Note to Instructor: Fire up HDP 2.5 image

Module 02 – Spark SQL – Read/Write DataFrames/SQL Tables

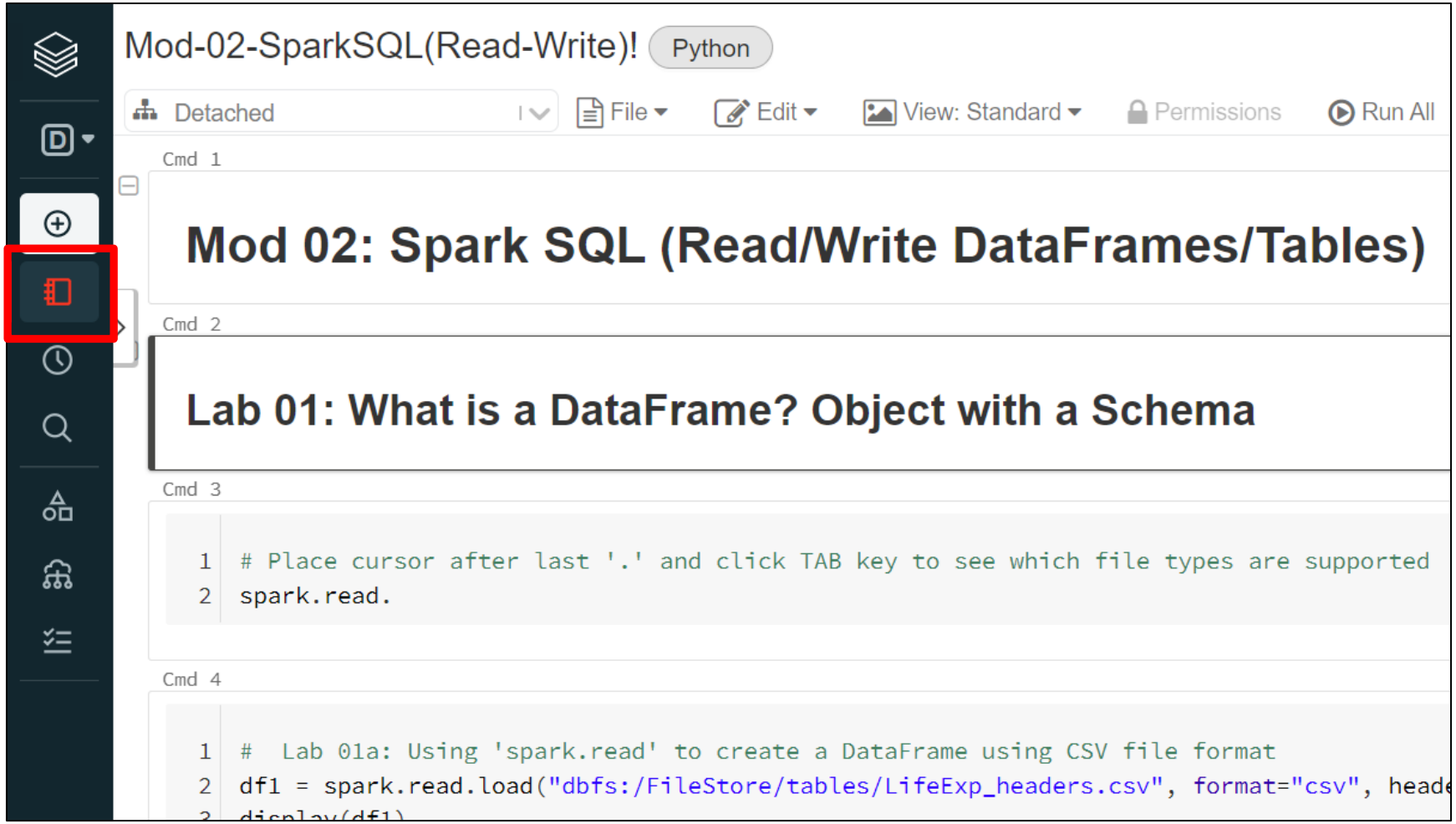
Copy

Table of Contents

Don't forget to start WebEx recording
Do Spark Jeopardy review

Go to: community.cloud.databricks.com and Logon
In Left-pane, Click on 'Clusters' or 'Compute' and Terminate old Cluster
Then click 'Create Cluster' button to create New one

Before we Begin: Open Notebook Mod-02



Module 02 – Spark SQL

After completing this module, you'll be able to work with Spark SQL including:

- What is Spark SQL?
  1. DataFrames
  2. Tables
  3. TempViews (Datasets, GlobalViews)

- The Catalyst Optimizer
- What is Hive?
- Creating DataFrames from five sources including:
  - Structured files, parallelize(), textFile(), Hive and mySQL
- CREATE TABLE function (Partition Tables and Bucket Tables)
- CreateOrReplaceTempView function and GlobalTempViews

Code displayed will be Python unless noted otherwise in Title bar

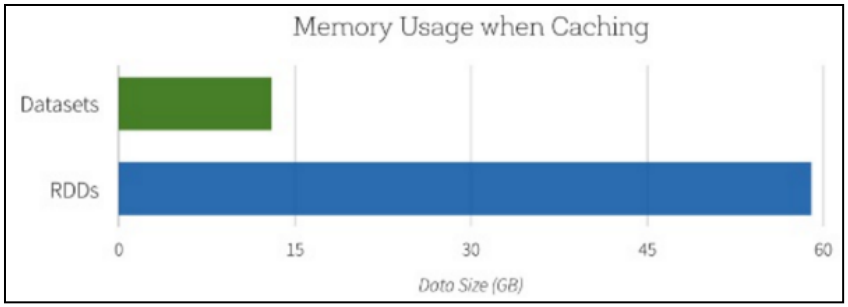
What is Apache Spark SQL?

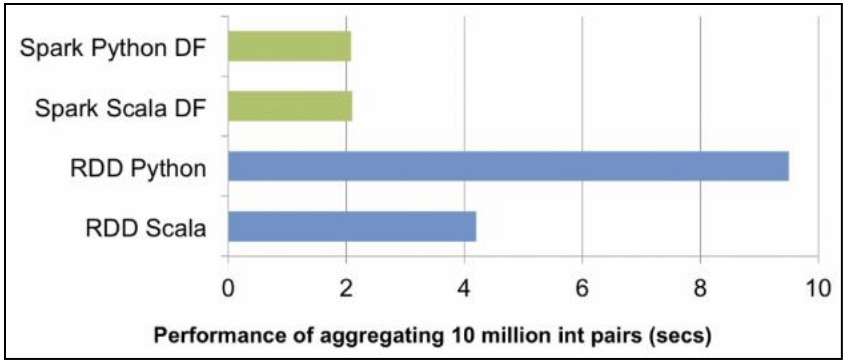
- Spark SQL brings native support for SQL to Spark and streamlines the process of querying data stored both in RDDs (and in external sources)
- Spark SQL conveniently blurs the lines between RDDs and relational tables. Unifying these powerful abstractions makes it easy for developers to intermix SQL commands querying external data with complex analytics, all within in a single application
- Concretely, Spark SQL will allow developers to:
  - Import relational data from Parquet files and Hive tables
  - Run SQL queries over imported data and existing RDDs
  - Easily write RDDs out to Hive tables or Parquet files
- Spark SQL also includes a Cost-based optimizer (Catalyst), columnar storage, and code generation to make queries fast

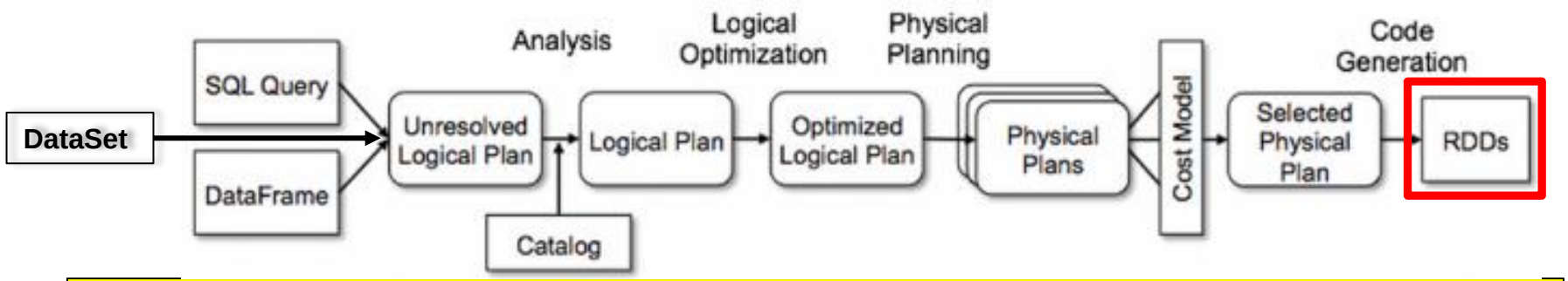
Bottom line: Spark SQL refers to both DataFrames and Tables/Views.
And there is a 3rd object called DataSets (Scala only)

Spark SQL uses Catalyst Optimizer/Tungsten

- Spark SQL uses the knowledge of the data types to more efficiently represent the data. For example, when caching data, it uses an in-memory columnar storage. In addition, instead of reading the entire data like the MapReduce engine normally does, Spark SQL can prune the data resulting in less Disk I/O
- Spark SQL uses the **Catalyst** optimizer which contains a general library for representing trees and applying rules to manipulate them. It can perform predicate push-down to optimize the query. In addition, Catalyst can efficiently serialize/deserialize JVM object using **Tungsten** Encoders that creates compact bytecode that can execute at superior speeds







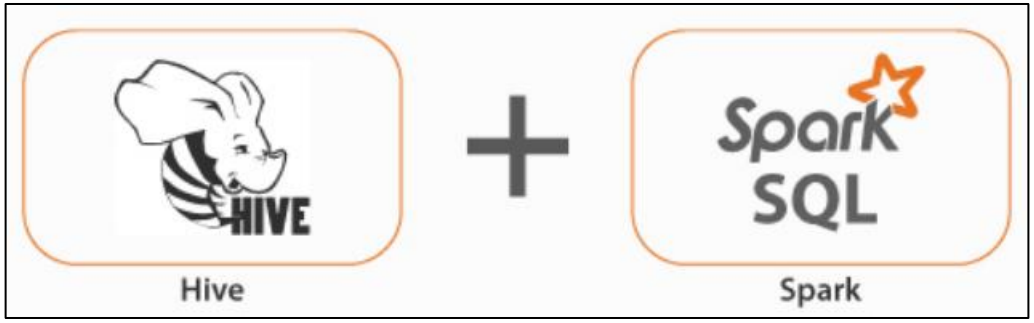
Also get push-down operations to remote data store (predicate pushdown, Index, etc)

What is Apache Hive?

http://hive.apache.org

- A Data warehousing infrastructure based on Hadoop
- Hive QL, which is an SQL-like language, enables users familiar with SQL to do ad-hoc querying, summarization and data analysis easily
- At the same time, Hive QL also allows traditional map/reduce programmers to be able to plug in their custom mappers and reducers to do more sophisticated analysis that may not be supported by the built-in capabilities of the language
- Invented at Facebook. Open sourced to Apache in 2008

Hive integration with Spark



- Spark SQL can read and write data stored in Apache Hive
- Hive has certain drawbacks. Initially, due to MapReduce jobs underneath, this process is slow. Secondly, it is only suitable for batch processing, and not for interactive queries or iterative jobs
- Spark SQL, on the other hand, addresses these issues remarkably well. We can directly access Hive tables on Spark SQL and use SQLContext queries or DataFrame APIs to work on those tables. The process is fast and highly efficient compared to Hive

Spark SQL and the three APIs

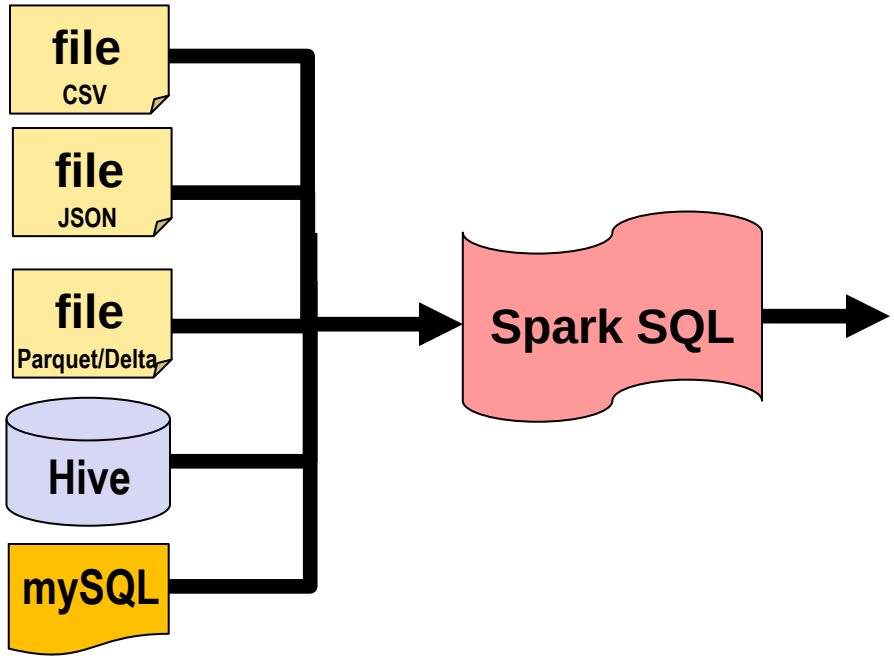
- Spark SQL is a Spark module for structured data processing. Unlike the basic Spark RDD API, the interfaces provided by Spark SQL give you more information about the structure of both the data and the computation being performed
- Internally, Spark SQL uses this extra information to perform additional optimizations. There are several ways to interact with Spark SQL including:
  - **DataFrame API** - via toDF() and createDataFrame() function
  - **Hive API** - via spark.table(), Spark can integrate into Hive and create Hive tables from Spark DataFrame. Once created, users can access tables globally (don't need to login to Spark)
  - **Spark SQL API** - via CREATE TABLE and createOrReplaceTempView

Bottom line: Spark SQL refers to both DataFrames and Tables/Views.
And there is a 3rd object called DataSets (Scala only)

Spark SQL - Big Picture

Parquet { JSON } amazon web services S3 openstack HIVE MySQL PostgreSQL HDFS and more ...

Similar in many aspects to Apache Hive, Spark SQL allows you to query using two API versions (Data Frame and SQL)



Dataframe (df)

name	gender	age	wt	eye
Juli	female	59	125	green
Mark	male	58	182	brown
Holly	female	25	130	hazel
Andrew	male	28	185	brown

01: DataFrame Reader (spark.read)

- SparkSQL provides for a special type of RDD called DataFrame (note in some documentation it is referred to as SchemaRDD)
- A DataFrame is an immutable RDD of Row objects which means it knows the Column names. It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations (ie: can use Catalyst optimizer)
- DataFrames can be constructed from a wide array of sources such as: structured data files (JSON, Parquet), existing RDDs, or Hive tables using 'spark.read' method

Column names (Explicit or Inferred)

```
3 df1 = spark.read.load("dbfs:/FileStore/tables/LifeExp_headers.csv",
4                       format="csv", header="True", inferSchema="True")
5 display(df1)
```

(3) Spark Jobs

df1: pyspark.sql.dataframe.DataFrame = [country: string, lifeexp: double ... 1 more fields

	country	lifeexp	region
1	Afghanistan	48.673	SAs
2	Albania	76.918	EuCA
3	Algeria	73.131	MENA
4	Angola	51.093	SSA

```
1 df1 = spark.read.load("dbfs:/FileStore/tables/LifeExp.csv",
2                       format="csv", inferSchema="True")
3 display(df1)
```

(3) Spark Jobs

df1: pyspark.sql.dataframe.DataFrame = [_c0: string, _c1: double ... 1 more fie

	_c0	_c1	_c2
1	Afghanistan	48.673	SAs
2	Albania	76.918	EuCA
3	Algeria	73.131	MENA
4	Angola	51.093	SSA

02a/b: Create DataFrame from Structured JSON file using spark.read()

file
JSON, ORC, parquet

- Here is the multi-structured JSON file we will first load into Spark

names1.json

```
{"name":"Juli","gender":"female","age":59,"wt":125,"eye":"green"}
{"name":"Mark","gender":"male","age":58,"wt":185,"eye":"brown"}
{"name":"Holly","gender":"female","age":25,"wt":135,"eye":"hazel"}
{"name":"Drew","gender":"male","age":29,"wt":180,"eye":"brown"}
```

- First, we read in DataFrame from an existing JSON file using the spark API
- Once loaded, we can use the DataFrame API to query via the show() function

```
df1 = spark.read.format("json").load("/user/zeppelin/names1.json")
df1.show()

+---+-----+------+-----+---+
|age|  eye|gender| name| wt|
+---+-----+------+-----+---+
| 59|green|female| Juli|125|
| 58|brown|  male| Mark|185|
| 25|hazel|female|Holly|135|
| 29|brown|  male| Drew|180|
+---+-----+------+-----+---+
```

- df1.printSchema() displays Schema in tree format →

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

Spark 3.x Scala syntax: **val df = spark.read.json("/user/zeppelin/names1.json").show()**

02c: DataFrame syntax – Querying using show()

- Depending on which language you are using, there are numerous ways to reference columns in DataFrames using the DataFrame API
- Here's some examples of querying a DataFrame named 'df1' using Python along with select argument

```
df1.select("name").show()
df1.select(df1.name).show()
df1.select(df1["name"], df1["age"]).show()
df1[df1.age<50].show()
```

```
+-----+
| name|
+-----+
| Juli|
| Mark|
|Holly|
| Drew|
+-----+
```

```
+-----+
| name|
+-----+
| Juli|
| Mark|
|Holly|
| Drew|
+-----+
```

```
+-----+---+
| name|age|
+-----+---+
| Juli| 59|
| Mark| 58|
|Holly| 25|
| Drew| 29|
+-----+---+
```

```
+---+-----+------+-----+---+
|age|  eye|gender| name| wt|
+---+-----+------+-----+---+
| 25|hazel|female|Holly|135|
| 29|brown|  male| Drew|180|
+---+-----+------+-----+---+
```

03: Read Parquet files using spark.read

file
JSON, ORC, parquet

Here we create 2 DataFrames from existing parquet files

Unlike CSV, Parquet files have Metadata that stores Schema automatically

```
1  %py
2
3  # Load Parquet files and show()
4
5  empDF = spark.read.format("parquet").load("dbfs:/FileStore/tables/parq_emp/")
6  deptDF = spark.read.format("parquet").load("dbfs:/FileStore/tables/parq_dept/")
7
8  empDF.show()
9  deptDF.show()
```

▸ (4) Spark Jobs
▸ empDF: pyspark.sql.dataframe.DataFrame = [emp: integer, mgr: integer ... 7 more fields]
▸ deptDF: pyspark.sql.dataframe.DataFrame = [dept: string, dept_name: string ... 2 more fields]

```
+----+----+----+------+---------+----------+----------+----------+---------+
| emp| mgr|dept|   job|last_name|first_name|      hire|     birth|   salary|
+----+----+----+------+---------+----------+----------+----------+---------+
|1018|1017| 501|512101| Ratzlaff|     Larry|1978-07-15|1954-05-31| 54000.00|
|1016| 801| 302|321100|   Rogers|      Nora|1978-03-01|1959-09-04| 56500.00|
|1014|1011| 402|422101|    Crane|    Robert|1978-01-15|1960-07-04| 24500.00|
```

04a: spark.read.load with StructType Library and format and schema arguments

If don't have structured file (like JSON, Parquet, ORC, etc) can create Schema using StructType and apply it to the unstructured file type (like CSV)

```
%py

## Import library so can create Schema
from pyspark.sql.types import StructType, StructField, IntegerType, StringType, FloatType

## Create schema to be used in creation of DataFrame
lifeSchema = StructType([StructField('Country',StringType(), True), \
                         StructField('LifeExp', FloatType(), True), StructField('Region', StringType(), True) ])

df1 = spark.read.load("dbfs:/FileStore/tables/LifeExp.csv", format="csv", schema=lifeSchema)

df1.show()
df1.printSchema()
```

▸ (1) Spark Jobs

▸ df1: pyspark.sql.dataframe.DataFrame = [Country: string, LifeExp: float ... 1 more fields]

```
+-----------+-------+------+
|    Country|LifeExp|Region|
+-----------+-------+------+
|Afghanistan| 48.673|   SAs|
|    Albania| 76.918|  EuCA|
|    Algeria| 73.131|  MENA|
|     Angola| 51.093|   SSA|
```

```
root
 |-- Country: string (nullable = true)
 |-- LifeExp: float (nullable = true)
 |-- Region: string (nullable = true)
```

04b: spark.read with Schema hard-coded sans StructType

With Spark 3.x, can forgo 'StructType' library when creating your Schema

```
1  # With Spark version 3, don't even have to mess around with 'StructType' anymore!!  Just do
2  DDLSchema = "Country string, LifeExp float, Region string"
3
4  df1 = spark.read.format("csv").load("dbfs:/FileStore/tables/LifeExp.csv", schema=DDLSchema)
5
6  display(df1)
7  df1.printSchema()
```

▸ (1) Spark Jobs

```
root
 |-- Country: string (nullable = true)
 |-- LifeExp: float (nullable = true)
 |-- Region: string (nullable = true)
```

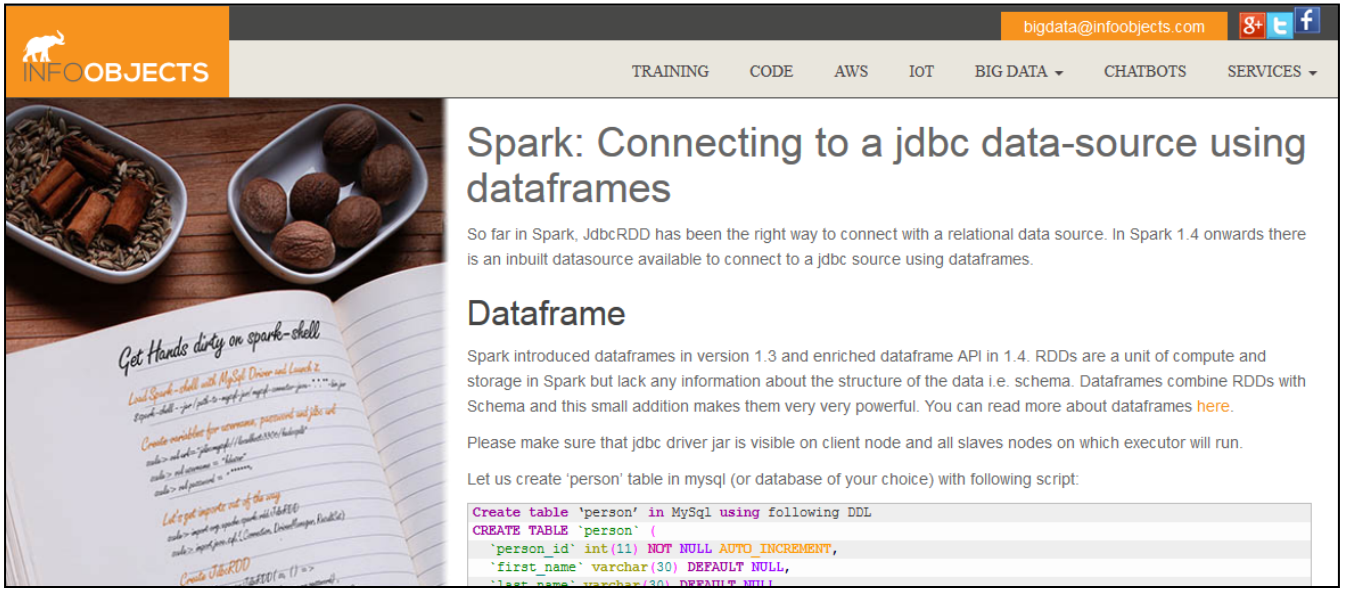
	Country	LifeExp	Region
1	Afghanistan	48.673	SAs
2	Albania	76.918	EuCA
3	Algeria	73.131	MENA

Movie: Create DF from RDBMS table using jdbc Connector to remote database

3

- To connect Spark to a relational database, jdbc is the way to go
  Open up 'mysql-conn.txt' file from your Desktop and follow the labs. You will

1. From cmd prompt, logon to mySQL and create table 'person' in 'test' database
2. Logon to Spark using the mySQL connector via Scala
3. Configure JDBC URL and mySQL properties so can connect to mySQL
4. Load mySQL table into Spark and run multiple queries
5. Save answer set into a new mySQL table
6. Log back into mySQL and confirm table exists



http://www.infoobjects.com/spark-connecting-to-a-jdbc-data-source-using-dataframes/

Lab 05: Create DF from RDBMS (mysql) (Scala)

Use HDP_2_5 image

```
// Open new Terminal. As Zeppelin user: ssh spark. Then point Spark shell to mySQL driver
spark-shell --driver-class-path /opt/mysql-connector-java-5.1.39-bin.jar

// Construct JDBC URL that  oints to mySQL database named 'test'
val url = "jdbc:mysql://172.17.0.2:3306/test"

// Create connection properties with username/password
val prop = new java.util.Properties
prop.setProperty("user", "mysql")
prop.setProperty("password", "")

// Load mySQL table 'person' into Spark
val people =       spark.read.jdbc(url, "person", prop)

// Display results
people.show()

// More queries
val below60 = people.filter(people("age") < 60)
below60.show()

val grouped = people.groupBy("gender")
val gender_count = grouped.count()
gender_count.show()

val avg_age = grouped.avg("age")
avg_age.show()

// Write table to mySQL table
gender_count.write.jdbc(url, "gender_count", prop)
```

```
+---------+----------+---------+------+---+
|person_id|first_name|last_name|gender|age|
+---------+----------+---------+------+---+
|        1|    Barack|    Obama|     M| 53|
|        2|      Bill|  Clinton|     M| 71|
|        3|   Hillary|  Clinton|     F| 68|
|        4|      Bill|    Gates|     M| 69|
|        5|  Michelle|    Obama|     F| 51|
+---------+----------+---------+------+---+
```

06: DataFrame Writer (write)

- Used to write a DataFrame to external storage
- Supports multiple file formats (CSV, Parquet, Delta, etc)
- Can even write as an SQL Table too

```
(df.write
  .option("compression", "snappy")
  .mode("overwrite")
  .parquet(outPath))
```

'Mode' include:
- **Append**
- **Overwrite**
- **ErrorIfExists**
- **Ignore**

Instead of writing to file format, can write as an SQL table too

```
eventsDF.write.mode("overwrite").saveAsTable("events_p")
```

Alternative coding:

```
usersDF.write.format("csv").mode("append").save(usersOutputPath2)
usersDF.write.format("delta").save(usersOutputPath3)
```

07: Saving DF as file using write() (1 of 2)

- write() function allows you to export the Data Frame into a file or external storage systems (ie: Cassandra, HDFS, etc.)
- There are several file formats available (ie: JSON, ORC, parquet, etc.)

```
df =      spark.table("emp")

df.write.format("json").save("/user/zeppelin/json7/")
df.write.format("orc").save("/user/zeppelin/orc7/")
# // If FORMAT not defined, defaults to Parquet
df.select("l_name", "dept").write.save("/user/zeppelin/parquet6")
df.write.format("parquet").save("/user/zeppelin/parquet7/")
df.write.format("parquet").save("/user/zeppelin/parquet7/", mode = "overwrite")
```

Took 11 sec. Last updated by anonymous at January 19 2017, 12:06:15 PM.

Parquet Files

- Apache project – format available to all Hadoop ecosystem projects
- Supports very efficient compression and encoding schemes
- Designed to work well on HDFS
- Encodes nested structures and sparsely populated data
- Is the default file type for Spark reading
- See http://parquet.apache.org/

25b: View HDFS files

```
%sh
hdfs dfs -ls /user/zeppelin/json7/
hdfs dfs -ls /user/zeppelin/parquet7/
```

```
Found 3 items
-rw-r--r--   1 zeppelin hdfs          0 2017-01-19 17:06 /user/zeppelin/json7/_SUCCESS
-rw-r--r--   1 zeppelin hdfs       1974 2017-01-19 17:06 /user/zeppelin/json7/part-r-00000-94bababb2-aafd-4f7d-9303-1bd3813fbe2b
-rw-r--r--   1 zeppelin hdfs       1703 2017-01-19 17:06 /user/zeppelin/json7/part-r-00001-94bababb2-aafd-4f7d-9303-1bd3813fbe2b
Found 5 items
-rw-r--r--   1 zeppelin hdfs          0 2017-01-19 17:06 /user/zeppelin/parquet7/_SUCCESS
-rw-r--r--   1 zeppelin hdfs        817 2017-01-19 17:06 /user/zeppelin/parquet7/_common_metadata
-rw-r--r--   1 zeppelin hdfs       2896 2017-01-19 17:06 /user/zeppelin/parquet7/_metadata
-rw-r--r--   1 zeppelin hdfs       2513 2017-01-19 17:06 /user/zeppelin/parquet7/part-r-00000-ed025126-b039-4f35-93d1-928a76746391.gz.parquet
-rw-r--r--   1 zeppelin hdfs       2484 2017-01-19 17:06 /user/zeppelin/parquet7/part-r-00001-ed025126-b039-4f35-93d1-928a76746391.gz.parquet
```

https://spark.apache.org/docs/1.6.1/sql-programming-guide.html

Saving DF as file using write() (2 of 2)

Save operations can optionally take a **SaveMode**, that specifies how to handle existing data if present. It is important to realize that these save modes do not utilize any locking and are not atomic. Additionally, when performing a **Overwrite**, the data will be deleted before writing out the new data.

Scala/Java	Any Language	Meaning
mode = ErrorIfExists(default)	**"error"(default)**	When saving a DataFrame to a data source, if data already exists, an exception is expected to be thrown.
mode = append	**"append"**	When saving a DataFrame to a data source, if data/table already exists, contents of the DataFrame are expected to be appended to existing data.
mode = overwrite	**"overwrite"**	Overwrite mode means that when saving a DataFrame to a data source, if data/table already exists, existing data is expected to be overwritten by the contents of the DataFrame.
mode = ignore	**"ignore"**	Ignore mode means that when saving a DataFrame to a data source, if data already exists, the save operation is expected to not save the contents of the DataFrame and to not change the existing data. This is similar to a CREATE TABLE IF NOT EXISTS in SQL.

08: Show Tables
(Displays Perm Tables and Temp View)

'SHOW TABLES' display any existing Tables you have created via the Spark 'CREATE TABLE' statement

```
%sql

-- May be empty if have yet to run 'CREATE TABLE' statement
-- 'isTemporary' = True earmarks any VIEWS you may have created via 'CreateOrReplaceTempView' statement
show tables;
```

	database	tableName	isTemporary
4	default	dept	false
5	default	diamonds	false
6	default	employees	false
7	default	flights_abbr	false
8	default	fly1	false
9	default	mpg	false
10		dept_view	true

Permanent Hive tables have 'is Temporary' = false
Temporary Views have 'is Temporary' = true

Create Table for Spark SQL: Generic Syntax

```
CREATE TABLE [ IF NOT EXISTS ] table_identifier
    [ ( col_name1 col_type1 [ COMMENT col_comment1 ], ... ) ]
    USING data_source
    [ OPTIONS ( key1=val1, key2=val2, ... ) ]
    [ PARTITIONED BY ( col_name1, col_name2, ... ) ]
    [ CLUSTERED BY ( col_name3, col_name4, ... )
        [ SORTED BY ( col_name [ ASC | DESC ], ... ) ]
        INTO num_buckets BUCKETS ]
    [ LOCATION path ]
    [ COMMENT table_comment ]
    [ TBLPROPERTIES ( key1=val1, key2=val2, ... ) ]
    [ AS select_statement ]
```

All 'CREATE TABLE' are stored as Hive tables. This is because Spark is a Processing Engine only and requires a 3rd Party application like Apache Hive to store Table objects

09: Create Table: USING / OPTIONS with header argument

```
1  %sql
2
3  # Lab 12a: DROP, then CREATE TABLE
4
5  DROP TABLE IF EXISTS mpg;
6
7  CREATE TABLE mpg
8  USING csv
9  OPTIONS (path "/databricks-datasets/Rdatasets/data-001/csv/ggplot2/mpg.csv", header "true")
```

Define file format here

Defined Column names in first row of file

```
5  SELECT * FROM mpg
```

▸ (1) Spark Jobs

	_c0	manufacturer	model	displ	year	cyl	trans
1	99	ford	mustang	5.4	2008	8	manual(m6)
2	98	ford	mustang	4.6	2008	8	auto(l5)
3	97	ford	mustang	4.6	2008	8	manual(m5)
4	96	ford	mustang	4.6	1999	8	manual(m5)

USING data_source: TEXT, CSV, JSON, JDBC, PARQUET, ORC, HIVE, DELTA, LIBSVM

OPTIONS: Point to Directory or file location along with other parameters

10: View Table details

Once you create a Table, can View it in UI via the left vertical pane.

Click 'Data', then select 'default' > 'mpg'

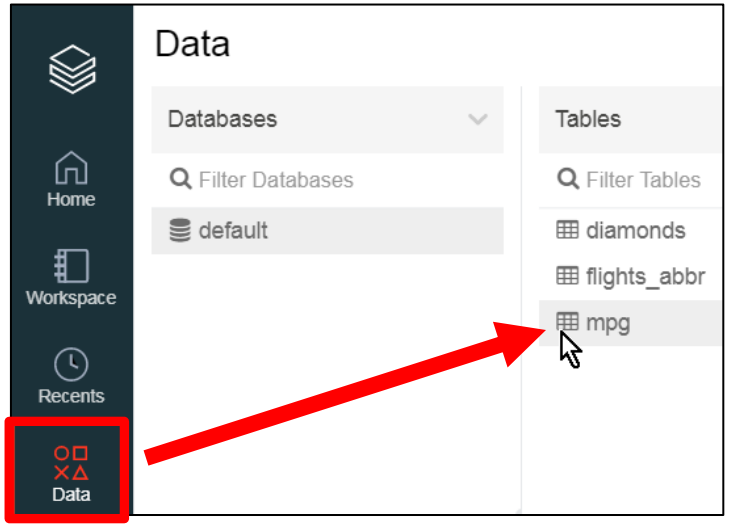


Table: mpg

Schema:

	col_name	data_type	comment
1	_c0	string	null
2	manufacturer	string	null
3	model	string	null
4	displ	string	null
5	year	string	null
6	cyl	string	null
7	trans	string	null
8	drv	string	null

Showing all 12 rows.

Sample Data:

	_c0	manufacturer	model	displ
1	1	audi	a4	1.8
2	2	audi	a4	1.8
3	3	audi	a4	2
4	4	audi	a4	2
5	5	audi	a4	2.8
6	6	audi	a4	2.8
7	7	audi	a4	3.1
8	8	audi	a4 quattro	1.8

11a-b: Create Table without Header info by creating Column names/data types manually

```
5  CREATE TABLE dept (dept_num INT, dept_name STRING, budget INT, mgr INT)
6  USING csv
7  OPTIONS (path "dbfs:/FileStore/tables/dept.csv")
8
9  -- After you create, confirm can see TABLE in 'DATA' icon UI
```

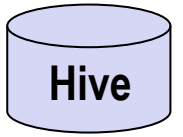
Cmd 27

```
1  %sql
2
3  SELECT * FROM dept
```

▸ (1) Spark Jobs

	dept_num	dept_name	budget	mgr
1	301	research and development	null	1019
2	501	marketing sales	null	1017
3	100	president	null	801
4	302	product planning	null	1016
5	402	software support	null	1011

11c: Create DF from (Hive) Table in prior Lab



- Using spark API, you can convert a Hive table to a DataFrame via table() function
- And since Hive table has pre-existing schema, don't need Row() or StructType()

```
5 df = spark.table("emp")
6
7 df.printSchema()
8 df.show()
```

▸ (1) Spark Jobs

▸ df: pyspark.sql.dataframe.DataFrame = [emp: integer, mgr: integer ... 6 more fields]

```
root
 |-- emp: integer (nullable = true)
 |-- mgr: integer (nullable = true)
 |-- job: integer (nullable = true)
 |-- l_name: string (nullable = true)
 |-- f_name: string (nullable = true)
 |-- hire: string (nullable = true)
 |-- birth: string (nullable = true)
 |-- salary: float (nullable = true)

+----+----+---+------+---------+-------+----------+------+
| emp| mgr|job|l_name|   f_name|   hire|     birth|salary|
+----+----+---+------+---------+-------+----------+------+
|1018|1017|501|512101| Ratzlaff|  Larry|1978-07-15|  null|
|1016| 801|302|321100|   Rogers|   Nora|1978-03-01|  null|
|1014|1011|402|422101|    Crane| Robert|1978-01-15|  null|
```

It is important to note you can query a Hive table directly from Spark client without the need to create a Dataframe as shown here

```
5 SELECT emp, l_name, f_name FROM emp LIMIT 5;
```

▸ (1) Spark Jobs

	emp	l_name	f_name
1	1018	512101	Ratzlaff
2	1016	321100	Rogers
3	1014	422101	Crane
4	1004	412101	Johnson
5	1002	413201	Brown

12a-d: Create Table: PARTITION BY

Partitioning is a Performance enhancement technique which can Dynamical read only Partition files when those Partition columns are in the WHERE clause

```
CREATE TABLE cust_part
(id INT, name STRING)
PARTITIONED BY (state STRING, city STRING);

INSERT INTO cust_part PARTITION (state = 'CA', city = 'Fremont') VALUES (100, 'Al');
INSERT INTO cust_part PARTITION (state = 'CA', city = 'San Jose') VALUES (200, 'Bo');
INSERT INTO cust_part PARTITION (state = 'AZ', city = 'Peoria') VALUES (300, 'Cy');
```

```
SELECT * from cust_part;
```

	id	name	state	city
1	300	Cy	AZ	Peoria
2	100	Al	CA	Fremont
3	200	Bo	CA	San Jose

```
SELECT * from cust_part WHERE state = 'CA'
```

Only had to read 1 Partition

```
CatalogPartition( Partition Values: [state=CA, city=San Jose] Location:
```

12e-g: Create Table: PARTITION BY

By creating a Table with Partitions, if the query has the Partitioning column in the WHERE clause, can reduce Disk I/O by reading only the file(s) pertaining to your query

Below you can see there were 2

```
1  # Before we begin, confirm have 'parq_emp' and 'parq_dept' folders
2  display(dbutils.fs.ls("dbfs:/user/hive/warehouse/cust_part/state=CA"))
```

▸ (3) Spark Jobs

	path	name	size
1	dbfs:/user/hive/warehouse/cust_part/state=CA/city=Fremont/	city=Fremont/	0
2	dbfs:/user/hive/warehouse/cust_part/state=CA/city=San Jose/	city=San Jose/	0

13a-d: CREATE TABLE: CLUSTERED BY / SORTED BY INTO NUM_BUCKETS and COMMENT

- Bucketing a table means that during a load, the Bucket column value is hashed and that determines which Bucket it will be stored
- Physically each Bucket is a file in the table directory
- Advantages include:
  - Like Partitioning, Bucketed tables provide faster query responses
  - 2 bucketed tables joined together is efficient if Bucket columns are Join columns since these rows guaranteed to be in the same Bucket
  - If include SORTED BY, Map-side joins are even more efficient since SORT is already done for Merge join
  - Bucket tables can also be an efficient way of Sampling

```
CREATE TABLE bucket_table (state STRING, population INTEGER,   yr INTEGER,)
      USING CSV
      COMMENT 'A bucketed sorted user table'
      CLUSTERED BY (state) SORTED BY (state) INTO 25 BUCKETS

INSERT OVERWRITE bucket_table VALUES ("Ohio",11664129,2017),
                                     ("Oklahoma",3932640,2017), …
```

STATE is the hash column

13-a/d: View Buckets read

Only had to read 1 out of the 25 buckets

```
1  %sql
2  EXPLAIN SELECT * FROM bucket_table WHERE state = 'Ohio'
```

	plan
	InMemoryFileIndex[dbfs:/user/hive/warehouse/bucket_table], PartitionFilters: [], PushedFilters: [IsNotNull(state), EqualTo(state,Ohio)], ReadSchema: struct<state:string,population:int,yr:int> SelectedBucketsCount: 1 out of 25

PARTITION versus BUCKET

- Although you can enact both Partitioning and Bucketing on the same table, each has their own advantages
- Partitioning gives effective results when:
  - There are a limited number of partitions
  - Comparatively equal sized partitions
- For example, if Partitioning by 'Country population', a few partitions will have large number of rows (China, India) while others will have small number of rows (Vatican, Monaco). In cases like this, Bucketing can be more effective since you can minimize Skew

14a-c: Create Table: TBLPROPERTIES

A List of Key-Value pairs that is used to tag the table definition

```
1  %sql
2  CREATE TABLE customer1(cust_code INT, name VARCHAR(100), cust_addr STRING)
3   TBLPROPERTIES ('created.by.user' = 'Mark', 'created.date' = '01-01-2021');
```

OK

Command took 0.90 seconds -- by ottmk@ucmail.uc.edu at 11/3/2020, 7:52:49 PM on quickstart

Cmd 35

```
1  %sql
2  SHOW TBLPROPERTIES customer1
```

	key	value
1	created.by.user	Mark
2	created.date	01-01-2021
3	transient_lastDdlTime	1604451170

15a/d: Create TempView from DataFrame via createOrReplaceTempView()

You can convert a DataFrame into a Spark Temp table via createOrReplaceTempView. Once done, you can now query via Spark SQL API

```
3  # Lab 19a: Create DataFrame from Parquet files and show()
4
5  deptDF = spark.read.format("parquet").load("dbfs:/FileStore/tables/parq_dept/")
6
7  deptDF.createOrReplaceTempView("dept_view")
```

```
3  -- Lab 19b: Query Temp View
4
5  SELECT * FROM dept_view;
```

▸ (1) Spark Jobs

	dept	dept_name	budget	mgr
1	301	research and development	465600.00	1019
2	501	marketing sales	308000.00	1017
3	100	president	400000.00	801

```
3  -- Lab 19c:  show tables
4
5  show tables;
```

	database	tableName	isTemporary
5	default	diamonds	false
6	default	emp	false
7	default	employees	false
8	default	flights_abbr	false
9	default	fly1	false
10	default	mpg	false
11		dept_view	true

Can convert a Temp View into a Perm Table via the 'CREATE TABLE AS' syntax

15e: Unlike TempViews, (Perm) Tables, can survive being queried in other Contexts

```
1  %py
2
3  # Lab 20b: Using 'py' Intrepreter
4  df = spark.table("emp")
5  df.show()
```

PYTHON() API

▸ (1) Spark Jobs

▸ df: pyspark.sql.dataframe.DataFrame = [emp: integer, mgr: i

```
+----+----+---+------+---------+-------+----------+
| emp| mgr|job|l_name|   f_name|   hire|     birth|
+----+----+---+------+---------+-------+----------+
|1018|1017|501|512101| Ratzlaff|  Larry|1978-07-15|
|1016| 801|302|321100|   Rogers|   Nora|1978-03-01|
```

- TempTables are materialized only in the Context they were created
- By making Permanent table, can now use in other Contexts as shown below
- Regardless of which API you use, rest assured you are using the Catalyst optimizer

TempViews are transient. If you lose your Cluster, all TempViews are de-materialized

In-line lab: Remove Cluster, create Cluster, then confirm TempView is removed from 'show tables'

```
1  %sql
2
3  -- Lab 20a:  Using 'sql' Intrepreter
4  SELECT * FROM emp;
```

SQL API

▸ (1) Spark Jobs

	emp	mgr	job	l_name	f_name	hire	birth
1	1018	1017	501	512101	Ratzlaff	Larry	1978-07-15
2	1016	801	302	321100	Rogers	Nora	1978-03-01

```
0: jdbc:hive2://localhost:10000> SELECT * FROM EMP;
+----------+----------+-----------+----------+----------------+-----------------+------------+------------+
| emp.emp  | emp.mgr  | emp.dept  | emp.job  | emp.last_name  | emp.first_name  |  emp.hire  |  emp.birth |
+----------+----------+-----------+----------+----------------+-----------------+------------+------------+
| 1018     | 1017     | 501       | 512101   | Ratzlaff       | Larry           | 1978-07-15 | 1954-05-31 |
| 1016     | 801      | 302       | 321100   | Rogers         | Nora            | 1978-03-01 | 1959-09-04 |
| 1014     | 1011     | 402       | 422101   | Crane          | Robert          | 1978-01-15 | 1960-07-04 |
```

Hive context

16a: One more Thing: Global Temporary Views

- Global Temporary Views are introduced in Spark 2.1.0 release. This feature is useful when you want to share data among different sessions and keep alive until your application ends. In Spark SQL, temporary views are session-scoped and will be automatically dropped if the session terminates.

- All the global temporary views are tied to a system preserved temporary database ' global_temp'

```
3  df = spark.read.json("dbfs:/FileStore/tables/names1.json")
4  df.createGlobalTempView("people")
5
6  # Global temporary view is tied to a system preserved database `global_temp`
7  spark.sql("SELECT * FROM global_temp.people").show()
8
9  # Global temporary view successful in another Session
10 spark.newSession().sql("SELECT * FROM global_temp.people").show()
```

```
+---+-----+------+-----+---+
|age|  eye|gender| name| wt|
+---+-----+------+-----+---+
| 59|green|female| Juli|125|
| 58|brown|  male| Mark|185|
| 25|hazel|female|Holly|135|
| 29|brown|  male| Drew|180|
+---+-----+------+-----+---+
```

16b: Global Temporary Views vs Temporary Views

Notice how a Temporary View fails when execute using a different Session

Spark SQL statements (Tables, Views)

DDL

- ALTER DATABASE
- ALTER TABLE
- ALTER VIEW
- CREATE DATABASE
- CREATE FUNCTION
- CREATE TABLE
- CREATE VIEW
- DROP DATABASE
- DROP FUNCTION
- DROP TABLE
- DROP VIEW
- REPAIR TABLE
- TRUNCATE TABLE
- USE DATABASE

DML

- INSERT
- INSERT INTO
- INSERT OVERWRITE DIRECTORY
- INSERT OVERWRITE DIRECTORY with Hive format
- INSERT OVERWRITE
- LOAD DATA

Retrieval

- SELECT
- EXPLAIN

Describe

- DESCRIBE DATABASE
- DESCRIBE FUNCTION
- DESCRIBE QUERY
- DESCRIBE TABLE

Show

- SHOW COLUMNS
- SHOW CREATE TABLE
- SHOW DATABASES
- SHOW FUNCTIONS
- SHOW PARTITIONS
- SHOW TABLE EXTENDED
- SHOW TABLES
- SHOW TBLPROPERTIES
- SHOW VIEWS

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

In Review: Spark SQL

After completing this module, you'll be able to work with Spark SQL including:

- What is Spark SQL?
  1. DataFrames
  2. Tables
  3. TempViews (Datasets, GlobalViews)
- The Catalyst Optimizer
- What is Hive?
- Creating DataFrames from five sources including:
  - Structured files, parallelize(), textFile(), Hive and mySQL
- CREATE TABLE function (Partition Tables and Bucket Tables)
- CreateOrReplaceTempView function and GlobalTempViews