
Science and Technology Facilities Council
Hartree Centre
Welcome

Science and Technology Facilities Council
Hartree Centre

Week 11: Advanced Spark

MLLib, GraphX and Streaming

Dom Richards
Artificial Intelligence Group Leader, Hartree Centre

Recap

- In weeks 2 and 3, we looked at big data middleware
  - Hadoop:
    - Provides a very simple computational abstraction: map followed by reduce over distributed collections of (key, value) pairs
  - Spark:
    - Extended the computational abstraction of Hadoop to include list programming functions (map, reduce, filter, fold, …) over Resilient Distributed Datasets (RDDs)
    - Also provides a performance boost over Hadoop

Today's Lecture

- Today, we'll see that Spark generalises the big data computational model further still.
- We'll see data structures and algorithms for:
  - Machine learning (Spark MLLib)
  - Graph algorithms (Spark GraphX)
  - Streaming data (Spark Streams)
- These packages extend the core computational abstraction of Spark (which is RDDs)
- They also provide further performance gains versus comparable code which is user-coded for RDDs.
- (Spark also supports SQL queries via the Spark SQL library. However, we'll skip this because database theory is out of scope for the current module.)

Recommended Reading

- This lecture is self-contained.
- However, if you'd like to know more, example code and documentation for today can be found here:
  - https://spark.apache.org/docs/2.2.0/ml-classification-regression.html
  - https://spark.apache.org/docs/2.2.0/graphx-programming-guide.html
  - https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#quick-example
  - https://graphframes.github.io/graphframes/docs/_site/user-guide.html

Spark MLLib

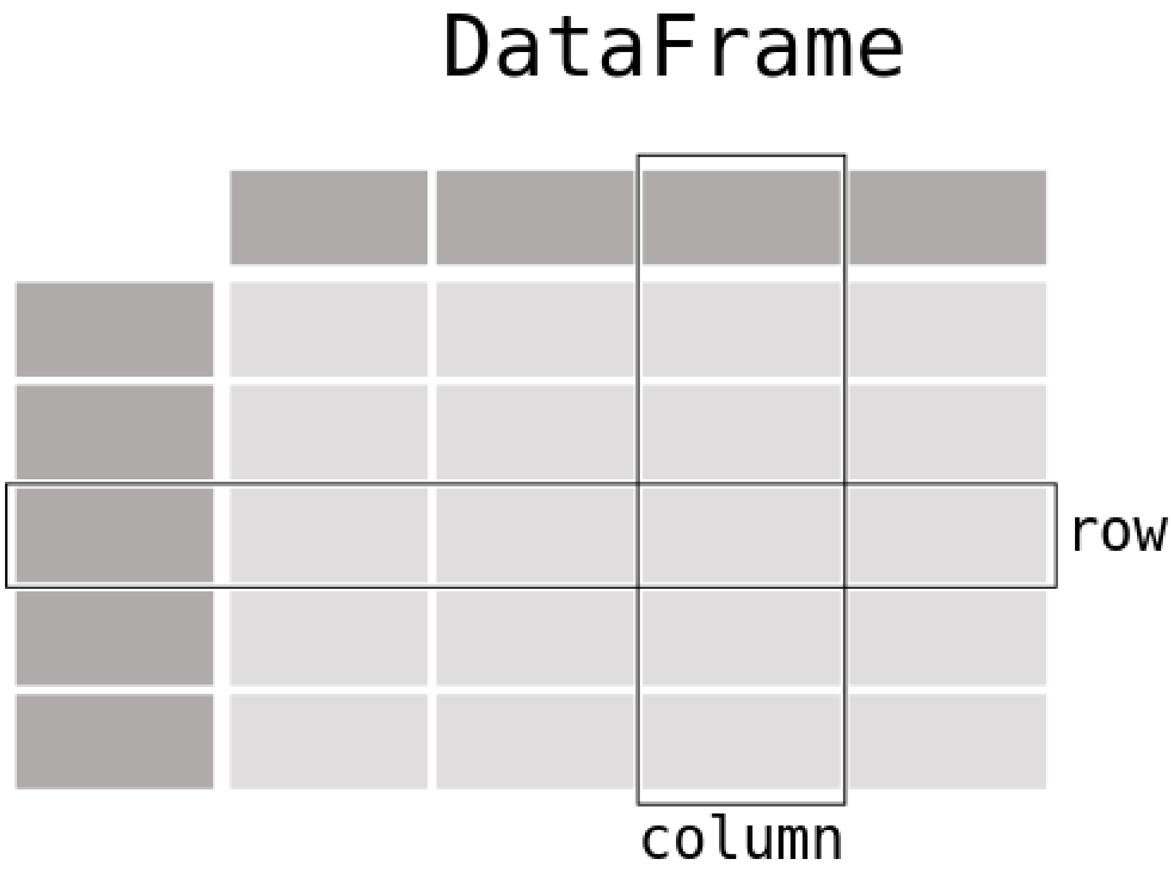
MLLib Overview

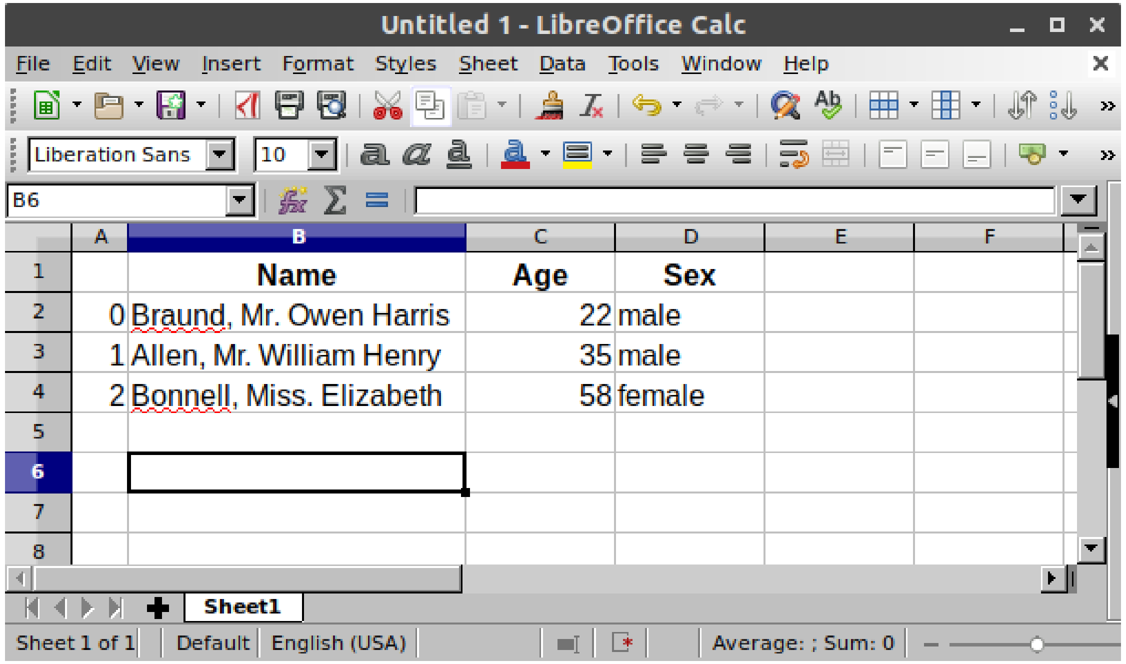
- MLlib is Spark’s machine learning library
- It allows machine learning to be applied to big datasets:
  - i.e. hundreds of TBs of training data, with thousands of features
- It provides tools such as:
  - Algorithms for supervised and unsupervised learning.
  - Tools for constructing end-to-end ML pipelines
  - Tools for saving and loading algorithms, models, pipelines etc.
  - Utility functions for linear algebra, statistics, data handling etc.

Spark DataFrames

- Before getting into the nuts and bolts of MLLib, we'll lay some groundwork by introducing DataFrames...
- All the packages we'll see today depend on DataFrames
- It is conceptually equivalent to a data frame in R/Python (or a table in a relational database)
- DataFrames are built on top of RDDs, and are distributed collections of rows with named columns.
- As with RDDs, they are:
  - Immutable (i.e. you can't change a DataFrame once it has been created).
  - Have lazy evaluation (i.e. a function on RDDs is not executed until its result is required).
  - Distributed over nodes in a Spark cluster
- DataFrames can be created in the following ways:
  - Loading data stored in various formats (e.g. JSON, CSV, RDBMS, XML, Parquet)
  - Loading data from an already existing RDD

DataFrames


DataFrame
row
column


Untitled 1 - LibreOffice Calc
File Edit View Insert Format Styles Sheet Data Tools Window Help
Liberation Sans
10
B6
A B C D E F
Name Age Sex
0 Braund, Mr. Owen Harris 22 male
1 Allen, Mr. William Henry 35 male
2 Bonnell, Miss. Elizabeth 58 female
Sheet1
Sheet 1 of 1 Default English (USA) Average: ; Sum: 0

Other MLLib Data Structures

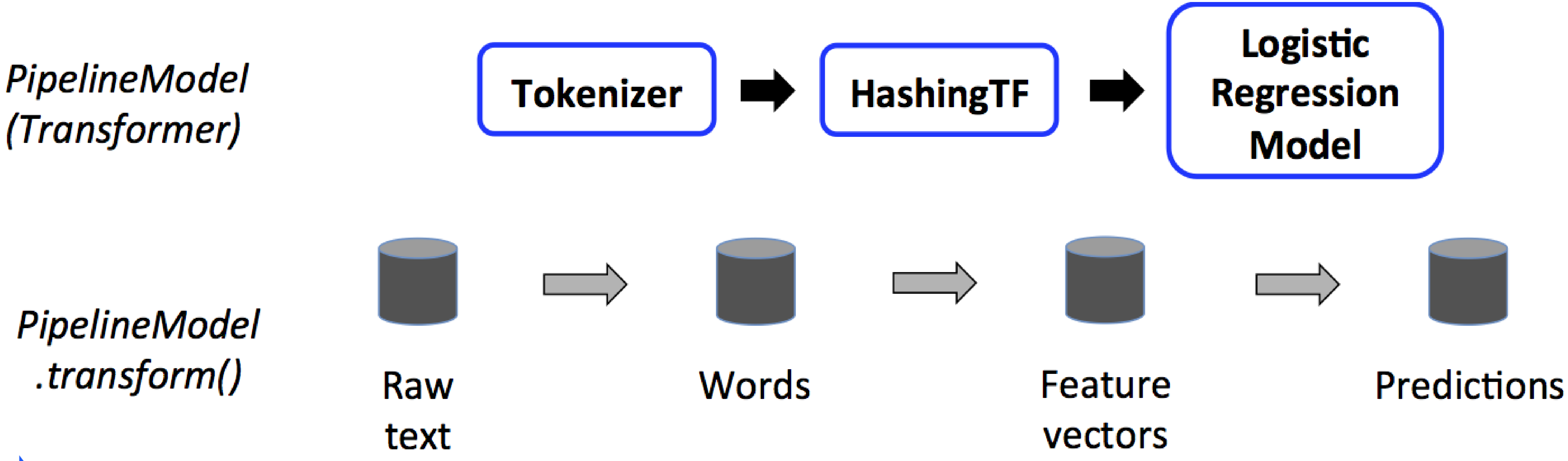
- MLLib also supports some other data structures:
- Vectors:
  - These can be dense or sparse, and local to a single spark node or distributed across several nodes
- Labelled points
  - A local vector, dense or sparse, which is associated with a response
- Local matrices
- Distributed matrices, of which there are several types:
  - RowMatrix
    - Backed by an RDD of rows
  - IndexedRowMatrix
    - Backed by an RDD of indexed rows
  - CoordinatedMatrix
    - Backed by an RDD of its individual entries
  - Block matrices
    - Backed by an RDD of matrix blocks - (int, int, matrix)

Machine Learning Pipelines

- In machine learning, it is common to run a sequence of steps to process and learn from data.
- E.g., a simple text document processing workflow might include several stages:
  - Split each document's text into words.
  - Convert each document's words into a numerical feature vector.
  - Learn a prediction model using the feature vectors and labels.
- More generally, the usual ML pipeline proceeds as follows:
  1. Load and Clean Data
  2. Extract Features
  3. Train Model
  4. Evaluate Model
  5. Repeat as necessary

MLLib Pipelines

- MLlib represents such a workflow as a Pipeline, which consists of a sequence of PipelineStages to be run in a specific order
- PipelineStages can be "Transformers" and "Estimators", as we'll see in the next few slides



Transformers

- Transformers are pipeline stages that transform DataFrames in some way:
  - E.g. feature transformation, model-bases estimation etc.
- In Spark, a Transformer implements the method transform(), which converts one DataFrame into another, generally by appending one or more columns. For example:
  - A feature transformer might take a DataFrame, read a column (e.g., text), map it into a new column (e.g., feature vectors), and output a new DataFrame with the mapped column appended.
  - A learning model might take a DataFrame, read the column containing feature vectors, predict the label for each feature vector, and output a new DataFrame with predicted labels appended as a column.

Estimators

- An Estimator abstracts the concept of a learning algorithm or any algorithm that fits or trains on data.
- An Estimator implements a method fit(), which accepts a DataFrame and produces a Model, which is a Transformer.
  - For example, a learning algorithm such as LogisticRegression (which we'll see shortly) is an Estimator, and calling fit() trains a LogisticRegressionModel, which is a Model and hence a Transformer.
- Transformer.transform() and Estimator.fit() are both stateless.

Evaluator

- Evaluators will evaluate the performance of a model, based on a performance certain metric:
  - E.g. mean squared error between predicted responses and actual responses
- Evaluator classes have an evaluate() method, which takes a DataFrame, and returns a double representing the evaluation metric
- Examples
  - BinaryClassificationEvaluator
  - CrossValidator

Automated Model Tuning

- An important task in ML is model selection, when one attempts to find the best model or parameters for a given task. (This is also called tuning.)
- In MLLib, tuning may be done for individual Estimators such as LogisticRegression, or for entire Pipelines which include multiple algorithms, feature extraction etc.
  - Hence, users can tune an entire Pipeline at once, rather than tuning each element in the Pipeline separately.
- MLlib supports model selection using tools such as CrossValidator and TrainValidationSplit. These tools require the following items:
  - An Estimator, which contains algorithm or Pipeline to tune
  - Set of ParamMaps, specifying the parameters to choose from, sometimes called a “parameter grid” to search over
  - An Evaluator, which implements the evaluation metric which should be measured in order to assess how well a fitted Model is performing

Automated Model Tuning

- Spark's model selection tools work as follows:
  1. They split the input data into separate training and test datasets.
  2. For each (training, test) pair, they iterate through the set of ParamMaps:
     - For each ParamMap, they fit the Estimator using those parameters, get the fitted Model, and evaluate the Model's performance using the Evaluator.
  3. They select the Model produced by the best-performing set of parameters.
- We'll see more about model tuning next week.

K-Means Clustering

```python
from pyspark.ml.clustering import KMeans
from pyspark.ml.evaluation import ClusteringEvaluator

dataset = spark.read.format("libsvm")\
    .load(f"{data_dir}/mllib/sample_kmeans_data.txt")

print(f"The type of 'dataset' is: {type(dataset)}")
dataset.show(10, False)

# Trains a k-means model.
kmeans = KMeans().setK(2).setSeed(1)
model = kmeans.fit(dataset)

# Make predictions
predictions = model.transform(dataset)

# Evaluate clustering by computing Silhouette score
evaluator = ClusteringEvaluator()
silhouette = evaluator.evaluate(predictions)
print("Silhouette with squared euclidean distance = " +
str(silhouette))

# Shows the result.
centers = model.clusterCenters()
print("Cluster Centers: ")
for center in centers:
print(center)
```

```
The type of 'dataset' is: <class 'pyspark.sql.dataframe.DataFrame'>
+-----+-------------------------+
|label|features                 |
+-----+-------------------------+
|0.0  |(3,[],[])                |
|1.0  |(3,[0,1,2],[0.1,0.1,0.1])|
|2.0  |(3,[0,1,2],[0.2,0.2,0.2])|
|3.0  |(3,[0,1,2],[9.0,9.0,9.0])|
|4.0  |(3,[0,1,2],[9.1,9.1,9.1])|
|5.0  |(3,[0,1,2],[9.2,9.2,9.2])|
+-----+-------------------------+

Silhouette with squared euclidean distance = 0.9997530305375207
Cluster Centers:
[9.1 9.1 9.1]
[0.1 0.1 0.1]
```

Linear Regression

```python
from pyspark.ml.regression import LinearRegression

# Load training data
training = spark.read.format("libsvm")\
.load(f"{data_dir}/mllib/sample_linear_regression_data.txt")

print(f"The type of 'training' is: {type(training)}")
training.show()

lr = LinearRegression(maxIter=10, regParam=0.3, elasticNetParam=0.8)

# Fit the model
lrModel = lr.fit(training)

# Print the coefficients and intercept for linear regression
print("Coefficients: %s" % str(lrModel.coefficients))
print("Intercept: %s" % str(lrModel.intercept))

# Summarize the model over the training set and print out some
metrics
trainingSummary = lrModel.summary
```

```
The type of 'training' is: <class
'pyspark.sql.dataframe.DataFrame'>
+-------------------+--------------------+
|              label|            features|
+-------------------+--------------------+
| -9.490009878824548|(10,[0,1,2,3,4,5,...|
| 0.2577820163584905|(10,[0,1,2,3,4,5,...|
| -4.438869807456516|(10,[0,1,2,3,4,5,...|
|-19.782762789614537|(10,[0,1,2,3,4,5,...|
| -7.966593841555266|(10,[0,1,2,3,4,5,...|
| -7.896274316726144|(10,[0,1,2,3,4,5,...|
| -8.464803554195287|(10,[0,1,2,3,4,5,...|
| 2.1214592666251364|(10,[0,1,2,3,4,5,...|
| 1.0720117616524107|(10,[0,1,2,3,4,5,...|
|-13.772441561702871|(10,[0,1,2,3,4,5,...|
| -5.082010756207233|(10,[0,1,2,3,4,5,...|
|  7.887786536531237|(10,[0,1,2,3,4,5,...|
| 14.323146365332388|(10,[0,1,2,3,4,5,...|
|-20.057482615789212|(10,[0,1,2,3,4,5,...|
|-0.8995693247765151|(10,[0,1,2,3,4,5,...|
| -19.16829262296376|(10,[0,1,2,3,4,5,...|
|  5.601801561245534|(10,[0,1,2,3,4,5,...|
|-3.2256352187273354|(10,[0,1,2,3,4,5,...|
| 1.5299675726687754|(10,[0,1,2,3,4,5,...|
| -0.250102447941961|(10,[0,1,2,3,4,5,...|
+-------------------+--------------------+
only showing top 20 rows

Coefficients: [0.0,0.3229251667740594,-
0.3438548034562219,1.915601702345841,0.05288058680386255,0.7659627
20459771,0.0,-0.15105392669186676,-
0.21587930360904645,0.2202536918881343]
Intercept: 0.15989368442397356
```

Science and Technology Facilities Council

Hartree Centre

Linear Support Vector Classifier

```python
from pyspark.ml.classification import LinearSVC

# Load training data
training = spark.read.format("libsvm")\
    .load(f"{data_dir}/mllib/sample_libsvm_data.txt")

print(f"The type of 'training' is: {type(training)}")
training.show()

lsvc = LinearSVC(maxIter=10, regParam=0.1)

# Fit the model
lsvcModel = lsvc.fit(training)

# Print the coefficients and intercept for linearsSVC
print("Coefficients: " + str(lsvcModel.coefficients))
print("Intercept: " + str(lsvcModel.intercept))
```

```
The type of 'training' is: <class 'pyspark.sql.dataframe.DataFrame'>

+-----+--------------------+
|label|            features|
+-----+--------------------+
|  0.0|(692,[127,128,129...|
|  1.0|(692,[158,159,160...|
|  1.0|(692,[124,125,126...|
|  1.0|(692,[152,153,154...|
|  1.0|(692,[151,152,153...|
|  0.0|(692,[129,130,131...|
|  1.0|(692,[158,159,160...|
|  1.0|(692,[99,100,101,...|
|  0.0|(692,[154,155,156...|
|  0.0|(692,[127,128,129...|
|  1.0|(692,[154,155,156...|
|  0.0|(692,[153,154,155...|
|  0.0|(692,[151,152,153...|
|  1.0|(692,[129,130,131...|
|  0.0|(692,[154,155,156...|
|  1.0|(692,[150,151,152...|
|  0.0|(692,[124,125,126...|
|  0.0|(692,[152,153,154...|
|  1.0|(692,[97,98,99,12...|
|  1.0|(692,[124,125,126...|
+-----+--------------------+
```

Science and Technology Facilities Council
Hartree Centre

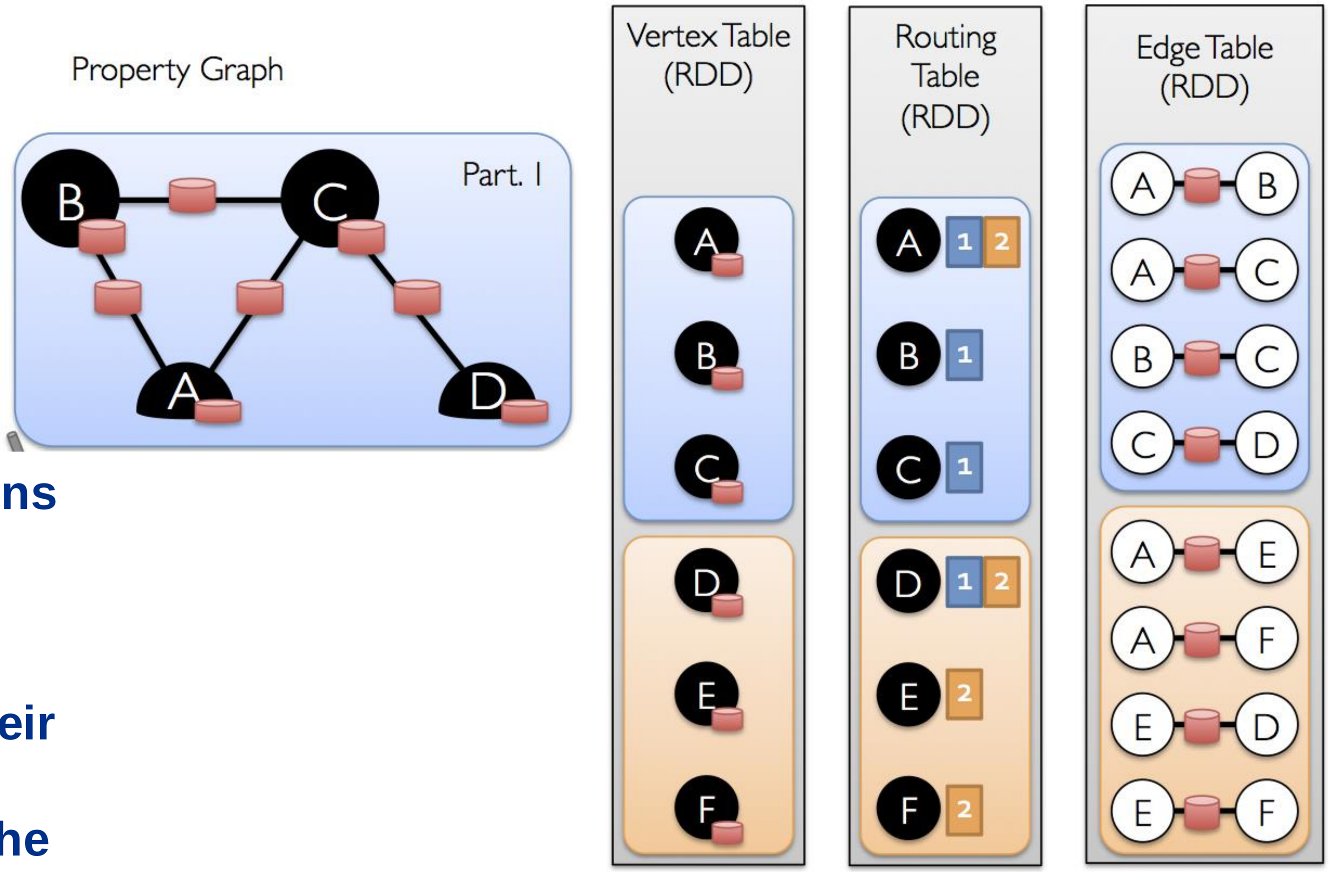
GraphX

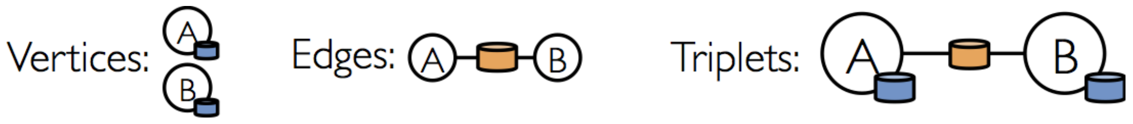
Introduction

- GraphX is a library for graph-based computation in Spark
- It is based around directed multigraphs:
  - Directed graphs have directions associated with the edges
  - Multigraphs can have multiple parallel edges going between the same vertices
- Specifically, GraphX uses "property graphs", which are directed multigraphs with user-defined objects attached to each vertex and edge.
- The ability to support parallel edges simplifies modeling scenarios where there can be multiple relationships (e.g., co-worker and friend) between the same vertices.
- Each vertex is keyed by a unique 64-bit long identifier (VertexId).
- Similarly, edges have corresponding source and destination vertex identifiers.

Data Structures for Property Graphs

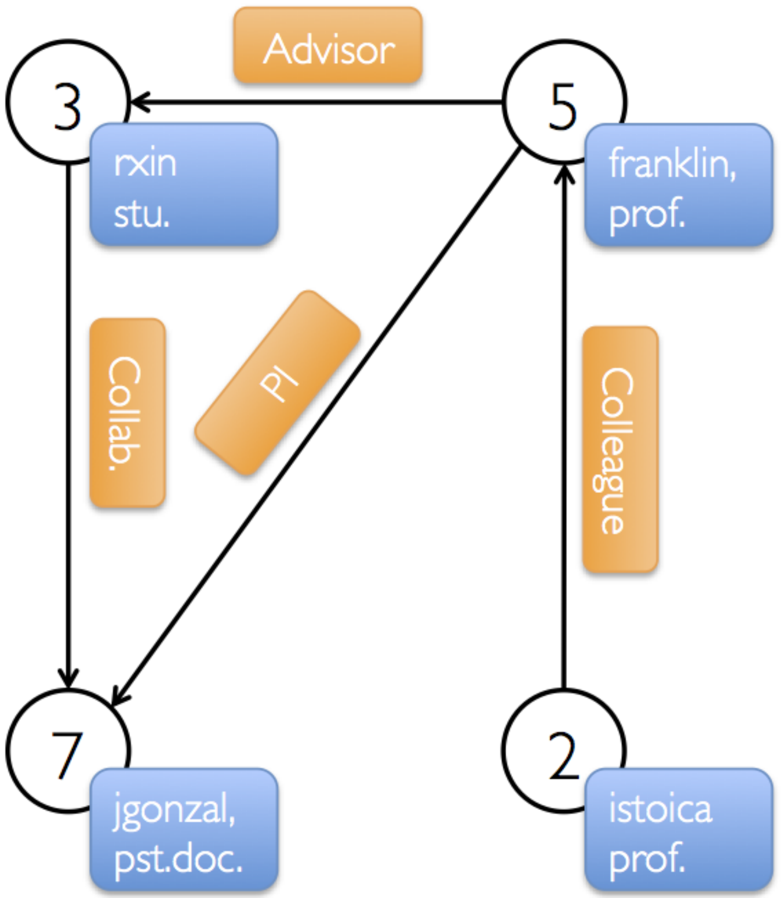
- Property graphs are represented with the following RDDs
- Vertex table:
  - Stores the vertices and their properties
- Edge table:
  - Stores the edges and their properties
- Routing Table:
  - Stores information about which partitions messages should be sent to
- Edge triplets:
  - Derived from the above data structures
  - Stores the graph's edges, along with their adjacent vertex properties
  - Triplets are a 3-way join of edges with the vertices on each side





Property Graph Example

Property Graph



Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

Edge Table

SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

GraphX Algorithms

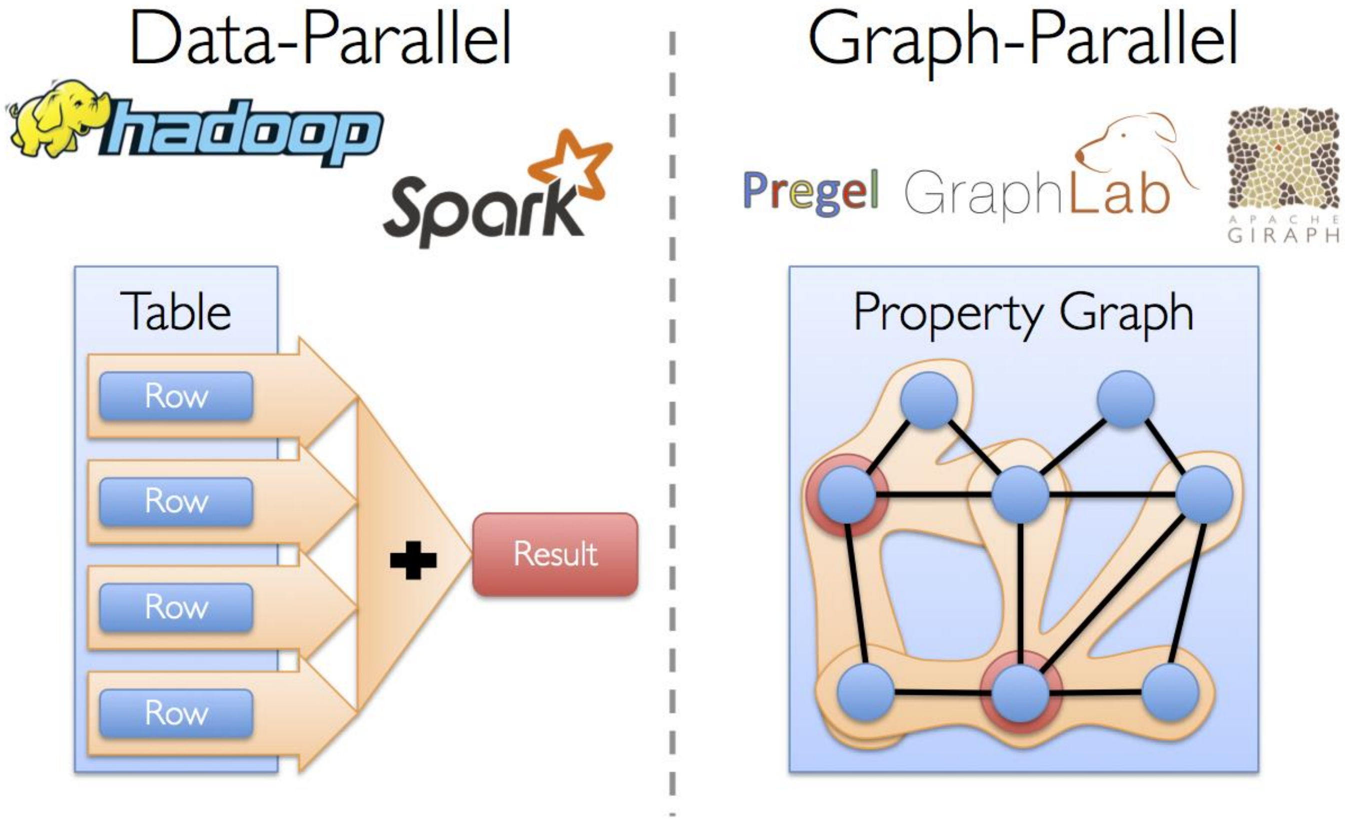
- GraphX algorithms are based around distributed message passing inside multigraphs.
- To support this, GraphX exposes an operator called aggregateMessages
  - As we'll see, aggregateMessages is analogous to MapReduce, except it runs on graphs instead of (key, value) collections
- AggregateMessages takes two user-defined functions:
  - sendMsg:
    - Analogous to the map function in map-reduce.
    - Takes an EdgeContext, which exposes the source and destination attributes along with the edge attribute and functions to send messages to the source and destination attributes (called sendToSrc, and sendToDst).
  - MergeMsg:
    - Analogous to the reduce function in map-reduce
    - Takes two messages destined to the same vertex and yields a single message.

GraphX Algorithms

- AggregateMessages works by:
  - Scanning the triplets on each edge partition
  - Running the sendMsg function for each of those triplets
  - Aggregating the messages to get a result
- The aggregateMessages operator returns a VertexRDD[Msg] containing the aggregate message (of type Msg) destined to each vertex. Vertices that did not receive a message are not included in the returned VertexRDD.

Distribution: The Graph Parallel Pattern

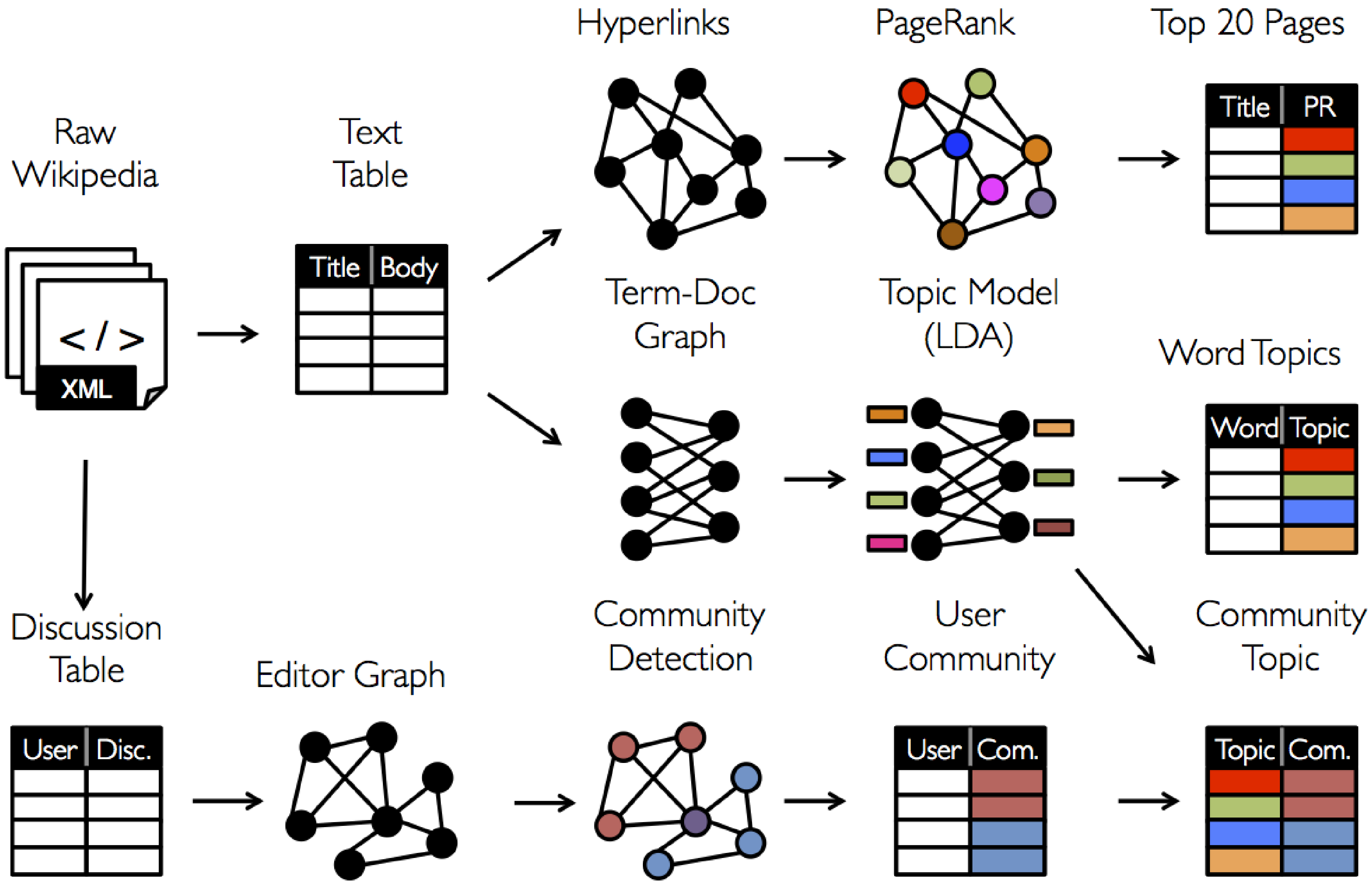
- Message passing graph algorithms can be parallelised because each node will only receive messages from nodes which are close to it in the graph
- GraphX relies on this fact to distribute graph algorithm computations across the nodes in a Spark cluster
- In fact, GraphX was created to express this pattern
  - It's called the *graph parallel pattern*



Motivating Examples

- Examples of graph problems in big data:
  - Social networks: community cohesiveness; page rank
  - Website links: page rank
  - Wikipedia links: graphs of top pages and top editors; community detection for editors.
- Page rank
  - Uses links as a vote of importance
  - Link counts for more if it's coming from an important page
- Triangle counting
  - Measures the cohesiveness of communities
  - Strong communities have more triangles – e.g. for social networks, a person's friends tend to know each other.
- Message passing is core to these and other graph algorithms, and hence they can be implemented in GraphX, as we shall see...

Graph-Based Examples for Web Data



Science and Technology Facilities Council

Hartree Centre

Basic Graph Operations

```
from graphframes import *
from graphframes.examples import Graphs

# Create a Vertex DataFrame with unique ID column "id"
v = sqlContext.createDataFrame([
("a", "Alice", 34),
("b", "Bob", 36),
("c", "Charlie", 30),
], ["id", "name", "age"])
# Create an Edge DataFrame with "src" and "dst" columns
e = sqlContext.createDataFrame([
("a", "b", "friend"),
("b", "c", "follow"),
("c", "b", "follow"),
], ["src", "dst", "relationship"])
# Create a GraphFrame
g = GraphFrame(v, e)
g.vertices.show()
g.edges.show()

# Query: Get in-degree of each vertex.
g.inDegrees.show()
```

```
+---+-------+---+
| id|   name|age|
+---+-------+---+
|  a|  Alice| 34|
|  b|    Bob| 36|
|  c|Charlie| 30|
+---+-------+---+

+---+---+------------+
|src|dst|relationship|
+---+---+------------+
|  a|  b|      friend|
|  b|  c|      follow|
|  c|  b|      follow|
+---+---+------------+

+---+--------+
| id|inDegree|
+---+--------+
|  b|       2|
|  c|       1|
+---+--------+
```

A Larger Graph

```
from graphframes import *
from graphframes.examples import Graphs

g = Graphs(sqlContext).friends() # Get example graph

# Display the vertex and edge DataFrames

g.vertices.show()

g.edges.show()

g.inDegrees.show()

# Find the youngest user's age in the graph.
# This queries the vertex DataFrame.
g.vertices.groupBy().min("age").show()
```

```
+---+-------+---+
| id|   name|age|
+---+-------+---+
|  a|  Alice| 34|
|  b|    Bob| 36|
|  c|Charlie| 30|
|  d|  David| 29|
|  e| Esther| 32|
|  f|  Fanny| 36|
+---+-------+---+
+---+---+------------+
|src|dst|relationship|
+---+---+------------+
|  a|  b|      friend|
|  b|  c|      follow|
|  c|  b|      follow|
|  f|  c|      follow|
|  e|  f|      follow|
|  e|  d|      friend|
|  d|  a|      friend|
+---+---+------------+
+---+--------+
| id|inDegree|
+---+--------+
|  b|       2|
|  c|       2|
|  f|       1|
|  d|       1|
|  a|       1|
+---+--------+
+--------+
|min(age)|
+--------+
|      29|
+--------+
```

Science and Technology Facilities Council
Hartree Centre

Shortest Paths and Triangle Count

- A vertex is part of a triangle when it has two adjacent vertices with an edge between them
- GraphX implements a triangle counting algorithm in the TriangleCount object that determines the number of triangles passing through each vertex, providing a measure of clustering
- We compute the triangle count of the social network from the previous slide

```
+---+-----------------+
| id|        distances|
+---+-----------------+
|  f|               {}|
|  e|{a -> 2, d -> 1}|
|  d|{a -> 1, d -> 0}|
|  c|               {}|
|  b|               {}|
|  a|         {a -> 0}|
+---+-----------------+
```

```
results = g.shortestPaths(landmarks=["a", "d"])
results.select("id", "distances").show()

results = g.triangleCount()
results.select("id", "count").show()
```

```
+---+-----+
| id|count|
+---+-----+
|  a|    0|
|  b|    0|
|  c|    0|
|  d|    0|
|  e|    0|
|  f|    0|
+---+-----+
```

PageRank

- PageRank measures the importance of each vertex in a graph, assuming an edge from u to v represents an endorsement of v's importance by u
  - For example, if a Twitter user is followed by many others, the user will be ranked highly
- GraphX comes with static and dynamic implementations of PageRank as methods on the PageRank object
- Static PageRank runs for a fixed number of iterations, while dynamic PageRank runs until the ranks converge (i.e., stop changing by more than a specified tolerance)

```
results = g.pageRank(resetProbability=0.01, maxIter=20)
results.vertices.select("id", "pagerank").show()
```

```
+---+--------------------+
| id|            pagerank|
+---+--------------------+
|  f|0.014950000000000028|
|  e|0.010000000000000018|
|  d|0.014950000000000028|
|  c|           2.9651995|
|  b|              2.9701|
|  a|0.024800500000000048|
+---+--------------------+
```

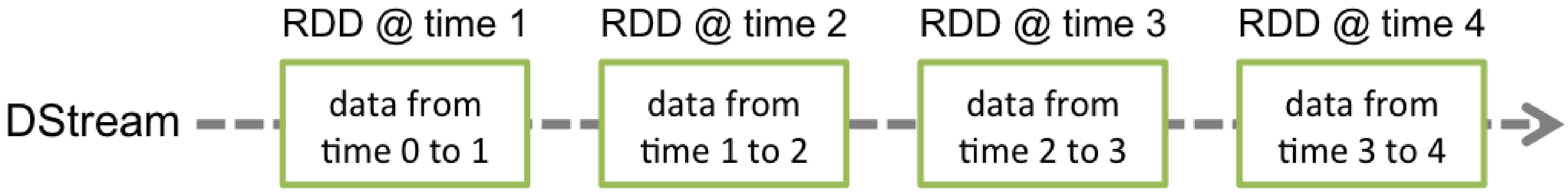
Spark Streaming

Spark Streaming Introduction

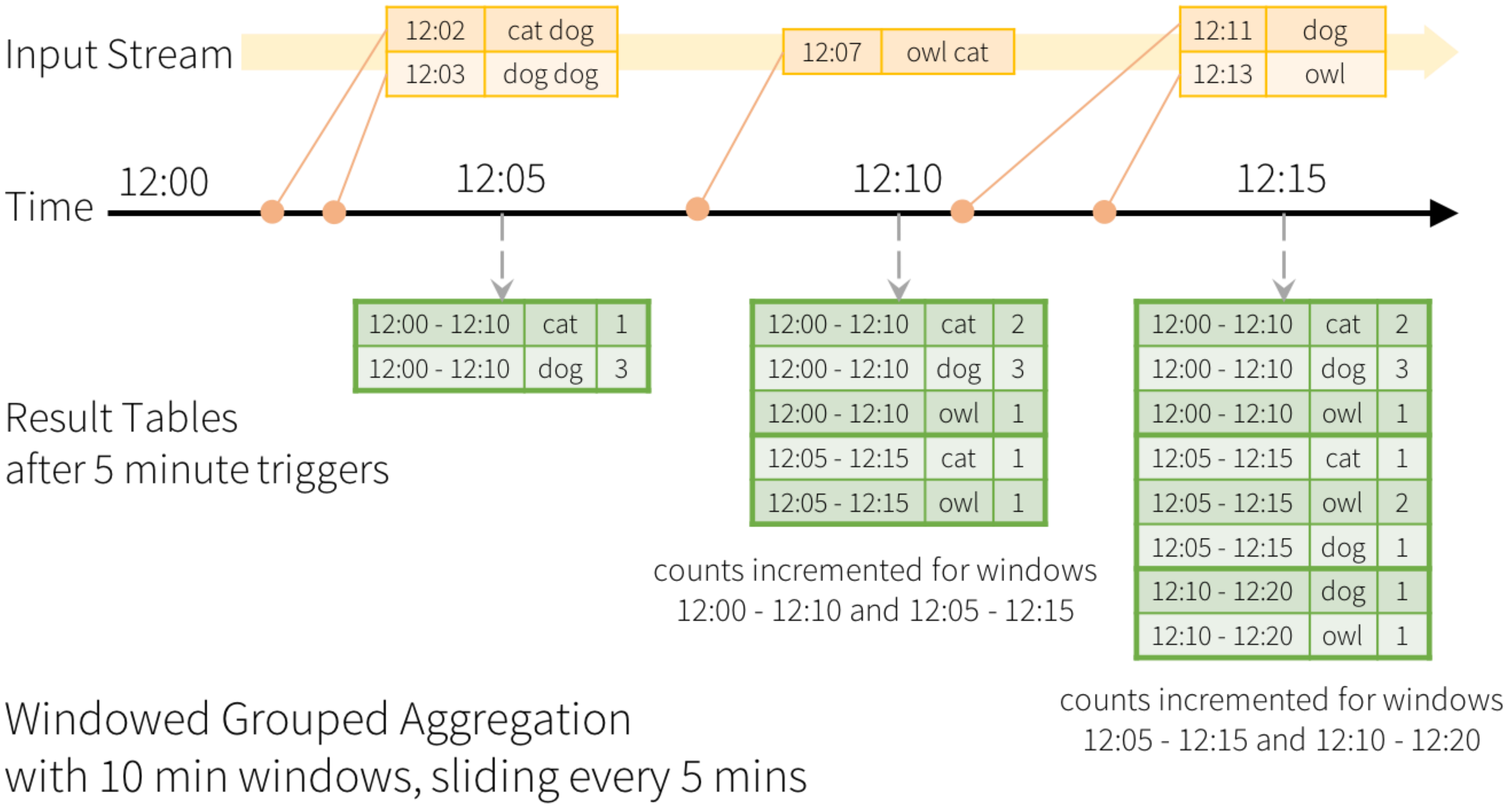
- Spark Streaming enables scalable, high-throughput, fault-tolerant stream processing of live data streams
- Data can be ingested from many sources, e.g. Kafka, Kinesis, or TCP sockets
- Streaming data can be processed using functions such as map, reduce, join and window (which we'll see shortly).
- Processed data can be directed to filesystems, databases, and live dashboards.
- You can also apply Spark's machine learning and graph processing algorithms to data produced by Spark Streaming.

Spark Streaming Data Structures

- DStream (i.e. Discretized Stream) is the basic abstraction provided by Spark Streaming.
- It represents a continuous stream of data, either the input data stream received from source, or the processed data stream generated by transforming the input stream.
- Internally, a DStream is represented by a continuous series of RDDs.
- Each RDD in a DStream contains data from a certain interval, as shown in the following figure.

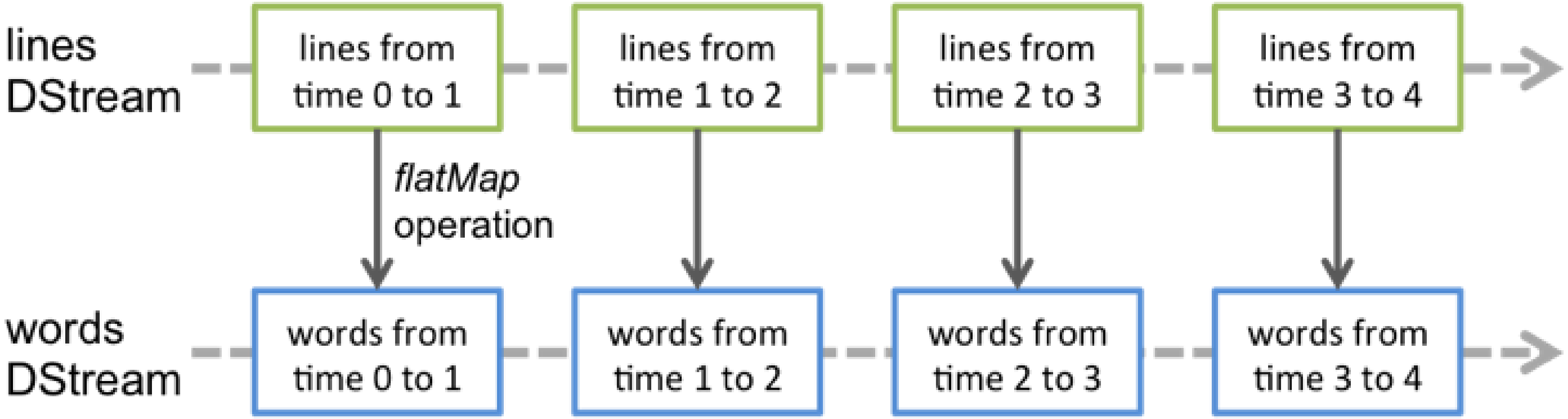


Spark Streaming Data Structures



Spark Streaming Algorithms

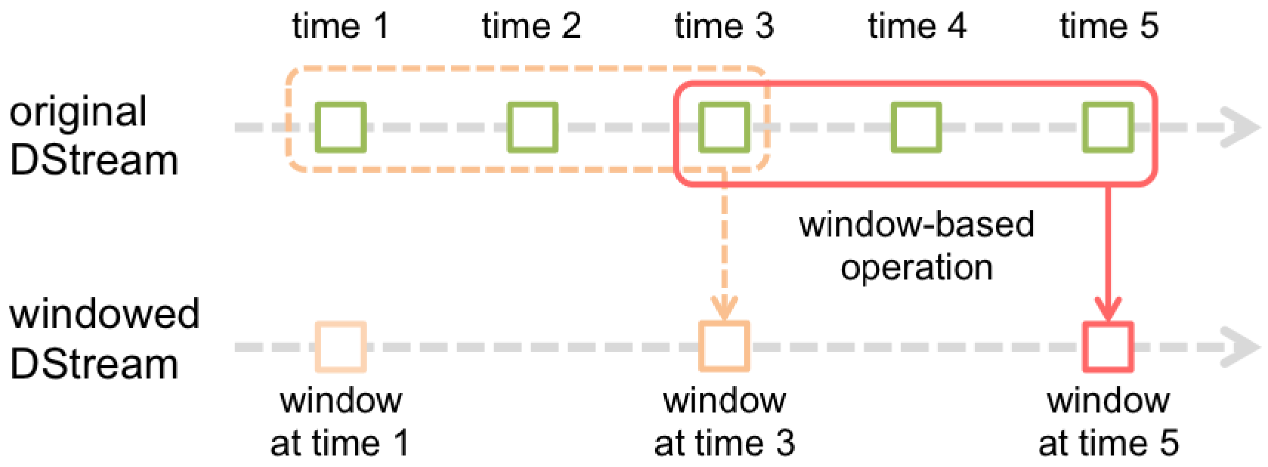
- Spark Streaming receives live input data streams and divides the data into batches
- These batches are processed by the Spark engine to generate the final stream of results in batches
- DStreams are created either from input data streams from sources such as Kafka, or by applying high-level operations on other DStreams
- Any operation applied on a DStream translates to operations on the underlying RDDs
  - This is shown in the figure below
- These underlying RDD transformations are computed by the Spark engine



Science and Technology Facilities Council
Hartree Centre

Window Operations

- Spark Streaming also provides windowed computations, which allow you to apply transformations over a sliding window of data
- Every time the window slides over a source DStream, the source RDDs that fall within the window are combined and operated upon to produce the RDDs of the windowed DStream.
- In this specific case, the operation is applied over the last 3 time units of data, and slides by 2 time units.
- Any window operation needs to specify two parameters.
  - window length - The duration of the window (3 in the figure).
  - sliding interval - The interval at which the window operation is performed (2 in the figure).
- These two parameters must be multiples of the batch interval of the source DStream (1 in the figure).



Science and Technology Facilities Council
Hartree Centre

Streaming Example: Word Counting

```
from pyspark import SparkContext
from pyspark.streaming import StreamingContext

sc = SparkContext(appName="PythonStreamingNetworkWordCount")
ssc = StreamingContext(sc, 1)

lines = ssc.socketTextStream("localhost", 9999)
counts = lines.flatMap(lambda line: line.split(" "))\
    .map(lambda word: (word, 1))\
    .reduceByKey(lambda a, b: a + b)
counts.pprint()

ssc.start()

ssc.awaitTermination()
```

```
> nc -lk 9999
hello world
```

```
-------------------------------------------
Time: 2022-12-04 18:05:36
-------------------------------------------
('hello', 1)
('world', 1)

-------------------------------------------
Time: 2022-12-04 18:05:37
-------------------------------------------
```

Science and
Technology
Facilities Council
Hartree Centre

Science and Technology Facilities Council

Hartree Centre

Questions?

Science and Technology Facilities Council

Hartree Centre

Thank you

Full Name

Full.name@stfc.ac.uk

hartree.stfc.ac.uk @HartreeCentre STFC Hartree Centre hartree@stfc.ac.uk