Spark Notes

bin/spark-shell —master yarn (to start spark-shell with yarn)
bin/spark-shell —master local (to start spark-shell in local mode)
bin/spark-shell —master spark://u10:7077 (to start spark-shell with a standalone spark cluster if spark standalone cluster)

sparkContext is the entry point of application
sparkSession is entry point of application which is tied to pyspark

The scala shell is REPL (Read Evaluate Print Loop)
Even though it appears as interpreter, all typed code is converted to Byte Code and executed)

Scala >

Spark.<tab>

Spark Operations
val creates immutable structures
Var creates mutable structures

scala>
#val rdd = spark.sparkContext.parallelize(1 to 1000)
Or
#val rdd = sc.parallelize(1 to 1000)
#rdd.count()
#rdd.getNumPartitions
#rdd.size
#val result = rdd.filter(x => x%2 == 0) ——> using filter transformation to filter out even numbers
#result.collect()
#result.foreach(println)

#val rdd_spec_partitions = sc.parallelize(1 to 1000, 4)
#rdd.getNumPartitions
——

Load a file in HDFS
For example : file1 : EdurekaTst.txt

Edureka is a known online training provider
Hadoop is a framework to work on large datasets
we are learning spark now
Edureka has many other courses
we are learning about RDD

: file2: EdurekaTst2.txt

Edureka is not a new company
we are learning from edureka on big data
we are experienced professionals
edureka takes courses for freshers and professionals
edureka is in bangalore and delhi

#val mydata = sc.textFile("hdfs://nameservice1/user/edureka_126370/mydataabc/EdurekaTst.txt")
—> creates an RDD using textFile
#val mydatauc = my data.map(line => line.toUpperCase())
#val mydataflt = mydatauc.filter(line => (line.startswith("E"))
#mydataflt.collect —> shows the result
#mydataflt.take(2).foreach(println) ——> to see 2 elements as result
#mydataflt.toDebugString ————> to look at RDD lineage captured by Spark
——

Chaining Transformations:
#val mydata = sc.textFile("hdfs://nameservice1/user/edureka_126370/mydataabc/EdurekaTst.txt").map(line => line.toUpperCase()).filter(line => (line.startswith("E")).collect
——

Difference of Map and flatMap Transformation

#val fruits = Seq("apple", "banana", "orange")
fruits: Seq[java.lang.String] = List(apple, banana, orange)

#fruits.map(_.toUpperCase)
res0: Seq[java.lang.String] = List(APPLE, BANANA, ORANGE)

#fruits.flatMap(_.toUpperCase)
res1: Seq[Char] = List(A, P, P, L, E, B, A, N, A, N, A, O, R, A, N, G, E)

flatMap is like a combination of map followed
by flatten, so it first runs map on the sequence, then runs flatten, giving the result shown.
#val mapResult = fruits.map(_.toUpperCase)
mapResult: Seq[String] = List(APPLE, BANANA, ORANGE)

#val flattenResult = mapResult.flatten
flattenResult: Seq[Char] = List(A, P, P, L, E, B, A, N, A, N, A, O, R, A, N, G, E)

Non Spark : Working with Scala
Scala >

```
#val l = List(10,20,30,40)
  l.size
  val l2 = l.map(x => x * 2)
  l2
  val l3 = l2.filter(x => x%3 == 0)
  l3
```

```
#val s = “this is a line”
  s.toLowerCase
  s.toUpperCase
  s.toLowerCase.contains(“spark”)
  <shows Boolean = false >
  s.toLowerCase.contains(“line”)
  <shows Boolean = true>

# 10 + 2

#val x = “Hello Guys”
  x.foreach(println)
#x ( shows result)
#x = “hi” (shows error  as reassignment to val is not
permitted)
#val x = “hi” ( works as new variable is created
here with new value)

#val x = { val a = 10; val b = 100 ; b - a }
#val file =
scala.io.Source.fromFile(“abc.txt”).mkString —>
shows error if file does’nt exist
#lazy val file =
scala.io.Source.fromFile(“abc.txt”).mkString —>
works as we are creating lazy variables
Note** In Spark, RDDs are implicitly lazy in nature.
```

#val x = 5
#val s = if (x > 0 && x < 6) 1 else 0 ——> return type is int
#val s = if (x > 0 && x < 6) “test” else 0 ——> return type is Any
Note** Any is a class in scala, return type of your control structure is always the immediate super type of your individual classes.

#val args = “Hello”
  args.foreach(arg => println(arg))
Note** For/Foreach loop, where we can take any collection and iterate through the collection without creating an iterator
In scala, string is treated as array of characters.

#val x = 1 to 5
  x.foreach(println)

#for (i <- 1 to 3; j <- 1 to 3) println(10*I+j)
#for (i <- 1 to 3; j <- 1 to 3; if i == j) println(10*I+j)

#Function examples
#def area(radius: Int): Double = { 3.14 * radius * radius }
#def area(radius: Int) = { 3.14 * radius * radius }
#def area(radius: Int) = 3.14 * radius * radius

#area(5)

#def concatStr(arg1: String, arg2: String = “world”,arg3: String = “scala”) = arg1+arg2+arg3
#concatStr(“hello”)
#cocatStr(arg1 = “Hello”,arg3 = “test”)
With names arguments , order is not important

#Scala supports these types of collections
Array
ArrayBuffer
Maps
Tuples
List

#val n = new Array[Int](10) ——> creates immutable array, ie length of array can’t be changed, elements can be.
#n —> list the array
#n(0) ——> shows element at index position 0
#n(0) = 1 ——> assigns a value 1 to array at index position 0
#n.foreach(println)

To create mutable data structures
#import scala.collection.mutable.ArrayBuffer or import scala.collection.mutable

#val a = ArrayBuffer[Int]() ——> creates mutable array structure, I.e. length of array can be changed and elements can be added
a += 2
a += 21
a += 22
a += (23,24,25)
#a ++= Array(26,27)
#a.remove(0) ——> to remove element at position 0
#a.map(x => x *2).foreach(println) ——> applying a map transformation and multiplying each element with 2
#a.trimEnd(2) —> removes last 2 elements
#a.insert(2,9) —> add element at 2nd index position
#a.insert(2,10,11,12) —> adds a list
#a.remove(2,3) —> removes 3 elements from index position 2

#mapping
#val mapping = Map("Aj" -> "test","cool" -> "guy")
Note** Creates a immutable map
#To create a mutable map
#import scala.collection.mutable.Map
#val mapping = Map("Aj" -> "test","cool" -> "guy")
#To update the value for a key

#mapping(“aj”) = “test1”

Tupples (List of different items where each Item can be of different data types)
#val tup = (“x”,234,3.4,”aj”)
#tup._1 ——> (to see element at first position ie here unlike arrays indexing starts at 1)

List
#val list = List(1,2,3)
#list.foreach(println)
#list.head ———> shows first element
#list.tail ————> shows remaining elements

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

import org.apache.spark.sql.SparkSession

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

// For implicit conversions like converting RDDs to DataFrames
import spark.implicits._
```

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:

```
val df = spark.read.json("examples/src/main/resources/people.json")
// Displays the content of the DataFrame to stdout
df.show()
```

DataFrames are just Dataset of Rows 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.

```
import spark.implicits._
df.printSchema()
df.select("name").show()
df.select($"name", $"age" + 1).show()
df.filter($"age" > 21).show()
df.groupBy("age").count().show()
```

Running SQL Queries Programmatically

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

```
// Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView("people")

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

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.

```
// 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()
```

Creating Datasets

Datasets are similar to RDDs, however, instead of using Java serialization or Kryo they use a specialized Encoder to serialize the objects for processing or transmitting over the network. While both encoders and standard serialization are responsible for turning an object into bytes, encoders are code generated dynamically and use a format that allows Spark to perform many operations like filtering, sorting and hashing without deserializing the bytes back into an object.

```
case class Person(name: String, age: Long)

// Encoders are created for case classes
val caseClassDS = Seq(Person("Andy",
32)).toDS()
caseClassDS.show()

// Encoders for most common types are
automatically provided by importing
spark.implicits._
val primitiveDS = Seq(1, 2, 3).toDS()
primitiveDS.map(_ + 1).collect() // Returns:
Array(2, 3, 4)

// DataFrames can be converted to a Dataset
by providing a class. Mapping will be done by
name
val path = "examples/src/main/resources/
```

```
people.json"
val peopleDS =
spark.read.json(path).as[Person]
peopleDS.show()
```

Interoperating with RDDs

Spark SQL supports two different methods for converting existing RDDs into Datasets. The first method uses reflection to infer the schema of an RDD that contains specific types of objects. This reflection based approach leads to more concise code and works well when you already know the schema while writing your Spark application. The second method for creating Datasets is through a programmatic interface that allows you to construct a schema and then apply it to an existing RDD. While this method is more verbose, it allows you to construct Datasets when the columns and their types are not known until runtime.

Inferring the Schema Using Reflection

```
import
org.apache.spark.sql.catalyst.encoders.Expr
essionEncoder
import org.apache.spark.sql.Encoder

// For implicit conversions from RDDs to
DataFrames
import spark.implicits._

// Create an RDD of Person objects from a text
file, convert it to a Dataframe
val peopleDF = spark.sparkContext
  .textFile("examples/src/main/resources/
people.txt")
  .map(_.split(","))
  .map(attributes => Person(attributes(0),
attributes(1).trim.toInt))
  .toDF()
// Register the DataFrame as a temporary view
peopleDF.createOrReplaceTempView("people")

// SQL statements can be run by using the sql
methods provided by Spark
val teenagersDF = spark.sql("SELECT name,
age FROM people WHERE age BETWEEN 13
AND 19")

// The columns of a row in the result can be
accessed by field index
teenagersDF.map(teenager => "Name: " +
teenager(0)).show()
// or by field name
teenagersDF.map(teenager => "Name: " +
teenager.getAs[String]("name")).show()
// No pre-defined encoders for
Dataset[Map[K,V]], define explicitly
implicit val mapEncoder =
org.apache.spark.sql.Encoders.kryo[Map[String
, Any]]
// Primitive types and case classes can be also
defined as
// implicit val stringIntMapEncoder:
Encoder[Map[String, Any]] =
ExpressionEncoder()

// row.getValuesMap[T] retrieves multiple
columns at once into a Map[String, T]
teenagersDF.map(teenager =>
teenager.getValuesMap[Any](List("name",
"age"))).collect()
```

// Array(Map("name" -> "Justin", "age" -> 19))

Programmatically Specifying the Schema
When case classes cannot be defined ahead of time (for example, the structure of records is encoded in a string, or a text dataset will be parsed and fields will be projected differently for different users),
a DataFrame can be created programmatically with three steps.

1. Create an RDD of Rows from the original RDD;
2. Create the schema represented by a StructType matching the structure of Rows in the RDD created in Step 1.
3. Apply the schema to the RDD of Rows via createDataFrame method provided by SparkSession.

import org.apache.spark.sql.types._

// Create an RDD
val peopleRDD **=**
spark.sparkContext.textFile("examples/src/main/

```
resources/people.txt")

// The schema is encoded in a string
val schemaString = "name age"

// Generate the schema based on the string of
schema
val fields = schemaString.split(" ")
  .map(fieldName => StructField(fieldName,
StringType, nullable = true))
val schema = StructType(fields)

// Convert records of the RDD (people) to Rows
val rowRDD = peopleRDD
  .map(_.split(","))
  .map(attributes => Row(attributes(0),
attributes(1).trim))

// Apply the schema to the RDD
val peopleDF = spark.createDataFrame(rowRDD,
schema)

// Creates a temporary view using the
DataFrame
peopleDF.createOrReplaceTempView("people")
```

```
// SQL can be run over a temporary view
created using DataFrames
val results = spark.sql("SELECT name FROM
people")

// The results of SQL queries are DataFrames
and support all the normal RDD operations
// The columns of a row in the result can be
accessed by field index or by field name
results.map(attributes => "Name: " +
attributes(0)).show()
```

Data Sources

Spark SQL supports operating on a variety of data sources through the DataFrame interface. A DataFrame can be operated on using relational transformations and can also be used to create a temporary view. Registering a DataFrame as a temporary view allows you to run SQL queries over its data.

```
val usersDF = spark.read.load("examples/src/
main/resources/users.parquet")
```

```
usersDF.select("name", "favorite_color").write.save("namesAndFavColors.parquet")
val peopleDF = spark.read.format("json").load("examples/src/main/resources/people.json")
peopleDF.select("name", "age").write.format("parquet").save("namesAndAges.parquet")
```

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 an Overwrite, the data will be deleted before writing out the new data.

Scala/Java	Any Language	Meaning
SaveMode.ErrorIfExists(default)	"error"(default)	When saving a DataFrame to a data source, if data already exists, an exception is expected to be thrown.
SaveMode.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.

SaveMode.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.
SaveMode.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.

DataFrames can also be saved as persistent tables into Hive metastore using the saveAsTable command. Notice that an existing Hive deployment is not necessary to use this feature. Spark will create a default local Hive metastore (using Derby) for you. Unlike the createOrReplaceTempViewcommand, saveAsTable will materialize the contents of the DataFrame and create a pointer to the data in the Hive metastore. Persistent tables will still exist even after your Spark program has restarted, as long as you maintain your connection to the same metastore. A DataFrame for a persistent table can be created by calling the table method on a SparkSession with the name of the table. By default saveAsTable will create a “managed table”, meaning that the location of the data will be controlled by the metastore. Managed tables will also have their data deleted automatically when a table is dropped.

Starting from Spark 2.1, persistent datasource tables have per-partition metadata stored in the Hive metastore. This brings several benefits:

- Since the metastore can return only necessary partitions for a query, discovering all the partitions on the first query to the table is no longer needed.
- Hive DDLs such as ALTER TABLE PARTITION ... SET LOCATION are now available for tables created with the Datasource API.

Loading Data Programmatically

Using the data from the above example:

```
// Encoders for most common types are
// automatically provided by importing
spark.implicits._
import spark.implicits._

val peopleDF = spark.read.json("examples/src/
main/resources/people.json")

// DataFrames can be saved as Parquet files,
maintaining the schema information
peopleDF.write.parquet("people.parquet")

// Read in the parquet file created above
// Parquet files are self-describing so the
schema is preserved
// The result of loading a Parquet file is also a
DataFrame
val parquetFileDF =
spark.read.parquet("people.parquet")

// Parquet files can also be used to create a
temporary view and then used in SQL
statements
parquetFileDF.createOrReplaceTempView("parqu
etFile")
val namesDF = spark.sql("SELECT name FROM
parquetFile WHERE age BETWEEN 13 AND 19")
namesDF.map(attributes => "Name: " +
attributes(0)).show()
// +------------+
// |       value|
// +------------+
// |Name: Justin|
// +------------+
```

Find full example code at "examples/src/main/scala/org/apache/spark/examples/sql/SQLDataSourceExample.scala" in the Spark repo.

Partition Discovery

Table partitioning is a common optimization approach used in systems like Hive. In a partitioned table, data are usually stored in different directories, with partitioning column values encoded in the path of each partition directory. The Parquet data source is now able to discover and infer partitioning information automatically. For example, we can store all our previously used population data into a partitioned table using the following directory structure, with two extra columns, gender and country as partitioning columns:

```
path
└── to
    └── table
        ├── gender=male
        │   ├── ...
        │   │
        │   ├── country=US
        │   │   └── data.parquet
        │   ├── country=CN
        │   │   └── data.parquet
        │   └── ...
        └── gender=female
            ├── ...
            │
            ├── country=US
            │   └── data.parquet
            ├── country=CN
            │   └── data.parquet
            └── ...
```

By passing path/to/table to either SparkSession.read.parquet or SparkSession.read.load, Spark SQL will automatically extract the partitioning information from the paths. Now the schema of the returned

DataFrame becomes:

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

Notice that the data types of the partitioning columns are automatically inferred. Currently, numeric data types and string type are supported. Sometimes users may not want to automatically infer the data types of the partitioning columns. For these use cases, the automatic type inference can be configured by spark.sql.sources.partitionColumnTypeInference.enabled, which is default to true. When type inference is disabled, string type will be used for the partitioning columns.
Starting from Spark 1.6.0, partition discovery only finds partitions under the given paths by default. For the above example, if users pass path/to/table/gender=male to either SparkSession.read.parquet or SparkSe

ssion.read.load, gender will not be considered as a partitioning column. If users need to specify the base path that partition discovery should start with, they can set basePath in the data source options. For example, when path/to/table/gender=male is the path of the data and users set basePath to path/to/table/, gender will be a partitioning column.

Schema Merging

Like ProtocolBuffer, Avro, and Thrift, Parquet also supports schema evolution. Users can start with a simple schema, and gradually add more columns to the schema as needed. In this way, users may end up with multiple Parquet files with different but mutually compatible schemas. The Parquet data source is now able to automatically detect this case and merge schemas of all these files.

Since schema merging is a relatively expensive operation, and is not a necessity in most cases, we turned it off by default starting from 1.5.0. You may enable it by

1. setting data source
   option mergeSchema to true when reading Parquet files (as shown in the examples below), or
2. setting the global SQL
   option spark.sql.parquet.mergeSchema to true
   .

// This is used to implicitly convert an RDD to a DataFrame.
import spark.implicits._

// Create a simple DataFrame, store into a partition directory
val squaresDF **=** spark.sparkContext.makeRDD(1 to 5).map(i **=>** (i, i * i)).toDF("value", "square")
squaresDF.write.parquet("data/test_table/key=1")

// Create another DataFrame in a new partition directory,
// adding a new column and dropping an existing column
val cubesDF **=** spark.sparkContext.makeRDD(6

```
to 10).map(i => (i, i * i * i)).toDF("value", "cube")
cubesDF.write.parquet("data/test_table/key=2")

// Read the partitioned table
val mergedDF =
spark.read.option("mergeSchema",
"true").parquet("data/test_table")
mergedDF.printSchema()

// The final schema consists of all 3 columns in
the Parquet files together
// with the partitioning column appeared in the
partition directory paths
// root
//  |-- value: int (nullable = true)
//  |-- square: int (nullable = true)
//  |-- cube: int (nullable = true)
//  |-- key: int (nullable = true)
```

Find full example code at "examples/src/main/scala/org/apache/spark/examples/sql/SQLDataSourceExample.scala" in the Spark repo.

Hive metastore Parquet table conversion

When reading from and writing to Hive

metastore Parquet tables, Spark SQL will try to use its own Parquet support instead of Hive SerDe for better performance. This behavior is controlled by the spark.sql.hive.convertMetastoreParquet configuration, and is turned on by default.

Hive/Parquet Schema Reconciliation

There are two key differences between Hive and Parquet from the perspective of table schema processing.

1. Hive is case insensitive, while Parquet is not
2. Hive considers all columns nullable, while nullability in Parquet is significant

Due to this reason, we must reconcile Hive metastore schema with Parquet schema when converting a Hive metastore Parquet table to a Spark SQL Parquet table. The reconciliation rules are:

1. Fields that have the same name in both schema must have the same data type regardless of nullability. The reconciled field should have the data type of the Parquet side, so that nullability is respected.

2. The reconciled schema contains exactly those fields defined in Hive metastore schema.
    - Any fields that only appear in the Parquet schema are dropped in the reconciled schema.
    - Any fields that only appear in the Hive metastore schema are added as nullable field in the reconciled schema.

Metadata Refreshing

Spark SQL caches Parquet metadata for better performance. When Hive metastore Parquet table conversion is enabled, metadata of those converted tables are also cached. If these tables are updated by Hive or other external tools, you need to refresh them manually to ensure consistent metadata.

```
// spark is an existing SparkSession
spark.catalog.refreshTable("my_table")
```

Configuration

Configuration of Parquet can be done using the setConf method on SparkSession or by running SET key=value commands using SQL.

Property Name	**Default**	**Meaning**
spark.sql.parquet.binaryAsString	false	Some other Parquet-producing systems, in particular Impala, Hive, and older versions of Spark SQL, do not differentiate between binary data and strings when writing out the Parquet schema. This flag tells Spark SQL to interpret binary data as a string to provide compatibility with these systems.
spark.sql.parquet.int96AsTimestamp	true	Some Parquet-producing systems, in particular Impala and Hive, store Timestamp into INT96. This flag tells Spark SQL to interpret INT96 data as a timestamp to provide compatibility with these systems.
spark.sql.parquet.cacheMetadata	true	Turns on caching of Parquet schema metadata. Can speed up querying of static data.
spark.sql.parquet.compression.codec	snappy	Sets the compression codec use when writing Parquet files. Acceptable values include: uncompressed, snappy, gzip, lzo.
spark.sql.parquet.filterPushdown	true	Enables Parquet filter push-down optimization when set to true.

spark.sql.hive.convertMetastoreParquet	true	When set to false, Spark SQL will use the Hive SerDe for parquet tables instead of the built in support.
spark.sql.parquet.mergeSchema	false	When true, the Parquet data source merges schemas collected from all data files, otherwise the schema is picked from the summary file or a random data file if no summary file is available.
spark.sql.optimizer.metadataOnly	true	When true, enable the metadata-only query optimization that use the table's metadata to produce the partition columns instead of table scans. It applies when all the columns scanned are partition columns and the query has an aggregate operator that satisfies distinct semantics.

JSON Datasets

Spark SQL can automatically infer the schema of a JSON dataset and load it as a Dataset[Row]. This conversion can be done using SparkSession.read.json() on either an RDD of String, or a JSON file. Note that the file that is offered as ***a json file*** is not a typical JSON file. Each line must contain a separate, self-contained valid JSON object. . As a consequence, a regular multi-line JSON file will most often fail.

```
// A JSON dataset is pointed to by path.
// The path can be either a single text file or a
directory storing text files
val path = "examples/src/main/resources/
people.json"
val peopleDF = spark.read.json(path)

// The inferred schema can be visualized using
the printSchema() method
peopleDF.printSchema()
// root
//  |-- age: long (nullable = true)
//  |-- name: string (nullable = true)

// Creates a temporary view using the
DataFrame
peopleDF.createOrReplaceTempView("people")

// SQL statements can be run by using the sql
methods provided by spark
val teenagerNamesDF = spark.sql("SELECT
name FROM people WHERE age BETWEEN 13
AND 19")
teenagerNamesDF.show()
// +------+
// |  name|
// +------+
// |Justin|
// +------+

// Alternatively, a DataFrame can be created for
a JSON dataset represented by
// an RDD[String] storing one JSON object per
string
val otherPeopleRDD =
spark.sparkContext.makeRDD(
  """{"name":"Yin","address":
{"city":"Columbus","state":"Ohio"}}""" :: Nil)
val otherPeople =
spark.read.json(otherPeopleRDD)
otherPeople.show()
// +---------------+----+
// |        address|name|
// +---------------+----+
// |[Columbus,Ohio]| Yin|
// +---------------+----+
```

Spark SQL also supports reading and writing data stored in Apache Hive. However, since Hive has a large number of dependencies, these dependencies are not included in the default Spark distribution. If Hive dependencies can be found on the classpath, Spark will load them automatically. Note that these Hive dependencies must also be present on all of the worker nodes, as they will need access to the Hive serialization and deserialization libraries (SerDes) in order to access data stored in Hive.
Configuration of Hive is done by placing your hive-site.xml, core-site.xml (for security configuration), and hdfs-site.xml (for HDFS configuration) file in conf/.
When working with Hive, one must instantiate SparkSession with Hive support, including connectivity to a persistent Hive metastore, support for Hive serdes, and Hive user-defined functions. Users who do not have an existing Hive deployment can still enable Hive support. When not configured by

the hive-site.xml, the context automatically creates metastore_db in the current directory and creates a directory configured
by spark.sql.warehouse.dir, which defaults to the directory spark-warehouse in the current directory that the Spark application is started. Note that
the hive.metastore.warehouse.dir property in hive-site.xml is deprecated since Spark 2.0.0. Instead,
use spark.sql.warehouse.dir to specify the default location of database in warehouse. You may need to grant write privilege to the user who starts the Spark application.

```
import org.apache.spark.sql.Row
import org.apache.spark.sql.SparkSession

case class Record(key: Int, value: String)

// warehouseLocation points to the default
location for managed databases and tables
val warehouseLocation = "spark-warehouse"
val spark = SparkSession
  .builder()
  .appName("Spark Hive Example")
  .config("spark.sql.warehouse.dir",
warehouseLocation)
  .enableHiveSupport()
  .getOrCreate()

import spark.implicits._
import spark.sql

sql("CREATE TABLE IF NOT EXISTS src (key INT,
value STRING)")
sql("LOAD DATA LOCAL INPATH 'examples/src/
main/resources/kv1.txt' INTO TABLE src")

// Queries are expressed in HiveQL
sql("SELECT * FROM src").show()
// +---+-------+
// |key|  value|
// +---+-------+
// |238|val_238|
// | 86| val_86|
// |311|val_311|
// ...
// Aggregation queries are also supported.
sql("SELECT COUNT(*) FROM src").show()
// +--------+
// |count(1)|
// +--------+
// |    500 |
// +--------+

// The results of SQL queries are themselves
DataFrames and support all normal functions.
val sqlDF = sql("SELECT key, value FROM src
WHERE key < 10 ORDER BY key")

// The items in DaraFrames are of type Row,
which allows you to access each column by
ordinal.
val stringsDS = sqlDF.map {
  case Row(key: Int, value: String) => s"Key:
$key, Value: $value"
}
stringsDS.show()
// +--------------------+
// |               value|
// +--------------------+
// |Key: 0, Value: val_0|
// |Key: 0, Value: val_0|
// |Key: 0, Value: val_0|
// ...

// You can also use DataFrames to create
temporary views within a SparkSession.
val recordsDF = spark.createDataFrame((1 to
100).map(i => Record(i, s"val_$i")))
recordsDF.createOrReplaceTempView("records")

// Queries can then join DataFrame data with
data stored in Hive.
sql("SELECT * FROM records r JOIN src s ON
r.key = s.key").show()
// +---+------+---+------+
// |key| value|key| value|
// +---+------+---+------+
// |  2| val_2|  2| val_2|
// |  4| val_4|  4| val_4|
// |  5| val_5|  5| val_5|
// ...
```

Extras

```
myDataFrame.show(Int.MaxValue)
```
It is generally not advisable to display an entire DataFrame to stdout, because that means you need to pull the entire DataFrame (all of its values) to the driver (unless DataFrame is already local, which you can check with df.isLocal).
Unless you know ahead of time that the size of your dataset is sufficiently small so that driver JVM process has enough memory available to accommodate all values, it is not safe to do this.
That's why DataFrame API's show() by default shows you only the first 20 rows.
You could use the df.collect which returns Array[T] and then iterate over each line and print it:
```
df.collect.foreach(println)
df.toJSON.collect.foreach(println)
println(df.show())
Df.show()
```

Spark 1.6
A Dataset is a new experimental interface added in Spark 1.6 that tries to provide 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.).

Spark 2.2.1
A Dataset is a distributed collection of data. Dataset is a new interface added in Spark 1.6 that 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.

Tungsten is a new Spark SQL component that provides more efficient Spark operations by working directly at the byte level.

Article

Background

Tungsten became the default in Spark 1.5 and can be enabled in earlier versions by setting spark.sql.tungsten.enabled to true (or disabled in later versions by setting this to false). Even without Tungsten, Spark SQL uses a columnar storage format with Kryo serialization to minimize storage cost.

Goal

The goal of Project Tungsten is to improve Spark execution by optimizing Spark jobs for CPU and memory efficiency (as opposed to network and disk I/O which are considered fast enough).

Scope

Tungsten focuses on the hardware architecture of the platform Spark runs on, including but not limited to JVM, LLVM, GPU, NVRAM, etc.

Optimization Features

- Off-Heap Memory Management using binary in-memory data representation aka Tungsten row format and managing memory explicitly,
- Cache Locality which is about cache-aware computations with cache-aware layout for

high cache hit rates,
- Whole-Stage Code Generation (aka *CodeGen*).

Design Improvements

- Tungsten includes specialized in-memory data structures tuned for the type of operations required by Spark, improved code generation, and a specialized wire protocol.
- Tungsten's representation is substantially smaller than objects serialized using Java or even Kryo serializers.
- As Tungsten does not depend on Java objects, both on-heap and off-heap allocations are supported. Not only is the format more compact, serialization times can be substantially faster than with native serialization. Since Tungsten no longer depends on working with Java objects, you can use either on-heap (in the JVM) or off-heap storage. If you use off-heap storage, it is important to leave enough room in your containers for the off-heap allocations - which you can get an approximate idea for from the web ui.
- Tungsten's data structures are also created

closely in mind with the kind of processing for which they are used. The classic example of this is with sorting, a common and expensive operation. The on-wire representation is implemented so that sorting can be done without having to deserialize the data again.
- By avoiding the memory and GC overhead of regular Java objects, Tungsten is able to process larger data sets than the same hand-written aggregations.

Benefits

The following Spark jobs will benefit from Tungsten:

- Dataframes: Java, Scala, Python, R
- SparkSQL queries
- Some RDD API programs via general serialization and compression optimizations

Next Steps

In the future Tungsten may make it more feasible to use certain non-JVM libraries. For many simple operations the cost of using BLAS, or similar linear algebra packages, from the JVM is dominated by the cost of copying the data off-heap.

K means:

```
def ageCategoriser(age: Int): String =
{
  age match{
   case t if t < 30 => "young"
   case t if t > 65 => "old"
   case t if t >=30 && t <= 65 => "mid"
  }
  }

while (tempDist > convergeDist) {
val closest = points.map(p => (closestPoint(p,
kPoints),(p,1)))
val pointStats = closest.reduceByKey{ case
((point1,n1),(point2,n2)) =>
(addPoints(point1,point2),n1+n2)}
 val newPoints = pointStats.map{case(i,
(point,n))=>(i,(point._1/n,point._2/
n))}.collectAsMap()
    tempDist = 0.0
    for (i <- 0 until K) {
    tempDist +=
distanceSquared(kPoints(i),newPoints(i)) }
    println("Distance between iterations:" +
tempDist)
```

```
    for (i <- 0 until K) {
    kPoints(i) = newPoints(i)
    }}

scala> kPoints.foreach(println)
(33.4958485281,-111.247263698)
(37.4529109631,-121.925959445)
(35.5840737229,-120.420453558)

scala> kPoints.foreach(println)
(35.08592000544937,-112.57643826547802
)
(39.93446153232589,-121.38638468623522
)
(34.55083229984546,-118.0643623654219)


import org.apache.spark.sql.SQLContext
val sqlcontext = new
org.apache.spark.sql.SQLContext(sc)
val dataframe_mysql =
sqlcontext.read.format("jdbc").option("url",
"jdbc:mysql://sn1:3306/test").option("driver",
"com.mysql.jdbc.Driver").option("dbtable",
"emp").option("user", "root").option("password",
"mapr").load()
dataframe_mysql.show()
```