Module 11

Spark SQL and Spark Streaming

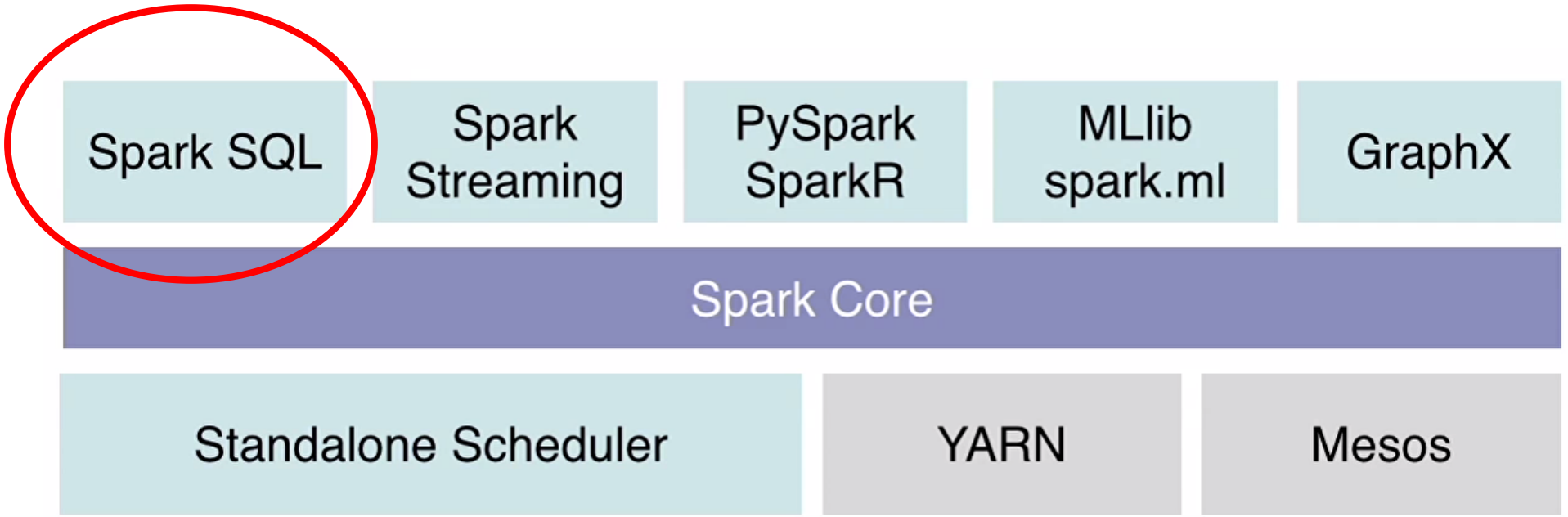
Thanachart Numnonda, Executive Director, IMC Institute

Mr.Aekanun Thongtae, Big Data Consultant, IMC Institute

Thanisa Numnonda, Faculty of Information Technology,

King Mongkut's Institute of Technology Ladkrabang

Spark Platform



Introduction to SparkSQL

- Spark SQL is a Spark module for structured data processing

- Spark SQL supports the execution of SQL queries written using either a basic SQL syntax or HiveQL
  - Catalyst optimizer: built inside Spark SQL:
    - Built-in mechanism to fetch data from some external data sources. For example, JSON, JDBC, Parquet, MySQL, Hive, PostgreSQL, HDFS, S3, and so on
    - Designed to optimize all phases of query execution: analysis, logical optimization, physical planning, and code generation to compile parts of queries to Java bytecode.

- Three ways to interact with Spark SQL: SQL, DataFrame API and Dataset API.

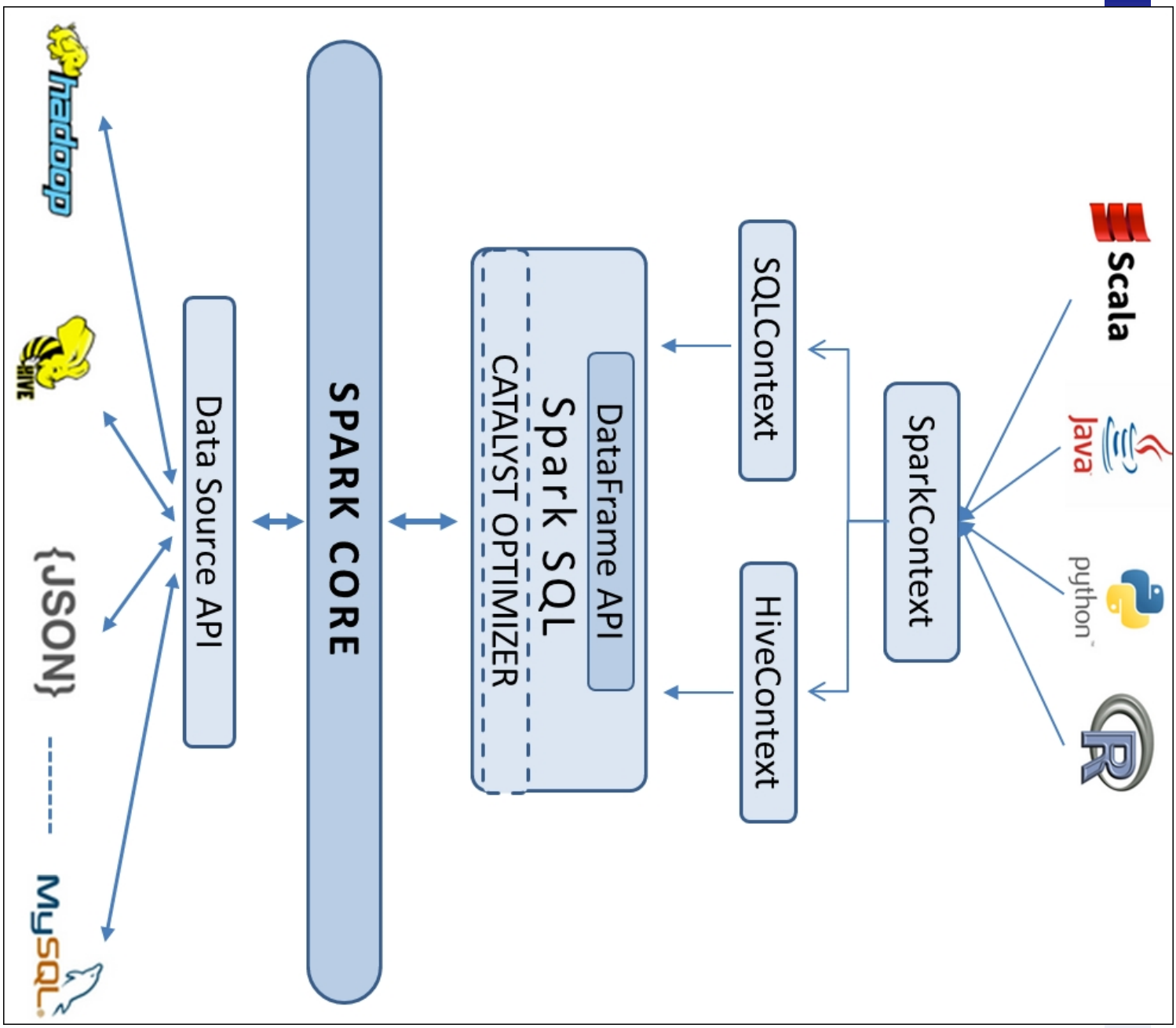
Introduction to DataFrame

- DataFrame is an immutable distributed collection of data. Unlike an RDD, data is organized into named columns, like a table in a relational database.

- DataFrame API was built as one more level of abstraction on top of Spark SQL.
  - Allow you use a two-dimensional data structure that usually has labelled rows and columns is called a DataFrame (R, Python-Pandas)

- The DataFrame API builds on the Spark SQL query optimizer to automatically execute code efficiently on a cluster of machines.

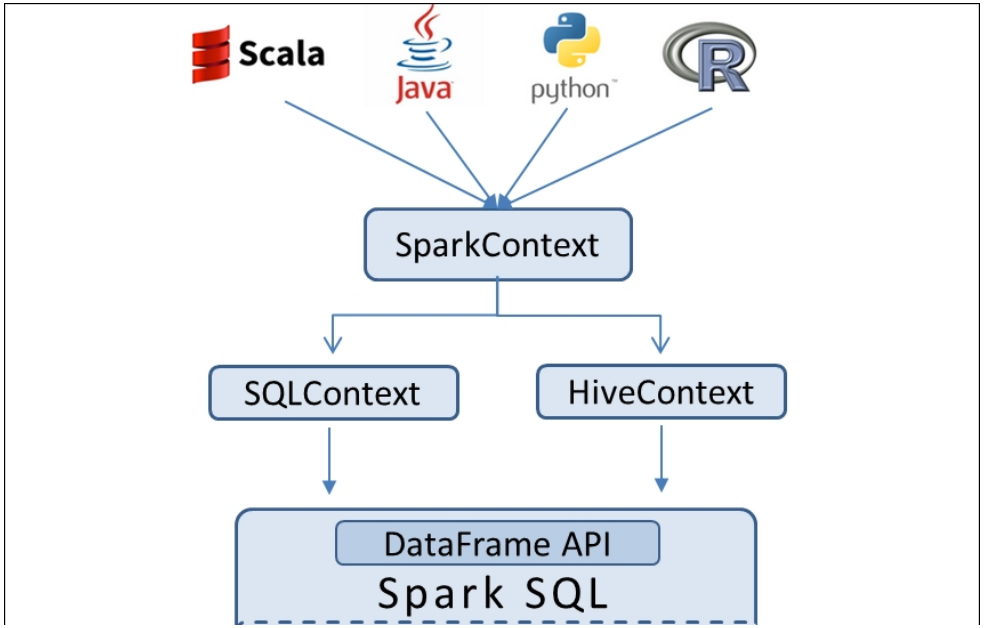
Architecture: Spark SQL & DataFrame



Source: Srinivas Duvvuri and Bikramaditya Singhal,"Spark for Data Science," Packt Publishing, 2016

Get Access to the SparkSQL

- Use DataFrame API to entry point:
  - SQLContext or
  - HiveContext
- SQLContext: RDDs, JSON, JDBC
- HiveContext: Hive Tables



When to use RDDs?

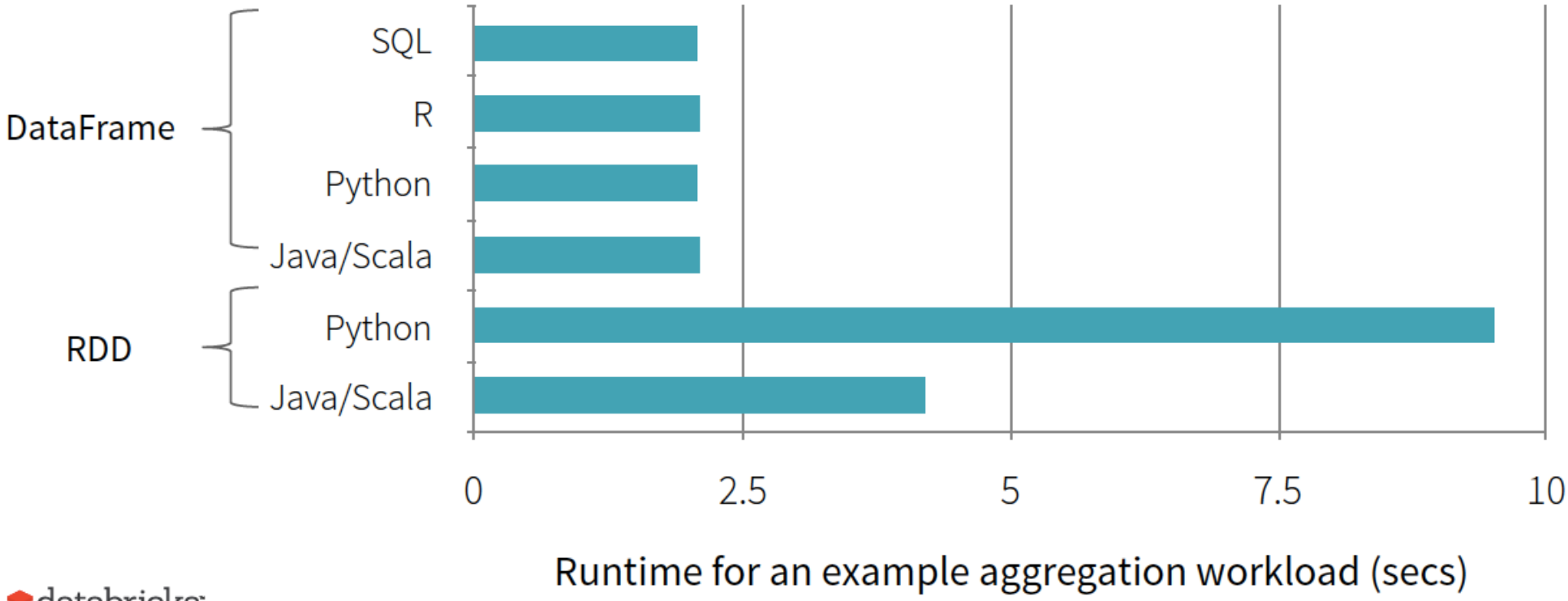
- Low-level transformation and actions and control on your dataset
- Data is unstructured, such as media streams or streams of text
- Manipulate your data with functional programming constructs than domain specific expressions
- Don't care about imposing a schema, such as columnar format, while processing or accessing data attributes by name or column
- Forgo some optimization and performance benefits available with DataFrames and Datasets for structured and semi-structured data

Adapted from databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html

RDDs vs. DataFrames: Differences

- DataFrames are a higher-level abstraction than RDDs.
- The definition of RDD implies defining a Directed Acyclic Graph (DAG) whereas defining a DataFrame leads to the creation of an Abstract Syntax Tree (AST). An AST will be utilized and optimized by the Spark SQL catalyst engine.
- RDD is a general data structure abstraction whereas a DataFrame is a specialized data structure to deal with two-dimensional, table-like data.

Benefit of Logical Plan: Performance Parity Across Languages



databricks

Source: Jump start into Apache Spark and Databricks

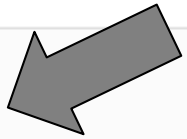
Creating DataFrames: RDDs

Python:

```
   //Create a list of colours
>>> colors = ['white','green','yellow','red','brown','pink']
//Distribute a local collection to form an RDD
//Apply map function on that RDD to get another RDD containing colour, length tuples
>>> color_df = sc.parallelize(colors)
        .map(lambda x:(x,len(x))).toDF(["color","length"])

>>> color_df
DataFrame[color: string, length: bigint]

>>> color_df.dtypes        //Note the implicit type inference
[('color', 'string'), ('length', 'bigint')]

>>> color_df.show()  //Final output as expected. Order need not be the same as shown
+------+------+
| color|length|
+------+------+
| white|     5|
| green|     5|
|yellow|     6|
|   red|     3|
| brown|     5|
|  pink|     4|
+------+------+
```

Creating DataFrames: JSON

Python:



HDFS (Hadoop Ecosystem)

```
//Pass the source json data file path
>>> df = sqlContext.read.json("./authors.json")
>>> df.show() //json parsed; Column names and data    types inferred implicitly
+----------+---------+
|first_name|last_name|
+----------+---------+
|      Mark|    Twain|
|   Charles|  Dickens|
|    Thomas|    Hardy|
+----------+---------+
```

Creating DataFrames: JDBC

Python:

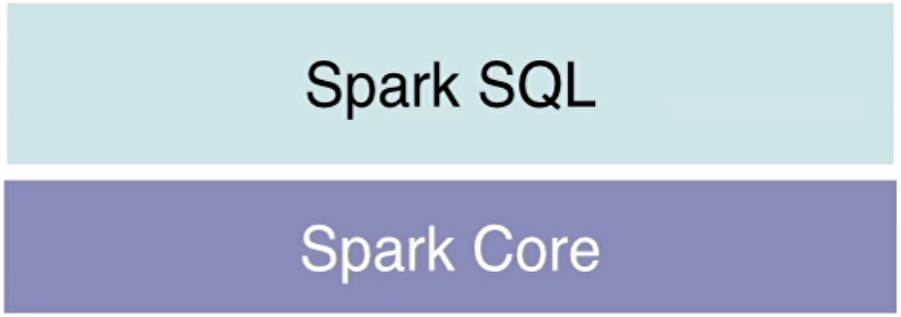
```
//Launch shell with driver-class-path as a command line argument
pyspark --driver-class-path /usr/share/   java/mysql-connector-java.jar
   //Pass the connection parameters
>>> peopleDF = sqlContext.read.format('jdbc').options(
                        url = 'jdbc:mysql://localhost',
                        dbtable = 'test.people',
                        user = 'root',
                        password = 'mysql').load()
   //Retrieve table data as a DataFrame
>>> peopleDF.show()
+----------+---------+------+----------+----------+---------+
|first_name|last_name|gender|       dob|occupation|person_id|
+----------+---------+------+----------+----------+---------+
|    Thomas|    Hardy|     M|1840-06-02|    Writer|      101|
|     Emily|   Bronte|     F|1818-07-30|    Writer|      102|
| Charlotte|   Bronte|     F|1816-04-21|    Writer|      103|
|   Charles|  Dickens|     M|1812-02-07|    Writer|      104|
+----------+---------+------+----------+----------+---------+
```

SparkSQL can leverage the Hive metastore

- Hive Metastore can also be leveraged by a wide array of applications
  - Hive
  - Impala
  - Spark
- Available from HiveContext

SparkSQL: HiveContext

```
context =    HiveContext(sc)

# query with SQL
results = context.sql(
  "SELECT * FROM people")

# apply Python transformation
names = results.map(lambda p: p.name)
```



Hands-on: DataFrames Operations

(LAB 1)

DataFrames Operations

Create a local collection of colors first

```
>>> colors = ['white','green','yellow','red','brown','pink']
```

Distribute the local collection to form an RDD
Apply map function on that RDD to get another RDD containing colour, length tuples and convert that RDD to a DataFrame

```
>>> color_df = sc.parallelize(colors)
      .map(lambda x:(x,len(x))).toDF(['color','length'])
```

Check the object type

```
>>> color_df
```

DataFrames Operations

Check the schema.

```
>>> color_df.printSchema()
>>> color_df.dtypes
```

Check row count

```
>>> color_df.count()
```

Look at the table contents. You can limit displayed rows by passing parameter to show .

```
>>> color_df.show(2)
```

DataFrames Operations

List out column names.

```
>>> color_df.columns
```

Drop a column. The source DataFrame color_df remains the same. Spark returns a new DataFrame which is being passed to show.

```
>>> color_df.drop('length').show()
```

Convert to JSON format.

```
>>> color_df.toJSON().first()
```

DataFrames Operations

Filter operation is similar to WHERE clause in SQL.
You specify conditions to select only desired rows.

Output of filter operation is another DataFrame object that is usually passed on to some more operations.

The following example selects the colors having a length of four or five only and label the column as "mid_length" filter.

```
>>> color_df.filter(color_df.length.between(4,5))
      .select(color_df.color.alias("mid_length")).show()
```

Adapted from Srinivas Duvvuri and Bikramaditya Singhal,"Spark for Data Science," Packt Publishing, 2016

DataFrames Operations

This example uses multiple filter criteria.

```
>>> color_df.filter(color_df.length > 4).filter(color_df[0]!="white").show()
```

Sort the data on one or more columns sort.
A simple single column sorting in default (ascending) order.

```
>>> color_df.sort('length').show()
```

You can use orderBy instead, which

```
>>> color_df.orderBy('length').show()
```

DataFrames Operations

First filter colors of length more than or equal to 4 and then sort on the column length in descending order.

```
>>> color_df.filter(color_df['length']>=4).sort('length',
ascending=False).show()
```

DataFrames Operations

GroupBy

```
>>> color_df.groupBy('length').count().show()
```

Basic Stat.

```
>>> color_df.describe().show()
```

DataFrames Operations for Data Prep.

(LAB 2)

DataFrames Operations for Data Prep.

Define a dataset.

```
from pyspark.sql.functions import *

# Define a dataset.

df = sc.parallelize([(10, '', 10000), (20, 'Female', 30000),
(None, 'Male', 80000), (None, 'Male', 5000)]).toDF(["age",
"gender", "income"])

df.show()
```

DataFrames Operations for Data Prep.

Data Cleansing: Null

```
# Treat Null Value (None) with Average one.

df.na.drop().agg(avg("age")).collect()[0][0]

no_null_df = df.na.fill(df.na.drop().agg(avg("age")).collect()[0][0])

no_null_df.show()
```

DataFrames Operations for Data Prep.

Data Cleansing: Outliners

```
# Treat Outliner with Remove one.

crunched_df = no_null_df.filter(col('income') >= 10000)

crunched_df.show()
```

DataFrames Operations for Data Prep.

Data Cleansing: Missing Values

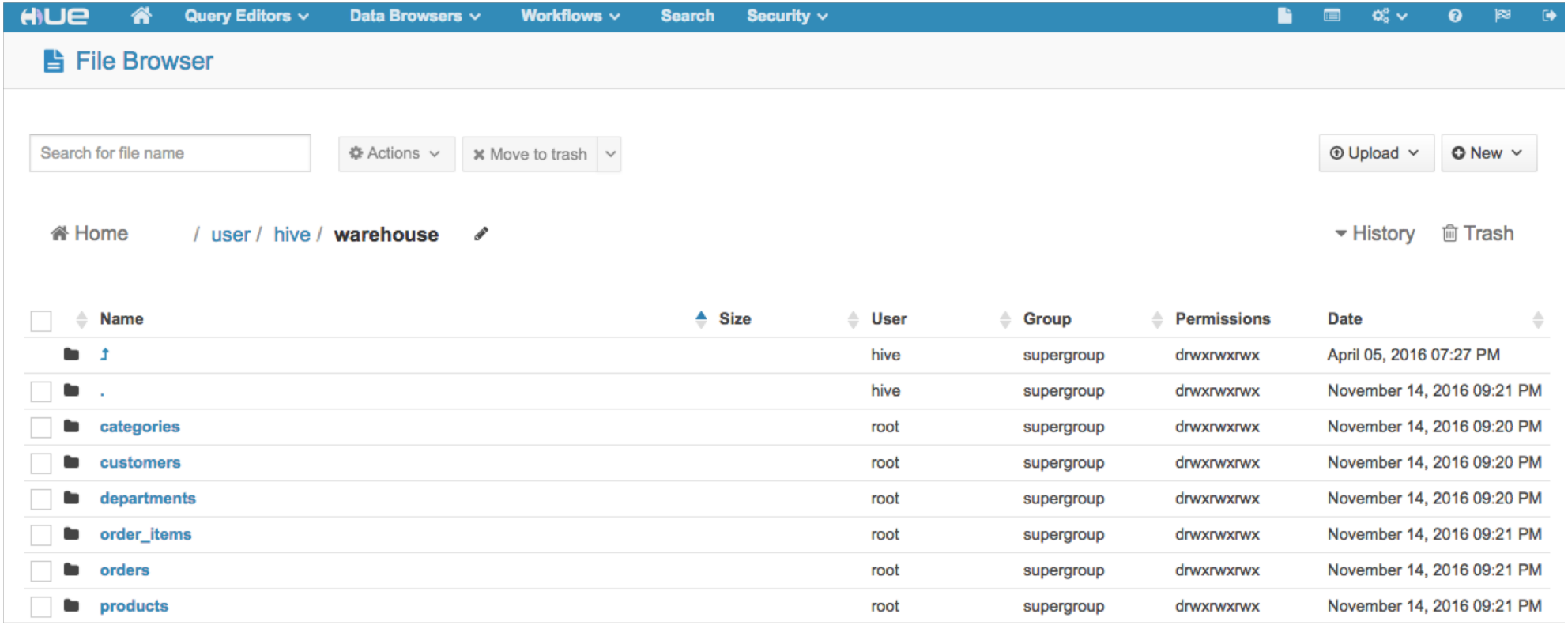
```
# Treat Missing Value with Defined Values.

treat_missing = udf(lambda x: "Male_Assume" if x == "" else x)

crunched_df.withColumn('new_gender',treat_missing(crunched_df.
gender)).show()
```

Create DataFrames from Hive Tables

(LAB 3)

Importing all tables to Hive with Compression

```
$sqoop import  --connect
"jdbc:mysql://quickstart.cloudera:3306/retail_db"  --username
root --password cloudera --table orders --hive-import --hive-
overwrite  --create-hive-table -m 1
```



Link Hive Metastore with Spark-Shell

Copy the configuration file

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

Running Spark

```
$ pyspark
```

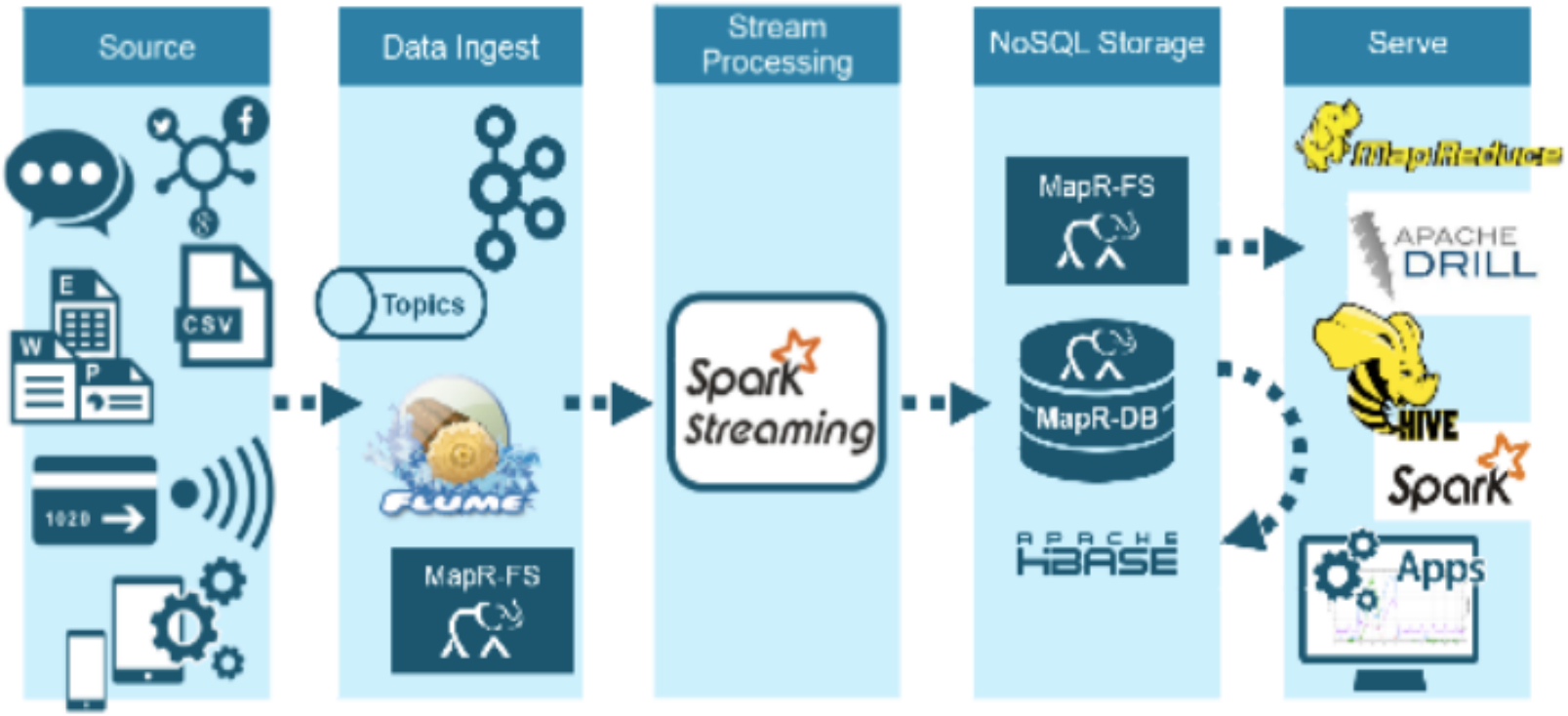
```
>>> sqlContext = HiveContext(sc)
>>> result = sqlContext.sql("SELECT * FROM orders")
>>> result.show()
```

order_id	order_date	order_customer_id	order_status
1	2013-07-25 00:00:...	11599	CLOSED
2	2013-07-25 00:00:...	256	PENDING_PAYMENT
3	2013-07-25 00:00:...	12111	COMPLETE
4	2013-07-25 00:00:...	8827	CLOSED
5	2013-07-25 00:00:...	11318	COMPLETE
6	2013-07-25 00:00:...	7130	COMPLETE
7	2013-07-25 00:00:...	4530	COMPLETE
8	2013-07-25 00:00:...	2911	PROCESSING
9	2013-07-25 00:00:...	5657	PENDING_PAYMENT
10	2013-07-25 00:00:...	5648	PENDING_PAYMENT
11	2013-07-25 00:00:...	918	PAYMENT_REVIEW
12	2013-07-25 00:00:...	1837	CLOSED
13	2013-07-25 00:00:...	9149	PENDING_PAYMENT
14	2013-07-25 00:00:...	9842	PROCESSING
15	2013-07-25 00:00:...	2568	COMPLETE
16	2013-07-25 00:00:...	7276	PENDING_PAYMENT
17	2013-07-25 00:00:...	2667	COMPLETE
18	2013-07-25 00:00:...	1205	CLOSED
19	2013-07-25 00:00:...	9488	PENDING_PAYMENT
20	2013-07-25 00:00:...	9198	PROCESSING

only showing top 20 rows

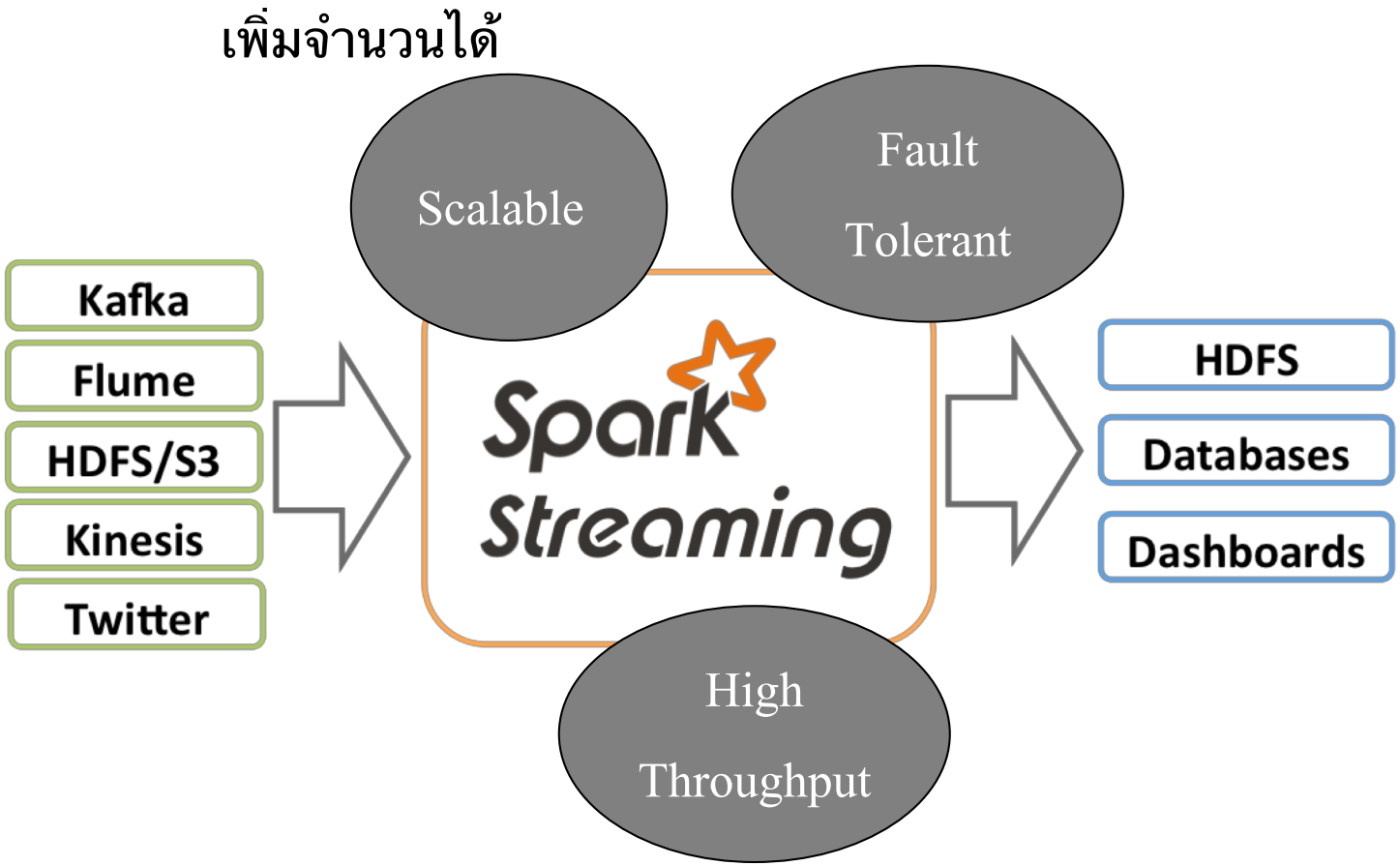
Spark Streaming

Stream Process Architecture



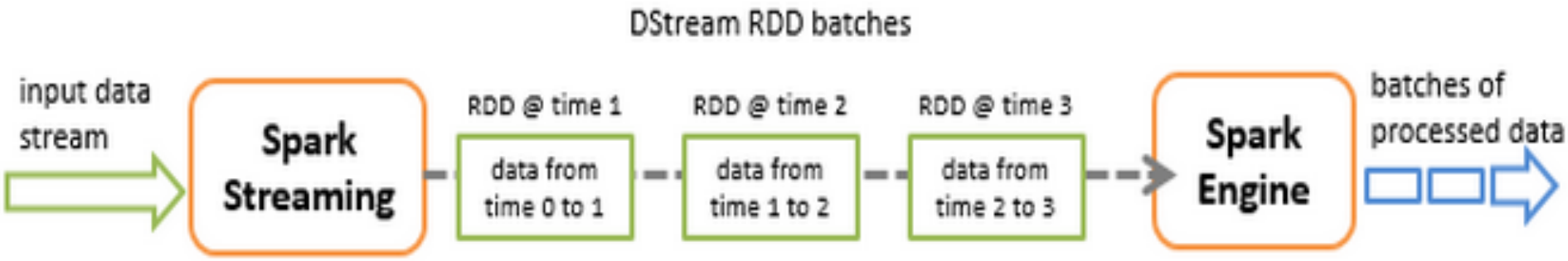
Source: MapR Academy

Overview: Various ingestion & Great features



Source: spark.apache.org

Overview: DStreams



ไม่ได้แปลว่า steam of data แต่หมายถึง distrct data เอาข้อมูลมาหันเป็นส่วนๆ

- DStreams can be created either from (1) input data streams or by (2) applying high-level operations on other DStreams.
- DStream is represented as a sequence of RDDs.

มี worker node หลายตัวช่วยกันประมวลผลได้ ทำงานเป็นแบบ paralle

Source: spark.apache.org

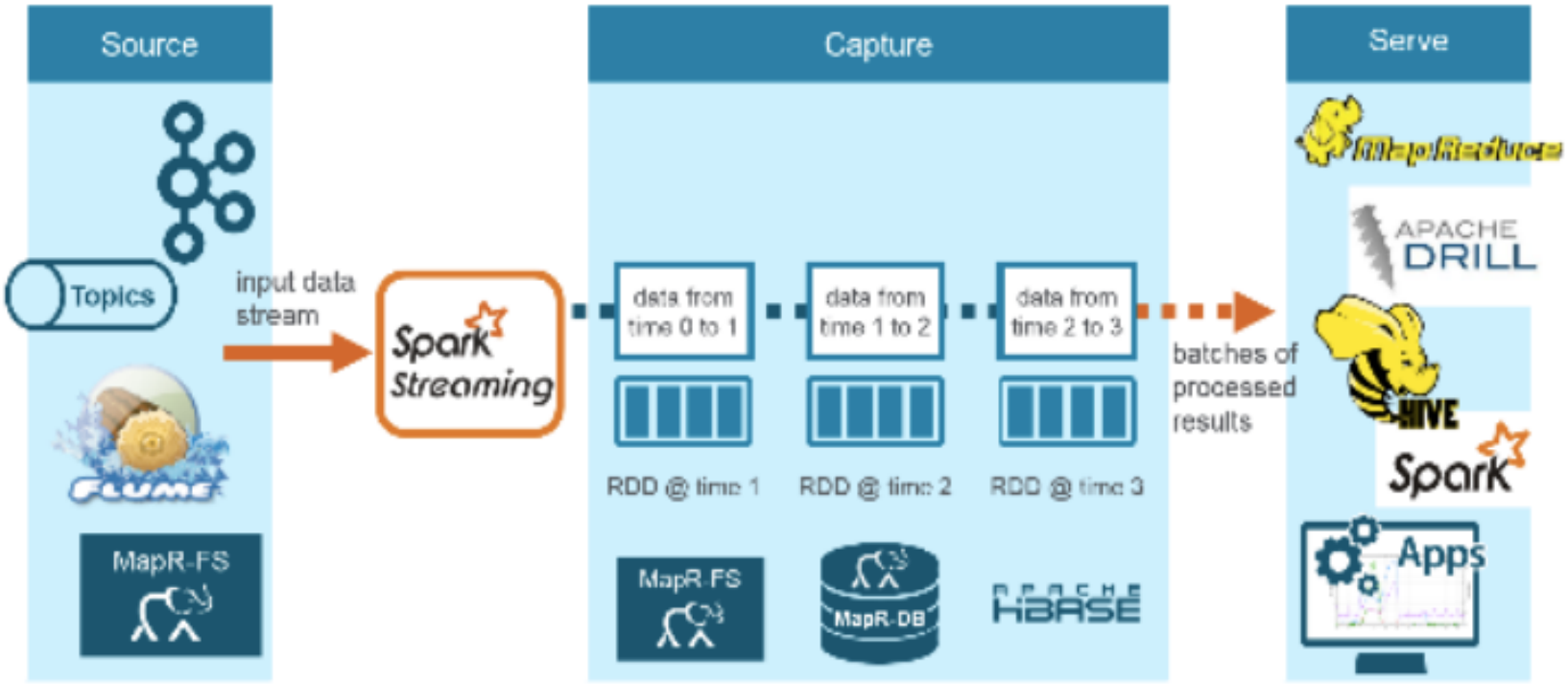
Overview: Micro-batches

"Spark Streaming receives live input data streams and divides the data into batches, which are then processed by the Spark engine to generate the final stream of results in batches."

Source: spark.apache.org

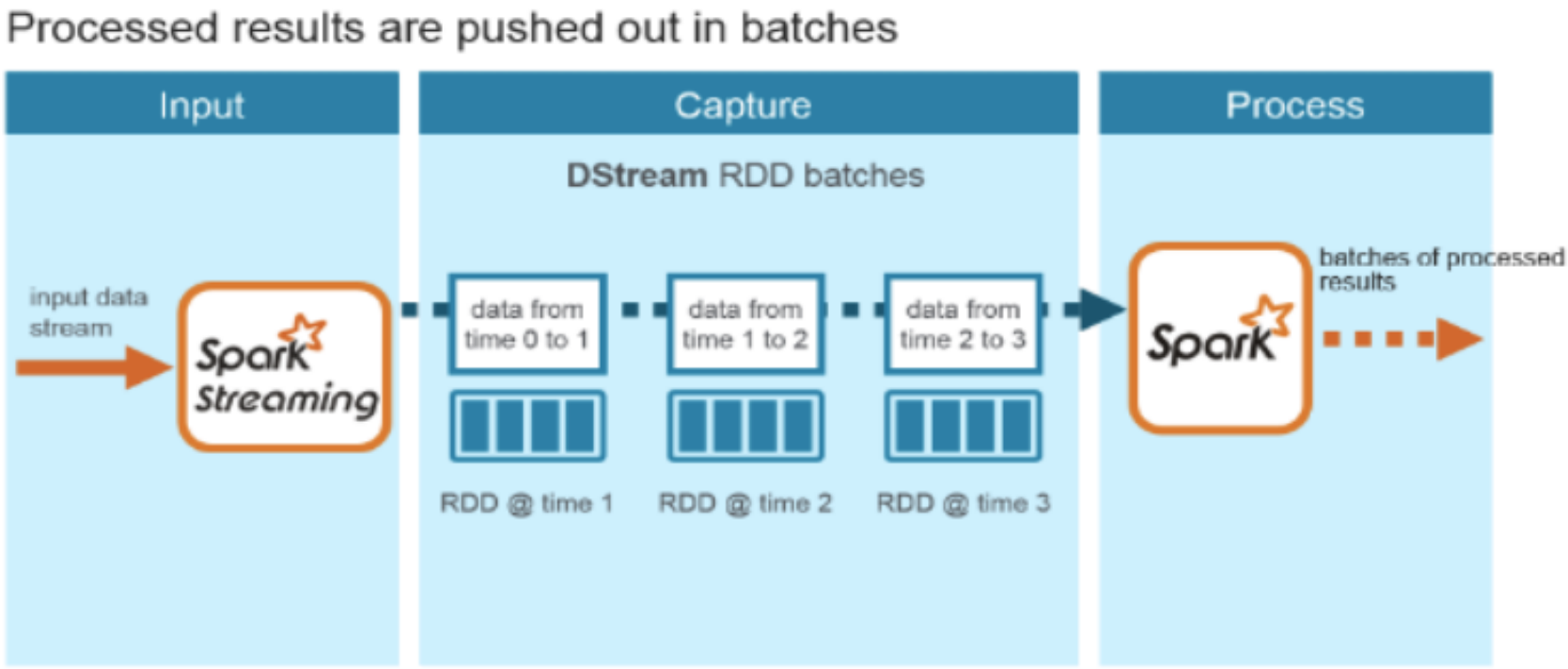
Spark Streaming Architecture

เอาไปทำงานต่อ



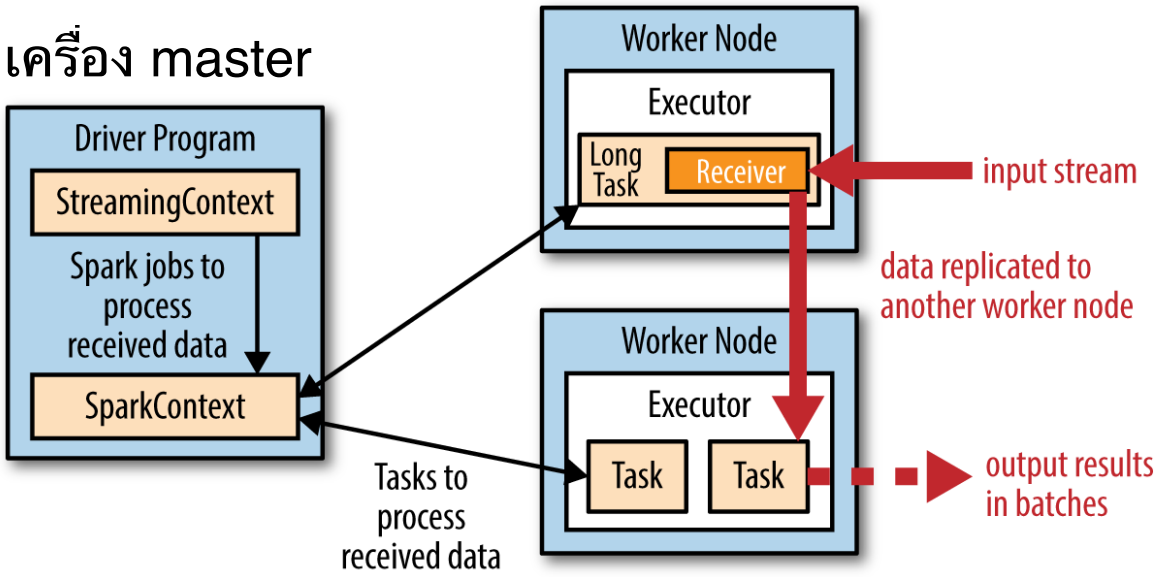
Source: MapR Academy

Processing Spark DStreams



Source: MapR Academy

Initializing StreamingContext



After a context is defined, you have to do the following.

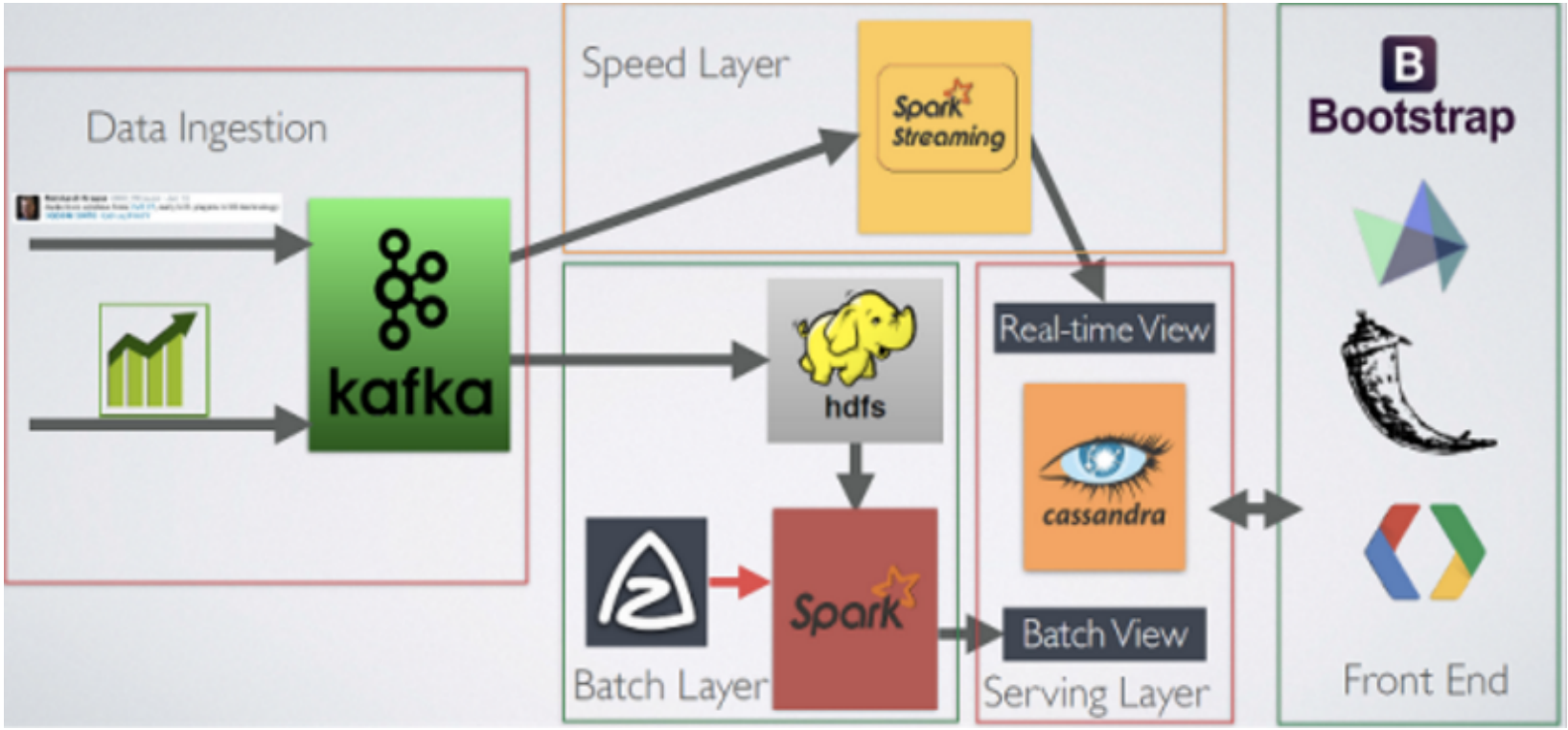
1.Define the input sources by creating input DStreams. ได้ ssc มา

2.Define the streaming computations by applying transformation and output operations to DStreams.

3.Start receiving data and processing it using streamingContext.start().

4.The processing can be manually stopped using streamingContext.stop().

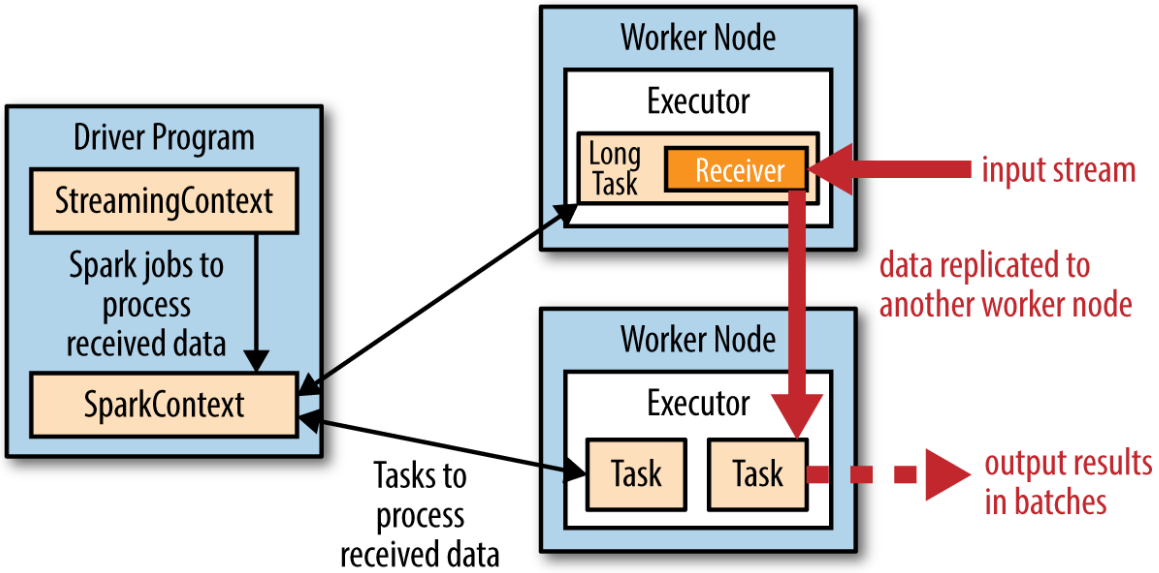
Use Case: Lambda Architecture



Source: http://www.insightdataengineering.com/

Hands-on: Spark Streaming

Initializing StreamingContext



 # Create a local StreamingContext and batch interval of 1 second

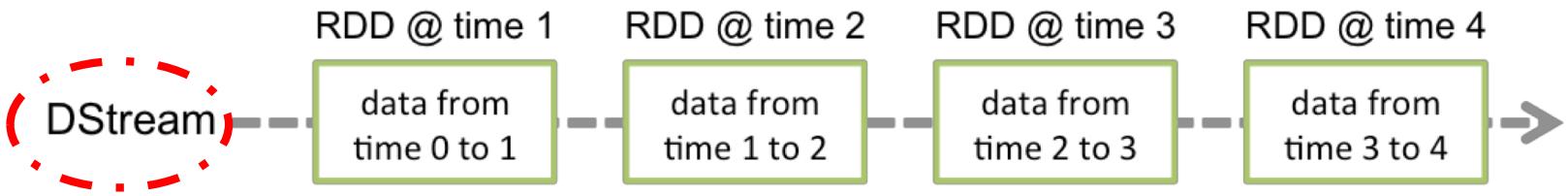
```
>>> from pyspark.streaming import StreamingContext

>>> ssc = StreamingContext(sc, 1)
```

Create a DStream & Define a source

 # Create a DStream that will connect to hostname:port, like localhost:9999

หั่นเป็นส่วนๆๆๆละ

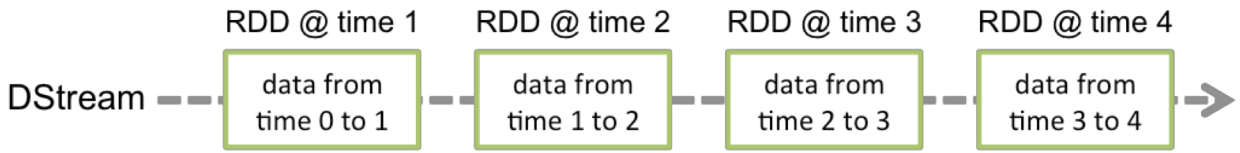
```
>>>lines = ssc.socketTextStream("localhost", 9999)
```



Source: spark.apache.org

Inside DStream

Each record (event) in this DStream is a line of text !

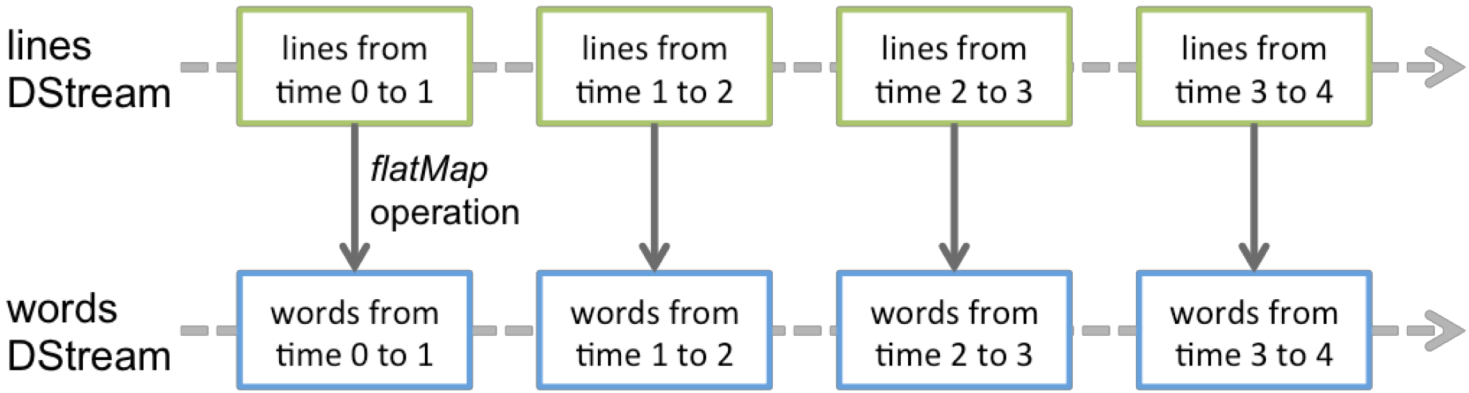


Get words from lines

 # Split each line into words

```
>>> words = lines.flatMap(lambda line: line.split(" "))
```

New DStream.



Source: spark.apache.org

Count and Print

```
# Count each word in each batch

>>> pairs = words.map(lambda word: (word, 1))

>>> wordCounts = pairs.reduceByKey(lambda x, y: x + y)


# Print the first ten elements of each RDD generated in this
DStream to the console

>>> wordCounts.pprint()
```

Source: spark.apache.org

Let's Start !

 # Start the computation

```
>>> ssc.start()
```

 # if you want to stop the computation, launch a following.

```
>>> ssc.stop()
```

Source: spark.apache.org

Netcat

 # Open new terminal (SSH) to connect same instance, and run a following ***on the Cloudera****.*

```
# nc -lk 9999
```

```
# TERMINAL 1:
# Running Netcat

$ nc -lk 9999

hello world



...
```

TERMINAL 2: Run PySpark-NetworkWordCount

```
-------------------------------------------
Time: 2014-10-14 15:25:21
-------------------------------------------
(hello,1)
(world,1)
...
```

Source: spark.apache.org