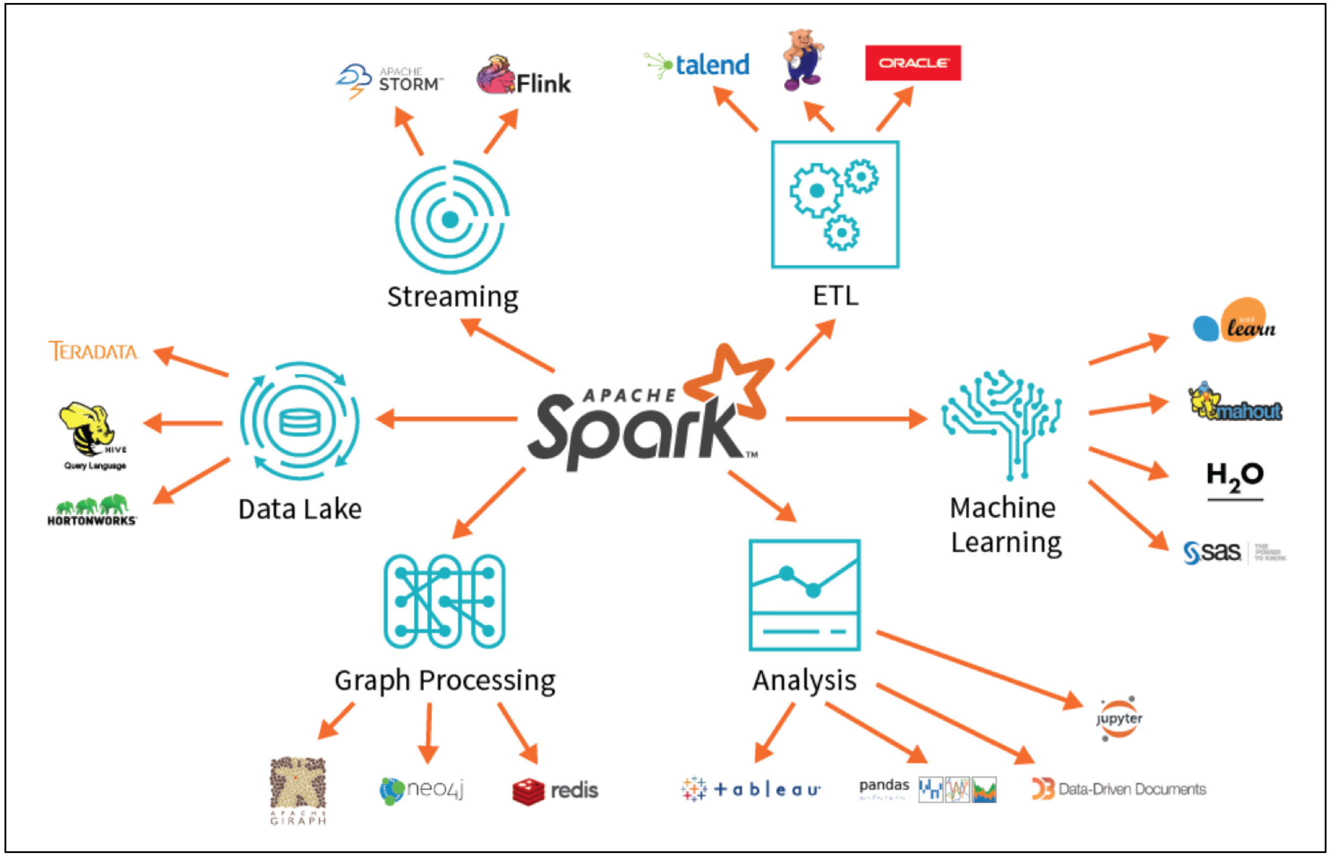
Spark analytics engine



Scheduling of sessions - Spark

- May 12th– Introduction to Spark
- May 15th– Introduction to Spark MLIB
- May 19th– Spark dataframes lab 2
- May 22nd – Spark MLIB lab 1
- May 26th – Spark MLIB lab 2
- May 29th – Spark MLIB lab 3

Processing data at scale

- Process data at scale: Terabytes
- Use common resources (local or cloud)
- From a Pandas-like framework (high level abstractions)

Common data processing pattern

Data Flow:

- Read / aggregate raw data from many sources
- Clean / normalize
- Transform
- Analyze
- Store

Apache Spark

- spark.apache.org (3.0.1 and 2.4.7)
- Unified analytics engine for large-scale data processing
- Parallel data processing on computer clusters

Spark as an unified platform

Unified platform for writing big data applications

- Data loading
- SQL queries
- machine learning
- streaming computation
- composable APIs

Spark computing engine

Move computation to data, not data to computing cores

Spark handles loading data and mix of storage systems

- Cloud: Amazon S3
- Key-value store: Cassandra
- Distributed file systems: HDFS
- Message buses: Kafka

Spark libraries

- Spark core engine
- Spark SQL
- MLib – machine learning
- Streaming
- graph analytics
- open source external libraries: https://spark-packages.org

Why Spark?

- **Speed**: DAG scheduler, query optimizer, physical execution engine
- **Ease of use**: Java, Scala, Python, R, SQL
- **Generality**: SQL, streaming, MLib, GraphX
- **Runs everywhere**: Hadoop, Mesos, Kubernetes, EC2

Get started with Spark

- Community:
  - http://spark.apache.org/community
  - Contributors: 300 companies, 1.200 developers
    - http://spark.apache.org/committers.html
- Getting Started:
  1. download spark-2.4.8-bin-hadoop2.7.tgz
  2. tar –xf spark-2.4.8-bin-hadoop2.7.tgz
  3. cd spark-2.4.8-bin-hadoop2.7
- Spark+AI summit: June 27-30th https://databricks.com/dataaisummit

Big data explosion

Speed	Cost	Cost
Applications need to add parallelism to run faster • from 2005: no faster CPUs, but an increase in CPU cores	cost drop of storage • 1TB storage cost cuts in half every 14 months	cost drop of collecting data technology • 12-megapixel webcam is less than 5€ • genome sequencing less than 1000€ per person

why is Spark popular now?

- Collecting data is inexpensive
- Need of large, parallel computations
- Difficult to scale large software solutions and traditional models (SQL)
- Spark proposal 2009:
  - Spark: cluster computing with working sets
  - UC Berkeley research project

Apache Spark: Core Concepts (1/4)

- Resilient Distributed Datasets: RDDs
  - Fault-tolerant, parallel data structures
  - Read-only, partitioned collection of records
  - Operated through **transformations** (e.g., `map`) and **actions** (e.g., `count`)

Apache Spark: Core Concepts (2/4)

- Spark is a distributed system to process very large volumes of data
- The world's largest **Spark cluster** has more than 8000 machines
- **Resource management systems**: master and worker concepts
  - Apache YARN, Apache Mesos, Kubernetes

Homework: services from Amazon and Google regarding Spark clusters?

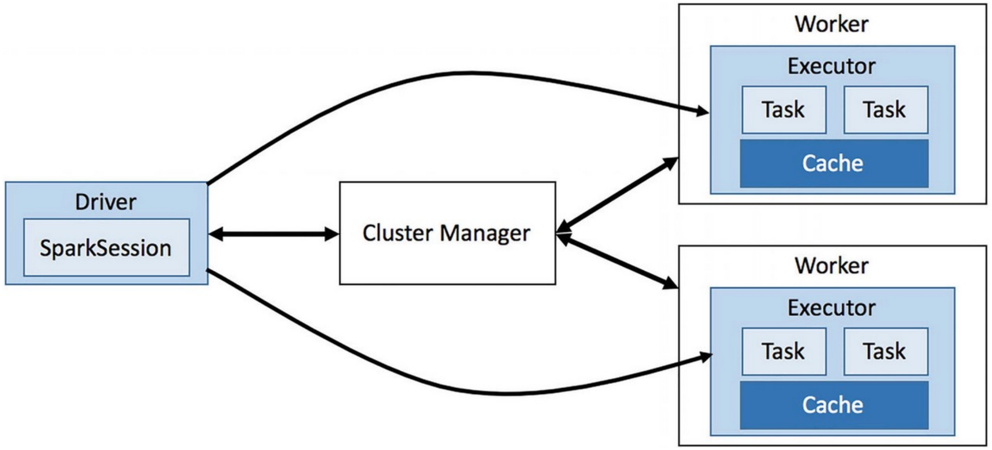
Apache Spark: Core Concepts (3/4)

- App logic with Spark APIs
- Spark Application
- Several runtime concepts

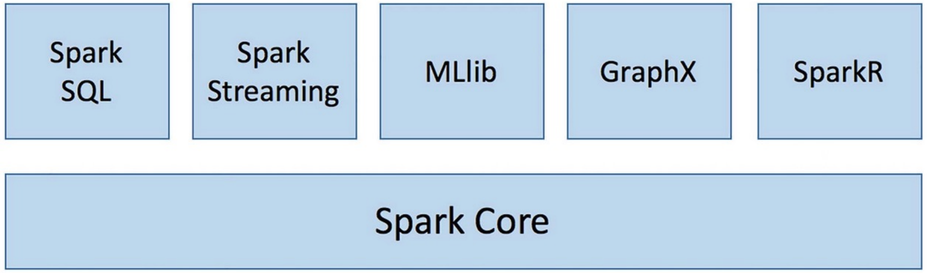
```python
from datetime import datetime
from pyspark.sql import SparkSession

spark = SparkSession.builder.getOrCreate()

rdd = spark.sparkContext.parallelize([
    (1, 'userA', datetime(2022, 3, 3, 15, 30, 12)),
    (2, 'userA', datetime(2022, 3, 3, 15, 31, 37)),
    (2, 'userB', datetime(2022, 3, 3, 15, 31, 46))
])

rdd.map(lambda x: (x[0], x[1:])).groupByKey().mapValues(list).collect()

# result:
# [('1', [['userA', '2022-03-03 15:30:00']]),
#  ('2', [['userA', '2022-03-03 15:31:37'], ['userB', '2022-03-03 15:31:46']])]
```

Apache Spark: Core Concepts (4/4)

- 1 **Driver** process per Spark App
- **N** Executors per Driver
- 1 CPU core per **Task**
- Master/Worker architecture
  - Master → Driver process
  - Worker → Executor process

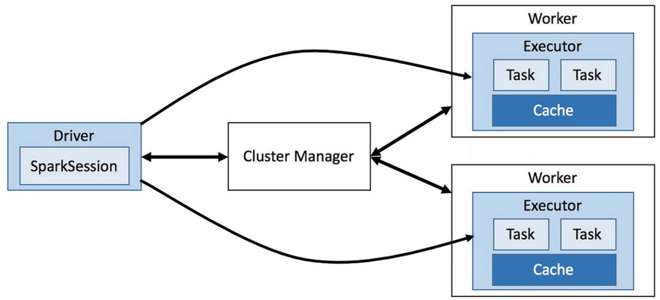


Spark Architecture (1/5)



- Unified stack built on top of Spark Core
- Separate libraries targeting specific data processing workloads
- Data flows through the APIs with no need of intermediate storage

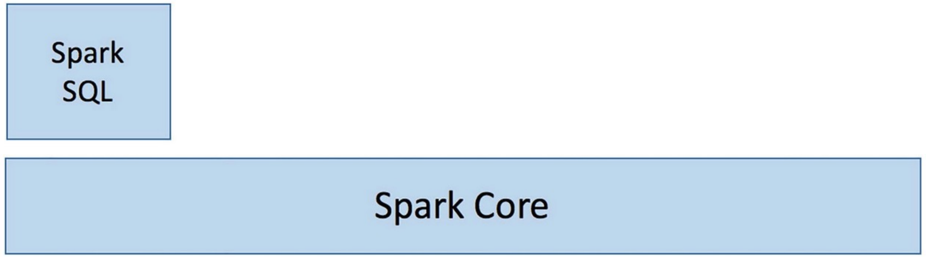
Spark Architecture (2/5)



Spark Core

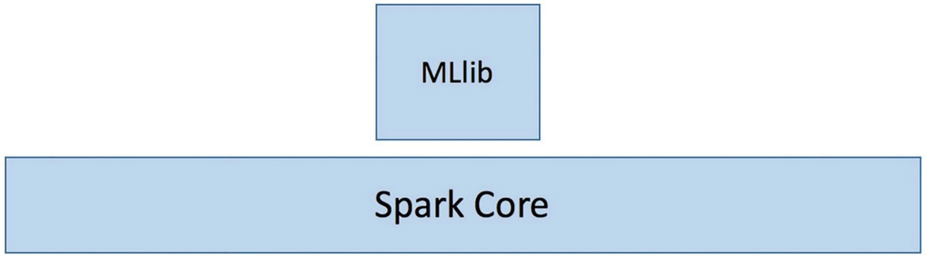
- Fault tolerance (i.e., RDDs)
- In-memory computation: cache() and persist()
- Scheduling and monitoring
- Interacting with storage systems (hdfs, s3, etc.)

Spark Architecture (3/5)



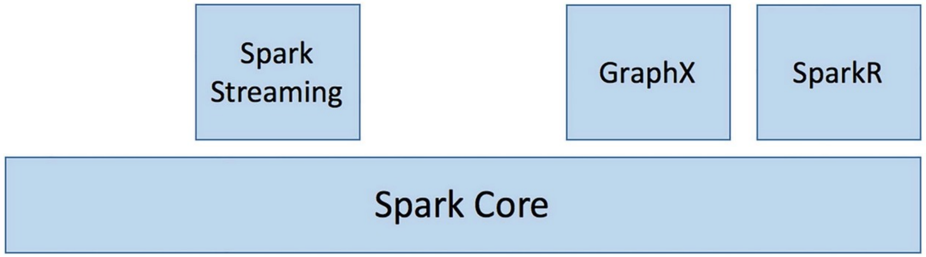
- Introduces the DataFrame high-level API
- Blurs the line between RDDs and relational tables
- Read/write JSON, CSV, Parquet, etc.
- Catalyst optimizer

Spark Architecture (4/5)



- Based on the DataFrame API (since version 2.0)
- +50 common ML algorithms out-of-the-box
- “Featurization” (Spark Features), Hyperparameter tuning, model persistence

Spark Architecture (5/5)



- Fault-tolerant streaming apps
- Graph-parallel computation (not available in Python)
- Large-scale data analysis using R

Spark DataFrames

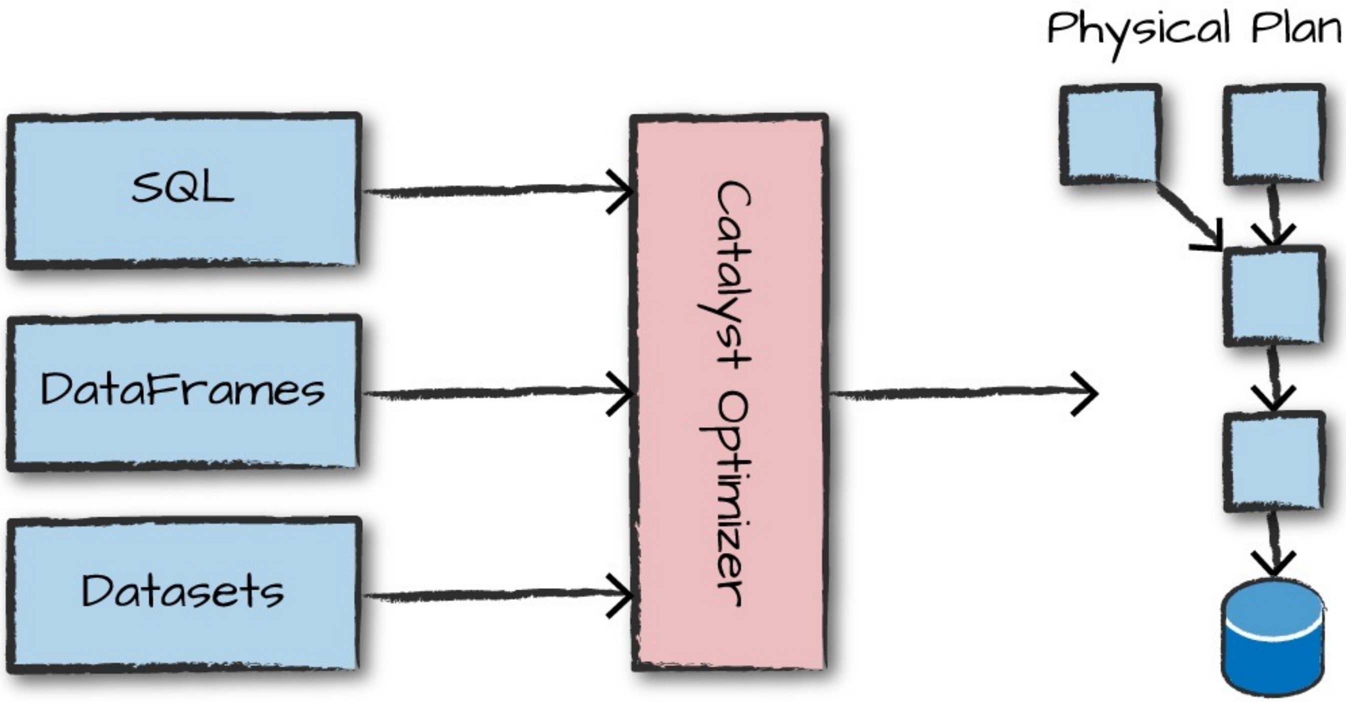
- Distributed table-like collections of well-defined rows and columns
- Each column must have the same number of rows
- Each column has the same type of data

- Action on DataFrames: Spark plans on how to manipulate rows and columns to compute the result for the user

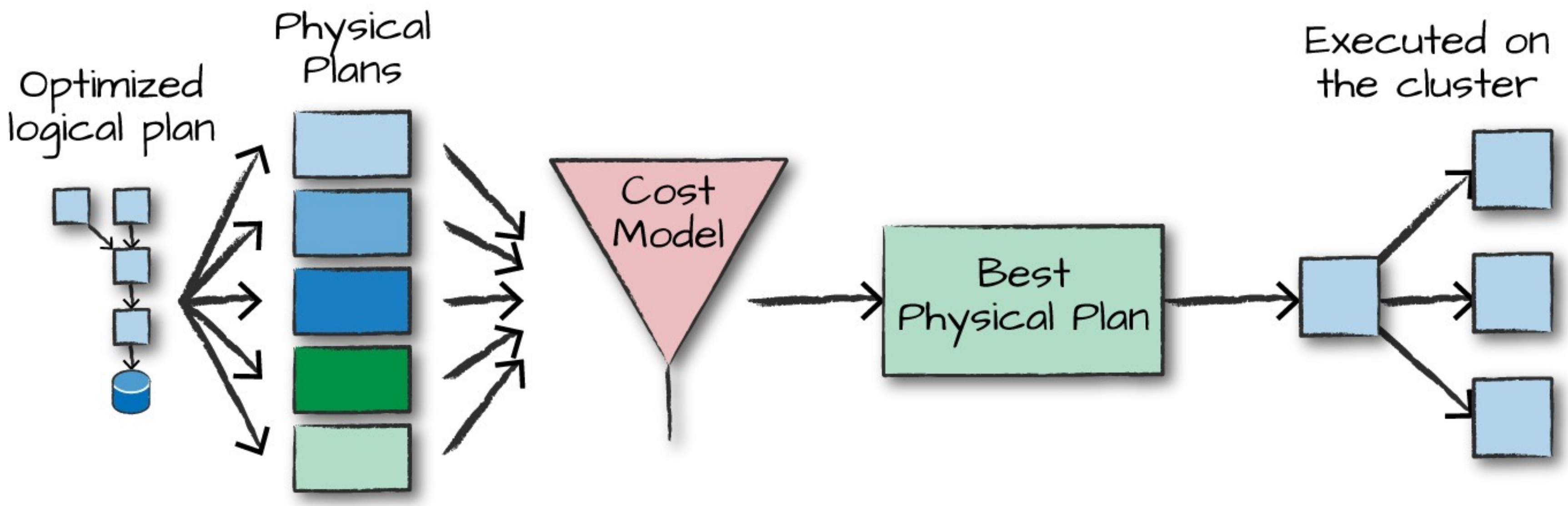
Structured API Execution

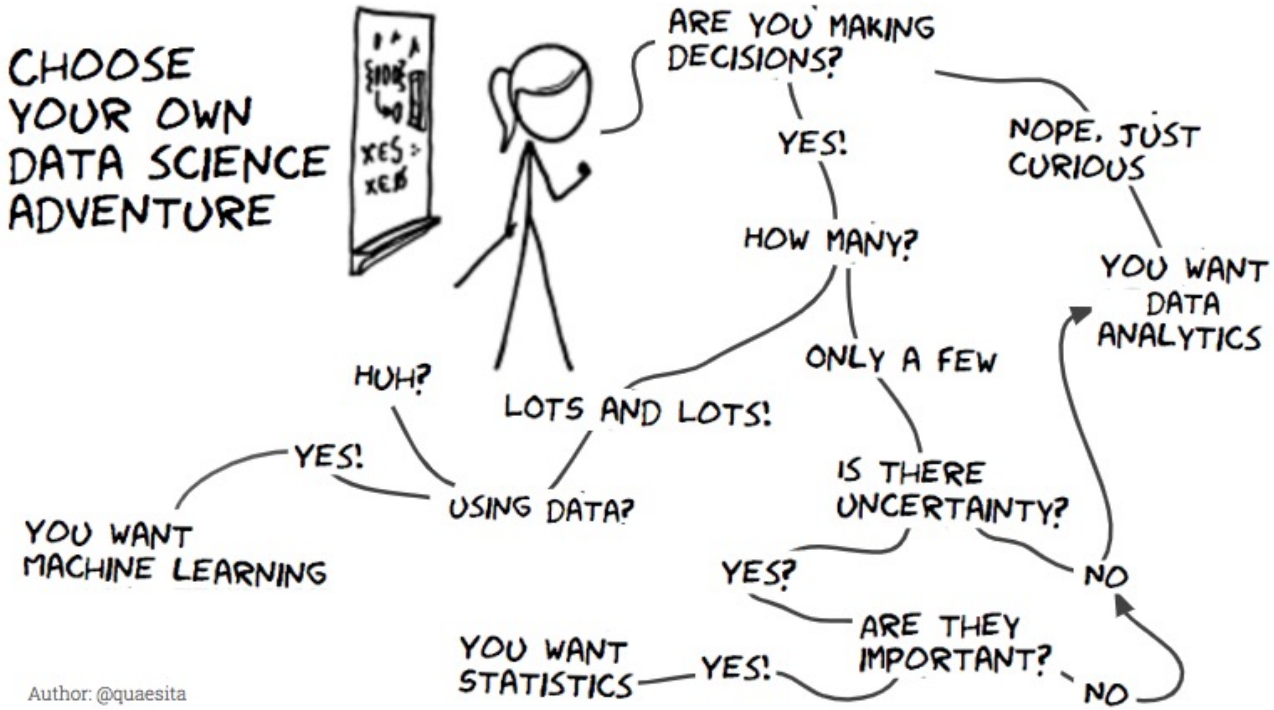
1. Write DataFrame code
2. If the code is valid it is converted to a Logical plan
3. Spark transforms logical plan to a physical plan: *Catalyst optimizer*
4. Spark executes physical plan on the cluster

Catalyst optimizer



Physical planning: Spark plan





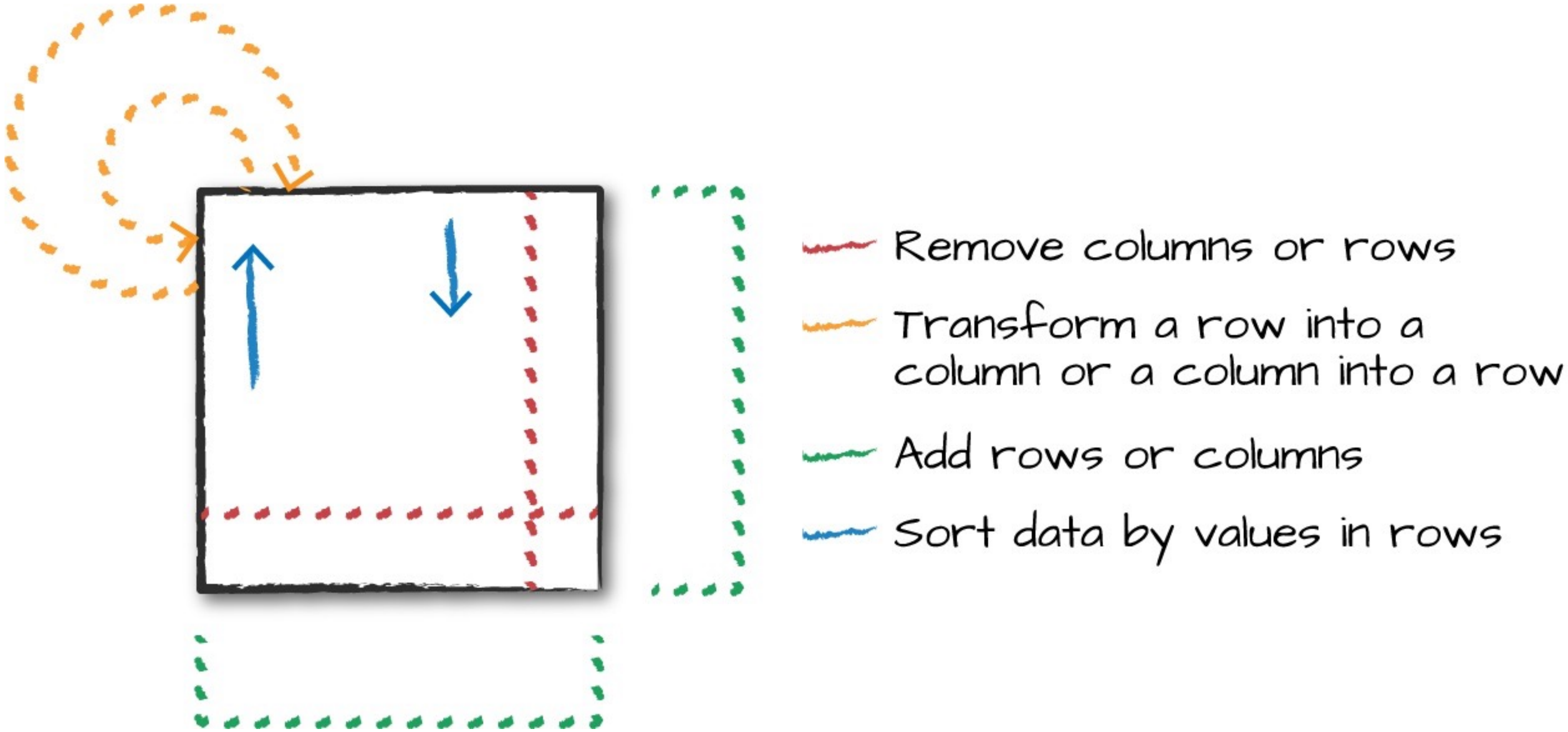
Author: @quaesita

DataFrame programming

a DataFrame is a list of records of type Row and a number of columns

```
val df = spark.read.csv(“json”).
    load(“/data/genomics/json/summary.csv”)

df.printSchema()
```

DataFrame transformations



Main query examples

Customer transactions

date	time	customer	name	product	price
01/10/2018	2:20 PM	100	John Smith	6	86
04/08/2018	11:38 AM	100	John Smith	8	79

WHERE IS MY PRIMARY KEY?

select and selectExpr

use select to manipulate columns in your DataFrame

```
c.select("customer").show
c.select("customer","product").show
```

```
+--------+
|customer|
+--------+
|     100|
|     200|
```

selectExpr

expr is the most flexible way to describe:

- a column
- or a string manipulation of a column

```
c.selectExpr("customer as customer_id").show
```

```
+-----------+
|customer_id|
+-----------+
|        100|
|        200|
```

Spark power

- Create new DataFrames with selectExpr!
- selectExpr is a simple way to build complex expressions

```
c.selectExpr("*", "(quantity = price) as equal_price")

+----------+--------+--------+-------+--------+-----+-----------+
|      date|    time|customer|product|quantity|price|equal_price|
+----------+--------+--------+-------+--------+-----+-----------+
|01/10/2018| 2:20 PM|     100|      1|       6|   86|      false|
|04/08/2018|11:38 AM|     200|      2|       8|   79|      false|
```

Aggregations on the entire DataFrame

```
c.selectExpr("avg(price)","count(customer)")
```

```
+----------+---------------+
|avg(price)|count(customer)|
+----------+---------------+
|    73.875|              9|
+----------+---------------+
```

Filtering Rows

- Create an expression that evaluates to true or false
- Used to filter out rows with an expression equal to false

```
c.where(expr(“product = 8”)).show

c.where(expr(“quantity < 9”)).
   where(expr(“customer != 100”)).show
```

```
+----------+--------+--------+-------+--------+-----+
|      date|    time|customer|product|quantity|price|
+----------+--------+--------+-------+--------+-----+
|01/10/2018| 2:20 PM|     200|      5|       8|  100|
|04/08/2018|11:10 AM|     101|      8|       5|   33|
```

Sorting rows

- Decide to have largest or smallest values at the top of DataFrame
- default is to sort in ascending order

```
c.orderBy("price").show
c.orderBy(desc("customer"),asc("price")).show
```

```
+----------+--------+--------+-------+--------+-----+
|      date|    time|customer|product|quantity|price|
+----------+--------+--------+-------+--------+-----+
|01/10/2018| 2:20 PM|     200|      5|       8|   33|
|04/08/2018|11:10 AM|     200|      8|       5|  111|
|20/10/2018| 1:10 AM|     100|      8|       5|   45|
|08/06/2018|12:10 AM|     100|      8|       5|   59|
```

Aggregations

Aggregate: collect data together from DataFrame

Summarize numerical data by custom grouping

key of grouping: column to focus on

aggregation function: how to transform column values

aggregations with groupBy

```
c.groupBy("customer").count.show

+--------+-------+
|customer|product|
+--------+-------+
|     100|      1|
|     101|      6|
|     200|      6|
+--------+-------+
```

Aggregations

- Aggregate: collect data together from DataFrame
- Summarize numerical data by custom grouping
- **key** of grouping: column to focus on (customer)
- aggregation **function**: how to transform column values (count)

- **groupBy**: one or more keys and one or more functions
- more advanced options: window, rollup, cube, ...

grouping with expressions

- Total number of products for each customer?

```
c.groupBy("customer").agg(sum("quantity")).show()
```

```
+--------+-------------+
|customer|sum(quantity)|
+--------+-------------+
|     100|           31|
|     101|            5|
|     200|           41|
+--------+-------------+
```

Grouping with expressions

- Define aggregation expression: “**sum(quantity)**”
- Use agg to apply function to key of interest

Grouping with expressions

- Define aggregation expression: "**sum(quantity)**"
- Use agg to apply function to key of interest

add all the products bought by each customer

Grouping with expressions

- Define aggregation expression: “**sum(quantity)**”
- Use agg to apply function to **key of interest**

add all the products bought by each customer

for each **customer** with the same **customer id number** ...
add all the products bought: **add product quantity**

Grouping with expressions

- Define aggregation expression: "**sum(quantity)**"
- Use agg to apply function to key of interest

add all the products bought by each customer

1: groupBy("customer") for those **customers** with the same value...

2: agg("sum(quantity)") add all the products bought by customer

Grouping with expressions

- Define aggregation expression: “**sum(quantity)**”
- Use agg to apply function to key of interest

1: groupBy(“customer”) for those **customers** with the same value...

2: agg(“sum(quantity)”) add all the products bought

```
c.groupBy("customer").agg(sum("quantity")).show()
```

Aggregation functions

- count
- countDistinct
- approx_count_distinct
- last
- min/max
- sum, sumDistinct
- avg
- var_pop/stddev_pop

DataFrame questions

Data file: transactions.csv

date	time	customer	product	quantity	price
01/10/2018	2:20 PM	100	1	6	86
04/08/2018	11:38 AM	200	2	8	79

1. How many elements?
2. How many DISTINCT customers?

3-How many products per customer?

Aggregation question: need to aggregate values per customer

which is the relevant key?

which is the calculation we need?

4-Sort customers by quantity

5-how many times customer id number 100 has purchased more than 5 items?

key?

conditions to meet for customer 100 transactions?

6-which were the products bought by customer with the **transaction with the largest number of products**?

Two questions to solve:

1-which is the customer with largest number of products in a transaction?

key: ?, value: ?

2-which are the products of selected customer?

key customer:?, value: ?

BONUS:
which is the product with highest price?