Apache Spark SQL and DataFrames

SMU, 2019

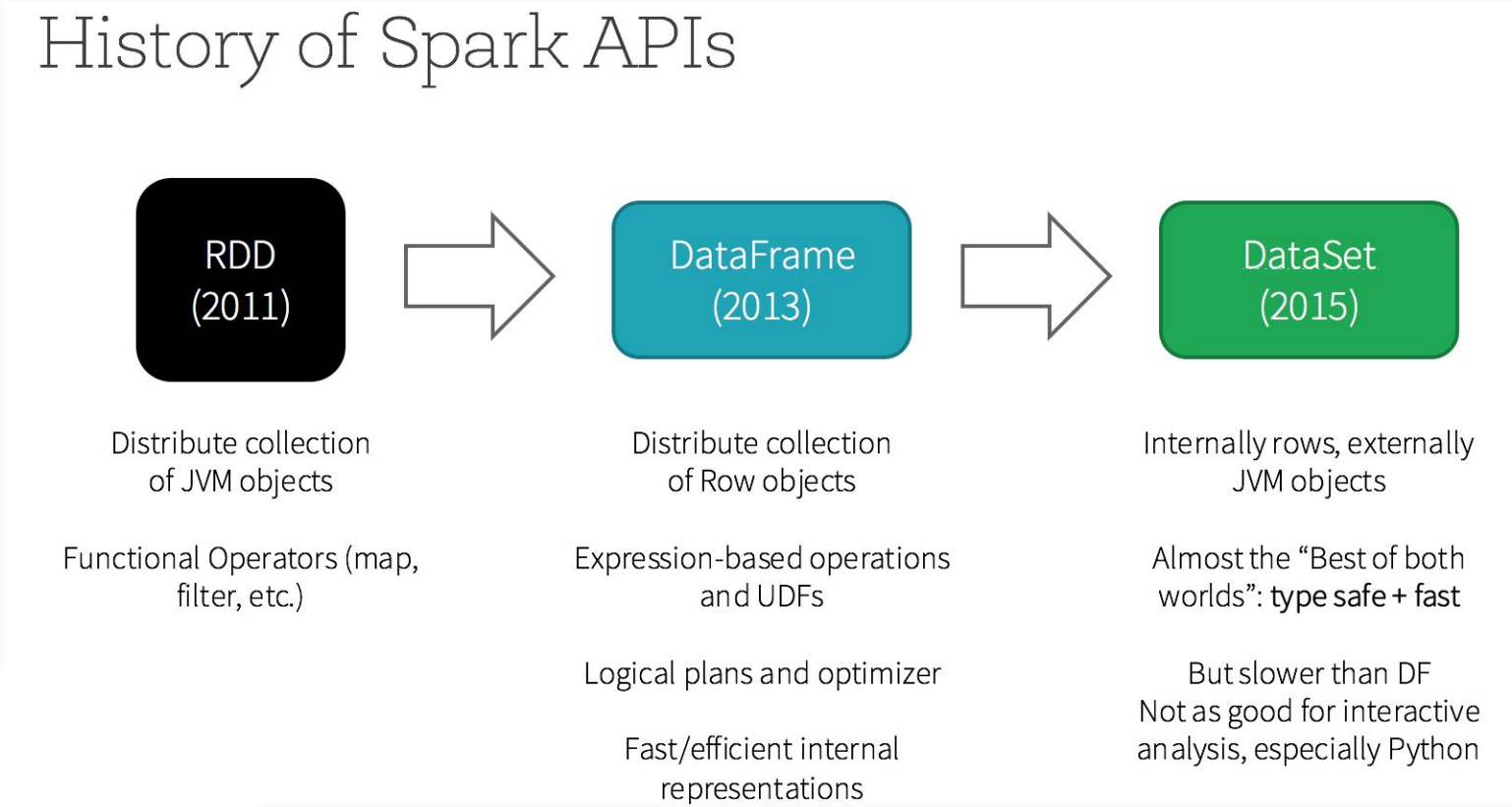
Nikita Neveditsin
nikita.neveditsin@smu.ca

What is Spark SQL?

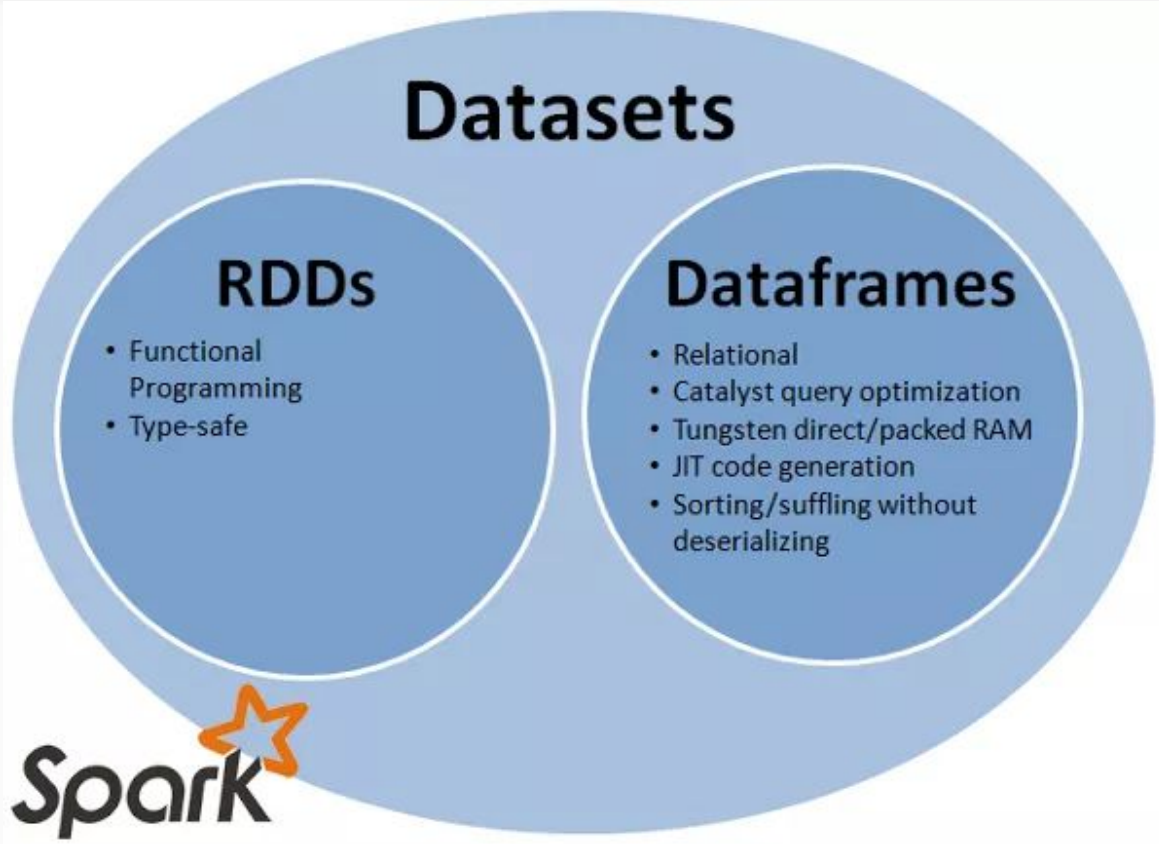
Spark SQL is a Spark module for **structured data processing**. Unlike the basic Spark RDD API, the interfaces provided by Spark SQL provide Spark with more information about the structure of both the data and the computation being performed. Internally, Spark SQL uses this extra information to perform extra optimizations. There are several ways to interact with Spark SQL including SQL and the Dataset API [l]

[link]

RDD, DataFrames, DataSets



RDD, DataFrames, DataSets (cont-d)



[link]

RDD, DataFrames, DataSets (cont-d)

Major points:

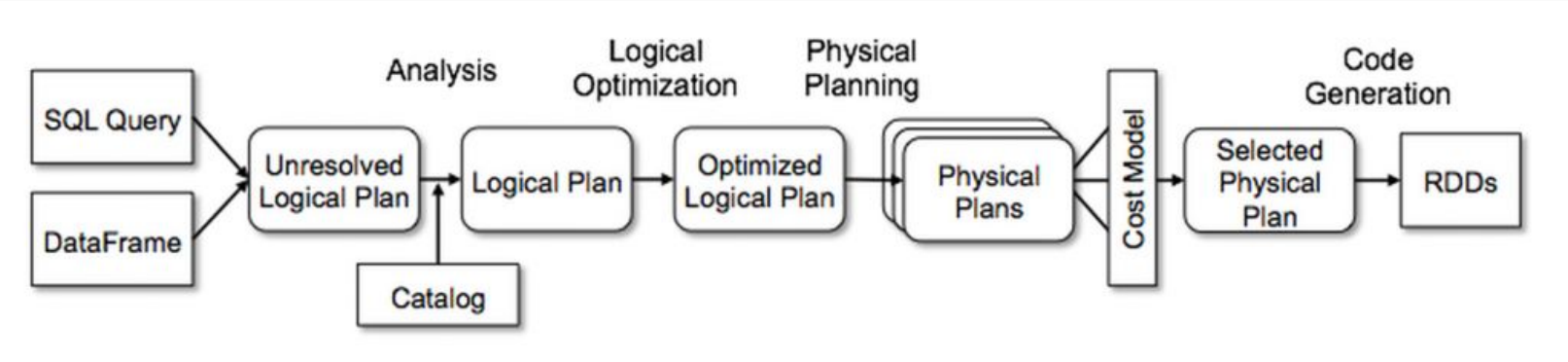
- All 3 interfaces provide **immutable** collections of objects
- RDD does not have a schema. DataFrame and Dataset both have a schema
- DataFrame and Dataset are better optimized in terms of performance
- A Dataset can identify more errors on compile time than a DataFrame
- Datasets are not supported in Python as Python uses dynamic typing (while Java and Scala are statically typed languages)

Do we still need RDD?

Some RDD use cases:

- Both DataFrame and Dataset use RDDs under the hood
- For working with unstructured data
- For low-level transformations and actions
- When you feel more comfortable with functional programming rather than SQL queries

Optimizations: Catalyst



[link]

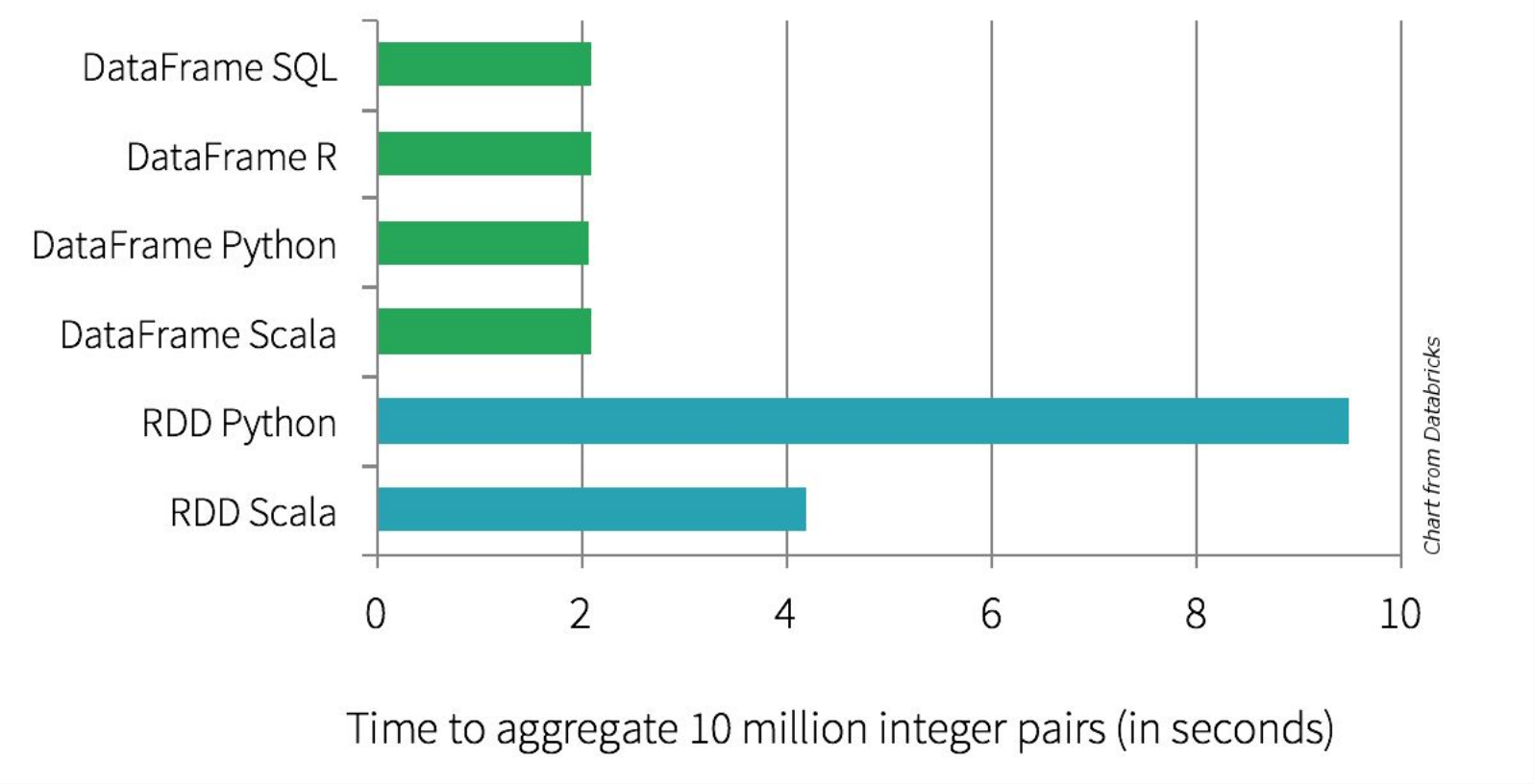
Optimizations: Tungsten

Tungsten is a codename for a set of projects that are focused on improving performance of the Spark execution engine by optimizing memory and CPU intensive operations.

Major optimizations:

- Instead of using JVM garbage collector, Spark engine manages memory manually (using Unsafe/ByteBuffer/etc.)
- Using cache-aware computations
- CPU-optimized code generation
- Moving intermediate data from memory to the CPU registers

Benchmarks



[link]

DataFrames and DataSets in action

- Open Zeppelin Notebook (http://sandbox-hdp.hortonworks.com:9995)
- Download a notebook http://wolly.cs.smu.ca/tmp/spsql.json and import it into Zeppelin
- Run the first 3 cells

DataFrames and DataSets: Word Count

Word count: RDD Way

```
%spark2.spark
val sc = spark.sparkContext
val textFile = sc.textFile("/__dsets/textdata/big.txt")
textFile.take(5)
val counts = textFile.flatMap(line => line.split(" "))
                     .map(word => (word, 1))
                     .reduceByKey(_ + _)
counts.take(10)
```

```
sc: org.apache.spark.SparkContext = org.apache.spark.SparkContext@37d1115d
textFile: org.apache.spark.rdd.RDD[String] = /__dsets/textdata/big.txt MapPartitionsRDD[1] at textFile at <console>:25
res0: Array[String] = Array(The Project Gutenberg EBook of The Adventures of Sherlock Holmes, by Sir Arthur Conan Doyle,
hanging all over the world. Be sure to check the)
counts: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[4] at reduceByKey at <console>:27
res1: Array[(String, Int)] = Array((Ah!,6), (reunion,4), (bone,350), (anaerobes,,2), ("opposed,2), (595;,1), (blandly,6)
```

DataFrames and DataSets: Word Count (cont-d)

Word count: DataSet Way

```
%spark2.spark
val textFile = spark.read.textFile("/__dsets/textdata/big.txt")
textFile.count()
textFile.first()
val counts = textFile.flatMap(line => line.split(" ")).
                                groupByKey(identity).count()
counts.take(10)
```

```
textFile: org.apache.spark.sql.Dataset[String] = [value: string]
res10: Long = 128457
res11: String = The Project Gutenberg EBook of The Adventures of Sherlock Holmes
counts: org.apache.spark.sql.Dataset[(String, Long)] = [value: string, count(1): bigint]
res13: Array[(String, Long)] = Array((By,186), (cold,,57), (those,1101), (some,1378), (few,443), ('Eg.',1), (Sit,5),
```

DataFrames and DataSets: Word Count (cont-d)

Python: DataFrames only

```
%spark2.pyspark
textFile = spark.read.text("/__dsets/textdata/big.txt")
textFile
```

```
DataFrame[value: string]
```

Conversion between RDD and DataSet(Frame)

Scala: DataSet of Strings can be easily converted to RDD of Strings:

```
%spark2.spark
val textFile = spark.read.textFile("/__dsets/textdata/big.txt")
val tfrdd = textFile.rdd
tfrdd.first()
```

```
textFile: org.apache.spark.sql.Dataset[String] = [value: string]
tfrdd: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[139] at rdd at <console>:25
res22: String = The Project Gutenberg EBook of The Adventures of Sherlock Holmes
res24: org.apache.spark.sql.DataFrame = [value: string]
```

Conversion between RDD and DataSet(Frame)

Python: Convert from DataFrame to RDD: we have RDD of Row objects first then convert Row objects to strings

```
%spark2.pyspark
tfrdd = textFile.rdd
print(tfrdd.first())
strRdd = tfrdd.flatMap(list)
strRdd.first()
```

```
Row(value=u'The Project Gutenberg EBook of The Adventures of Sherlock Holmes')
u'The Project Gutenberg EBook of The Adventures of Sherlock Holmes'
```

Took 1 sec. Last updated by anonymous at March 13 2019, 4:04:29 PM.

Creating a DataFrame from CSV files

```
%sh
wget http://wolly.cs.smu.ca/tmp/drivers.csv
wget http://wolly.cs.smu.ca/tmp/timesheet.csv
hadoop fs -mkdir -p /dsets/trucks
hadoop fs -put drivers.csv /dsets/trucks
hadoop fs -put timesheet.csv /dsets/trucks
hadoop fs -ls /dsets/trucks
```

```
--2019-03-06 19:58:35--  http://wolly.cs.smu.ca/tmp/drivers.csv
Resolving wolly.cs.smu.ca (wolly.cs.smu.ca)... 140.184.230.23
Connecting to wolly.cs.smu.ca (wolly.cs.smu.ca)|140.184.230.23|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 2043 (2.0K) [text/csv]
Saving to: ‘drivers.csv’
```

Creating a DataFrame from CSV files (cont-d)

Create DataFrame from CSV with inferred schema

```
%spark2.pyspark
drivers = spark.read.csv("/dsets/trucks/drivers.csv", header=True, inferSchema = "true")
timesheet = spark.read.csv("/dsets/trucks/timesheet.csv", header=True,  inferSchema = "true")
```

DataFrame API: print schema

Show Schema

```
%spark2.pyspark
drivers.printSchema()
print(drivers.columns)
```

```
root
 |-- driverId: integer (nullable = true)
 |-- name: string (nullable = true)
 |-- ssn: integer (nullable = true)
 |-- location: string (nullable = true)
 |-- certified: string (nullable = true)
 |-- wage-plan: string (nullable = true)
['driverId', 'name', 'ssn', 'location', 'certified', 'wage-plan']
```

DataFrame API: get row objects

Get first 2 Row objects

```
%spark2.pyspark
drivers.take(2)
```

```
[Row(driverId=10, name=u'George Vetticaden', ssn=621011971, location=u'244-4532 Nulla Rd.', certified=u'N',
Engesser', ssn=262112338, location=u'366-4125 Ac Street', certified=u'N', wage-plan=u'miles')]
```

Took 1 sec. Last updated by anonymous at March 13 2019, 4:15:53 PM.

DataFrame API: printing rows

Print first 2 rows

```
%spark2.pyspark
drivers.show(2)
```

```
+--------+----------------+---------+------------------+---------+---------+
|driverId|            name|      ssn|          location|certified|wage-plan|
+--------+----------------+---------+------------------+---------+---------+
|      10|George Vetticaden|621011971|244-4532 Nulla Rd.|        N|    miles|
|      11|   Jamie Engesser|262112338|366-4125 Ac Street|        N|    miles|
+--------+----------------+---------+------------------+---------+---------+
only showing top 2 rows
```

DataFrame API: how many rows are there?

Show row count

```
%spark2.pyspark
print(drivers.count())
print(timesheet.count())
```

```
34
1768
```

DataFrame API: print summary

Show stats for each column

```
%spark2.pyspark
drivers.describe().show()
```

```
+-------+-----------------+---------+--------------------+--------------------+---------+---------+
|summary|         driverId|     name|                 ssn|            location|certified|wage-plan|
+-------+-----------------+---------+--------------------+--------------------+---------+---------+
|  count|               34|       34|                  34|                  34|       34|       34|
|   mean|             26.5|     null|4.90337713117647...|                null|     null|     null|
| stddev|9.958246164193104|     null| 3.304606057937715E8|                null|     null|     null|
|    min|               10|Adam Diaz|           123808238|       1027 Quis Rd.|        N|    hours|
|    max|               43|Wes Floyd|           977706052|P.O. Box 945- 601...|        Y|    miles|
+-------+-----------------+---------+--------------------+--------------------+---------+---------+
```

DataFrame API: select columns

Select particular columns

```
%spark2.pyspark
drivers.select('driverId','name').show(3)
drivers.select(drivers.columns[:2]).show(3)
driv = drivers.select('driverId','name')
```

```
+--------+----------------+
|driverId|            name|
+--------+----------------+
|      10|George Vetticaden|
|      11|   Jamie Engesser|
|      12|      Paul Coddin|
+--------+----------------+
only showing top 3 rows
+--------+----------------+
|driverId|            name|
+--------+----------------+
|      10|George Vetticaden|
|      11|   Jamie Engesser|
|      12|      Paul Coddin|
+--------+----------------+
```

DataFrame API: select columns - drop

```
%spark2.pyspark
timesheet.show(2)
tsheet = timesheet.select('driverId', 'hours-logged', 'miles-logged')
tsheet.show(2)
timesheet.drop('week').show(2)
```

```
+--------+----+------------+------------+
|driverId|week|hours-logged|miles-logged|
+--------+----+------------+------------+
|      10|   1|          70|        3300|
|      10|   2|          70|        3300|
+--------+----+------------+------------+
only showing top 2 rows
+--------+------------+------------+
|driverId|hours-logged|miles-logged|
+--------+------------+------------+
|      10|          70|        3300|
|      10|          70|        3300|
+--------+------------+------------+
only showing top 2 rows
+--------+------------+------------+
|driverId|hours-logged|miles-logged|
+--------+------------+------------+
|      10|          70|        3300|
```

DataFrame API: grouping and aggregating

Group by and Aggregate

```
%spark2.pyspark
aggsheet = tsheet.groupby('driverId').agg({'hours-logged': 'sum', 'miles-logged': 'sum'})
aggsheet.show(2)
```

```
+--------+-----------------+-----------------+
|driverId|sum(miles-logged)|sum(hours-logged)|
+--------+-----------------+-----------------+
|      31|           137057|             2704|
|      34|           137728|             2811|
+--------+-----------------+-----------------+
only showing top 2 rows
```

DataFrame API: joins

Join

```
%spark2.pyspark
joined = driv.join(aggsheet, ["driverId"], 'inner')
joined.show(2)
```

```
+--------+-------------+-----------------+-----------------+
|driverId|         name|sum(miles-logged)|sum(hours-logged)|
+--------+-------------+-----------------+-----------------+
|      31|Rommel Garcia|           137057|             2704|
|      34| Frank Romano|           137728|             2811|
+--------+-------------+-----------------+-----------------+
only showing top 2 rows
```

Drivers and timesheets exercise (Pig code):

```
drivers = LOAD '/__dsets/trucks/drivers.csv' USING PigStorage(',');
raw_drivers = FILTER drivers BY $0>1;
drivers_details = FOREACH raw_drivers GENERATE $0 AS driverId, $1 AS name;
timesheet = LOAD '/dsets/trucks/timesheet.csv' USING PigStorage(',');
raw_timesheet = FILTER timesheet by $0>1;
timesheet_logged = FOREACH raw_timesheet GENERATE $0 AS driverId, $2 AS
hours_logged, $3 AS miles_logged;
grp_logged = GROUP timesheet_logged by driverId;
sum_logged = FOREACH grp_logged GENERATE group as driverId,
SUM(timesheet_logged.hours_logged) as sum_hourslogged,
SUM(timesheet_logged.miles_logged) as sum_mileslogged;
join_sum_logged = JOIN sum_logged by driverId, drivers_details by driverId;
join_data = FOREACH join_sum_logged GENERATE $0 as driverId, $4 as name, $1 as
hours_logged, $2 as miles_logged;
dump join_data;
```

Drivers and timesheets exercise (Spark code):

Complete example

```
%spark2.pyspark
drivers = spark.read.csv("/dsets/trucks/drivers.csv", header=True, inferSchema = "true")
timesheet = spark.read.csv("/dsets/trucks/timesheet.csv", header=True,  inferSchema = "true")
driv = drivers.select('driverId','name')
tsheet = timesheet.select('driverId', 'hours-logged', 'miles-logged')
aggsheet = tsheet.groupby('driverId').agg({'hours-logged': 'sum', 'miles-logged': 'sum'})
joined = driv.join(aggsheet, ["driverId"], 'inner')
joined.show(2)
```

```
+--------+-------------+-----------------+-----------------+
|driverId|         name|sum(miles-logged)|sum(hours-logged)|
+--------+-------------+-----------------+-----------------+
|      31|Rommel Garcia|           137057|             2704|
|      34| Frank Romano|           137728|             2811|
+--------+-------------+-----------------+-----------------+
only showing top 2 rows
```

Took 2 sec. Last updated by anonymous at March 12 2019, 1:39:27 PM.

DataFrame API: filtering

Filtering

```
%spark2.pyspark
joined.filter(joined["driverId"] < 13).show()
joined.filter(joined.driverId < 13).filter(joined["name"].rlike("^Paul.*")).show()
```

```
+--------+----------------+----------------+----------------+
|driverId|            name|sum(miles-logged)|sum(hours-logged)|
+--------+----------------+----------------+----------------+
|      12|     Paul Coddin|          135962|            2639|
|      10|George Vetticaden|          147150|            3232|
|      11|  Jamie Engesser|          179300|            3642|
+--------+----------------+----------------+----------------+

+--------+-----------+----------------+----------------+
|driverId|       name|sum(miles-logged)|sum(hours-logged)|
+--------+-----------+----------------+----------------+
|      12|Paul Coddin|          135962|            2639|
+--------+-----------+----------------+----------------+
```

DataFrame API: rename columns

Rename columns

```
%spark2.pyspark
joined = joined.withColumnRenamed('sum(miles-logged)', 'miles_logged')
joined.show(2)
```

```
+--------+-------------+------------+-----------------+
|driverId|         name|miles_logged|sum(hours-logged)|
+--------+-------------+------------+-----------------+
|      31|Rommel Garcia|      137057|             2704|
|      34| Frank Romano|      137728|             2811|
+--------+-------------+------------+-----------------+
only showing top 2 rows
```

DataFrame API: sorting

Order By

```
%spark2.pyspark
joined.orderBy(joined["driverId"].asc()).show(3)
```

```
+--------+-----------------+------------+-----------------+
|driverId|             name|miles_logged|sum(hours-logged)|
+--------+-----------------+------------+-----------------+
|      10|George Vetticaden|      147150|             3232|
|      11|   Jamie Engesser|      179300|             3642|
|      12|      Paul Coddin|      135962|             2639|
+--------+-----------------+------------+-----------------+
only showing top 3 rows
```

DataFrame API: new columns

New column from existing ones

```
%spark2.pyspark
joined.withColumn('miles_and_hours', joined['miles_logged'] + joined['sum(hours-logged)']).show(3)
```

```
+--------+---------------+------------+-----------------+---------------+
|driverId|           name|miles_logged|sum(hours-logged)|miles_and_hours|
+--------+---------------+------------+-----------------+---------------+
|      31|  Rommel Garcia|      137057|             2704|         139761|
|      34|   Frank Romano|      137728|             2811|         140539|
|      28|Olivier Renault|      137469|             2723|         140192|
+--------+---------------+------------+-----------------+---------------+
only showing top 3 rows
```

DataFrame API: new columns (cont-d)

Transformations with regular expressions

```
%spark2.pyspark
from pyspark.sql.functions import *
joined.withColumn("first_name", regexp_replace('name', "(.+) (.+)","$1")).show(3)
```

```
+--------+--------------+------------+-----------------+----------+
|driverId|          name|miles_logged|sum(hours-logged)|first_name|
+--------+--------------+------------+-----------------+----------+
|      31| Rommel Garcia|      137057|             2704|    Rommel|
|      34|  Frank Romano|      137728|             2811|     Frank|
|      28|Olivier Renault|      137469|             2723|   Olivier|
+--------+--------------+------------+-----------------+----------+
only showing top 3 rows
```

Took 1 sec. Last updated by anonymous at March 15 2019, 5:19:55 PM. (outdated)

Exercise:

In the Exercise 1 cell write the following queries using DataFrame API:

- How many weeks are logged?
- What is the week number with the maximum value of ratio of the average miles logged to average hours logged?
- Is it different from the week number with the maximum average of ratio of miles logged to hours logged?
- (If time permits): Add an extra column to the drivers DataFrame - it should show the first letter of the first name and last name (e.g., Rommel Garcia -> R. Garcia)

SQL way to work with DataFrames

Spark SQL: register tables first

```
%spark2.pyspark
drivers = spark.read.csv("/dsets/trucks/drivers.csv", header=True, inferSchema = "true")
timesheet = spark.read.csv("/dsets/trucks/timesheet.csv", header=True,  inferSchema = "true")
drivers.createOrReplaceTempView('drivers')
timesheet.createOrReplaceTempView('timesheet')
```

Took 52 sec. Last updated by anonymous at March 14 2019, 1:37:09 PM.

registerTempTable() method creates an in-memory view that is scoped to the current cluster and the current session.

SQL way

```
%spark2.pyspark
spark.sql("select * from drivers").show(2)
```

```
+--------+-----------------+---------+------------------+---------+---------+
|driverId|             name|      ssn|          location|certified|wage-plan|
+--------+-----------------+---------+------------------+---------+---------+
|      10|George Vetticaden|621011971|244-4532 Nulla Rd.|        N|    miles|
|      11|   Jamie Engesser|262112338|366-4125 Ac Street|        N|    miles|
+--------+-----------------+---------+------------------+---------+---------+
only showing top 2 rows
```

Drivers and timesheets exercise (SQL code):

Example in Spark SQL

```
%spark2.pyspark
spark.sql("SELECT driverId,\
                    SUM(`hours-logged`) as hours, \
                    SUM(`miles-logged`) as miles \
                    FROM timesheet \
                    GROUP BY driverId").createOrReplaceTempView('aggregated_ts');

spark.sql("SELECT drivers.driverId,\
                    drivers.name,\
                    aggregated_ts.* \
                    from drivers JOIN aggregated_ts ON drivers.driverId == aggregated_ts.driverId").show(2)
```

Data sources and sinks

Spark SQL supports the following sources and sinks of data:

- CSV files
- Json files
- XML files
- RDBMS sources
- RDDs
- Parquet files
- Other

Parquet files

Parquet is a columnar format that is supported by many other data processing systems. Spark SQL provides support for both reading and writing Parquet files that automatically preserves the schema of the original data. When writing Parquet files, all columns are automatically converted to be nullable for compatibility reasons[l].

Parquet files (cont-d)

Working with Parquet files: write to parquet

```
%spark2.pyspark
drivers.write.parquet("/__dsets/drivers.parquet")
```

Took 1 sec. Last updated by anonymous at March 12 2019, 1:39:53 PM.

Parquet files (cont-d)

```
[root@sandbox-hdp ~]# hadoop fs -ls /__dsets
Found 6 items
drwxr-xr-x   - root     hdfs          0 2019-02-13 14:28 /__dsets/census
drwxr-xr-x   - zeppelin hdfs          0 2019-03-12 16:25 /__dsets/drivers.parquet
drwxr-xr-x   - root     hdfs          0 2019-02-13 14:28 /__dsets/fuel
drwxr-xr-x   - root     hdfs          0 2019-02-13 14:28 /__dsets/logs
drwxr-xr-x   - root     hdfs          0 2019-02-21 16:34 /__dsets/textdata
drwxr-xr-x   - root     hdfs          0 2019-02-13 14:28 /__dsets/trucks
[root@sandbox-hdp ~]# hadoop fs -get /__dsets/drivers.parquet .
[root@sandbox-hdp ~]# cat drivers.parquet/
cat: drivers.parquet/: Is a directory
[root@sandbox-hdp ~]# cd drivers.parquet/
[root@sandbox-hdp drivers.parquet]# ls
part-00000-9ccff658-8ccb-4d42-a951-83993f6759e1-c000.snappy.parquet  _SUCCESS
[root@sandbox-hdp drivers.parquet]# du -h part-00000-9ccff658-8ccb-4d42-a951-83993f6759e1-c000.snappy.parquet
4.0K    part-00000-9ccff658-8ccb-4d42-a951-83993f6759e1-c000.snappy.parquet
[root@sandbox-hdp drivers.parquet]# head part-00000-9ccff658-8ccb-4d42-a951-83993f6759e1-c000.snappy.parquet
PAR1□□□□□□,□D□□□□□□+□
□□□D□


□□□□□□□□□□□□□ !"#$%&'()*+□,□D□□□□        Wes Floyd       Adam Diaz□□<□D□□George VetticadenJ▒└VT LF  E┼±LF□□□□□P▒┤┌ C□□□□┼□□!R□□ NVT LF └VT LF FF
ACR VT□ CLF□□□▒
         R□▒□├ B▒┐□▒□
T□└ MFFC┤FF▒□E□□□ MVT≥LF ┌┌     G□▒┼┤ LVT┤
□(mmel Garcia!K4Ryan Templeton!r@Sridhara Sabbella□,Frank Romano□(Emil Siemes□WAndrew !-8deOliviWes Floyd□$Scott Shaw□K|David Kaiser□Nicolas Maill

 P▒├┤□┼□□□□□□←□D□□□□□□□DF:□□)▒□□□□D□□□%Rw
                                  I0+>P□)▒□±▒1
```

[link]

Parquet files (cont-d)

Working with Parquet files: read from parquet

```
%spark2.pyspark
drv_parq = spark.read.parquet("/__dsets/drivers.parquet")
drv_parq.printSchema()
drivers.printSchema()
```

```
root
 |-- driverId: integer (nullable = true)
 |-- name: string (nullable = true)
 |-- ssn: integer (nullable = true)
 |-- location: string (nullable = true)
 |-- certified: string (nullable = true)
 |-- wage-plan: string (nullable = true)
root
 |-- driverId: integer (nullable = true)
 |-- name: string (nullable = true)
 |-- ssn: integer (nullable = true)
 |-- location: string (nullable = true)
 |-- certified: string (nullable = true)
 |-- wage-plan: string (nullable = true)
```

Save a DataFrame as CSV files

```
%spark2.pyspark
spark.sql("SELECT * FROM aggregated_ts").write.csv("/__dsets/agg")
```

Took 7 sec. Last updated by anonymous at March 14 2019, 4:57:47 PM. (outdated)

```
%sh
hadoop fs -ls /__dsets/agg
```

```
-rw-r--r--   1 zeppelin hdfs         15 2019-03-14 19:57 /__dsets/agg/part-00011-b4e14c38-9fee-4aa2-a235-56885188cb25-c000.csv
-rw-r--r--   1 zeppelin hdfs         15 2019-03-14 19:57 /__dsets/agg/part-00014-b4e14c38-9fee-4aa2-a235-56885188cb25-c000.csv
-rw-r--r--   1 zeppelin hdfs         30 2019-03-14 19:57 /__dsets/agg/part-00019-b4e14c38-9fee-4aa2-a235-56885188cb25-c000.csv
-rw-r--r--   1 zeppelin hdfs         15 2019-03-14 19:57 /__dsets/agg/part-00024-b4e14c38-9fee-4aa2-a235-56885188cb25-c000.csv
-rw-r--r--   1 zeppelin hdfs         15 2019-03-14 19:57 /__dsets/agg/part-00030-b4e14c38-9fee-4aa2-a235-56885188cb25-c000.csv
-rw-r--r--   1 zeppelin hdfs         15 2019-03-14 19:57 /__dsets/agg/part-00048-b4e14c38-9fee-4aa2-a235-56885188cb25-c000.csv
-rw-r--r--   1 zeppelin hdfs         15 2019-03-14 19:57 /__dsets/agg/part-00049-b4e14c38-9fee-4aa2-a235-56885188cb25-c000.csv
-rw-r--r--   1 zeppelin hdfs         30 2019-03-14 19:57 /__dsets/agg/part-00053-b4e14c38-9fee-4aa2-a235-56885188cb25-c000.csv
-rw-r--r--   1 zeppelin hdfs         15 2019-03-14 19:57 /__dsets/agg/part-00069-b4e14c38-9fee-4aa2-a235-56885188cb25-c000.csv
-rw-r--r--   1 zeppelin hdfs         15 2019-03-14 19:57 /__dsets/agg/part-00075-b4e14c38-9fee-4aa2-a235-56885188cb25-c000.csv
-rw-r--r--   1 zeppelin hdfs         30 2019-03-14 19:57 /__dsets/agg/part-00077-b4e14c38-9fee-4aa2-a235-56885188cb25-c000.csv
-rw-r--r--   1 zeppelin hdfs         15 2019-03-14 19:57 /__dsets/agg/part-00084-b4e14c38-9fee-4aa2-a235-56885188cb25-c000.csv
-rw-r--r--   1 zeppelin hdfs         15 2019-03-14 19:57 /__dsets/agg/part-00089-b4e14c38-9fee-4aa2-a235-56885188cb25-c000.csv
-rw-r--r--   1 zeppelin hdfs         15 2019-03-14 19:57 /  dsets/agg/part-00095-b4e14c38-9fee-4aa2-a235-56885188cb25-c000.csv
```

Save a DataFrame as CSV files (cont-d)

```
%sh
hadoop fs -cat /__dsets/agg/part-00163-b4e14c38-9fee-4aa2-a235-56885188cb25-c000.csv
```

```
11,3642,179300
33,2759,139285
```

Took 2 sec. Last updated by anonymous at March 14 2019, 9:40:01 PM. (outdated)

Save a DataFrame as CSV files (cont-d)

```
%spark2.pyspark
spark.sql("SELECT * FROM aggregated_ts").coalesce(1)\
                                        .write.csv("/__dsets/agg1")
```

Took 1 sec. Last updated by anonymous at March 14 2019, 9:48:02 PM. (outdated)

```
%sh
hadoop fs -ls /__dsets/agg1
```

```
Found 2 items
-rw-r--r--   1 zeppelin hdfs          0 2019-03-15 00:48 /__dsets/agg1/_SUCCESS
-rw-r--r--   1 zeppelin hdfs        510 2019-03-15 00:48 /__dsets/agg1/part-00000-7ae6019d-f083-4d7d-afc7-0d4adc8ce760-c000.csv
```

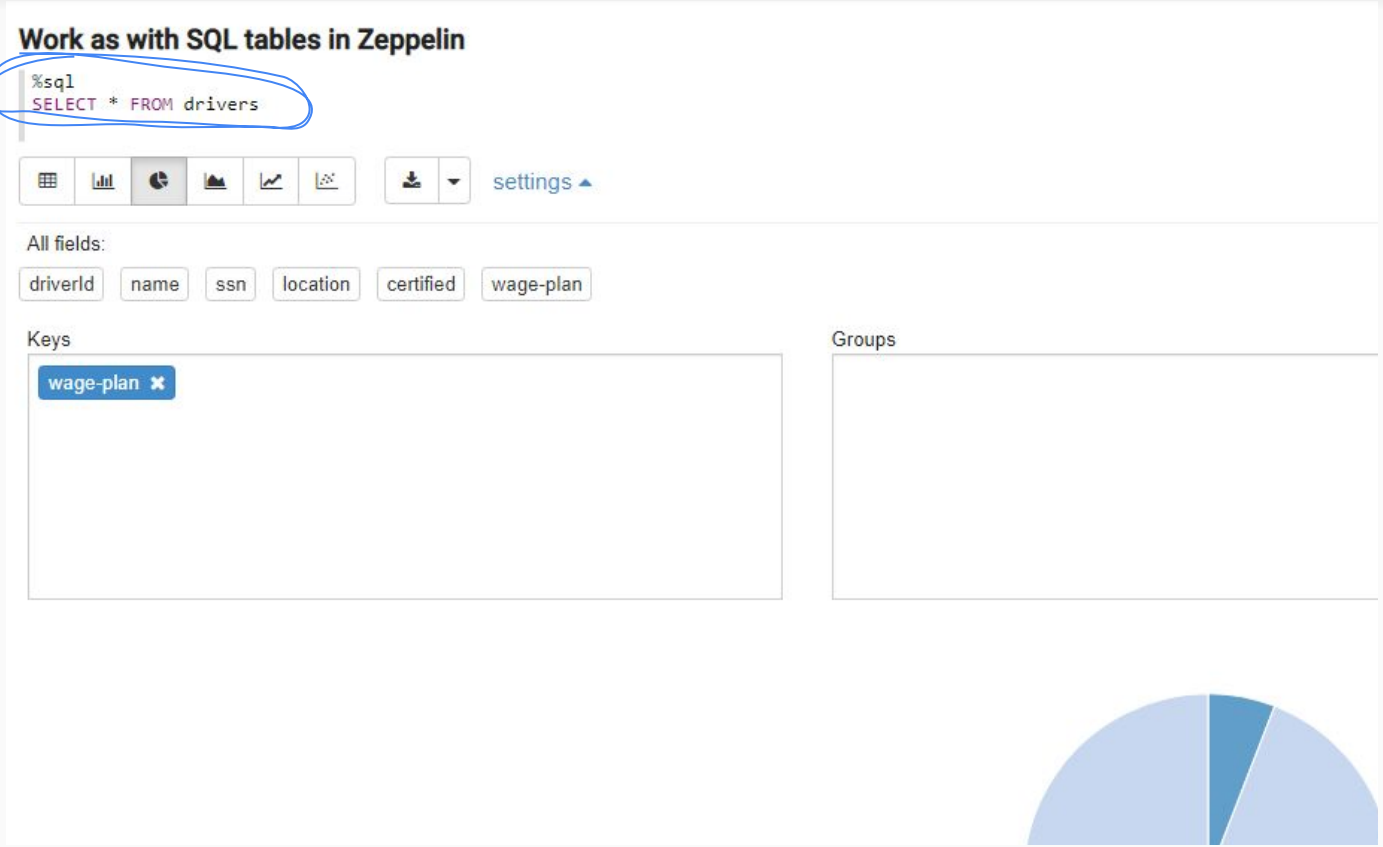
Took 2 sec. Last updated by anonymous at March 14 2019, 9:48:13 PM.

Exercise:

In the Exercise 2 cell write the following queries using Spark SQL API:

- **How many different wage plans are in our dataset?**
- **Compare wage plans: which one is more popular?**
- **What about value of ratio of the average miles logged to average hours logged within the wage plans?**
- **Is it different from average of ratio of miles logged to hours logged within the wage plans?**
- **Save results as parquet files**
- **Which one is easier to work with: SQL API or DataFrame API?**

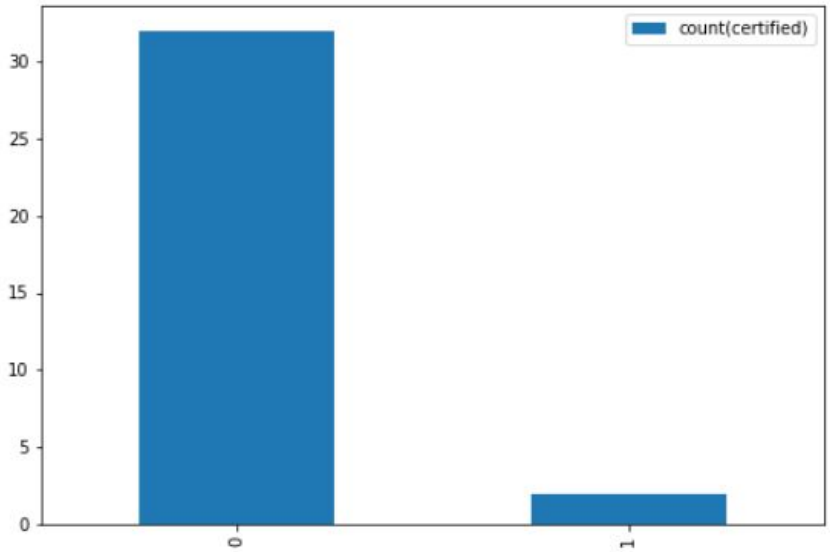
You can work with TempViews as with tables



Spark DataFrames and Pandas DataFrames

```
%spark2.pyspark
df = spark.sql("SELECT COUNT(certified) FROM drivers GROUP BY certified").toPandas()
df.plot.bar();
```

```
<matplotlib.axes._subplots.AxesSubplot object at 0x7fea922ab150>
```



After converting Spark DataFrame to Pandas DataFrame you can use all benefits of Pandas DataFrame including powerful visualization tools like PyPlot, machine learning libraries for Python, etc.

Integrating with Hive

You can load data from Hive tables and save DataSets as Hive tables

- saveAsTable() method creates a permanent, physical table stored in the external datastore (HDFS/S3/etc.) using the Parquet format. Table **metadata** is stored in *Hive metastore*. The table is accessible from any application that has access to the cluster
- To load data from a Hive table you can just use a Spark SQL API if Hive support is enabled in your SparkSession (e.g, spark.sql("SELECT * FROM table_in_hive"));

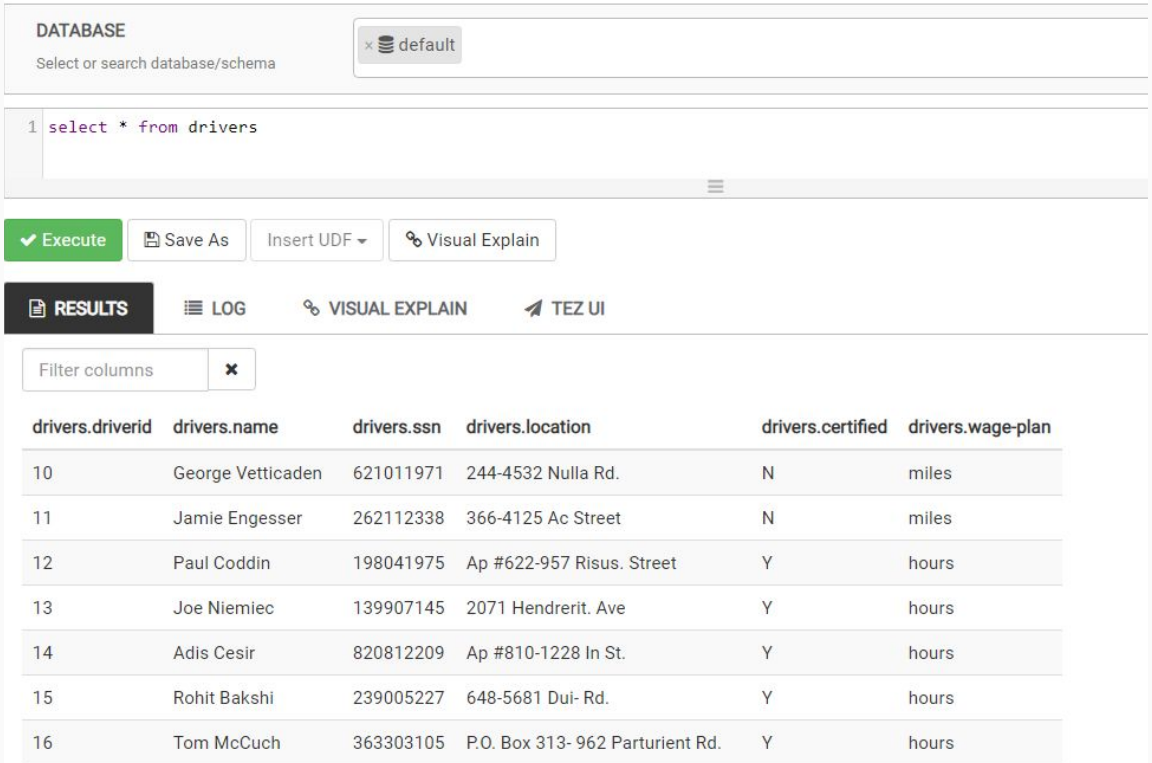
[link]

Save table in Hive metastore (permanently)

```
%spark2.pyspark
drivers.write.mode("overwrite").saveAsTable("drivers")
```

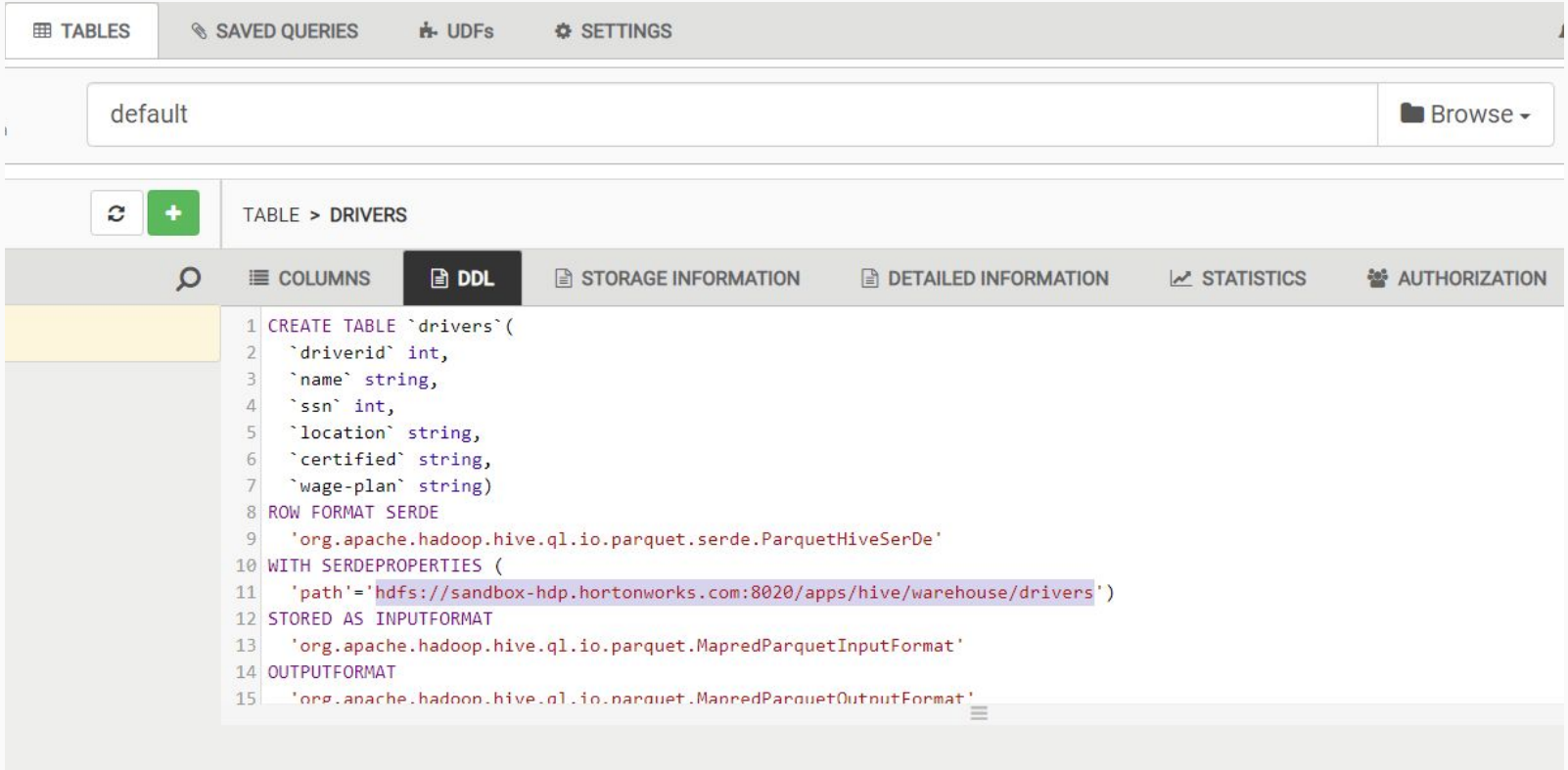
Took 1 sec. Last updated by anonymous at March 12 2019, 2:16:43 PM.

Save table in Hive metastore (permanently)

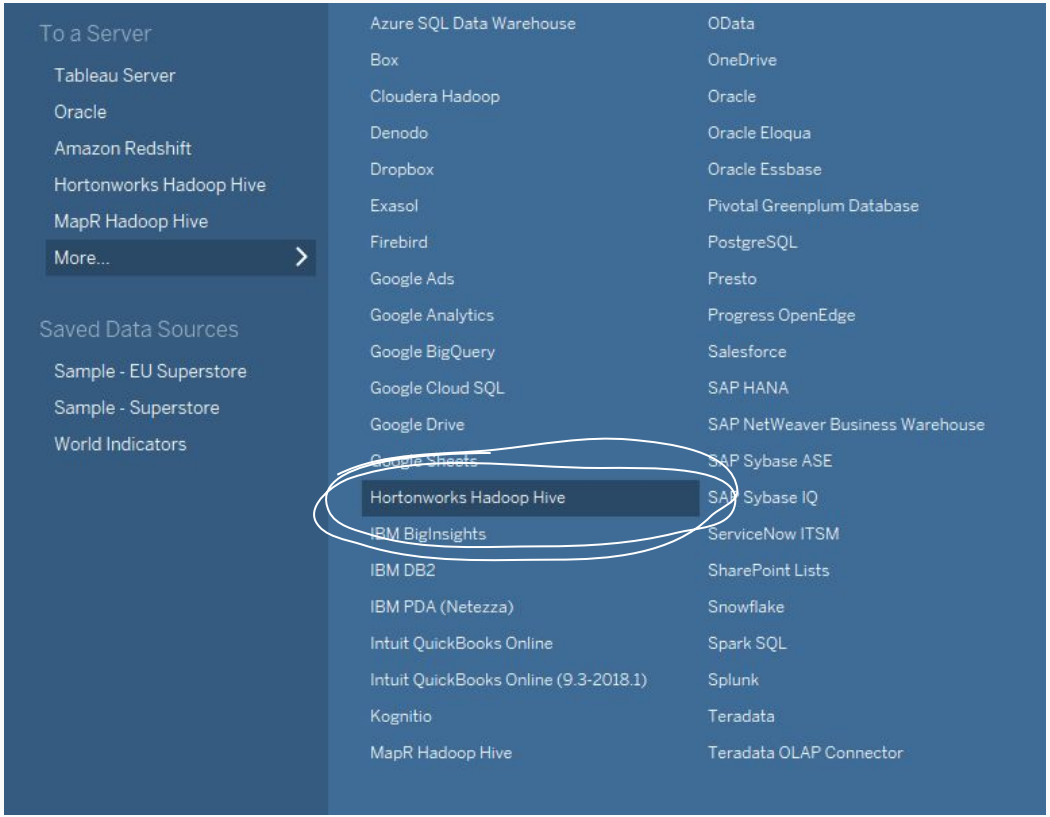


drivers.driverid	drivers.name	drivers.ssn	drivers.location	drivers.certified	drivers.wage-plan
10	George Vetticaden	621011971	244-4532 Nulla Rd.	N	miles
11	Jamie Engesser	262112338	366-4125 Ac Street	N	miles
12	Paul Coddin	198041975	Ap #622-957 Risus. Street	Y	hours
13	Joe Niemiec	139907145	2071 Hendrerit. Ave	Y	hours
14	Adis Cesir	820812209	Ap #810-1228 In St.	Y	hours
15	Rohit Bakshi	239005227	648-5681 Dui- Rd.	Y	hours
16	Tom McCuch	363303105	P.O. Box 313- 962 Parturient Rd.	Y	hours

Save table in Hive metastore (permanently)



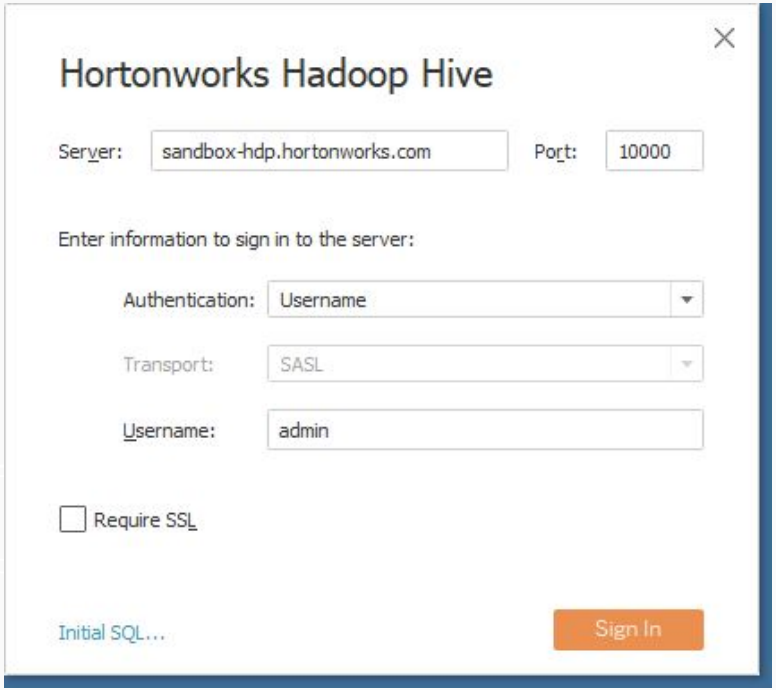
Connect to Tableau



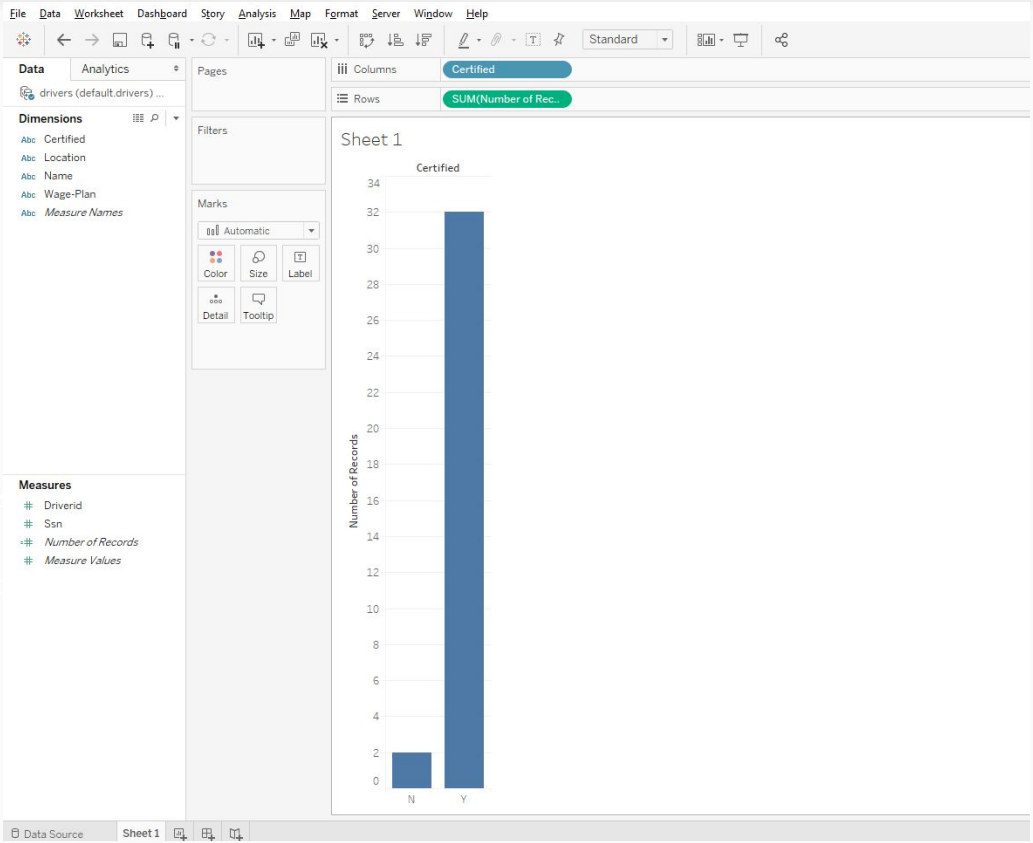
Note: you should download and install a Hive ODBC driver from https://hortonworks.com/downloads/

/HDP Add-Ons section/

Connect to Tableau (cont-d)



Connect to Tableau (cont-d)



Structured Streaming

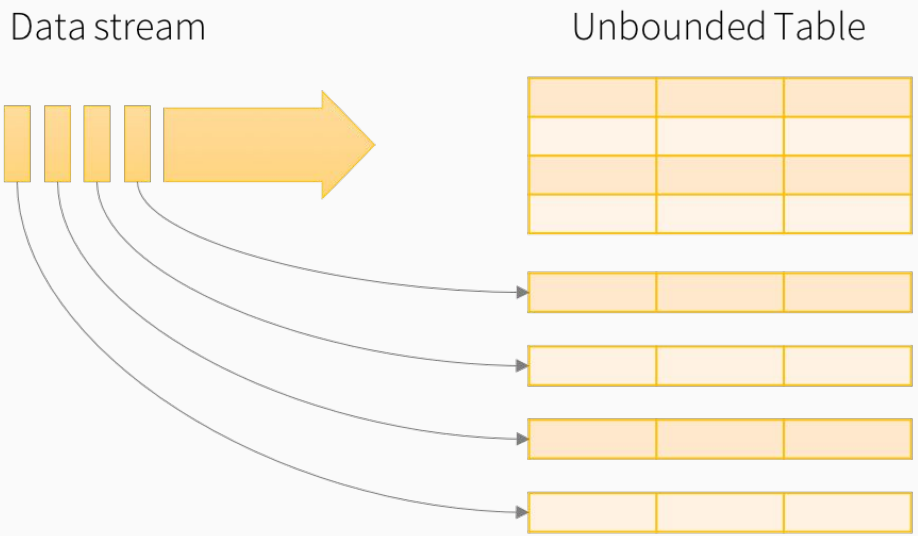
“Structured Streaming is a scalable and fault-tolerant stream processing engine built on the Spark SQL engine. You can express your streaming computation the ***same way you would express a batch computation on static data***. The Spark SQL engine will take care of running it incrementally and continuously and updating the final result as streaming data continues to arrive. You can use the Dataset/DataFrame API in Scala, Java, Python or R to express streaming aggregations, event-time windows, stream-to-batch joins, etc. The computation is executed on the same optimized Spark SQL engine” [I].

Structured Streaming: major features

Major features:

- Uses micro-batches
- Fault tolerance is ensured by using *checkpoints* and *write-ahead logging*
- Spark Structured Streaming runs incremental queries on the *unbounded input table*

Structured Streaming: Programming model



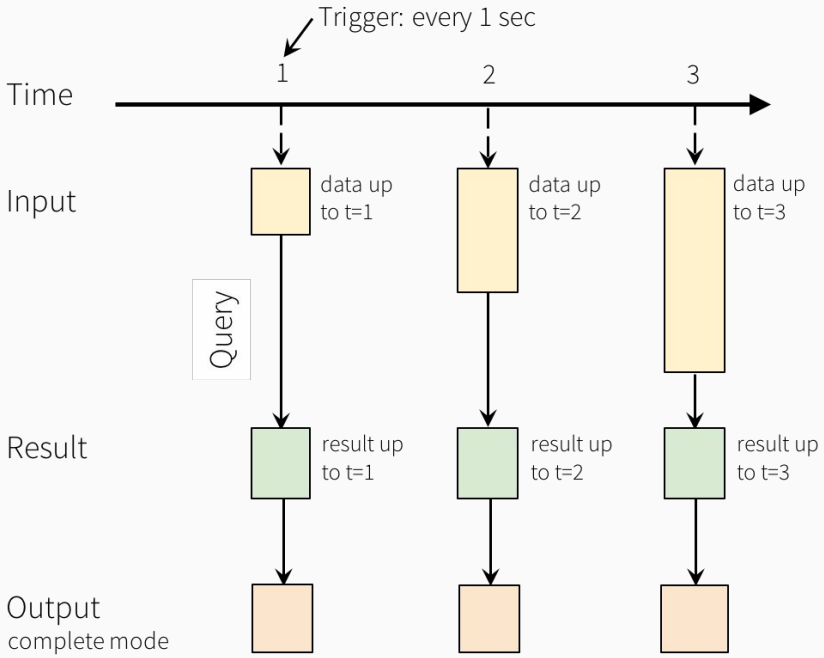
new data in the
data stream
=
new rows appended
to a unbounded table

[link]

Data stream as an unbounded table

Structured Streaming: Progr. model (cont-d)

A query on the input will generate the "**Result Table**". Every trigger interval (say, every 1 second), new rows get appended to the Input Table, which eventually updates the Result Table. Whenever the result table gets updated, we would want to write the changed result rows to an external sink [].



Programming Model for Structured Streaming [link]

Structured Streaming: Modes of output

The “Output” is defined as what gets written out to the **external storage (HDFS/other DFS).** Modes:

- **Complete Mode** - The entire updated Result Table will be written to the external storage.
- **Append Mode** - Only the new rows appended in the Result Table since the last trigger will be written to the external storage.
- **Update Mode** - Only the rows that were **updated** in the Result Table since the last trigger will be written to the external storage (available since Spark 2.1.1)[I].

NOTE: available modes of output depend on type of your queries and on output sinks

Structured Streaming: Modes of output (cont-d)

Query types and output modes. More: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

Query Type		Supported Output Modes	Notes
Queries with aggregation	Aggregation on event-time with watermark	Append, Update, Complete	Append mode uses watermark to drop old aggregation state. But the output of a windowed aggregation is delayed the late threshold specified in `withWatermark()` as by the modes semantics, rows can be added to the Result Table only once after they are finalized (i.e. after watermark is crossed). See the Late Data section for more details. Update mode uses watermark to drop old aggregation state. Complete mode does not drop old aggregation state since by definition this mode preserves all data in the Result Table.
	Other aggregations	Complete, Update	Since no watermark is defined (only defined in other category), old aggregation state is not dropped. Append mode is not supported as aggregates can update thus violating the semantics of this mode.
Queries with `mapGroupsWithState`		Update	
Queries with `flatMapGroupsWithState`	Append operation mode	Append	Aggregations are allowed after `flatMapGroupsWithState`.
	Update operation mode	Update	Aggregations not allowed after `flatMapGroupsWithState`.
Queries with `joins`		Append	Update and Complete mode not supported yet. See the support matrix in the Join Operations section for more details on what types of joins are supported.
Other queries		Append, Update	Complete mode not supported as it is infeasible to keep all unaggregated data in the Result Table.

[link]

Structured Streaming: sources of data

There are the following built-in sources:

- **File source** - Reads files written in a directory as a stream of data. Supported file formats are text, csv, json, orc, parquet. See the docs of the DataStreamReader interface for a more up-to-date list, and supported options for each file format. Note that the files must be atomically placed in the given directory, which in most file systems, can be achieved by file move operations.
- **Kafka source** - Reads data from Kafka. It's compatible with Kafka broker versions 0.10.0 or higher.
- **Socket source (for testing)** - Reads UTF8 text data from a socket connection. The listening server socket is at the driver. Note that this should be used only for testing as this does not provide end-to-end fault-tolerance guarantees.
- **Rate source (for testing)** - Generates data at the specified number of rows per second, each output row contains a timestamp and value. Where timestamp is a Timestamp type containing the time of message dispatch, and value is of Long type containing the message count, starting from 0 as the first row. This source is intended for testing and benchmarking[!].

Structured Streaming: output sinks

Sink	Supported Output Modes	Options	Fault-tolerant	Notes
File Sink	Append	`path`: path to the output directory, must be specified. For file-format-specific options, see the related methods in DataFrameWriter (Scala/Java/Python/R). E.g. for "parquet" format options see `DataFrameWriter.parquet()`	Yes (exactly-once)	Supports writes to partitioned tables. Partitioning by time may be useful.
Kafka Sink	Append, Update, Complete	See the Kafka Integration Guide	Yes (at-least-once)	More details in the Kafka Integration Guide
Foreach Sink	Append, Update, Complete	None	Depends on ForeachWriter implementation	More details in the next section
ForeachBatch Sink	Append, Update, Complete	None	Depends on the implementation	More details in the next section
Console Sink	Append, Update, Complete	`numRows`: Number of rows to print every trigger (default: 20) `truncate`: Whether to truncate the output if too long (default: true)	No	
Memory Sink	Append, Complete	None	No. But in Complete Mode, restarted query will recreate the full table.	Table name is the query name.

[link]

Structured Streaming: example

Demo Structured Streaming Application:

Login to http://sandbox-hdp.hortonworks.com:4200/ and get it with the command:

wget wolly.cs.smu.ca/tmp/strustr.py

And run:

spark submit strustr.py

Important notes:

- You need to define a **schema** for input files (no automatic inference by default)
- Before running the app create an input files directory in HDFS:

hadoop fs -mkdir /__dsets/s_out

```
  GNU nano 2.5.3         File: /var/www/html/tmp/strustr.py
from pyspark.sql.types import StructType
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName('StructuredStreamingExample').getOrCreate()
spark.sparkContext.setLogLevel("ERROR")
schema = StructType().add("driverId", "integer")\
                     .add("week", "integer")\
                     .add("hours-logged", "integer")\
                     .add("miles-logged", "integer")
df = spark \
    .readStream \
    .option("sep", ",") \
    .schema(schema) \
    .csv("/__dsets/s_out")

out = df.groupby('week').agg({'hours-logged': 'sum', 'miles-logged': 'sum'})

out = out.writeStream \
    .format("console") \
    .outputMode("update") \
    .start()

out.awaitTermination()
```

Structured Streaming: example (cont-d)

Python script that writes chunks of the input CSV file to HDFS with defined interval:

Login to http://sandbox-hdp.hortonworks.com:4200/ **in another tab** and get it with the command:

wget wolly.cs.smu.ca/tmp/csvsender.py

Get a CSV file to read:

wget wolly.cs.smu.ca/tmp/timesheet.csv

Run the script (read the next 10 lines from csvsender.py every 5 seconds):

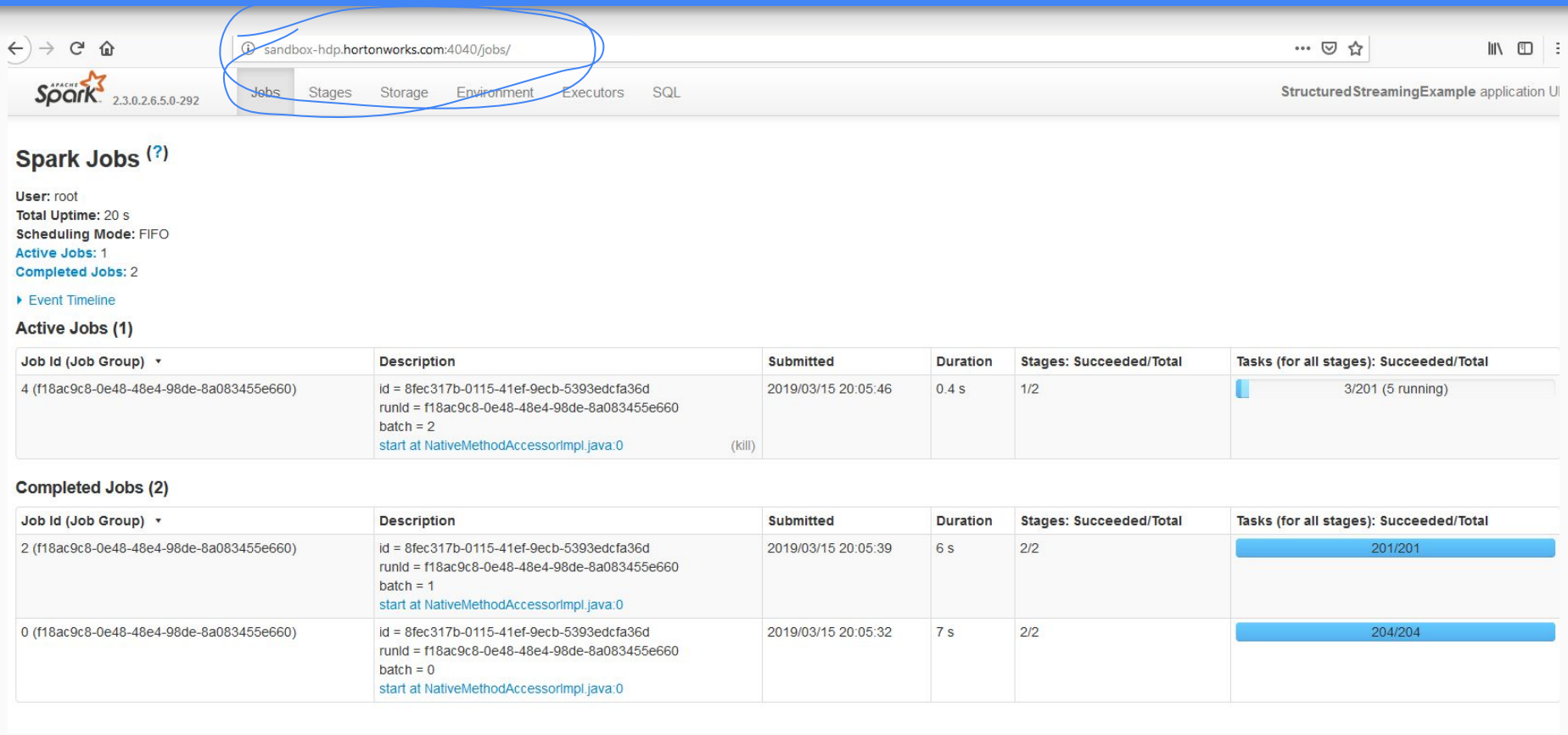
python csvsender.py timesheet.csv 10 5

```
  GNU nano 2.5.3                File: csvsender.py                Modifi

import time, sys, os

f=open(sys.argv[1], "r")
nlines = int(sys.argv[2])
interval = int(sys.argv[3])
content = f.read().splitlines()
if not os.path.exists("out"):
    os.makedirs("out")

idx = 0
leng = len(content)
while True:
    if idx+nlines > leng:
        idx = 0

    chunk = content[idx:idx+nlines]
    str_chunk = "\n".join(chunk)
    chunk_name = "out/part" + str(time.time()) + ".csv"
    chunk_file = open(chunk_name, "w")
    chunk_file.write(str_chunk)
    chunk_file.close()

    os.system("hadoop fs -put " + chunk_name + " /__dsets/s_out; rm " + chunk_name)
    idx += nlines

    time.sleep(interval)

p.close()


^G Get Help    ^O Write Out    ^W Where Is    ^K Cut Text    ^J Justify    ^C Cur Pos
```

Structured Streaming: example (cont-d)

```
19/03/15 20:01:57 INFO ContextHandler: Started o.s.j.s.ServletContextHandler@302c2a99{/SQL,null,AVAILABLE,@Spark}
19/03/15 20:01:57 INFO ContextHandler: Started o.s.j.s.ServletContextHandler@52e21116{/SQL/json,null,AVAILABLE,@Spark}
19/03/15 20:01:57 INFO ContextHandler: Started o.s.j.s.ServletContextHandler@53d03d49{/SQL/execution,null,AVAILABLE,@Spark}
19/03/15 20:01:57 INFO ContextHandler: Started o.s.j.s.ServletContextHandler@44aa5683{/SQL/execution/json,null,AVAILABLE,@Spark}
19/03/15 20:01:57 INFO ContextHandler: Started o.s.j.s.ServletContextHandler@3cab5200{/static/sql,null,AVAILABLE,@Spark}
19/03/15 20:01:57 INFO StateStoreCoordinatorRef: Registered StateStoreCoordinator endpoint
-------------------------------------------
Batch: 0
-------------------------------------------
+----+------------------+------------------+
|week|sum(miles-logged)|sum(hours-logged)|
+----+------------------+------------------+
|  31|             35853|               740|
|  34|             36921|               739|
|  28|             34666|               690|
|  26|             35535|               712|
|  27|             35924|               717|
|  44|             36863|               723|
|  12|             39140|               746|
|  22|             38503|               781|
|  47|             36562|               748|
|null|              null|              null|
|   1|             38325|               727|
|  52|             19131|               400|
|  13|             38627|               791|
|   6|             38787|               749|
|  16|             35008|               705|
|   3|             38317|               781|
|  40|             35967|               760|
|  20|             37261|               752|
|  48|             36702|               735|
|   5|             39268|               779|
+----+------------------+------------------+
only showing top 20 rows

-------------------------------------------
Batch: 1
-------------------------------------------
+----+------------------+------------------+
```

Structured Streaming: example (cont-d)



End of the class

Thanks!