Apurva Nandan
Tommi Jalkanen

Analyzing Large Datasets using Apache Spark

November 16– 17, 2017
CSC – IT Center for Science Ltd, Espoo


```
>>>rdd = sc.parallelize([('Python',2),
('Java',3), ('Scala',4), ('R',5), ('Java', 8),
('Python',9), ('Scala',3)])
>>>rdd_avg = rdd.map(lambda item:
(item[0], [item[1],
1])).reduceByKey(lambda a,b: (a[0] + b[0],
a[1] + b[1])).map(lambda item: (item[0],
item[1][0] / item[1][1]))
>>>rdd_avg.collect()
```

```
>>>rdd_for_df = rdd.map(lambda
item: Row(language=item[0],
count=item[1]))
>>>df = rdd_for_df.toDF()
>>>df_avg =
df.groupBy('language').avg('count')
>>>df_avg.show()
```

Agenda

	Thu		Fri
9.00-9.30	Overview and architecture of Spark	9.00-9.30	Spark Dataframes and SQL overview
9.30-10.15	Basics of RDDs + Demo	9.30-10.15	Exercises
10.15-10.30	Coffee break	10.15-10.30	Coffee break
10.30-11.00	RDD: Transformations and Actions	10.30-10.45	Dataframes and SQL contd.
11.00-12.00	Exercises	10.45-12.00	Exercises
12.00-13.00	Lunch break	12.00-13.00	Lunch break
13.00-13.30	Word Count Example	13.00-14.00	Exercises contd.
13.30-14.00	Exercises		
14.00-14.15	Short Overview of MLlib	14.00-14.30	Best practices and other useful stuff
14.15-14.30	Coffee break	14.30-14.45	Coffee break
14.30-15.30	Final Day 1 Exercise	14.45-15.00	Brief overview of Spark Streaming
		15.00-15.30	Demo:Processing live twitter stream
15.30-16.00	Summary of the first day & exercises walk-through	15.30-16.00	Summary of the second day and exercises walk-through

Workstations
Username: **cscuser**
Password: **ump1hank1**

Wireless accounts for CSC-guest network behind the badges. Alternatively, use the eduroam network with your university accounts or the LAN cables on the tables.
Accounts to Sisu supercomputer delivered separately.

Analyzing large datasets using Apache Spark

Apurva Nandan
Tommi Jalkanen

CSC – Finnish research, education, culture and public administration ICT knowledge center

Overview

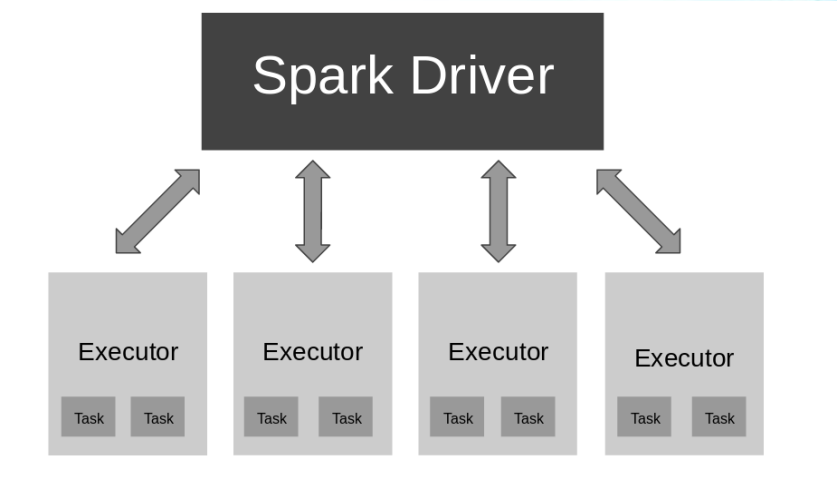
Apache Spark

- Apache Spark is a fast, Open Source, big data based engine for large scale data analysis and distributed processing
- Developed in Scala and runs on Java Virtual Machine
- Can be Used with Scala, Java, Python or R
- Works on the Map-Reduce concept to do the distributed processing

Apache Spark: Key Concepts

- Stores the data into memory when needed
- Follows distributed processing concepts for distributing the data across the nodes of the cluster

Apache Spark: Execution of code



SparkContext

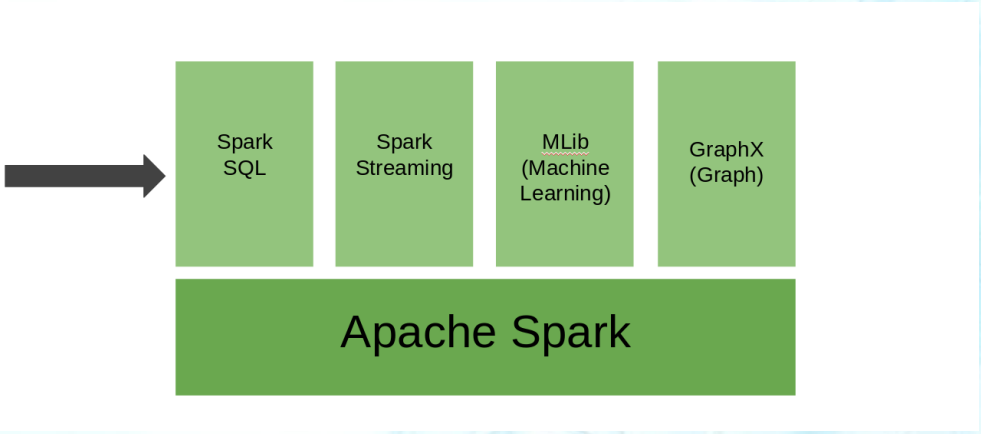
- Main entry point of a spark application
- Sparkcontext sets up internal services and establishes a connection to a Spark execution environment.
- Used to create RDDs, access Spark services and run jobs.

```
>>> from pyspark import SparkContext
>>> sc = SparkContext()
```

Spark vs Hadoop

- Faster than Hadoop in many cases specially iterative algorithms
- Spark Stores the data in memory which allows faster chaining of operations vs Hadoop Map Reduce which stores the output to disk for any map or reduce operation
- Much more simpler programming API as compared to Hadoop – Less time writing complex map reduce scripts
- Equally good fault tolerance

Spark Stack



Spark: Parallelism

- Spark allows the user to control parallelism by providing partitions when creating an RDD / Dataframe or changing the configuration
- Example: >>>sc.textFile(inputfile, numPartitions)
- Partitioning : accessing all hardware resources while executing a Job.
- More Partition = More Parallelism
- The user should check the hardware and see how many tasks can run on each executor. There has to be a balance between under utilizing an executor and over utilizing it

Spark Programming

Map-Reduce Paradigm

- Programming model developed by Google
- As the name suggests, there are two phases for processing the data – Map and Reduce
  - Map: Transform each element in the data set
  - Reduce: Combine the data according to some criteria
- Spark uses the Map Reduce programming model for distributed processing

Spark: RDD

- RDD a.k.a Resilient Distributed Datasets
- Spark's fault tolerant dataset which can be operated in a distributed manner by running the processing across all the nodes of a cluster
- Can be used to store any type of element in it
- Some Special types of RDDs:
  - Double RDD: for storing numerical data
  - Pair RDD: For storing key pair values

Spark: Pair RDD

- Special form of RDDs which are used for map reduce based functions
- They are in a form of key/value pairs <K,V>. Key and Value can be of any type
- They can be used for running map, reduce, filter, sort, join or other functions present in the spark API
- Generally, Pair RDDs are used more often

Spark: Creating RDDs

- By distributing the native python collection:

```
>>> sc.parallelize(['this', 'is', 'an', 'example'])
>>> sc.parallelize(xrange(1,100))
```

- By reading text files from local file system , HDFS or object storage:

```
>>> sc.textFile('file:///example.txt')
>>> sc.textFile('hdfs:///hdfsexample.txt')
```

This output RDD is an RDD of strings

Spark: RDD Transformations and Actions

- Spark programming resembles functional programming style
- Declaritive function calls without side effects vs. imperative data handling which changes state of the program

Normal function example:

```
num = 0
def increment():
  global num
  num += 1
```

Functional example:

```
def increment(num):
  return num + 1
```

Spark: RDD Transformations

- **map**(func) :Return a new RDD formed by passing each element of the source through a function func.
- **filter**(func) :Return a new RDD formed by selecting elements of the source on which func returns true.
- **flatMap**(func) : Similar to map, but flattens the final result.
- **groupByKey**([numTasks]) :When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. Note: If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using reduceByKey or aggregateByKey will yield much better performance.
- **reduceByKey**(func, [numTasks]) :When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function func, which must be of type (V,V) => V.
- **sortByKey**([ascending], [numTasks]) : When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument.
- **join**(otherDataset, [numTasks]) : When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are also supported
- **cogroup**(otherDataset, [numTasks]) : When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable<V>, Iterable<W>)) tuples. This operation is also called groupWith.

Spark: RDD Actions

- **collect**() : Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
- **count**() :Return the number of elements in the dataset.
- **first**() :Return the first element of the dataset (similar to take(1)).
- **take**(n) :Return an array with the first n elements of the dataset.
- **takeSample**(withReplacement, num, [seed]) : Return an array with a random sample of num elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
- **takeOrdered**(n, [ordering]) Return the first n elements of the RDD using either their natural order or a custom comparator.
- **saveAsTextFile**(path) Write the elements of the dataset as a text file (or set of text files) in the local filesystem, HDFS
- **CountByKey**() : Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
- **foreach**(func) : Run a function func on each element of the dataset. This is usually done for side effects

Spark: Lambda Expressions

- Lambda expressions: simple functions that can be written as an expression.
- Do not support multi-statement functions or statements that do not return a value
- Very useful in Spark transformations. *Why?*
- For eg. **lambda x: x*x**
- *Will take x as an input and return x^2 as the output*

RDD: Transformations - Map

- Used to apply a function to each element in the RDD
- This is used to modify elements of the RDD. One could completely change the format of the RDD

```
>>> rdd = sc.parallelize(range(1,10)) # [1,2,.....10]
>>> rdd_map = rdd.map(lambda x: (x, x))
>>> rdd_map.collect()  # [(1,1), (2,2) ..... (10,10)]
>>> rdd_map = rdd_map.map(lambda x: (x[0]*2, x[1])).collect()
```

- Special case: flatMap:

```
>>> rdd_flatmap = rdd_map.flatMap(lambda x: (x[0], x[1] + 1))
```

RDD: Transformations - Filter

- Used to filter out the RDD according to a defined criteria

```
>>> rdd = sc.textFile('HP.txt')
```

the place where things are hidden
if you have to ask you will never know
if you know you need only ask

```
>>> rdd = rdd.filter(lambda x: x.startswith('the'))
```

the place where things are hidden

RDD: Tabular to PairRDD

Always remember, if your RDD turns out to be something like this:
col1, col2, col3
1,[1,2],'hello'
2,[3,4],'moi'
3,[5,6],'hola'
4,[7,8],'ciao'

Convert it into a key/pair format like:
1, ([1,2], 'hello')
…..

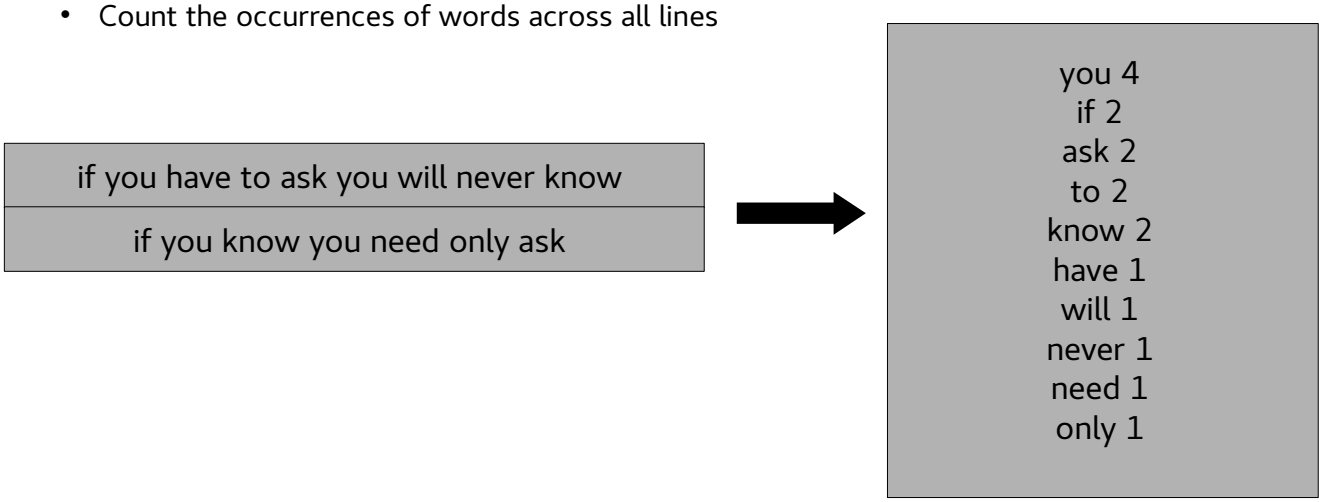
- This means, now you have one key and the rest of the things are values. This helps Spark operate the RDD as a PairRDD. (Also, reduceByKey, countByKey will get their 'Key' ;-))

RDD: Shuffles

- Happens for some Spark transformations like groupByKey and reduceByKey.
- Data required in computations of records can be in different partitions of the parent RDD.
- The tasks require that all the tuples with same key have to be in the same partition and should be processed by the same task.
- Spark executes a shuffle, which involves transfer of data across the workers of the cluster and results in a new stage with completely new set of partitions.
- Expensive operations as they incur heavy disk and network I/O
- Primary goal when choosing an arrangement of operators is to reduce the number of shuffles and the amount of data shuffled. Because shuffles are expensive.
- All the shuffle data is written to disk and transferred over the network.
- join, cogroup, *By or *ByKey transformations result in shuffles. Each of these operations have different costs and as a novice one has to be careful when selecting one over another

Example: Word Count

- Count the occurrences of words across all lines



Example: Word Count

```
# Load the text file
>>> rdd = sc.textFile('HP.txt')
# Split the text into words and flatten the results. Why?
>>> words = rdd.flatMap(lambda line: line.split())
>>> words.collect()
('if' , 'you', 'have', 'to', 'ask', 'you' 'will', 'never', 'know', 'if', 'you', 'know' , 'you' ,
'need' , 'only', 'to', 'ask')
```

Example: Word Count

```
# Load the text file
>>> rdd = sc.textFile('HP.txt')
# Split the text into words and flatten the results. Why?
>>> words = rdd.flatMap(lambda line: line.split())

>>> words_map = words.map(lambda x: (x,1))
```

(('if',1) , ('you',1), ('have',1), ('to',1), ('ask',1), ('you',1), ('will',1), ('never',1), ('know',1), ('if',1), ('you',1), ('know',1) , ('you',1) , ('need',1) , ('only',1), ('to',1), ('ask',1))

Example: Word Count

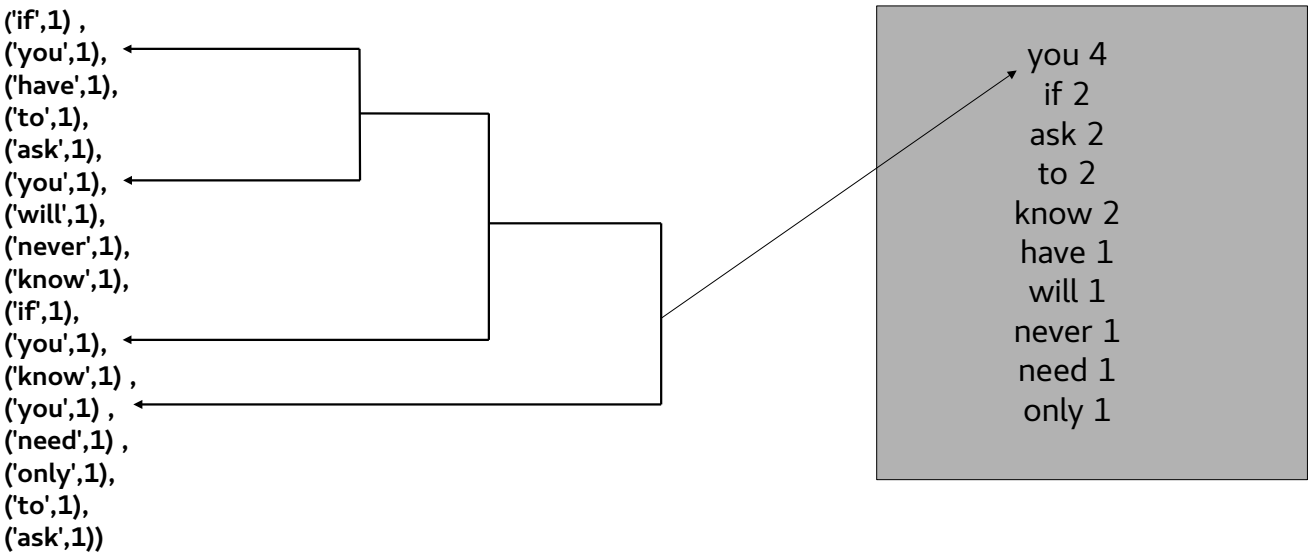
```
# Load the text file
>>> rdd = sc.textFile('HP.txt')
# Split the text into words and flatten the results. Why?
>>> words = rdd.flatMap(lambda line: line.split())

>>> words_map = words.map(lambda x: (x,1))
```

(('if',1) , ('you',1), ('have',1), ('to',1), ('ask',1), ('you',1), ('will',1), ('never',1), ('know',1), ('if',1), ('you',1), ('know',1) , ('you',1) , ('need',1) , ('only',1), ('to',1), ('ask',1))

```
>>> words_count = words_map.reduceByKey(lambda a,b: a+b)
```

Example: Word Count



Machine learning in Spark

Spark Mllib – Machine learning in Spark

- Apache Spark's scalable machine learning library.
- Used with Spark's APIs and compatible with NumPy in Python
- 100x faster than Hadoop
- Don't reinvent the wheel: If you are looking to do use any of the machine learning based techniques in your work , it is always a best idea to see if Spark has the algorithm in MLlib library
- The algorithms provided are much more optimized and ready to use
- This course does not focus on detailed Machine learning concepts but just gives an overview

Spark Mllib – Machine learning in Spark

- **Classification**: logistic regression, naive Bayes,...
- **Regression**: generalized linear regression, survival regression,...
- **Decision trees**, random forests, and gradient-boosted trees
- **Recommendation**: alternating least squares (ALS)
- **Clustering**: K-means, Gaussian mixtures (GMMs),...
- **Topic modeling**: latent Dirichlet allocation (LDA)
- Frequent itemsets, association rules, and sequential pattern mining

- Feature transformations: standardization, normalization, hashing,...
- ML Pipeline construction
- Model evaluation and hyper-parameter tuning
- ML persistence: saving and loading models and Pipelines

- Other utilities include:
  - Distributed linear algebra: SVD, PCA,...
  - Statistics: summary statistics, hypothesis testing,...

Why Spark SQL?

Calculate AVG : RDD vs DF

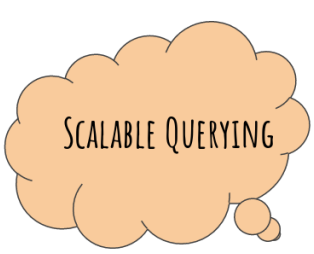
- *Using RDD*

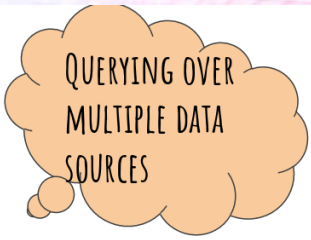
```
>>>rdd = sc.parallelize([('Barcelona',2), ('Rome',3), ('Paris',4), ('Vegas',5), ('Barcelona', 8), ('Vegas',9), ('Rome',3)])
>>>rdd_avg = rdd.map(lambda item: (item[0], [item[1], 1])).reduceByKey(lambda a,b: (a[0] + b[0], a[1] + b[1])).map(lambda item: (item[0], item[1][0] / item[1][1]))
>>>rdd_avg.collect()
```

- *Using Dataframes*

```
>>>rdd_for_df = rdd.map(lambda item: Row(city=item[0], count=item[1]))
>>>df = rdd_for_df.toDF()
>>>df_avg = df.groupBy('city').avg('count')
>>>df_avg.show()
```

Spark SQL





OPTIMIZATIONS!

SparkSession

- Entry point to Spark SQL.
- First object which needs to be created while developing Spark SQL applications

```
>>> from pyspark.sql import SparkSession
>>> spark = SparkSession.builder.getOrCreate()
```

OR

```
>>>spark = SparkSession \
.builder \
.appName("Python Spark SQL basic example") \
.getOrCreate()
```

Spark SQL: Dataframes

- A distributed collection of data organized into named columns.
- Conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations under the hood (Catalyst Optimizer)
- Can be constructed from JSON files, Parquet files, external databases, Text files, existing RDDs.
- Can be thought of as: RDDs with Schema
- Allows data to be fetched using SQL Statements

Spark SQL: Dataframes

- You can try to fetch the data using SQL queries
- Or, You can use the functions of the dataframe api.
- Both are optimized by Catalyst optimizer
- Immutable
- *In case if you want to perform any complex operation : the dataframe can be trasnformed back to an RDD, then you could use the core spark functions*

Spark SQL: Dataframes

```
>>>df = spark.read.json("people.json")
>>>df.show() # Displays the content
>>>df.printSchema()

# Select only the "name" column
>>>df.select("name").show()

# Write the file as Parquet format
>>>df.write.parquet("people.parquet")
# Read the parquet file
>>>df_people spark.read.parquet("people.parquet")

# Filter the records based on age
>>>df_people_filtered = df_people.filter(df_people['age'] >21)
```

Spark SQL: Dataframes

```
# Read the file
>>>lines = sc.textFile("people.txt")
# Split the columns of text file into two
>>>parts = lines.map(lambda l: l.split(","))

# Remove unwanted spaces
>>>people = parts.map(lambda p: (p[0], p[1].strip()))

# Select only the "name" column
>>>df.select("name").show()
```

Spark SQL: Dataframes

```
# Infer the schema, and register the DataFrame as a table.
>>schemaPeople = spark.createDataFrame(people)
>>schemaPeople.printSchema()
```

```
# Declare schema fields with the name of the field and the type of the field
>>>fields = [StructField("name", StringType(), True), StructField("age",
IntegerType(), True)]

# Create schema
>>>schema = StructType(fields)

# Applying the generated schema to the RDD.
>>>schemaPeople = sqlContext.createDataFrame(people, schema)
>>>schemaPeople.printSchema()
```

Spark SQL: Dataframes

- **filter** – same as SQL where
- **select** – same as sql select
- **join** – joins two based on a given expression
- **groupBy** – Groups the dataframe by specified columns for performing aggregation (min,max,sum,avg,count)
- **orderBy** – sorted by specific cols
- **union** – combining rows of two dataframes
- **limit** – limits result count to specified value

- **distinct** – returns distinct rows in a dataframe
- **drop** – drops a specific col from df
- **dropDuplicates** – duplicate rows removed
- **fillna** – replaces null values
- **dropna** – drops null values
- **replace** – replaces a value in a dataframe with another

- **rdd** – returns the rdd representation of the dataframe
- **foreach** – Applies a function to each record

Spark SQL: Dataframes - Columns

- **alias** – new name for the col

- **startswith** – check if the col value starts with the given string
- **endswith** – check if the col value ends with the given string
- **contains** – contains a string
- **isNull** – check if the col value is null
- **isNotNull**
- **like** – SQL like match

- **cast** – change the column data type

- Example:
  # shows the records which have retro in between for specified column
df.filter(df['col_name'].**like**('%Retro%')

Spark SQL: Dataframes

- In order to use SQL statements, we create → Temporary view
- Temporary views in Spark SQL are session-scoped
- Disappear if the session that creates it terminates.

```
# Create a temporary view from a dataframe
>>>dataframe.createOrReplaceTempView("table_name")
# now, use the SQL statements
>>>result = spark.sql("SELECT * FROM table_name”)
```

Spark SQL: Dataframes – SQL Query

- Query statements:
  - SELECT
  - GROUP BY
  - ORDER BY
- Operators:
  - Relational operators (=, ⇔, ==, <>, <, >, >=, <=, etc)
  - Arithmetic operators (+, -, *, /, %, etc)
  - Logical operators (AND, &&, OR, ||, etc)
- Joins
  - JOIN
  - {LEFT|RIGHT|FULL} OUTER JOIN
  -
- Unions
- Sub-queries
  -

Spark SQL: Dataframes - Optimization

- Only reads the data which is required by:
- Skipping a partition while reading the data
- Skipping the data using statistics (mix, max)
- Using Parquet file format (explained in the next slide)
- Using Partitioning (department=sales)

Apache Parquet

- Compressed , space efficient columnar data format
- Available for any Hadoop ecosystem.
- Can be used with Spark API for reading and saving dataframes
- Read Example: *dataframe = spark.read.parquet('filename.parquet')*
- Write Example: *dataframe.write.parquet('filename.parquet')*
- Most preferred way of storing/reading data from/to Spark
- **ONLY reads the columns that are needed rather than reading the whole file. Limiting I/O cost**

Other Useful Stuff

PySpark Shell

- Allows you to write your code using python in an interactive shell
- Easy way to get started with spark and start writing code, allows quick testing
- To access the pyspark shell
  - sh $SPARK_HOME/bin/pyspark

```
>>>sc #  SparkContext is the main entry point for Spark API (Created by default in pyspark shell)
>>>rdd = sc.parallelize(range(1,10**5))  # create a parallel collection (rdd) with the specified range
>>>rdd.count()  # count the number of elements
>>> rdd.collect() # print the elements currently in the collection
```

Submitting Jobs to Spark

- Everything we did was on a “local” environment i.e. we did not have a real cluster
- The cluster needs to be setup using Apache Ambari / Kubernetes etc
- Once you're on the cluster:
  - You would have to write your code in textfiles with *.py extension
  - Always better to test the code with a pyspark notebook or using pyspark shell first
  - Always try to run the code with a subset of data and then submit it to the cluster
  - Make sure you know the configuration of your cluster first in order to manage the resources better
  - Always store the output in the form of Parquet files, if possible
  - It is perfectly possible to connect a Jupyter Notebook with the cluster

Submitting Jobs to Spark

```
from pyspark import SparkContext, SparkConf # SparkContext Needs to be called first

conf = SparkConf()  # Used to provide optional configurations to the spark application
conf.setAppName('Test app')

sc = SparkContext(conf=conf)
rdd = sc.parallelize(range(1, 10**5))

print(rdd.count())
print(rdd.collect())
```

- # Submit your code : **sh $SPARK-HOME/bin/spark-submit filename.py**

HDFS

- Hadoop Distributed File System - Java-based file system for scalable and reliable data storage
- Designed for spanning large clusters with nominal hardware
- Scalability of up to 200 PB of storage and a single cluster of 4500 servers, supporting close to a billion files and blocks in production
- Fault-tolerant, distributed storage system working with various concurrent data access applications, coordinated by YARN.
- Usually, default storage type with our Spark clusters.

Different Filesystems

- There is a possiblity of using a variety of filesystems considering your storage
- Storing in local filesystem is always an option
- Apart from HDFS and Local FS, we could also store our data on cloud
- As an example:
  - Text file RDDs created using SparkContext's textFile method takes an URI for the file (either a local path on the machine, or a hdfs://, s3n://, etc URI) and reads it as a collection of lines.
  - For example, if you have a swift based storage (An object store on Openstack based cloud environment), the URI becomes: *swift://containerTest.SparkTest/features.parquet*

Repartition vs Coalesce

- Used for changing the number of partitions in RDD or DF
- Why change the partitions?
- Repartition :
  - Tries to shuffle the data all over the network to increase/decrease the number of partitions.
  - Any shuffle operation is costly when it comes to large operations.
- Coalesce:
  - Tries to combine the number of partitions without actually performing the costly shuffle.
  - Can be only used to decrease the number of partitions
- partitionBy:
  - Custom partitions
  - Makes shuffling functions more efficient i.e reduceByKey(), join(), cogroup() etc

More Parallelism

The following parameters can be provided as configs to adjust the parallel behavior too

- spark.default.parallelism
  - Works for raw RDDs – parallelize, join, reduceByKey
  - Ignores the dataframes
- spark.sql.shuffle.partitions
  - For dataframes
  - Number of partitions that are used when shuffling data for joins or aggregations

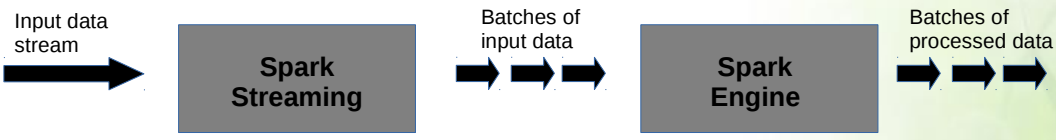
Random Thoughts

- Use *reduceByKey* to perform aggregation or counting instead of *groupByKey*
- Avoid printing huge data sets using *rdd.collect()*
- Try to convert the RDD to a dataframe whenever you can, they offer better performance
- Convert the DF back to RDD in case you need complex operations
- Always be sure to check the partitions for parallelization. You might need to scale up or down accordingly
- Try to accomplish the things using Spark's map, flatMap functions when you need to apply something over the elements.

Running SQL Queries over Live Data

Spark Streaming

- Enables scalable, high-throughput, fault-tolerant stream processing of live data streams.
- Data can be ingested from many sources like Kafka, Flume, Kinesis, or TCP sockets
- Can be processed using complex algorithms expressed with high-level functions like map, reduce, join and window.
- Spark Streaming receives live input data streams
- Divides the data into batches
- Processed by the Spark engine to generate the final stream of results in batches.



Spark Streaming

- Continuous stream of data is called discretized stream or Dstream
- Internally, a DStream is represented as a sequence of RDDs.
- Any operation applied on a DStream translates to operations on the underlying RDDs.

Spark Streaming: SQL

- Real time data over a stream
- Live SQL queries over a streaming data
- *What could be more thrilling than real time data analysis?*
- To use DataFrames and SQL on streaming data: We need to create a SparkSession using the SparkContext that the StreamingContext is using. This is done by creating a lazily instantiated singleton instance of SparkSession.

Spark Streaming: SQL - Twitter

- Create a twitter account (If you do not already have one)
- Go to apps.twitter.com → Create New App
- Put the details, website can be any website, blog, page etc you have
- Click on App Name → Keys and Access Tokens → Regenerate My Access Token and Token Secret
- The values that you need to copy are:
  - Consumer Key
  - Consumer Secret
  - Access Token
  - Access Token Secret
- Put all the values in the jupyter notebook (twitter_auth_socket)
- In the 'track' parameter, add the value that you would want twitter to search by

Exercises

General instructions

- Open notebooks.csc.fi
- Login with HAKA (if you have one) or else request for a non-HAKA account
- Go to *Account* tab → *Join Group*
- Enter the following join code: **sparkcsc-5vhzh**
- Go back to the *Dashboard* tab
- Launch the environment called **Apache Spark – CSC - Nov 17**
- Wait for the provisioning to complete and then click on "*Open In Browser*"
- Enter the *work* directory -> *spark-csc*
- You should now see the required Jupyter Ipython Notebooks and datasets

Day 1

1. RDD: Transformation and Actions

a) Open the file called Apache-Spark-Getting-Started.ipynb
b) The examples are given as a demonstration
c) Do the user exercises

2. Word Count Example

a) Open the file called Spark-Word-Count.ipynb
b) The word count example is solved
c) Followed by user exercises

3. Day-1 Final Exercise

a) Open the file called Spark-Day1-Final.ipynb
b) Do the user exercises

DAY 2

4. Spark Dataframes and SQL Overview

a) Open the file called SparkSQL-Basics.ipynb
b) Try out the examples
c) Try out the user exercises

5. Dataframes and SQL

a) Open the file called SparkSQL-Analysis.ipynb
b) Try out the examples
c) Try out the user exercises

6. Spark Streaming

a) Open files SparkSQL-Streaming-Twitter.ipynb and twitter_auth_socket.ipynb
b) Create a twitter account (If you do not already have one)
c) Go to apps.twitter.com → Create New App
d) Put the details, website can be any website, blog, page etc you have
e) Click on App Name → Keys and Access Tokens → Regenerate My Access Token and Token Secret
f) The values that you need to copy are:
    i. Consumer Key
    ii. Consumer Secret
    iii. Access Token
    iv. Access Token Secret
g) Put all the values in the jupyter notebook (twitter_auth_socket.ipynb)
h) In the 'track' parameter, add the value that you would want twitter to search by
i) Continue with the exercises of SparkSQL-Streaming-Twitter.ipynb