Big Data Analytics and Stream Processing on Apache Spark

Krešimir Pripužić, PhD

Associate Professor, Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia

Creative Commons

You are free to

- **Share** — copy and redistribute the material in any medium or format
- **Adapt** — remix, transform, and build upon the material

under the following terms

- **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- **NonCommercial** — You may not use the material for commercial purposes.
- **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

- https://creativecommons.org/licenses/by-nc-sa/4.0/

Defining Big Data

- *"Big Data - **the massive amounts of data collected over time that are difficult to analyze and handle using common database management tools**. The data are analyzed for marketing trends in business as well as in the fields of manufacturing, medicine and science. The types of data include **business transactions**, **e-mail messages**, **photos**, **surveillance videos**, **activity logs** and **unstructured text from blogs** and **social media**, as well as **the huge amounts of data** that can be collected **from sensors** of all varieties.*"

pcmag.com-Encyclopedia

- *„Big data analytics is the process of **examining large amounts of data** of a variety of types **to uncover hidden patterns, unknown correlations and other useful information**."*

Vignesh Prajapati: „Big Data Analytics with R and Hadoop", Packt Publishing, 2013.

Four Dimensions of Big Data (four V's)

- Volume – scale of data
- Variety – different forms of data
- Velocity – speed of data generation
- Veracity – uncertainty of data

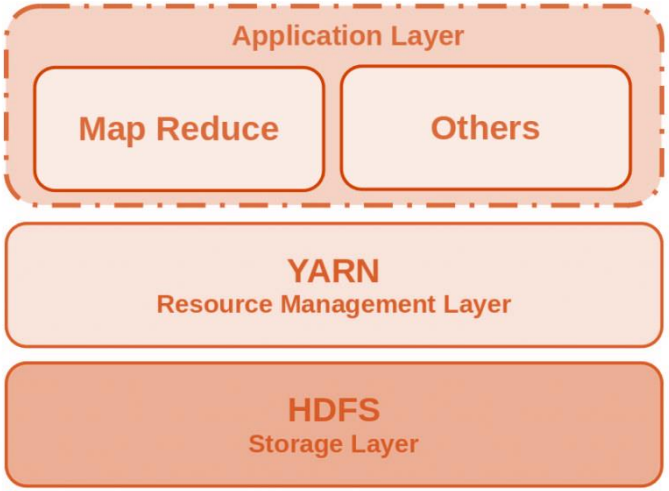
Source: http://www.slideshare.net/findwise/ibm-big-dataanalytics

Apache Hadoop

- A framework that allows a **distributed processing of large data sets** across clusters of computers
- Uses simple programming models
- Written in **Java**
- Scale up from single servers to thousands of machines
- Detects and handles failures at the application layer
- Deliverers **a highly-available service on top of a cluster of computers**, each of which may be prone to failures
- Many other FLOSS (Free/Libre and Open-Source Software) projects depend on it: https://hadoopecosystemtable.github.io/

Apache Hadoop Modules

- **Hadoop Common:** The common utilities that support the other Hadoop modules
- **Hadoop Distributed File System (HDFS):** A distributed file system that provides high-throughput access to application data
- **Hadoop YARN:** A framework for job scheduling and cluster resource management
- **Hadoop MapReduce:** A YARN-based system for parallel processing of large data sets



Source: https://starship-knowledge.com/category/framework

Essential HDFS Commands

- **mkdir** – Creates a directory

```
hdfs dfs -mkdir /user/alice/new_dir
```

- **ls** – Displays the contents of a directory

```
hdfs dfs -ls /user/alice/dir_to_list
```

- **put** – Copies a file from the local filesystem to the DFS

```
hdfs dfs -put /home/alice/file_to_put.txt /user/alice/file_to_put.txt
```

- **get** – Copies a file from the DFS to the local filesystem

```
hdfs dfs -get /user/alice/file_to_get.txt /home/alice/file_to_get.txt
```

- **cat** – Displays the contents of a file

```
hdfs dfs -cat /user/alice/file_to_show.txt
```

- **mv** – Moves a file in the DFS

```
hdfs dfs -mv /user/alice/file_to_move.txt /user/alice/moved_file.txt
```

- **cp** – Copies a file in the DFS

```
hdfs dfs -cp /user/alice/file_to_copy.txt /user/alice/copied_file.txt
```

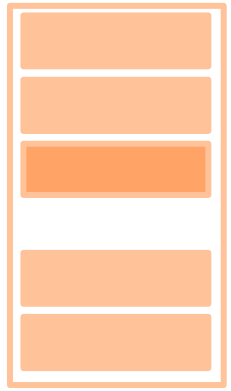
- **rm** – Deletes a file or directory in the DFS

```
hdfs dfs -rm /user/alice/file_to_delete.txt
hdfs dfs -rm –R /user/alice/folder_to_delete.txt
```

Running a Hadoop Cluster

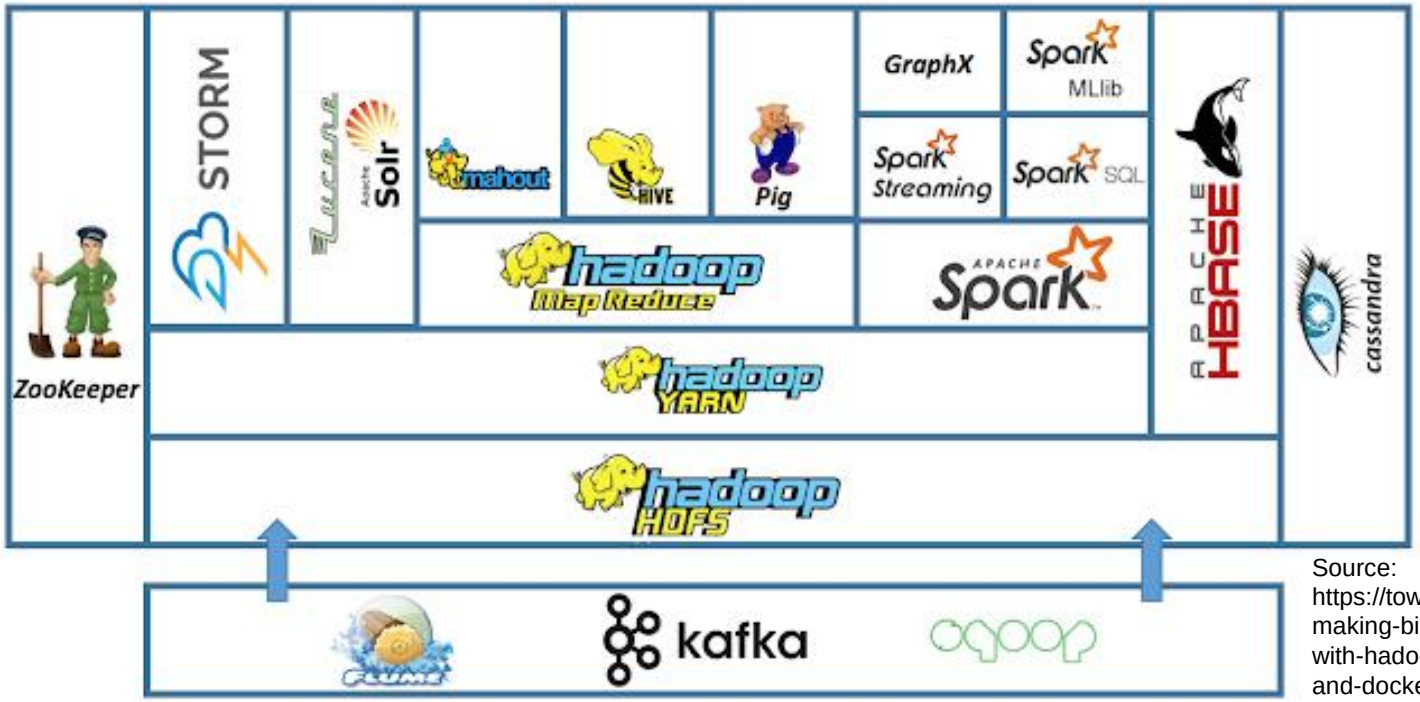
- **Standalone Operation**
  - The default running mode
  - Hadoop runs as a single Java process
  - Useful for debugging purposes
  - HDFS and YARN do not run in this mode

- **Pseudo-Distributed Operation**
  - Hadoop is run on a single-node
  - Each Hadoop daemon runs in a separate Java process
  - Useful for simulating the actual Hadoop cluster
  - Represents a fully-fledged test environment



- **Fully-Distributed Operation**
  - Hadoop daemons run on different nodes

Apache Hadoop Stack



Source: https://towardsdatascience.com/making-big-moves-in-big-data-with-hadoop-hive-parquet-hue-and-docker-320a52ca175

- **HBase**: A scalable, distributed database that supports structured data storage for large tables.
- **Mahout**: A Scalable machine learning and data mining library.
- **Spark:** A fast and general compute engine for Hadoop data. Spark provides a simple and expressive programming model that supports a wide range of applications, including ETL, machine learning, stream processing, and graph computation.
- **Solr**: A popular, blazing-fast, open-source enterprise search platform built on Apache Lucene.
- **ZooKeeper**: A high-performance coordination service for distributed applications.

Apache Spark

- Apache Spark is a **unified analytics engine for large-scale data processing**
- Spark **runs on** a cluster manger (e.g. **Hadoop YARN**, Apache Mesos, Kubernetes), standalone (pseudo or fully-distributed), locally (in a single JVM) or in the cloud (with a pay-as-you-go model)
- Apache Spark runs applications **up to 100x faster in memory and 10x faster on disk than Hadoop**
- Spark offers over 80 high-level operators that make it **easy to build parallel apps**
- Written in **Scala**, but supports writing of applications in Java, Scala, **Python**, R, and SQL
- Access data in **Apache HDFS**, Alluxio, Apache Cassandra, Apache HBase, Apache Hive, and hundreds of other data sources

Apache Spark Components

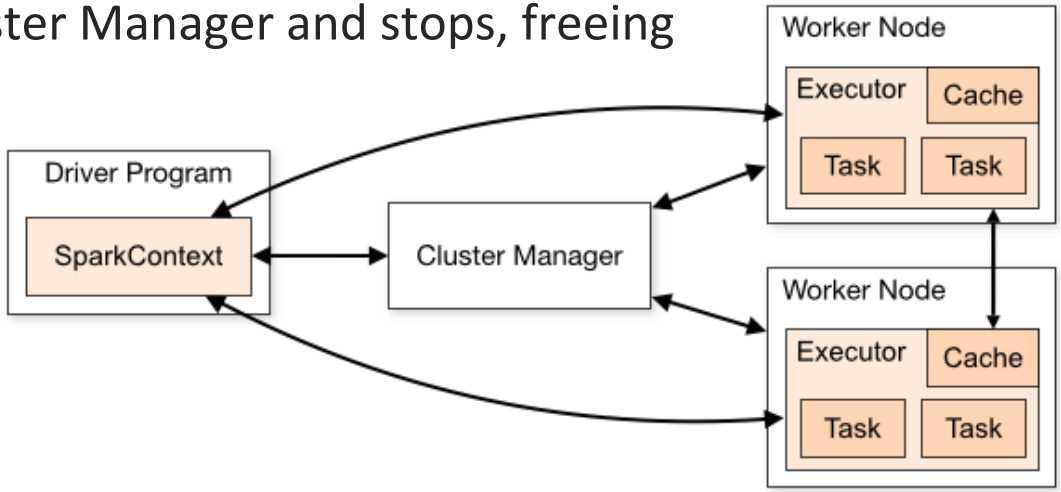
Source: https://devopedia.org/apache-spark

- **Spark Core:** a component for task scheduling, memory management, fault recovery, interacting with storage systems, and more
- **Spark SQL & DataFrames:** a component for working with structured data
- **Spark Streaming:** a component that enables processing of live streams of data
- **Spark MLlib:** a component containing common machine learning (ML) functionality
- **Spark GraphX:** a component for manipulating graphs

(Simplified) Execution of a Spark Application

Source: https://luminousmen.com/post/spark-anatomy-of-spark-application

1. An Spark application is copied to the gateway node and started.
2. The Driver Program starts and implicitly converts user code containing transformations and actions into a **logical execution plan** called a DAG (Directed Acyclic Graph).
3. The Driver Program then converts the DAG into a **physical execution plan** which contain stages. After conversion to a physical execution plan, the driver creates physical execution units called tasks at each stage.
4. The Driver Program now communicates with the Cluster Manager and negotiates resources. Cluster Manager asks worker nodes to run the executors.
5. When the application finishes executing, the Driver Program disconnects itself from the Cluster Manager and stops, freeing up its resources.



Source: https://spark.apache.org/docs/latest/cluster-overview.html

Spark Glossary

Term	Meaning
Application	User program built on Spark. Consists of a driver program and executors on the cluster.
Driver Program	The process running the main() function of the application and creating the SparkContext.
Cluster Manager	An external service for acquiring resources on the cluster (e.g. standalone manager, Mesos, YARN)
Deploy mode	Distinguishes where the driver process runs. In "cluster" mode, the framework launches the driver inside of the cluster. In "client" mode, the submitter launches the driver outside of the cluster.
Worker Node	Any node that can run application code in the cluster
Executor	A process launched for an application on a worker node, that runs tasks and keeps data in memory or disk storage across them. Each application has its own executors.
Job	A parallel computation that gets spawned in response to a Spark action.
Stage	Each job gets divided into smaller sets of tasks called stages that depend on each other.
Task	A unit of work that will be sent to one executor

Spark Data (Programming) Abstractions

All are immutable and read-only!!!

	RDD	DataFrame	Dataset
Release version	Spark 1.0	Spark 1.3	Spark 1.6
Data Representation	Distributed collection of elements.	Distributed collection of data organized into columns.	Combination of RDD and DataFrame.
Data Formats	Structured and unstructured are accepted.	Structured and semi-structured are accepted.	Structured and unstructured are accepted.
Compile-time type safety	Available compile-time type safety.	No compile-time type safety. Errors detect on runtime.	Available compile-time type safety.
Optimization	No built-in optimization engine. Each RDD is optimized individually.	Query optimization through the Catalyst optimizer.	Query optimization through the Catalyst optimizer, like DataFrames.
Serialization	Uses serialization for sending both the data and structure between nodes.	Serialization happens in memory in binary format.	Encoder handles conversions between objects and tables, which is faster than serialization.
Programming Language Support	Java, Scala, Python and R.	Java, Scala, Python and R.	Only Java and Scala.
Schema Projection	Schemas need to be defined manually.	Auto-discovery of file schemas.	Auto-discovery of file schemas.

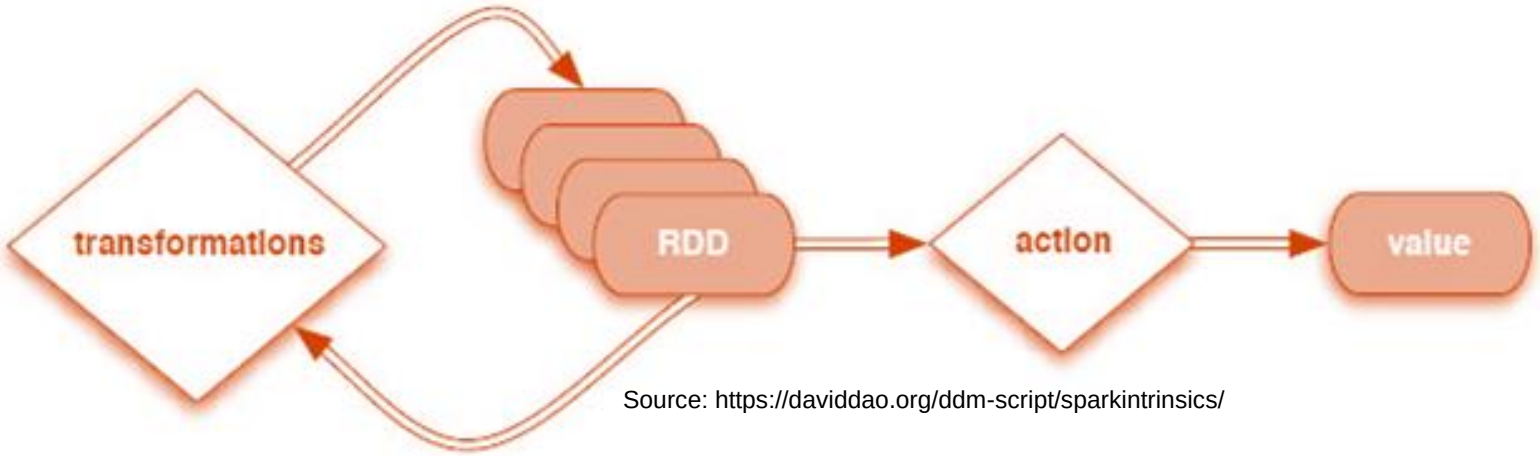
Source: https://phoenixnap.com/kb/rdd-vs-dataframe-vs-dataset

PySpark

- Aa Python API for Spark
- Based on Py4J library that allows Python to dynamically interface with JVM objects
- Supports RDD and DataFrame abstractions
- Detailed API Reference
  - Spark SQL (DataFrame)
  - Structured Streaming (DataFrame-based)
  - MLlib (DataFrame-based)
  - Spark Streaming (RDD-based)
  - MLlib (RDD-based)

RDD (*Resilient Distributed Dataset*)

- Processing based on the dataflow programming – programming paradigm that models a program as a directed graph of the data flowing between operations
- Supports two types of operations
  - *Transformations* transform an RDD to another RDD
    - Lazy execution (i.e. they are not executed immediately)
  - *Actions* consume an RDD
    - An action triggers the execution of transformations



Source: https://daviddao.org/ddm-script/sparkintrinsics/

Essential RDD Transformations 1/2

Transformations	
map(f[, preservesPartitioning])	Return a new RDD by applying a function to each element of this RDD.
filter(f)	Return a new RDD containing only the elements that satisfy a predicate.
flatMap(f[, preservesPartitioning])	Return a new RDD by first applying a function to all elements of this RDD, and then flattening the results.
sample(withReplacement, fraction[, seed])	Return a sampled subset of this RDD.
union(other)	Return the union of this RDD and another one.
distinct([numPartitions])	Return a new RDD containing the distinct elements in this RDD.

Essential RDD Transformations 2/2

Transformations	
groupByKey([numPartitions, partitionFunc])	Group the values for each key in the RDD into a single sequence.
reduceByKey(func[, numPartitions, …])	Merge the values for each key using an associative and commutative reduce function.
sortByKey([ascending, numPartitions, …])	Sorts this RDD, which is assumed to consist of (key, value) pairs.
join(other[, numPartitions])	Return an RDD containing all pairs of elements with matching keys in self and other.
cogroup(other[, numPartitions])	For each key k in self or other, return a resulting RDD that contains a tuple with the list of values for that key in self as well as other.
cartesian(other)	Return the Cartesian product of this RDD and another one.

Essential RDD Actions

Actions	
reduce(f)	Reduces the elements of this RDD using the specified commutative and associative binary operator.
collect()	Return a list that contains all of the elements in this RDD.
count()	Return the number of elements in this RDD.
take(num)	Take the first num elements of the RDD.
saveAsTextFile(path[, compressionCodecClass])	Save this RDD as a text file, using string representations of elements.
foreach(f)	Applies a function to all elements of this RDD.
min([key]) max([key])	Find the minimum/maximum item in this RDD.

Example 1: Counting Words Using PySpark

```
# Imports are excluded

ss = SparkSession.builder.getOrCreate()
sc = ss.sparkContext

result = sc.textFile("1661-0.txt") \
    .flatMap(lambda line: line.strip().split()) \
    .map(lambda word: re.sub("[^a-zA-Z]+", "", word).lower().strip()) \
    .filter(lambda word: len(word) > 0) \
    .map(lambda word: (word, 1)) \
    .reduceByKey(lambda a, b: a + b)

result.saveAsTextFile("word_counts.txt")
```

```
Pseudo-distributed - Time elapsed in seconds:  2.814466714859009

Fully Distributed - Time elapsed in seconds:  6.459198474884033
```

Example 2: Counting Words in Java using Spark

```
//imports are excluded

SparkSession ss = SparkSession.builder().getOrCreate();
SparkContext sc = ss.sparkContext();

JavaPairRDD<String, Long> result = sc.textFile("1661-0.txt").
  flatMap(line -> Arrays.asList(line.trim().split("\\s")).iterator()).
  map(word -> word.replaceAll("[^a-zA-Z]", "").toLowerCase().trim()).
  filter(word -> word.length() > 0).
  mapToPair(word -> new Tuple2<>(word, 1L)).
  reduceByKey((x, y) -> x + y);

result.saveAsTextFile("word_counts.txt");
```

Example 3: Counting Words in Java

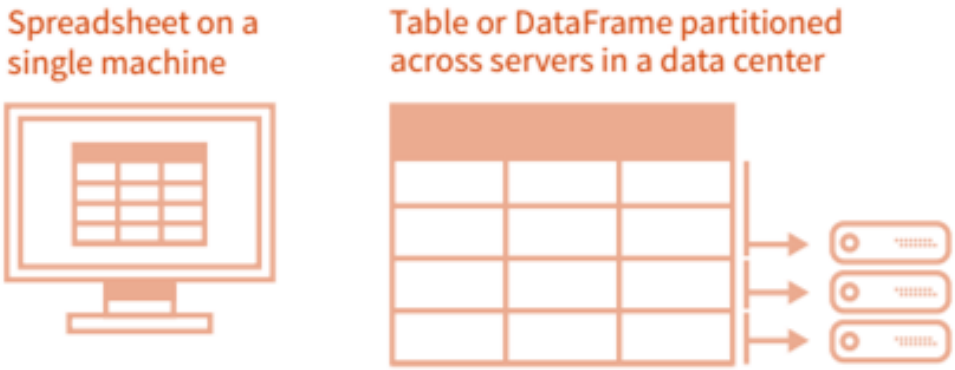
```
//imports are excluded

Map<String, Long> result = Files.lines(Paths.get(args[0])))).
  flatMap(line -> Arrays.stream(line.trim().split("\\s")))).
  map(word -> word.replaceAll("[^a-zA-Z]", "").toLowerCase().trim()).
  filter(word -> word.length() > 0).
  map(word -> new SimpleEntry<>(word, 1L)).
  collect(groupingBy(SimpleEntry::getKey, reducing(0L,
SimpleEntry::getValue, (v1,v2) -> v1 + v2)));

//write results to a file
try (FileWriter fw = new FileWriter(args[1])) {
  result.forEach((k, v) -> writeToFile(fw, k + "," + v + "\n"));
}
```

(Java-like) Streams in Python

- pyxtension is a pure Python GNU-licensed library that includes Scala-like streams (using Fluent Interface pattern), Json with attribute access syntax, and other common-use stuff.
- There are other Python libraries that support Fluent Interface streams as alternatives to pyxtension, but being much poorer in features:
  - https://pypi.org/project/lazy-streams/
  - https://pypi.org/project/pystreams/
  - https://pypi.org/project/fluentpy/
  - https://github.com/matthagy/scalaps
  - https://pypi.org/project/infixpy/
  - https://github.com/sspipe/sspipe

Spark DataFrame

- A DataFrame simply represents a table of data with rows and columns
- A simple analogy would be a spreadsheet with named columns
- The list of columns and the types in those columns represent the schema
- Similar concepts exist in R and Python (Pandas)
- While an R/Python DataFrame or spreadsheet sits on a single computer, a Spark DataFrame is distributed across (possibly) thousands of computers



Source: https://databricks.com/glossary/what-are-dataframes

Essential DataFrame Operations 1/2

Data Manipulation Operations	
toPandas()	Returns the contents of this DataFrame as Pandas DataFrame.
createDataFrame(data[, schema, …])	Creates a DataFrame from an RDD, a list or a pandas.DataFrame.
drop(*cols)	Returns a new DataFrame that drops the specified column.
select(*cols)	Projects a set of expressions and returns a new DataFrame.
filter(condition)	Filters rows using the given condition.
withColumn(colName, col)	Returns a new DataFrame by adding a column or replacing the existing column that has the same name.
SparkSession.read	Returns a DataFrameReader that can be used to read data in as a DataFrame.
write	Interface for saving the content into external storage.

Essential DataFrame Operations 2/2

Essential Operations	
printSchema()	Prints out the schema in the tree format.
show([n, truncate, vertical])	Prints the first n rows to the console
sort(*cols, **kwargs)	Returns a new DataFrame sorted by the specified column(s).
count()	Returns the number of rows in this DataFrame.
crossJoin(other)	Returns the cartesian product with another DataFrame.
foreach(f)	Applies the f function to all Row of this DataFrame.
groupBy(*cols)	Groups the DataFrame using the specified columns, so we can run aggregation on them.
join(other[, on, how])	Joins with another DataFrame, using the given join expression.
corr(col1, col2[, method])	Calculates the correlation of two columns of a DataFrame as a double value.
describe(*cols)	Computes basic statistics.

Example 4: Processing Data using Pandas DataFrame

```
import pandas as pd
import matplotlib.pyplot as plt


df = pd.read_csv("trip_data_small.csv", header="infer")
df.dtypes

df_sel = df[["medallion", "passenger_count", "trip_distance", "trip_time_in_secs"]]
df_sel[["trip_distance", "trip_time_in_secs"]].corr(method='pearson')
df_sel[["trip_time_in_secs", "passenger_count"]].corr(method='pearson')

plt.scatter(x='trip_distance', y='trip_time_in_secs', data=df_sel)
plt.show()
df_sel.describe()

df_sum = df_sel.groupby("medallion").sum()
df_spd = df_sum
df_spd["average_speeed"] = df_sum["trip_distance"] / df_sum["trip_time_in_secs"]

df_sort = df_spd.sort_values("average_speeed", ascending=False)
df_sort.to_csv("sorted_by_avg_speed.csv")
```

Example 5: Processing Data using Spark DataFrame

```
from pyspark.sql import SparkSession
import matplotlib.pyplot as plt

session = SparkSession.builder.getOrCreate()
context = session.sparkContext
df = session.read.csv("trip_data_small.csv", header=True, inferSchema=True)
df.printSchema()
df_sel = df.select("medallion", "passenger_count", "trip_distance", "trip_time_in_secs
")
df_sel.stat.corr('trip_distance', 'trip_time_in_secs', method='pearson')
df_sel.stat.corr('trip_time_in_secs', 'passenger_count', method='pearson’)

plt.scatter(x='trip_distance', y='trip_time_in_secs', data=df_sel.toPandas())
plt.show()
df_sel.describe().show()

df_sum = df_sel.groupBy("medallion").sum()
df_spd = df_sum.withColumn("average_speeed", df_sum["sum(trip_distance)"] / df_sum["su
m(trip_time_in_secs)"])

df_sort = df_spd.sort("average_speeed", ascending=False)
df_spd.write.csv("sorted_by_avg_speed.csv")
session.stop()
```

Spark MLlib

- MLlib is Spark MLlib is a distributed and scalable machine-learning framework
- Usable in Java, Scala, Python, and R
- MLlib fits into Spark's APIs and interoperates with NumPy in Python and R libraries
- You can use any Hadoop data source (e.g. HDFS, HBase, or local files), making it easy to plug into Hadoop workflows
- There are DataFrame-based and RDD-based APIs for MLlib
- At a high level, it provides tools such as:
  - ML Algorithms: common learning algorithms such as classification, regression, clustering, and collaborative filtering
  - Featurization: feature extraction, transformation, dimensionality reduction, and selection
  - Pipelines: tools for constructing, evaluating, and tuning ML Pipelines
  - Persistence: saving and load algorithms, models, and Pipelines
  - Utilities: linear algebra, statistics, data handling, etc.

Example 6 - 1/3: Preparing Data for ML

```
from pyspark.sql import SparkSession
session = SparkSession.builder.getOrCreate()
context = session.sparkContext

df = session.read.csv("trip_data_small.csv", header=True, inferSchema=True)
df_sel = df.select("medallion", "passenger_count", "trip_distance", "trip_time_in_secs
")
df_sel.stat.corr('trip_distance', 'trip_time_in_secs', method='pearson')
df_sel.stat.corr('trip_distance', 'passenger_count', method='pearson’)

from pyspark.ml.feature import VectorAssembler
vectorAssembler = VectorAssembler(inputCols = ["passenger_count", "trip_time_in_secs"]
, outputCol = "features")
df_vec = vectorAssembler.transform(df_sel)
df_prep = df_vec.select(['features', 'trip_distance'])
splits = df_prep.randomSplit([0.7, 0.3])
df_train = splits[0]
df_test = splits[1]
```

Example 6 - 2/3: Linear Regression

```
from pyspark.ml.regression import LinearRegression
lr = LinearRegression(featuresCol = 'features', labelCol='trip_distance', maxIter=10,
regParam=0.3, elasticNetParam=0.8)
lr_model = lr.fit(df_train)
print("Coefficients: " + str(lr_model.coefficients))
print("Intercept: " + str(lr_model.intercept))

lr_model_summ = lr_model.summary
print("RMSE: %f" % lr_model_summ.rootMeanSquaredError)
print("R2: %f" % lr_model_summ.r2)
print("numIterations: %d" % lr_model_summ.totalIterations)
print("objectiveHistory: %s" % str(lr_model_summ.objectiveHistory))
lr_model_summ.residuals.show(5)
df_train.describe().show()

from pyspark.ml.evaluation import RegressionEvaluator
evaluator_rmse = RegressionEvaluator(labelCol="trip_distance", predictionCol="predicti
on", metricName="rmse")
evaluator_r2 = RegressionEvaluator(labelCol="trip_distance", predictionCol="prediction
", metricName="r2")
lr_predictions = lr_model.transform(df_test)
print("RMSE on test data = %g" % evaluator_rmse.evaluate(lr_predictions))
print("R2 on test data = %g" % evaluator_r2.evaluate(lr_predictions))
```

Example 6 - 3/3: DT and GBR Regression

```
# Decision tree learning algorithm for regression
from pyspark.ml.regression import DecisionTreeRegressor
dt = DecisionTreeRegressor(featuresCol ='features', labelCol = 'trip_distance')
dt_model = dt.fit(df_train)
dt_predictions = dt_model.transform(df_test)
print("RMSE on test data = %g" % evaluator_rmse.evaluate(dt_predictions))
print("R2 on test data = %g" % evaluator_r2.evaluate(dt_predictions))

# Gradient-Boosted Trees learning algorithm for regression
from pyspark.ml.regression import GBTRegressor
gbt = GBTRegressor(featuresCol = 'features', labelCol = 'trip_distance', maxIter=10)
gbt_model = gbt.fit(df_train)
gbt_predictions = gbt_model.transform(df_test)
print("RMSE on test data = %g" % evaluator_rmse.evaluate(gbt_predictions))
print("R2 on test data = %g" % evaluator_r2.evaluate(gbt_predictions))
```

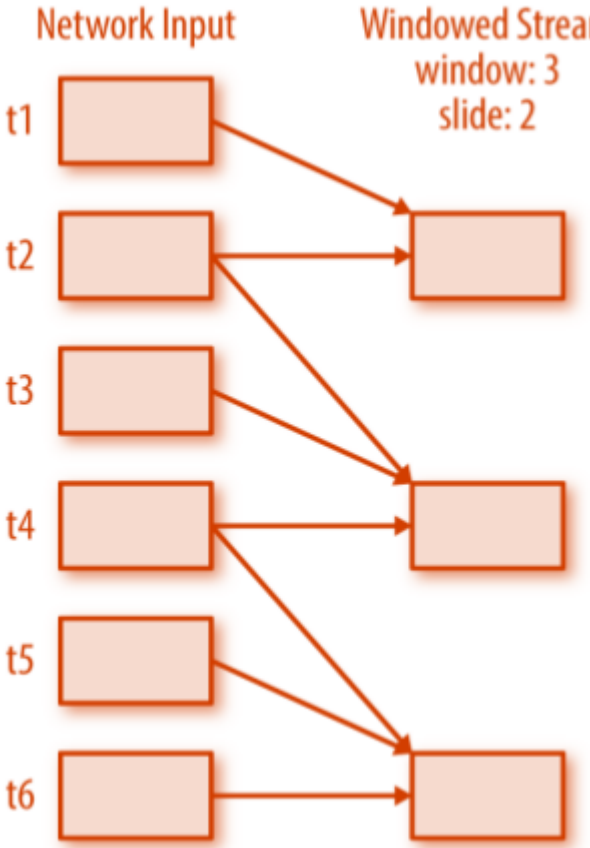
Spark Streaming

- Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams
- Spark Streaming receives live input data streams and divides the data into batches to which we often refer as micro-batches
- Bathes are processed by the Spark engine to generate the final stream of results also in batches
- Spark Streaming provides a high-level abstraction called discretized stream or DStream, which is internally represented as a sequence of RDDs
- DStreams can be created either from input data streams from sources such as Kafka, and Kinesis, or by applying high-level operations on other Dstreams
- Batch duration (in seconds) is defined at the beginning of an application within the `StreamingContext(sparkContext, batchDuration=None)`

Source: https://spark.apache.org/docs/latest/streaming-programming-guide.html

Spark Streaming Traansformations

- There are transformations with and without an internal state
- Those with the internal state (`window`, `countByWindow`, `reduceByWindow`, etc.) are using time windows
  - They require two parameters: `windowDuration` and `slidingDuration`, which are both a multiple of `batchDuration`
    - `windowDuration` - The duration of the window
    - `slidingDuration` - The interval at which the window operation is performed



Source: https://www.oreilly.com/library/view/learning-spark/9781449359034/

Spark Streaming and MLlib

- Supports three *online machine learning* models
  - Streaming Linear Regression with Stochastic Gradient Descent
  - Streaming Logistic Regression
  - Streaming K Means

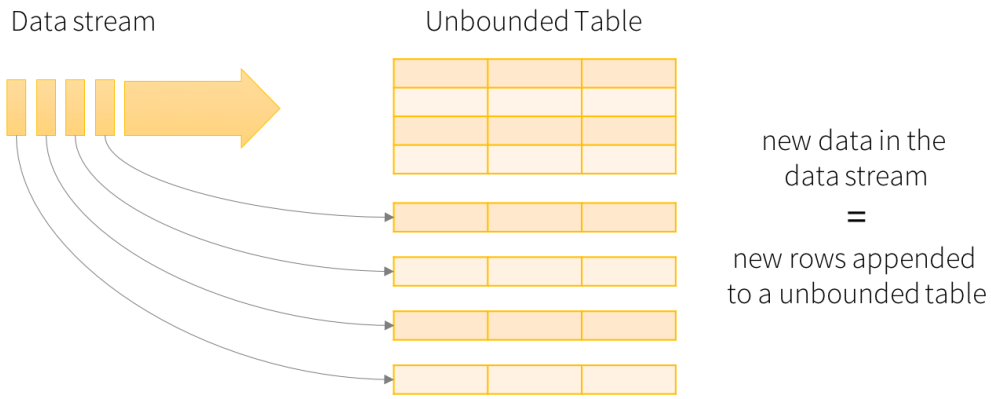
- *Online machine learning* is a method of machine learning in which data becomes available in a sequential order and is used to update the best predictor for future data at each step, as opposed to batch learning techniques which generate the best predictor by learning on the entire training data set at once (definition from Wikipedia)

Spark Structured Streaming

- Structured Streaming is a **scalable and fault-tolerant stream processing engine** built on the Spark SQL engine
- It supports both the **end-to-end** and **exactly-once** stream processing
- Streaming computation is expressed in the same way as a batch computation on static data
- The Spark SQL engine takes care of running the streaming computation incrementally and continuously and updates the result as streaming data arrives
- The Dataset/DataFrame API in Scala, Java, Python or R is used with it to express **streaming aggregations, event-time windows, stream-to-batch joins**, etc.
- Supports both the **event and processing times**

Structured Streaming Processing Model

- Structured Streaming treats a live data stream as an Input Table that is being continuously appended
- Every data item that is arriving on the stream is like a new row being appended to the Input Table
- This stream processing model that is very similar to a batch processing model
- A continuous query generates the Result Table
- Every trigger interval, new rows get appended to the Input Table, which eventually updates the Result Table



Data stream as an unbounded table

Source:
https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

The Output in Spark Streaming

- The Output is defined as what gets written out to the external storage
- It can be defined in three different 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

Structured Streaming Operations

- Operations on streaming DataFrames/Datasets
  - Basic Operations - Selection, Projection, Aggregation
  - Window Operations on Event Time
    - Handling Late Data and Watermarking
  - Join Operations
    - Stream-static Joins
    - Stream-stream Joins
      - Inner Joins with optional Watermarking
      - Outer Joins with Watermarking
      - Semi Joins with Watermarking
      - Support matrix for joins in streaming queries
  - Streaming Deduplication
  - Policy for handling multiple watermarks
  - Arbitrary Stateful Operations
  - Unsupported Operations
  - Limitation of global watermark

Example 7 - 1/2 – Structured Streaming

```
# Imports are excluded

session = SparkSession.builder.getOrCreate()
schema = StructType([StructField('medallion',StringType(),True),
                     StructField('hack_license',StringType(),True),
                     StructField('vendor_id',StringType(),True),
                     StructField('rate_code',IntegerType(),True),
                     StructField('store_and_fwd_flag',StringType(),True),
                     StructField('passenger_count',IntegerType(),True),
                     StructField('trip_time_in_secs',IntegerType(),True),
                     StructField('trip_distance',DoubleType(),True),
                     StructField('pickup_longitude',DoubleType(),True),
                     StructField('pickup_latitude',DoubleType(),True),
                     StructField('dropoff_longitude',DoubleType(),True),
                     StructField('dropoff_latitude',DoubleType(),True)])

stream_df = (
  session
    .readStream
    .schema(schema)
    .option("maxFilesPerTrigger", 1) \
    .csv("trip_data_small_dir/*.csv")
```

Example 7 - 2/2 – Structured Streaming

```
# add processing time as a new column
time_stream_df = stream_df.withColumn("processingTime",current_timestamp())
import pyspark.sql.functions as F
import random
time_stream_df = time_stream_df.withColumn("processingTime", (F.unix_timestamp("proces
singTime") + F.rand() * 15).cast('timestamp'))

stream_count_df = (
  time_stream_df
    .groupBy(
      time_stream_df.passenger_count,
      window(time_stream_df.processingTime, "5 seconds"))
    .count()
)

stream_count_df.writeStream \
    .outputMode("update") \
    .format("console") \
    .option("truncate","false") \
    .start() \
    .awaitTermination()
```

Summary

- We defined Big Data and present the main challenges
- We presented Apache Hadoop and Apache Spark
- We presented PySpark API and its corresponding RDD and DataFrame APIs
- We showed that DataFrame API is like Python's Pandas library
- We learned to build, train and test basic ML models using PySpark DataFrame API
- We presented Spark Streaming and Structured Streaming