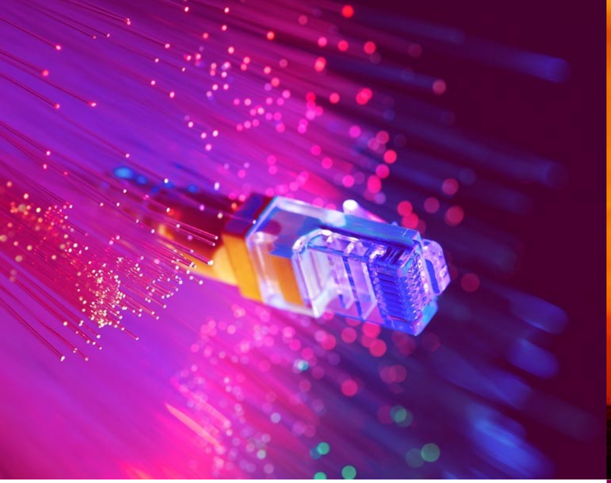
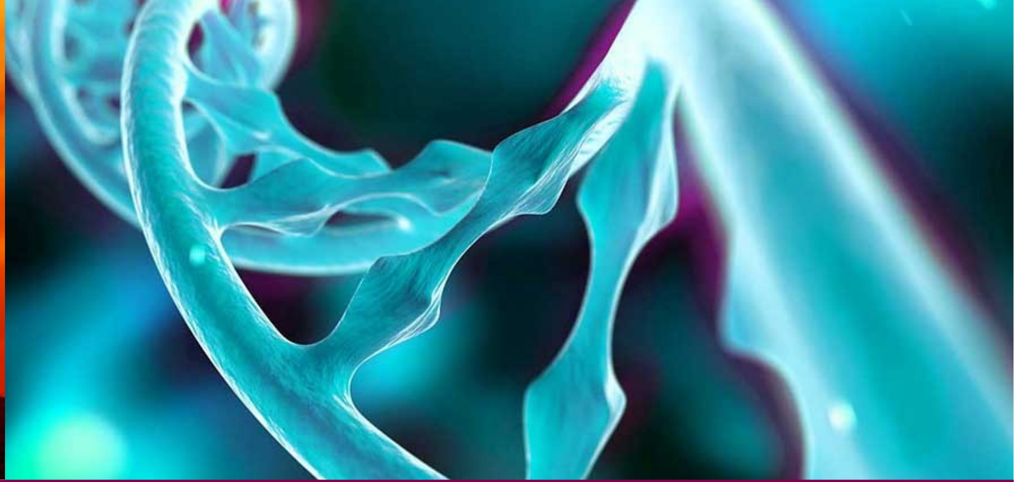




Spark SQL

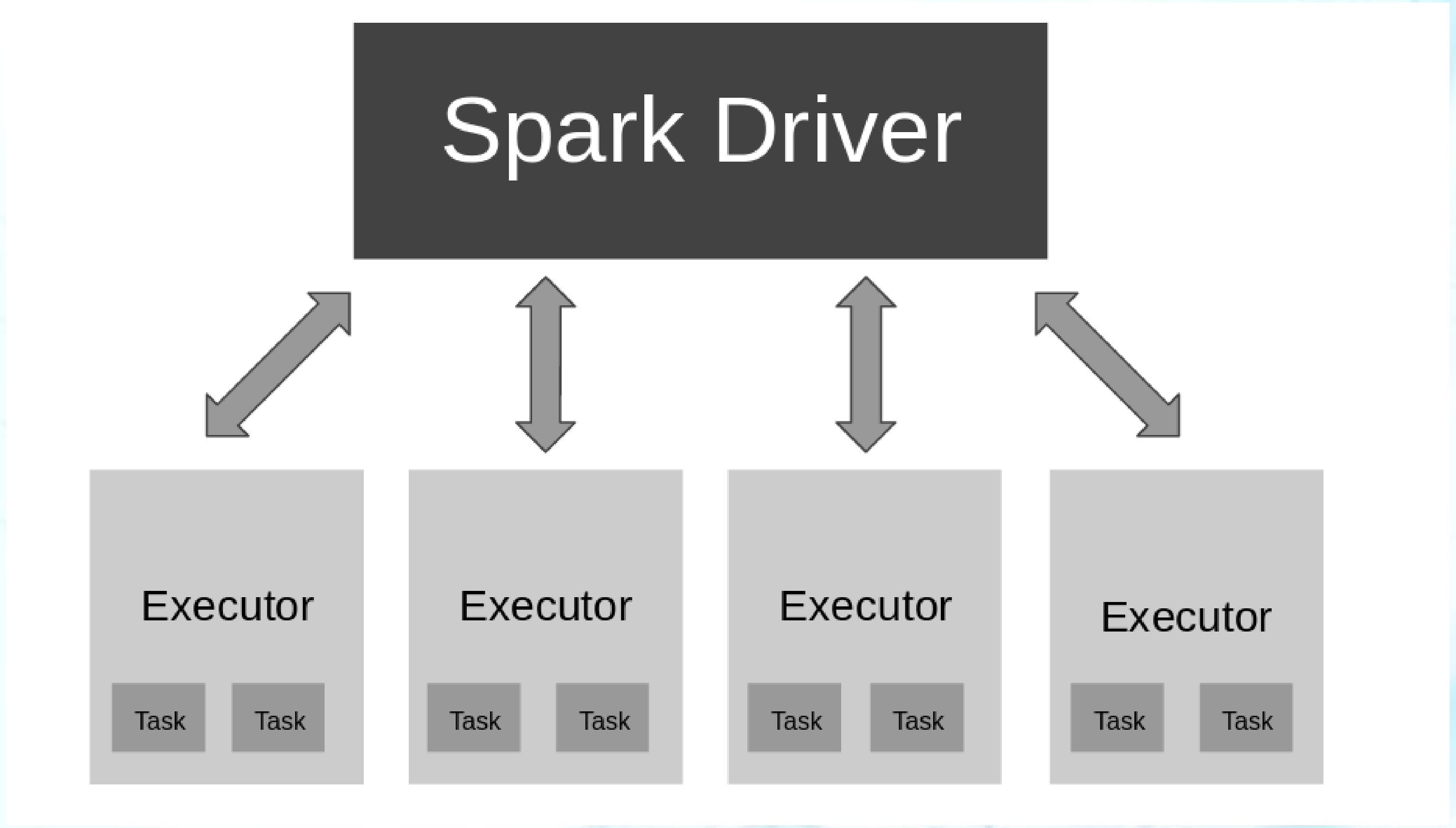
Apurva Nandan

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

CSC

Spark : Brief Technical Overview

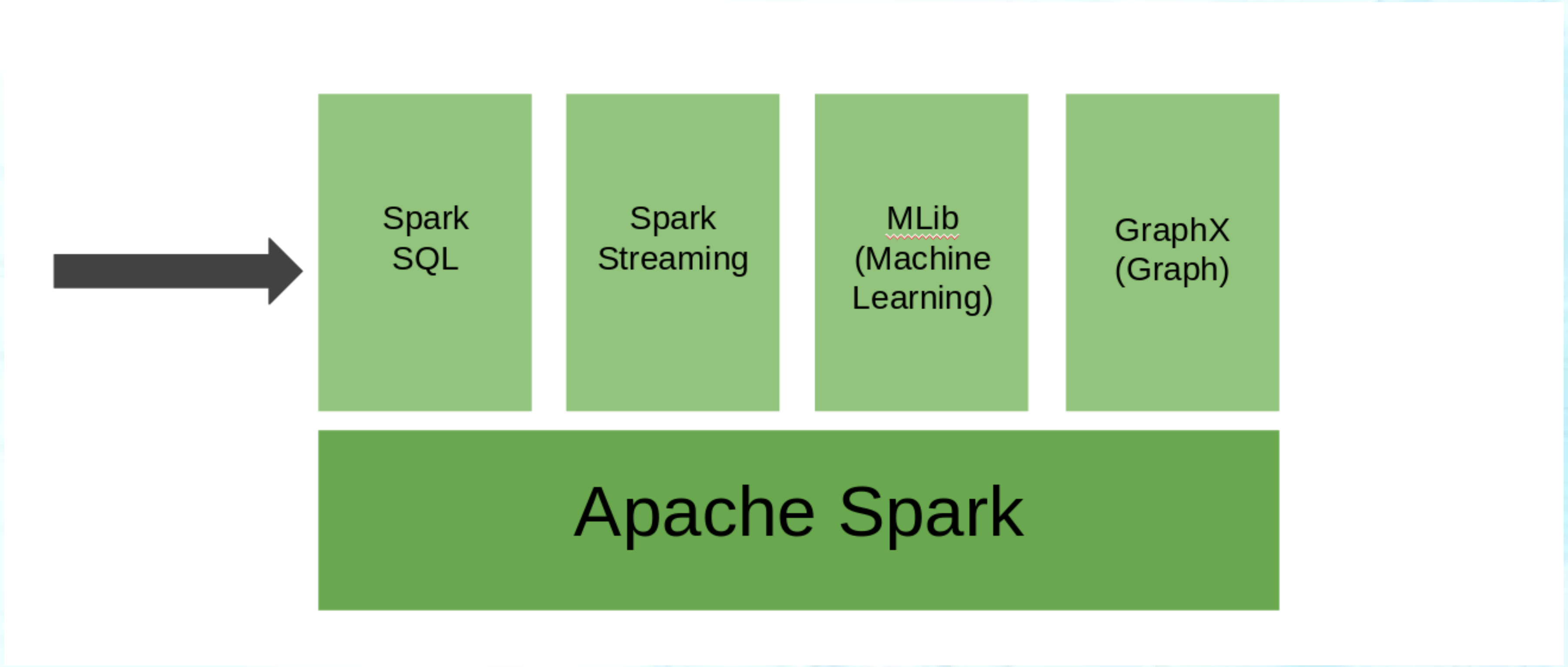
Spark: Execution of code



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 Stack


Spark SQL
Spark Streaming
MLib (Machine Learning)
GraphX (Graph)
Apache Spark

Spark: RDD & Lambdas

RDD - Resilient Distributed Datasets

- Spark's fault tolerant dataset, operated in a distributed manner by running the processing across all the nodes of a cluster
- 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

Lambda Expressions

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

SparkContext, SparkSession

- **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()
```

- **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()
```

Why Spark SQL?

Calculate AVG : RDD vs DF

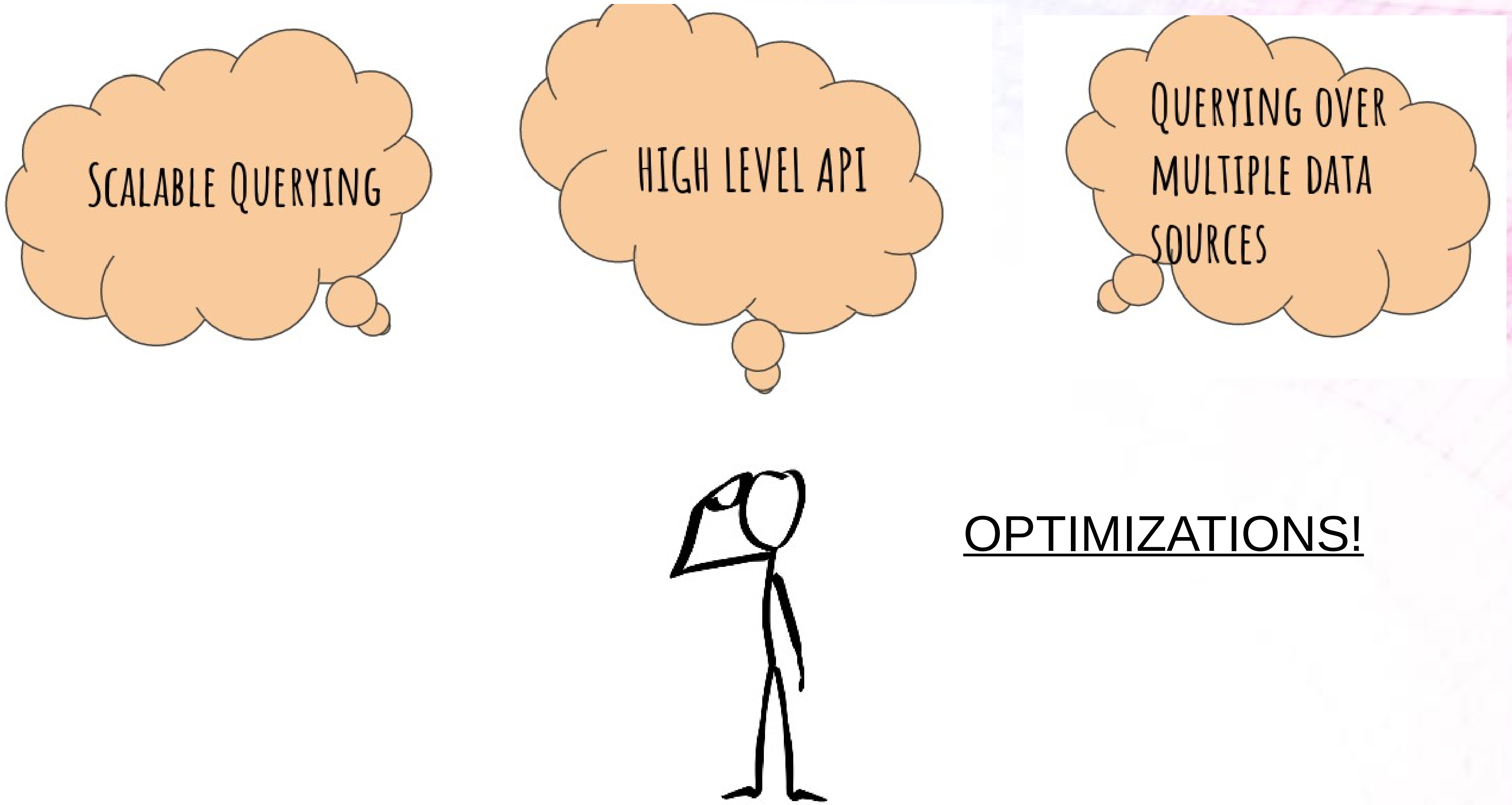
Using RDD

- 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()

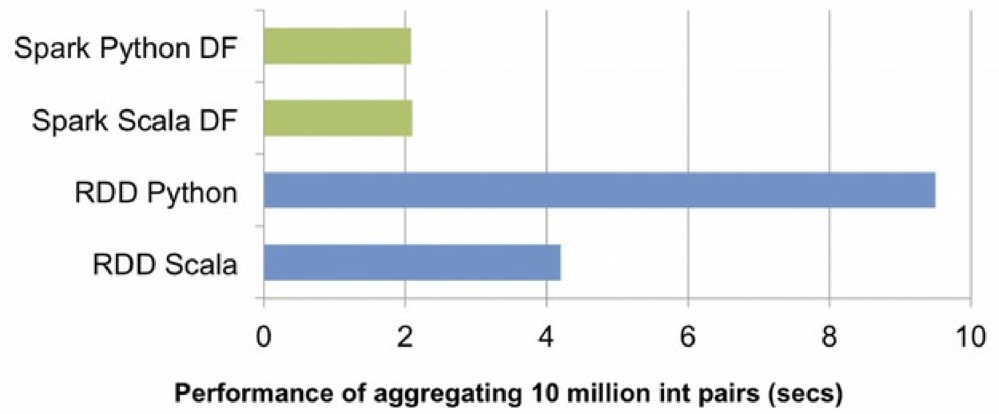
Spark SQL



OPTIMIZATIONS!

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 Catalyzed 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()
```

```
# 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

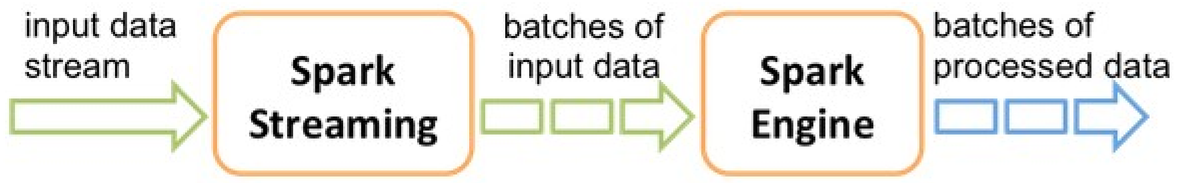
- Compressed , space efficient columnar data format
- Available for any Hadoop ecosystem.
- Can be used with Spark API for reading and saving dataframes
- Read Example: *dataframe = spark.read.parquet('filename.parquet')*
  Write Example: *dataframe.write.parquet('filename.parquet')*
- Most preferred 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**

CSC

Running SQL Queries over Live Data

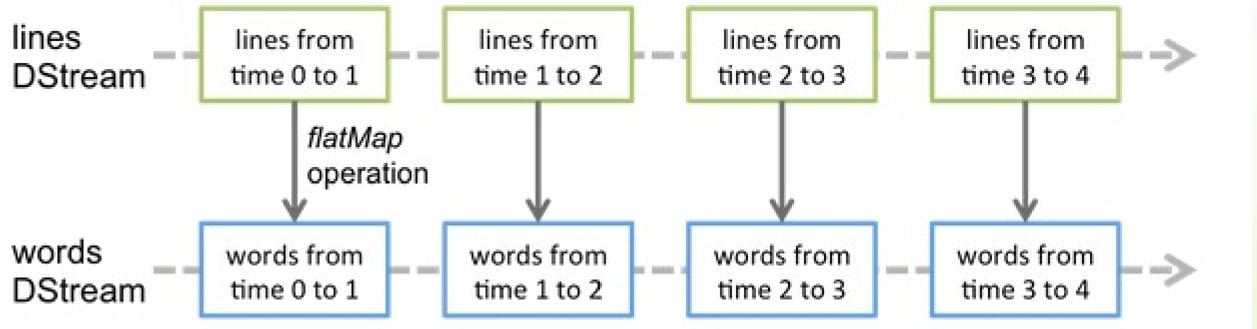
Spark Streaming

- 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

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

CSC

Thank you!

- Spark SQL Official Documentation
- Databricks Spark Reference
- PySpark Dataframe Reference (for functions)
- Learning Spark: Lightning-Fast Big Data Analysis – Book (O'REILLY)

https://www.facebook.com/CSCfi

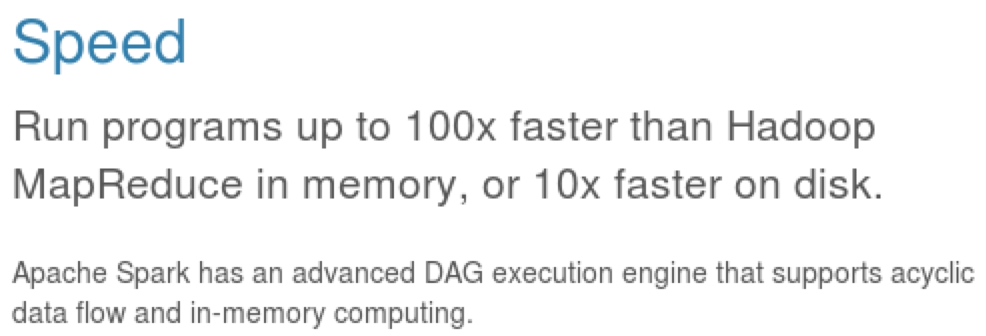
https://twitter.com/CSCfi

https://www.youtube.com/c/CSCfi

https://www.linkedin.com/company/csc---it-center-for-science

Apache Parquet : Metadata

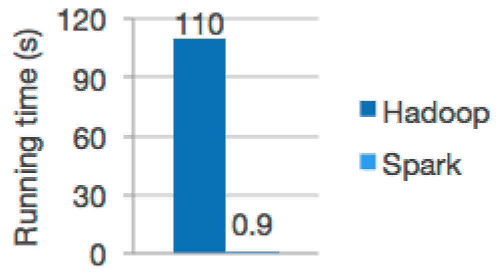
- 3 types of metadata: file metadata, column (chunk) metadata and page header metadata.
- Readers are expected to first read the file metadata to find all the column chunks they are interested in. The columns chunks should then be read sequentially.
- So Spark run 1st (separate) job to read the footer to understand layout of the data and 2nd job is actually to access the data or columns of data.
- Column 0 , Values: 2840100, Null Values: 66393, Distinct Values: 0
  Max: 75599, Min: 28800
  Compression: SNAPPY, Encodings: RLE PLAIN_DICTIONARY BIT_PACKED
  Uncompressed Size: 5886233, Compressed Size: 2419027
- Column chunks are composed of pages written back to back. The pages share a common header and readers can skip over page they are not interested in. The data for the page follows the header and can be compressed and/or encoded. The compression and encoding is specified in the page metadata.



Speed

Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk.

Apache Spark has an advanced DAG execution engine that supports acyclic data flow and in-memory computing.



Logistic regression in Hadoop and Spark

HDFS – Hadoop distributed file system

- Distributed file system that is meant for storing large data sets and being fault tolerant.
- In a production system, the Spark cluster should ideally be on the same machines as the Hadoop cluster to make it easy to read files.