides a structure/schema to the underneath dataset and helps perform computations using direct SQL like queries.

No additional configuration needed to use Spark SQL. Developers can write Spark SQL queries along with their existing Spark Core's code in the Spark scala shell.

Listed below are the features of Spark SQL:

* Ability to use queries for data analysis
* Ability to run analytics directly against Hive tables
* Spark SQL jobs run 100 times faster than Hive jobs
* Spark SQL supports different file formats like Parquet, JSON, XML, Text, CSV/TSV etc

**Spark SQL - Programmatic Components**

Below are the Spark SQL programmatic components which helps provide structure to the dataset and run queries on top of it

* DataFrame
* Dataset

Let us see what are DataFrames and Datasets in Spark SQL and how their APIs could be used for writing Spark SQL jobs

Below are the steps for writing Spark SQL queries using DataFrame

1. Create a SparkContext object and load the dataset using the SparkContext object's textFile() method

1. //File in args(0) is loaded in to Spark RDD using SparkContext object 'sc'
2. val Car\_Info = sc.textFile(args(0));

2. Create a SQLContext object using SQLContext class

1. //Creation of a SQLContext object in Spark SQL
2. val sqlContext = new org.apache.spark.sql.SQLContext(sc);

3. Create a DataFrame and load a normal RDD in to the DataFrame using Case class

1. //Creation of a case class mapping the fields in the dataset
2. case class Cars(sensorid: String,carid: String,latitude:Double,longitude:Double ....);
3. /\*Creation of a Spark SQL DataFrame by delimitting the fields, and loading them as case class properties.
4. Note: "toInt", "toDouble" are Scala methods for converting text fields in to numerical\*/
5. val DF = Car\_Info.map(\_.split(",")).map(c => Cars(c(0), c(1),c(2).toDouble,c(3).toDouble,c(4).toInt,c(5).toInt,c(6).toInt,c(7).toInt,
6. c(8).toDouble,c(9),c(10).toDouble)).toDF();

4. Register the DataFrame as a normal table using registerTempTable() method

1. //Registering the DataFrame as a temporary table
2. DF.registerTempTable("cars");

5. Write and execute Spark SQL queries against the table using SQLContext object's sql() method

1. /\* Query using sql method to filter out relevant fields and counting the number of errors
2. generated by every car using count() method grouping by carid \*/
3. val errors= sqlContext.sql("SELECT sensorid,carid,latitude,longitude,vehicle\_speed,count(TypeOfMessage) FROM cars WHERE carid!='?'
4. AND TypeOfMessage Like 'ERR.\*' group by carid");

**Note:** DataFrame also offers special API methods apart from SQL queries.

Various API methods of DataFrame are specified below:

df.show() == shows the contents of the dataframe

df.select(‘Col\_Name’).show() == select specific columns of df

df.filter(df.col(‘age’).gt(19)).show() = shows age > 19 values only

df.groupBy(‘age’).count().show() == groups by age col and gives count() we ca give min(),max(),mean() etc also

# we will load the dataset, filter records, apply schema, and persist the output as a Parquet file.

**1.1 Load/Structure the Dataset**

Load the sensor dataset in to Spark RDD

1. //SQL Context object creation in Spark SQL
2. val sqlContext = new org.apache.spark.sql.SQLContext(sc);
3. //File in args(0) is loaded in to Spark RDD using SparkContext object 'sc'
4. val Car\_Info = sc.textFile("hdfs://vimsmys-42:9000/user/jai\_trng/jaihdfs/ArisconnCars.txt");

Create a Case class to map the fields of the dataset

1. // Creating a case class mapping the fields in the dataset
2. case class Cars(sensorid: String,carid: String,latitude:Double,longitude:Double,engine\_speed:Int,accelerator\_pedal\_position:Int,
3. vehicle\_speed:Int,torque\_at\_transmission:Int,fuel\_level:Int,
4. TypeOfMessage:String,timestamp:Double)

Create a DataFrame and register as temporary table

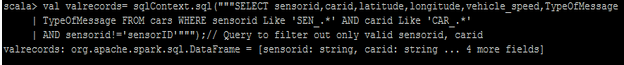
1. //Pointing the header to remove it before SQL analysis
2. val header = Car\_Info.first()
3. // Creating a Spark SQL DataFrame by delimitting the fields and loading them as Case class properties.
4. Note that, "toInt", "toDouble" are Scala methods for converting text fields in to numericals
5. val DF = Car\_Info.filter(c => c!=header).map(\_.split(",")).map(c => Cars(c(0), c(1),c(2).toDouble,c(3).toDouble,
6. c(4).toInt,c(5).toInt,c(6).toInt,c(7).toInt,c(8).toDouble,c(9),c(10).toDouble)).toDF();
7. //Registering the DataFrame as a temporary table
8. DF.registerTempTable("cars");

Write Spark SQL query to filter valid records

1. //Spark SQL Query to fetch only valid records where sensorid starts with SEN and carid starts with CAR

val valrecords= sqlContext.sql("SELECT sensorid,carid,latitude,longitude,vehicle\_speed,TypeOfMessage FROM cars WHERE sensorid Like 'SEN\_.\*' AND carid Like 'CAR\_.\*'order by vehicle\_speed desc LIMIT 20")

Note :: While writting this select query in the unix scala environment we need to place lie below



To print the records in the output :

===== Valrecords.collect.foreach(println)

**1.2 Store as Parquet**

1. //Store the results of the DataFrame as a parquet file in an HDFS directory
2. valrecords.write.parquet("/user/jai\_trng/jaihdfs/Cars.parquet")

== To see the file hadoop fs –ls /user/jai\_trng/jaihdfs

**What is a Parquet file?**

* Parquet is columnar file format supported across many data processing systems
* Parquet files preserves the schema of a file while storing it
* Spark SQL offers API functions to read/write parquet files

**2.1 Load Parquet File**

In this step, create a SQLContext object and use **read.parquet()** method to load the parquet file created in our earlier requirement

1. val sqlContext = new org.apache.spark.sql.SQLContext(sc)
2. val carsparq = sqlContext.read.parquet("hdfs://vimsmys-42:9000/user/jai\_trng/jaihdfs/CarsParq.parquet")

**2.2 Parquet to Hive Table**

In this step, **write.option** is used to specify the Hive matastore directory for the Hive table. **saveAsTable()** is used to persist the DataFrame as a hive table

1. //write.option method use to specify the path for the Hive table and saveAsTable method creates a Hive table in Hive metastore
2. carsparq.write.option("path",'/home/jai\_trng/jaicarsparq').saveAsTable("SparkHiveCarsTable")

**2.3 Spark SQL on Hive**

In this step, HiveContext object is created and using which Spark SQL queries are written against the existing Hive tables. HiveContext object provides an entry point for Spark to Hive.

1. //Creation of HiveContext object for Spark to connect to Hive
2. val hContext = new org.apache.spark.sql.hive.HiveContext(sc);
3. //Spark SQL query on a hive table to compute the top 10 vehicles which were running on maximum speed
4. val CarHiveData = hContext.sql("select carid,latitude,longitude,MAX(vehicle\_speed) from SparkHiveCarsTable GROUP BY carid limit 10")
5. //Output of the query to be displayed
6. CarHiveData.show();

# Joining the datasets

Arisconn Cars third requirement is to join the current dataset with the **Error\_Code** dataset to fetch the complete meaning of the error codes.

Here, **Error\_Code** dataset will be loaded into a Hive table called **SparkHiveCarsErrors** and Spark SQL join will be applied against the two tables.

**3.1 Load the Second Dataset**

Create another Hive table, **SparkHiveCarsErrors**and load the dataset (Engine\_ErrorCodes.txt) into the table as shown below

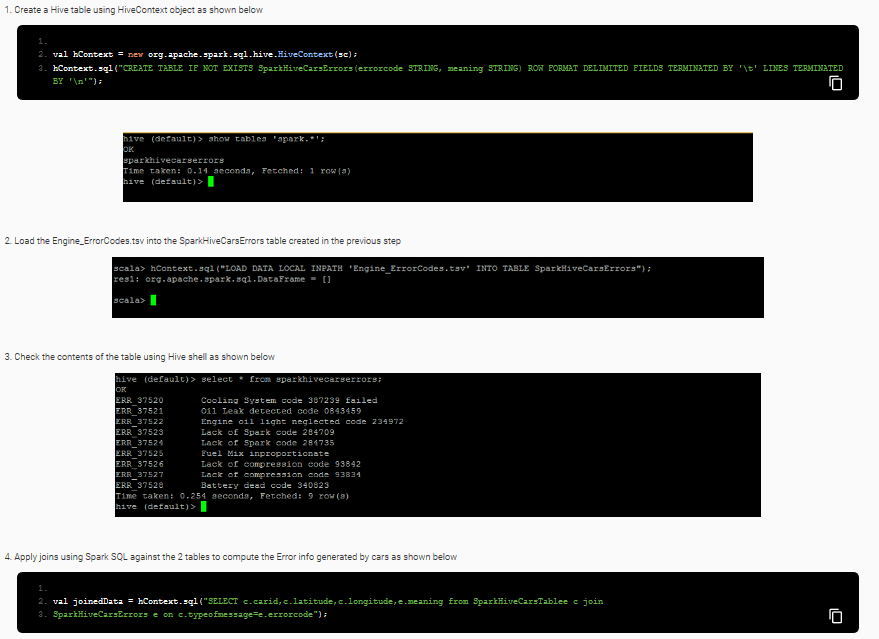
1. //Creating a Hive table using HContext
2. val hContext = new org.apache.spark.sql.hive.HiveContext(sc)
3. hContext.sql("CREATE TABLE IF NOT EXISTS SparkHiveCarsErrors(errorcode STRING, meaning STRING) ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' LINES TERMINATED BY '\n'");
4. hContext.sql("LOAD DATA LOCAL INPATH 'Engine\_ErrorCodes.txt' INTO TABLE SparkHiveCarsErrors")

In the above snippet, HiveContext object was created using SparkContext 'sc' and sql method of HiveContext object used to create a Hive table and load a local dataset 'Engine\_ErrorCodes.txt' into the Hive table.

**3.2 Join Datasets**

Apply Spark SQL join against the 2 tables (SparkHiveCarsErrors and SparkHiveCarsTable) to compute the error info generated by every car

1. //sql method of HiveContext used to create a Spark SQL join query against 2 Hive tables
2. val joinedData = hContext.sql("SELECT c.sensorid,c.carid,c.vehicle\_speed,e.meaning from SparkHiveCarsTable c join SparkHiveCarsErrors e on c.typeofmessage=e.errorcode")
3. joinedData.show();



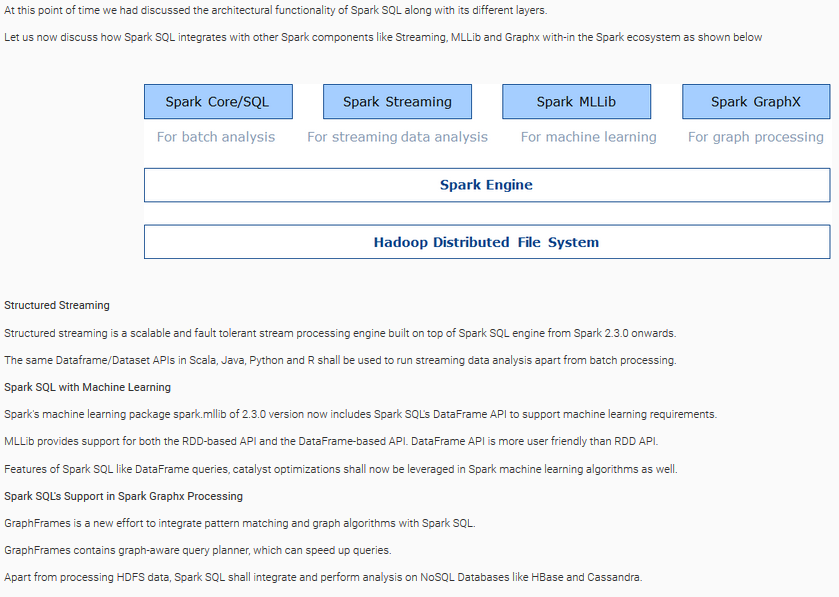
joinedData.show() = = gives the output

### Q) Shane works in a Spark SQL project and likes to load a local dataset (Customer.txt) into a Hive table (CustHive) using Spark SQL.

Ans) **val hContext = new org.apache.spark.sql.hive.HiveContext(sc);  
hContext.sql("CREATE TABLE IF NOT EXISTS CustHive(cid INT, custname STRING, email STRING, pid INT) ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' LINES TERMINATED BY '\n'");  
hContext.sql("LOAD DATA LOCAL INPATH 'Customer.txt' INTO TABLE CustHive");**

### Q) In a Banking project Shane has a requirement to join the Bank's Customer data in Hive with the Sale data (Sale.txt file) to compute the type of loans availed by the customer.

### 1. Create a HiveContext object 2. Use sql() method of HiveContext to create a sale Hive table and load Sale.txt into it 3. Use sql() method of HiveContext to create a join query to apply join on both these Hive tables



Consider a scenario where a leading retail client wants to run SQL analysis against their data stored in Cassandra and not on HDFS. Here, Spark SQL need to be integrated with Cassandra.

**Spark SQL on Cassandra**

Apache Spark does not provide a direct support for accessing Cassandra table data.

DataStax enterprise platform when configured on the Spark cluster provides provision to integrate Apache Spark with Cassandra.

Click :: <https://www.datastax.com/products/datastax-enterprise>

to learn more about installing and configuring DataStax Enterprise platform.

Once configured, Start the Spark shell,

As shown in the below code, **setKeySpace()** of Cassandra's Spark Context object (csc) is used to set the Cassandra Key Space. sql() of csc is used to run Spark SQL query analysis.

1. csc.setKeyspace("Cassandra\_keyspace\_name");
2. val results = csc.sql("SELECT \* from Cassandra\_keyspace\_name.Cassandra\_table");

What if the clients need to maintain data with-in a Hadoop database where Spark SQL analysis will run on top for OLAP, and other tools may run for OLTP real time read and write.

A leading Banking client maintains all its historical data in a HBase cluster. Client needs to store the data in HBase as real time read-write analysis to happen in parallel to OLAP (Online Analytical Processing) batch data analysis.

HBase is a distributed storage database solution which runs on top of HDFS.

The Client prefers to run Spark SQL for OLAP analysis against HBase for performance reasons. Let us understand how to configure Spark SQL on HBase.

**Spark SQL on HBase**

Spark-on-HBase connector is used to connect Spark with HBase directly.

It implements standard Spark Datasource API, and leverages Spark Catalyst engine for query optimizations.

Click :: <https://hortonworks.com/blog/spark-hbase-dataframe-based-hbase-connector/> to learn how to connect Spark SQL with HBase.

Till this point of time we had learnt Spark SQL architecture, and its integration with Spark Ecosystem components and NoSQL Databases.

Let us now discuss the other Spark SQL concepts like Dataset API, accessing Hive UDFs using Spark SQL etc.

Dataset is a programmatic component of Spark SQL introduced from version 1.6 onwards. Dataset can be used instead of Dataframes for Spark SQL analysis.

It supports lambda expressions and offers high performance.

It is a specialized Dataframe where elements map to specific JVM object type instead of Row class.

In Dataset, data is stored in the encoded form and a special encoder does the job of mapping Dataset with the schema.

**Interoperability**

A Dataset can be converted in to a DataFrame by calling ds.toDF().

A DataFrame can be converted in to a Dataset using df.as[Element Type] as shown below

1. case class ArisconData(name: String, age: Int)
2. val ds: Dataset[ArisconData] = df.as[ArisconData]

where ds is the dataset created from a DataFrame with the help of a type class ArisconData.

**How to use Dataset programmatically?**

Let us find out how to use Dataset programmatically through a quick retail example below

**Example 1**

A leading Retail client needs to compute a 20% discount value against all its product prices owing to a festive season. Price values are extracted and maintained in a separate list

**Input Data** : List(100,300,500,800,1000)

The retail client requires us to use Spark SQL Dataset API to implement the above requirement

**Code Solution**

1. //Dataset API offers toDS() method to convert a normal collection in to a Dataset
2. val price = List(100,300,500,800,1000).toDS();
3. //Computing the final price after a 20% discount for all items
4. val price\_after\_discount = price.map((\_\*80)/100)

A leading banking client provides loans to its customers and need to compute the average age of customers under every profession. Below is a sample dataset of Customer.txt file which need to be analyzed. The banking client requires us to implement the requirement using Dataset API of Spark SQL.

**Schema** - custid, firstname, lastname, age, profession



**Solution Path**

* Map Customer dataset with a Customer class
* Provide schema using Dataset API
* Load the customer file as a temporary view
* Analyse using Spark SQL queries against the View

**Code Solution**

1. val sqlContext = new org.apache.spark.sql.SQLContext(sc);
2. val Cust\_Info = sc.textFile("hdfs://volgalnx012:9000/jaihdfs/Customer.txt");
3. case class Customer(custid: String,firstname: String,lastname:String,age:Int,profession:String);
4. //Creating a Dataset using toDS() method
5. val DS = Cust\_Info.map(\_.split(",")).map(c => Customer(c(0),c(1),c(2),c(3).toInt,c(4)).toDS();
6. //Creating and registering the Dataset as a temporary view
7. DS.createOrReplaceTempView("customer");
8. //Spark SQL query to compute Profession, Average age of that profession
9. val pilotavg= sqlContext.sql("SELECT profession,AVG(age) FROM customer group by profession");
10. pilotavg.show();

### Sam works in a Spark SQL Big Data project and need to read the below file and convert in to a Dataset.

### case class Customer(Transid: String,Custid: String,Product:String,Price:Long); val Cust\_Info = sc.textFile("hdfs://volgalnx012:9000/jaihdfs/Bank\_Customer.txt"); val DS = Cust\_Info.map(\_.split(",")).map(c => Customer(c(0),c(1),c(2),c(3)).toDS();

**1. Dataset is a programmatic component introduced in 1.6**

**2. A Dataframe can be converted into a Dataset by calling ds.toDF() method**

So far we have seen how Spark SQL supports loading and analyzing text files using DataFrame/Dataset. Let us find out what are the other file formats supported by Spark SQL.

**Support for JSON**

Spark SQL supports loading JSON files into a Dataframe or Dataset, infer schema, runs Spark SQL queries against it as shown below

**Example code**

1. //Creation of SQLContext object
2. val sqlContext = new org.apache.spark.sql.SQLContext(sc);
3. //Spark SQL provides read.json() method to load JSON files as DataFrames directly
4. val cars = sqlContext.read.json("hdfs://vimsmys-42:9000/user/jai\_trng/jaihdfs/Cars.json");
5. //Schema of JSON files shall be printed using printSchema() method
6. cars.printSchema();
7. //Creation of a temporary table
8. cars.registerTempTable("carstable");
9. //Spark SQL query against carstable
10. val op = sqlContext.sql("SELECT \* from carstable");
11. //show() method to print output
12. op.show();

As shown above, **read.json()** is the API method used to load a JSON file into a Dataframe by inferring schema.

**Support for AVRO**

Spark SQL supports reading and writing Avro using the spark-avro library which includes **avro** methods **read.avro("inputdir")** and **write.avro("outputdir")**as shown below

**Example code**

1. import com.databricks.spark.avro.\_
2. //Creation of SQLContext object
3. val sqlContext = new org.apache.spark.sql.SQLContext(sc);
4. //Spark SQL provides read.avro() method to load Avro files as DataFrames directly
5. val cars = sqlContext.read.avro("hdfs://vimsmys-42:9000/user/jai\_trng/jaihdfs/Cars.avro");
6. //Schema of avro files shall be printed using printSchema() method
7. cars.printSchema();
8. //Creation of a temporary table
9. cars.registerTempTable("carstable");
10. //Spark SQL query against carstable
11. val op = sqlContext.sql("SELECT \* from carstable");
12. //show() method to print output
13. op.show();

**Benefits of Avro files**

The major benefit of Avro file is its performance in read and write, and optimized storage in compressed format. Hence, businesses prefer maintaining their datasets in AVRO format.

It uses JSON to define datatypes and protocols. JSON is the standard serialization component, across businesses for representing data. AVRO serialize data in a binary format with the help of JSON.

### Sara works on a Spark SQL project for a retail client. She need to load a JSON file representing the customer data and perform analysis on top of it.

### val sqlContext = new org.apache.spark.sql.SQLContext(sc); val customers = sqlContext.read.json("hdfs://vimsmys-42:9000/user/jai\_trng/jaihdfs/Customers.json");

### Sara works on a Spark SQL project for a banking client and need to load a AVRO file representing the sale data and perform analysis on top of it.

### val sqlContext = new org.apache.spark.sql.SQLContext(sc); val customers = sqlContext.read.avro("hdfs://vimsmys-42:9000/user/jai\_trng/jaihdfs/Customers.avro");

**What if a Hive UDF need to be accessed directly in Spark SQL for custom functionalities?**  
Spark SQL also supports accessing Hive UDFs directly with in its shell

Let us understand this through the below scenario:

A Retail client has a requirement to convert the values of price field in its dataset to floating type data. Hive UDF need to be created and accessed with in Spark SQL for this scenario.

Hive UDF creation is a part of our earlier course on Hive. To understand Hive UDF creation refer to the course [here](https://lex.infosysapps.com/viewer/lex_27350079123234040000).

Once the Hive UDF called **sparksql\_hiveudf.jar** is created let us understand how to register it and use it with-in Spark shell through below steps.

**Solution Path - Steps**

1. Open the spark-shell by registering the Spark SQL Hive udf jar using --jars sparksql\_hiveudf.jar

Spark-shell –jars sparksql\_hiveudf.jar

2. Create a Hive Context object and create a temporary function called convertToFloat as below

1. //Creation of HiveContext object
2. val sqlContext = new org.apache.spark.sql.hive.HiveContext(sc);
3. //Create a temporary function convertToFloat for the Hive UDF convert under package sparksql\_hiveudf
4. sqlContext.sql("""create temporary function convertToFloat as 'sparksql\_hiveudf.convert'""");

3. Apply the UDF's temporary function against the price field as shown below

1. //Using the Hive UDF convertToFloat()
2. sqlContext.sql("select id,name,convertToFloat(price) from empsparksql")

**Best Practices**

Ensure,

* To cache Spark SQL's DataFrame using **dataFrame.cache()** to improve performance. It uses in-memory columnar format to scan only required columns resulting in less memory usage
* To store any results in Parquet format which is an optimized standard serialization format for schema persistence. Parquet format enables quick read and write of DataFrames across applications

**Performance Tuning**

Performance of Spark SQL applications increase dramatically when below parameters are tuned

1. Number of partitions - Partitions refers to the logical split of the distributed dataset and number of tasks will be equal to number of partitions. Default paralleism is 12 which means 12 tasks would run against the splitted dataset by default. This value need to be either upgraded or downgraded to achieve performance benefit  
  
Parallelism can be controlled as shown below while entering into Spark shell:

1. spark-submit --conf spark.default.parallelism=23

2. Caching of parquet schema metadata can be turned on as this speeds up read/write of Parquet files.  
**Example** : spark.sql.parquet.cacheMetadata = "true"

### 3. Specify the respective compression algorithm while compressing the Parquet file. Below property sets the compression algorithm as gzip. needs to compress the Parquet file using gzip algorithm to enable optimized storage. Example : spark.sql.parquet.compression.codec = gzip

4. Set spark.sql.tungsten.enabled = true. When true, an optimized Tungsten physical execution backend explicitly manages memory and dynamically generates bytecode for expression evaluation.

Source :: [Spark SQL Programming Guide](https://spark.apache.org/docs/1.6.0/sql-programming-guide.html)