# SparkSQL and Dataframes

Spark SQL

## What We'll Cover

- Overview
- SQLContext and HiveContexts
- Creating data frames
- Dataframe operations
- Running SQL programmatically
- Data sources including interoperation with Hive

# History: SparkSQL Replaces Shark

- Hive had/has an execution engine for Spark called Shark
  - set hive.execution.engine=spark
  - (please note: this does not work on the EMR Hive because it wasn't compiled for this option)
- However, much of its internals still planned for a MapReduce dataflow
- Instead of just making Shark bigger, the Berkeley people started over again with SparkSQL

# Remember All Those Actions and Transformations?

THINKBIG ANALYTICS
A TERADATA COMPANY

***Transformations***

map
filter
flatMap
mapPartitions
mapPartitionsWithIndex
sample
union
intersection
distinct
cartesian
pipe
coalesce
groupByKey
reduceByKey
aggregateByKey
sortByKey
join
cogroup
keys
values
repartition
repartitionAndSortWithinPartitions

***Actions***

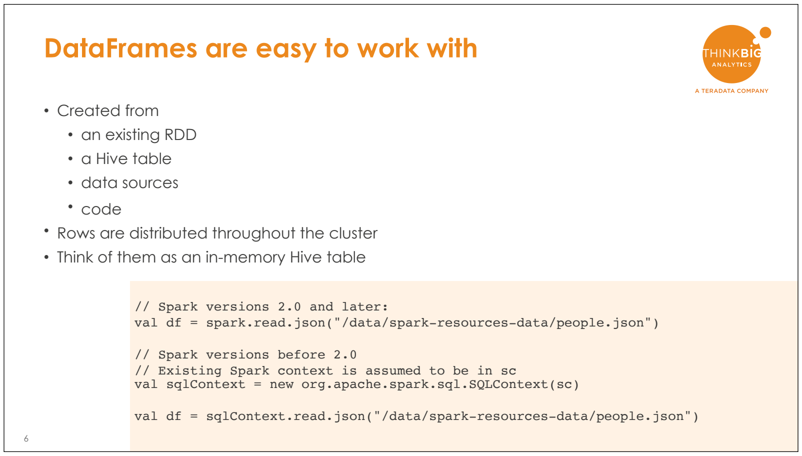
reduce
collect
count
first
take
takeSample
takeOrdered
saveAsTextFile
saveAsSequenceFile
saveAsObjectFile
countByKey
foreach

***Aren't they terribly low level?***

***Wouldn't it be great if we had standard data manipulation operations?***

## SparkSQL Gives Programmers Higher Level Abstractions for RDDs

**DataFrame:**
**a distributed collection of data organized into rows and columns (i.e., like relational tables)**



# DataFrames are easy to work with

- Created from
  - an existing RDD
  - a Hive table
  - data sources
  - code
- Rows are distributed throughout the cluster
- Think of them as an in-memory Hive table

```
// Spark versions 2.0 and later:
val df = spark.read.json("/data/spark-resources-data/people.json")

// Spark versions before 2.0
// Existing Spark context is assumed to be in sc
val sqlContext = new org.apache.spark.sql.SQLContext(sc)

val df = sqlContext.read.json("/data/spark-resources-data/people.json")
```

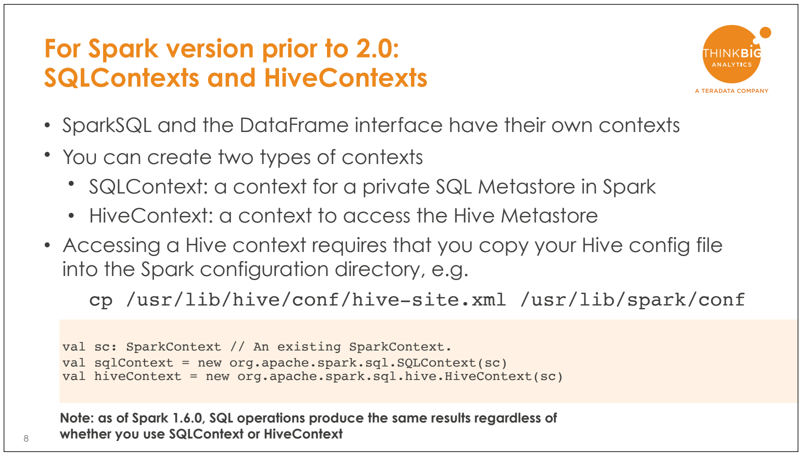
# For Spark Versions 2.0 and Above, Access SQL Operations through your Spark Session

- Accessing Hive tables requires that you copy your Hive config file into the Spark configuration directory, e.g.

```
cp /usr/lib/hive/conf/hive-site.xml /usr/lib/spark/conf
```

- Spark-shell automatically creates a Spark session object called `spark.`
- Create your own Spark session using the SparkSession method.

```
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._
```



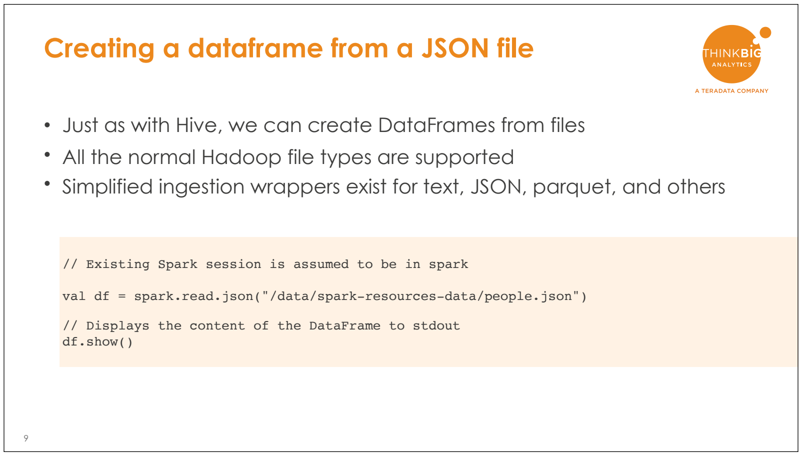
## For Spark version prior to 2.0: SQLContexts and HiveContexts

- SparkSQL and the DataFrame interface have their own contexts
- You can create two types of contexts
  - SQLContext: a context for a private SQL Metastore in Spark
  - HiveContext: a context to access the Hive Metastore
- Accessing a Hive context requires that you copy your Hive config file into the Spark configuration directory, e.g.

```
cp /usr/lib/hive/conf/hive-site.xml /usr/lib/spark/conf
```

```
val sc: SparkContext // An existing SparkContext.
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
val hiveContext = new org.apache.spark.sql.hive.HiveContext(sc)
```

**Note: as of Spark 1.6.0, SQL operations produce the same results regardless of whether you use SQLContext or HiveContext**



## Creating a dataframe from a JSON file

- Just as with Hive, we can create DataFrames from files
- All the normal Hadoop file types are supported
- Simplified ingestion wrappers exist for text, JSON, parquet, and others

```
// Existing Spark session is assumed to be in spark

val df = spark.read.json("/data/spark-resources-data/people.json")

// Displays the content of the DataFrame to stdout
df.show()
```

- Show is how you display the data frame

# Creating a dataframe from a Parquet file

- Just as with JSON, we can create DataFrames from Parquet files too

```
val df = spark.read.parquet("/data/spark-resources-data/users.parquet")
df.show()

val s3flights = spark.read.parquet("s3n://thinkbig.academy.aws/ontime/parquet")
```

## For Spark 2.0 and Later:
## Use the Spark Session To apply SQL

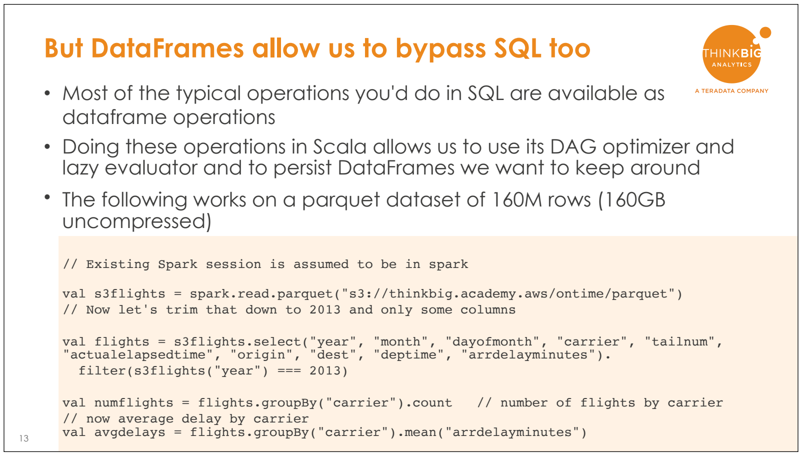
- Just as with Hive, we can create DataFrames from files
- All the normal Hadoop file types are supported
- Simplified ingestion wrappers exist for text, JSON, parquet, and others

```
// Existing Spark context is assumed to be in sc
val hiveContext = new org.apache.spark.sql.hive.HiveContext(sc)

spark.sql("USE carl")                     // Select my database
val stocks = spark.sql("SELECT * FROM STOCKS")
stocks.show()
```

# For Spark versions before 1.6.0: A HiveContext allows us to read Hive files

- Just as with Hive, we can create DataFrames from files
- All the normal Hadoop file types are supported
- Simplified ingestion wrappers exist for text, JSON, parquet, and others

```
// Existing Spark context is assumed to be in sc
val hiveContext = new org.apache.spark.sql.hive.HiveContext(sc)

hiveContext.sql("USE carl")                     // Select my database
val stocks = hiveContext.sql("SELECT * FROM STOCKS")
stocks.show()
```



## But DataFrames allow us to bypass SQL too

THINKBIG ANALYTICS

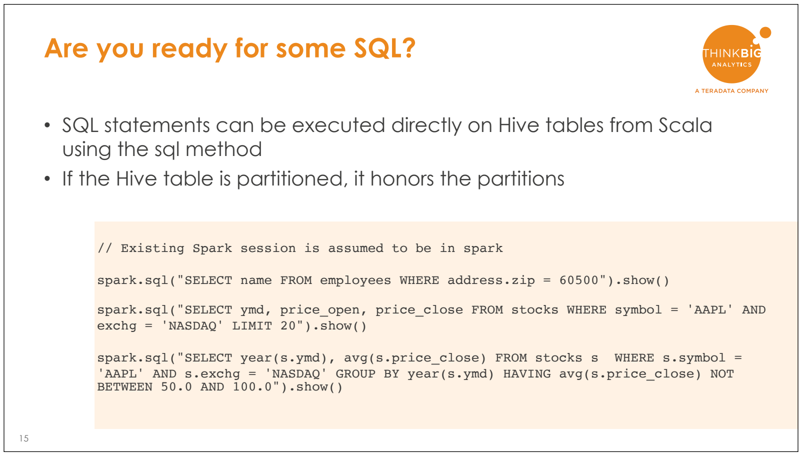
A TERADATA COMPANY

- Most of the typical operations you'd do in SQL are available as dataframe operations
- Doing these operations in Scala allows us to use its DAG optimizer and lazy evaluator and to persist DataFrames we want to keep around
- The following works on a parquet dataset of 160M rows (160GB uncompressed)

```
// Existing Spark session is assumed to be in spark

val s3flights = spark.read.parquet("s3://thinkbig.academy.aws/ontime/parquet")
// Now let's trim that down to 2013 and only some columns

val flights = s3flights.select("year", "month", "dayofmonth", "carrier", "tailnum",
"actualelapsedtime", "origin", "dest", "deptime", "arrdelayminutes").
  filter(s3flights("year") === 2013)

val numflights = flights.groupBy("carrier").count   // number of flights by carrier
// now average delay by carrier
val avgdelays = flights.groupBy("carrier").mean("arrdelayminutes")
```

- SQL 1978
- Adding a dot at the end of your select statement calls it the first portion for execution
- RDD catalyst optimizer allows the computation to run faster
-

## Lab: 01-sparkSQL-walkthrough.{md,scala}



## Are you ready for some SQL?

- SQL statements can be executed directly on Hive tables from Scala using the sql method
- If the Hive table is partitioned, it honors the partitions

```
// Existing Spark session is assumed to be in spark

spark.sql("SELECT name FROM employees WHERE address.zip = 60500").show()

spark.sql("SELECT ymd, price_open, price_close FROM stocks WHERE symbol = 'AAPL' AND
exchg = 'NASDAQ' LIMIT 20").show()

spark.sql("SELECT year(s.ymd), avg(s.price_close) FROM stocks s  WHERE s.symbol =
'AAPL' AND s.exchg = 'NASDAQ' GROUP BY year(s.ymd) HAVING avg(s.price_close) NOT
BETWEEN 50.0 AND 100.0").show()
```

- If the hive table is partitioned, the efficiencies are maintained in sql

# The spark-sql shell

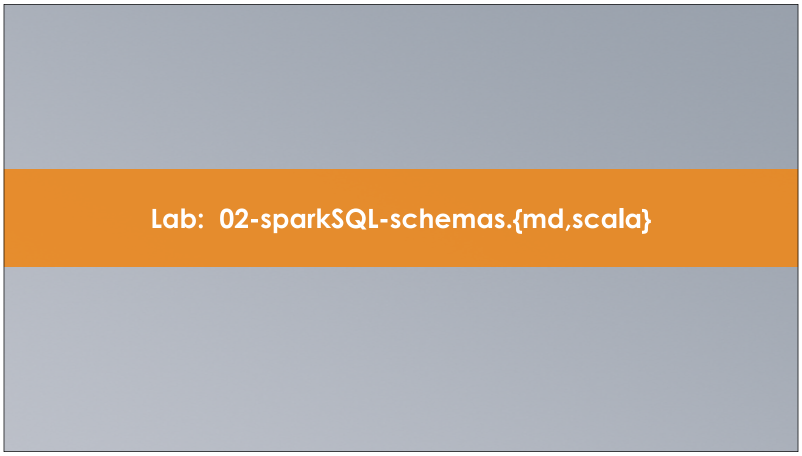
- Hive contexts understand HiveQL, but use a different execution engine
- Compare the speed of SparkSQL with Hive running the same queries
- A good way to do this is to run the command spark-sql
- It uses a HiveContext by default if you have Hive configured

```
// from the Unix shell type the following:

spark-sql

USE carl;

SELECT year(s.ymd), avg(s.price_close) FROM stocks s  WHERE s.symbol = 'AAPL' AND s.exchg =
'NASDAQ' GROUP BY year(s.ymd) HAVING avg(s.price_close) NOT BETWEEN 50.0 AND 100.0;
```

# We can run SQL on DataFrames from RDDs too!

- You can build your own DataFrames from RDDs and query them with SQL statements
- In fact, DataFrames are RDDs

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

/// Create an RDD of Person objects and register it as a table.
val people = spark.read.textFile("hdfs:///data/spark-resources-data/people.txt").
  map(_.split(",")).                          // split by commas
  map(p => Person(p(0), p(1).trim.toInt)). // first field is the person string, second is an integer
  toDF()
people.registerTempTable("people")

// SQL statements can be run by using the sql methods provided by sqlContext.
val teenagers = spark.sql("SELECT name FROM people WHERE age >= 13 AND age <= 19")

// 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 ordinal.
teenagers.map(t => "Name: " + t(0)).collect().foreach(println)
```



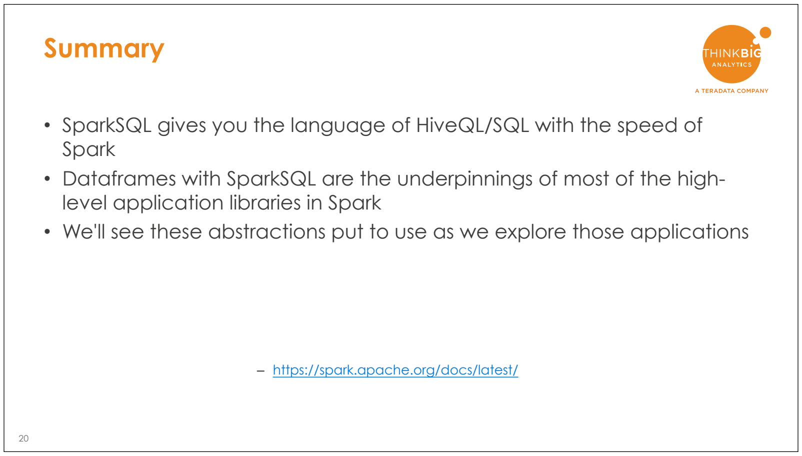
Lab: 02-sparkSQL-schemas.{md,scala}

## Quick exercise

Using the Scala wordcount as an example, count up the number of words in all of Shakespeare

If you know how to use regular expressions, can you suggest a better way to split up the words in Shakespeare so we don't get all the punctuation?

Can you modify your function to count all words longer than 10 characters? The function length gives you the length of a string.



## Summary

- SparkSQL gives you the language of HiveQL/SQL with the speed of Spark
- Dataframes with SparkSQL are the underpinnings of most of the high-level application libraries in Spark
- We'll see these abstractions put to use as we explore those applications

– https://spark.apache.org/docs/latest/