# Spark for data Engineers

SQLBits 2022, March 8th, 2022

Tomaž Kaštrun

# Agenda

- 6 modules; 50min + 10min work
- Start: 10AM CET/9AM GMT – 5PM CET/4PM GMT
- 2x coffee break, 1x lunch break
- Ask anytime, discussion welcome
- Mute your mic if you are not asking

# Material

- Material available on Github; clone the repository
- Downloadable programs (spark, JVM, Hadoop)

# Module 1

General overview, Apache Spark, Environment setting

# Module 1

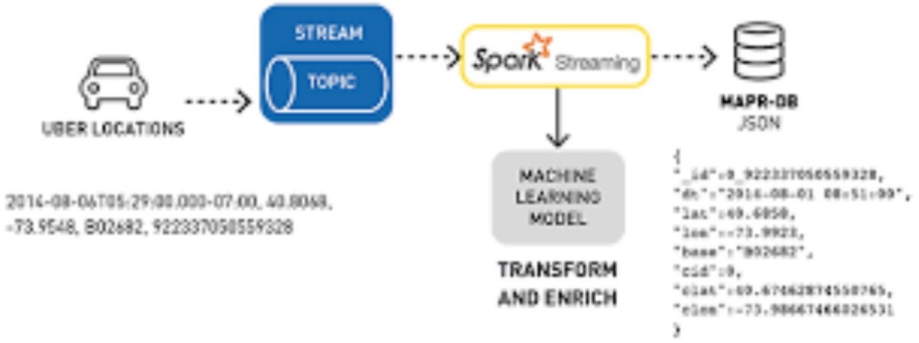
- About Apache Spark
- Spark execution modes
- Installation
- Running Spark

# On Apache Spark

- Fast, expressive, general-purpose in-memory cluster computing framework compatible with Apache Hadoop and built around speed, ease of use and streaming analytics
  Faster and easier than Hadoop MapReduce*
  Large community and 3rd party libraries
- Provides high-level APIs (Java, Scala, Python, R) Supports variety of workloads
  - interactive queries, streaming, machine learning and graph processing

# Spark Use-Cases

- Logs processing (Uber)
- Event detection and real-time analysis
- Interactive analysis
- Latency reduction
- Advanced ad-targeting (Yahoo!)
- Recommendation systems (Netflix, Pinterest)
- Fraud detection
- Sentiment analysis (Twitter)



# Spark Use-Cases

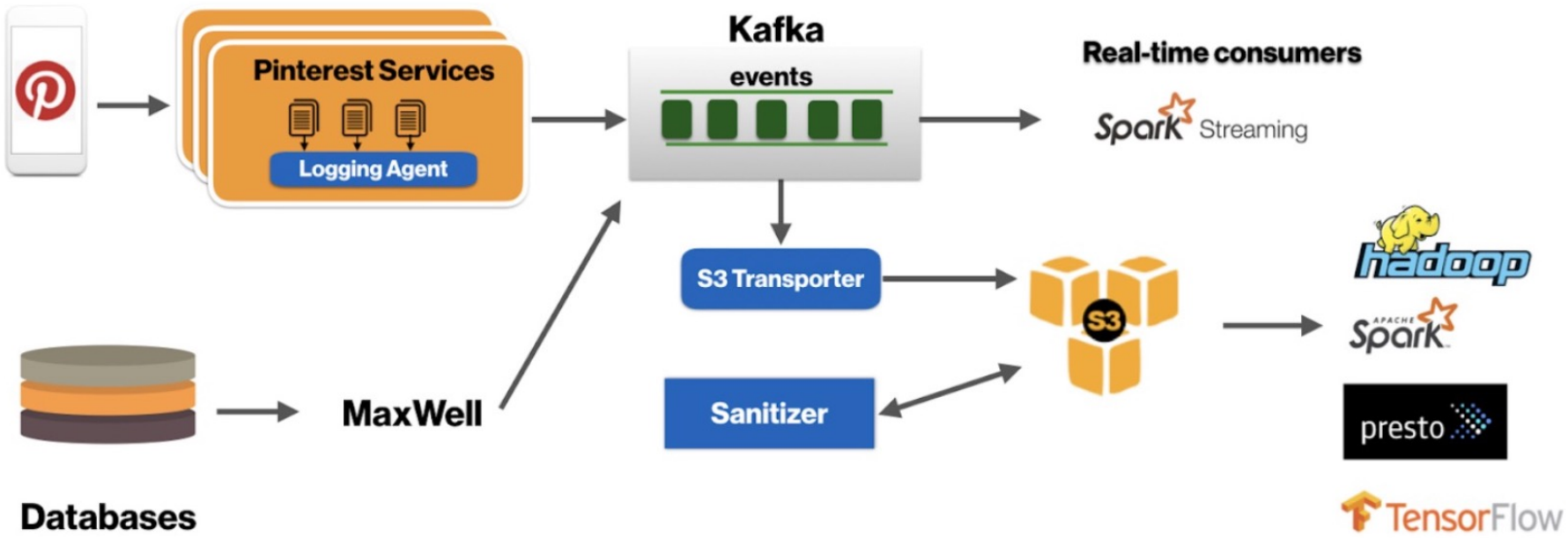
- Logs processing (Uber)
- Event detection and real-time analysis
- Advanced ad-targeting (Yahoo!)

# Spark Use-Cases

- Logs processing (Uber)
- Event detection and real-time analysis
- Interactive analysis
- Latency reduction
- Advanced ad-targeting (Yahoo!)
- Recommendation systems (Netflix, Pinterest)
- Fraud detection
- Sentiment analysis (Twitter)

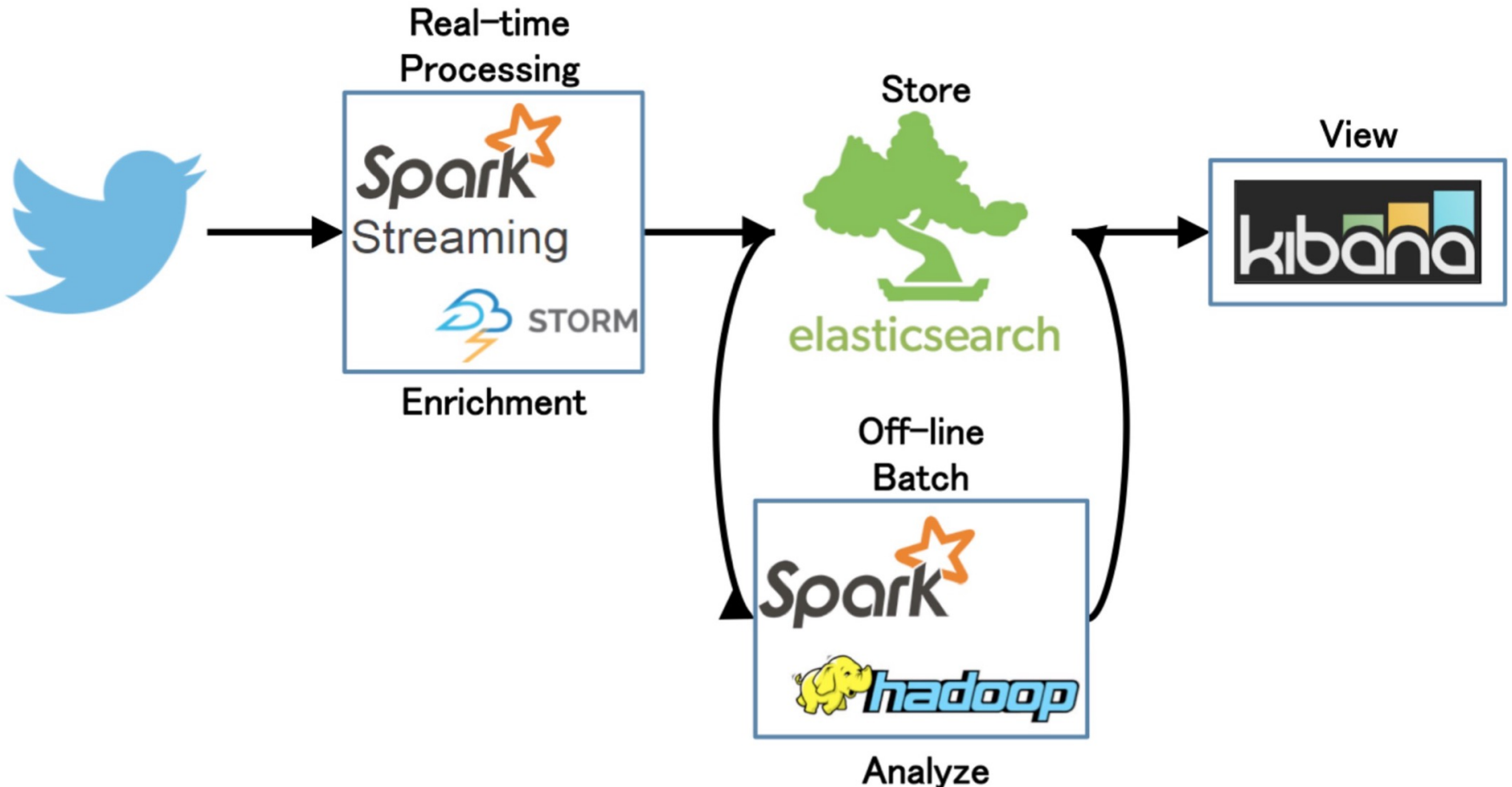
# Spark Use-Cases

- Recommendation systems (Netflix, Pinterest)
- Fraud
- Senti



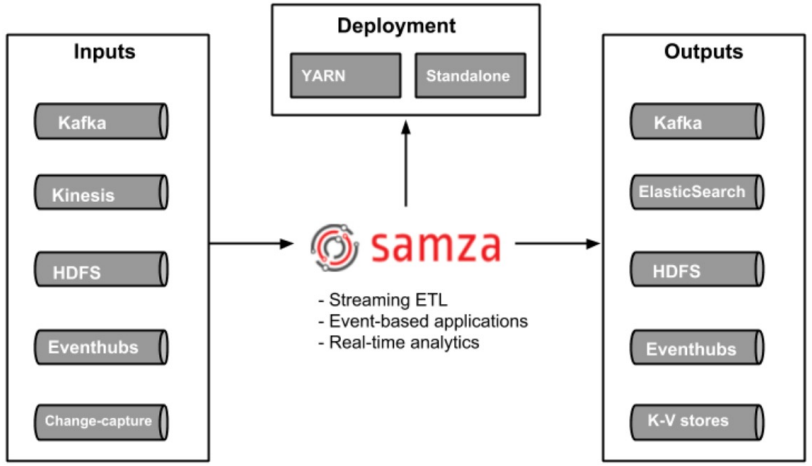
# Spark Use-Cases

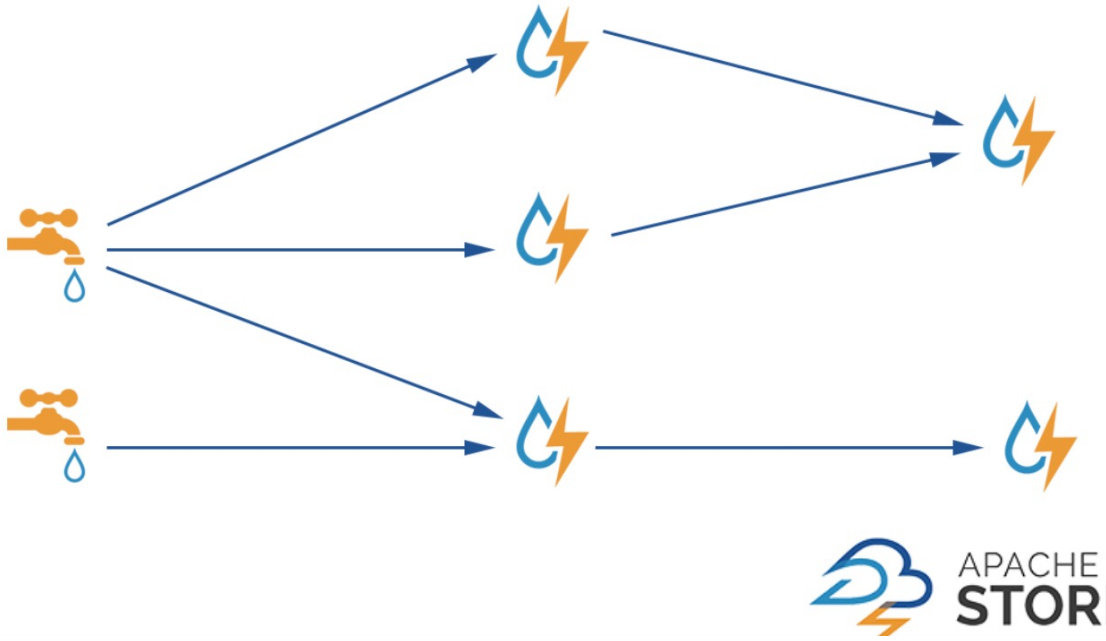
- Sentiment analysis (Twitter)

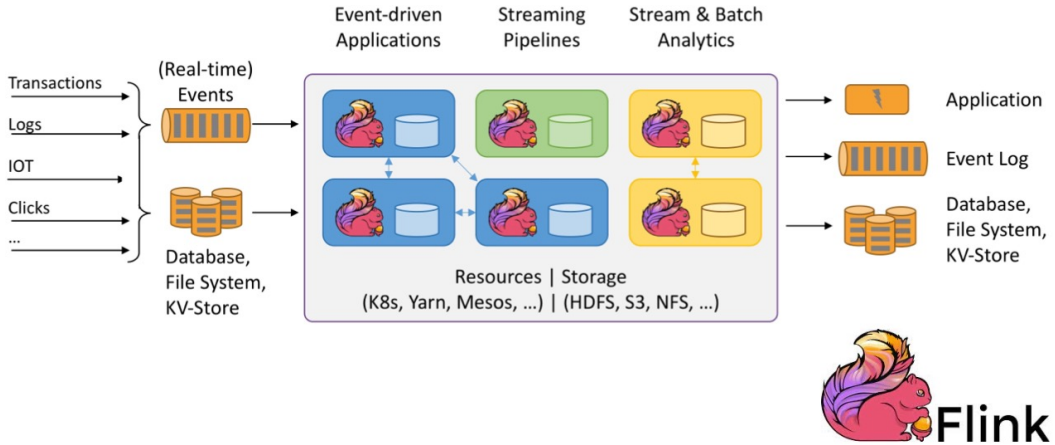


# Apache Spark “real-time”

- Apache Samza (library/framework)
- Apache Storm (real-time stream processing)
- Apache Flink (native streaming support for all workloads)



# Apache Spark 3.x

- Latest release – v3.1.1. (March 2, 2021)
- Improvements over Spark 2 (v2.4.1)
  - ▶️ Python 2 deprecated
  - ▶️ Adaptive execution of Spark SQL (merging intermediate results among workers)
  - ▶️ Dynamic partition pruning
  - ▶️ Support for deep learning (GPU support for Nvidia, AMD, Intel)
  - ▶️ Better Kubernetes integration
  - ▶️ Graph features (Morpheus as extension of Cypher, neo4j support)
  - ▶️ ACID transactions (for Delta Lake storage)
  - ▶️ Apache Arrow data format integration (columnar format for analytics)

# Hadoop MapReduce vs. Apache Spark

Big data frameworks

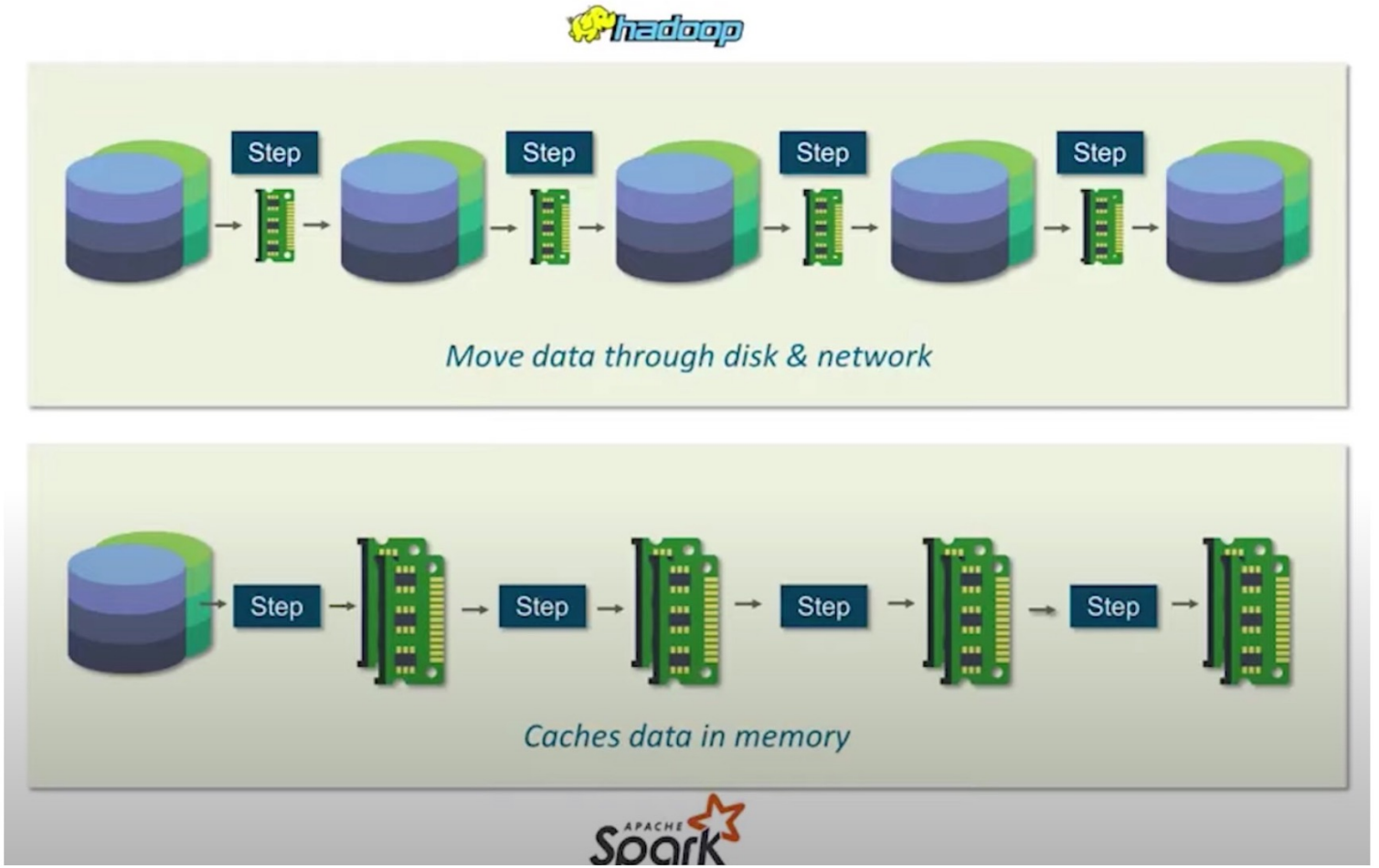
- Performance
- Ease of use
- Costs
- Data processing
- Fault tolerance
- Security

Hadoop

Archival data analysis

Spark

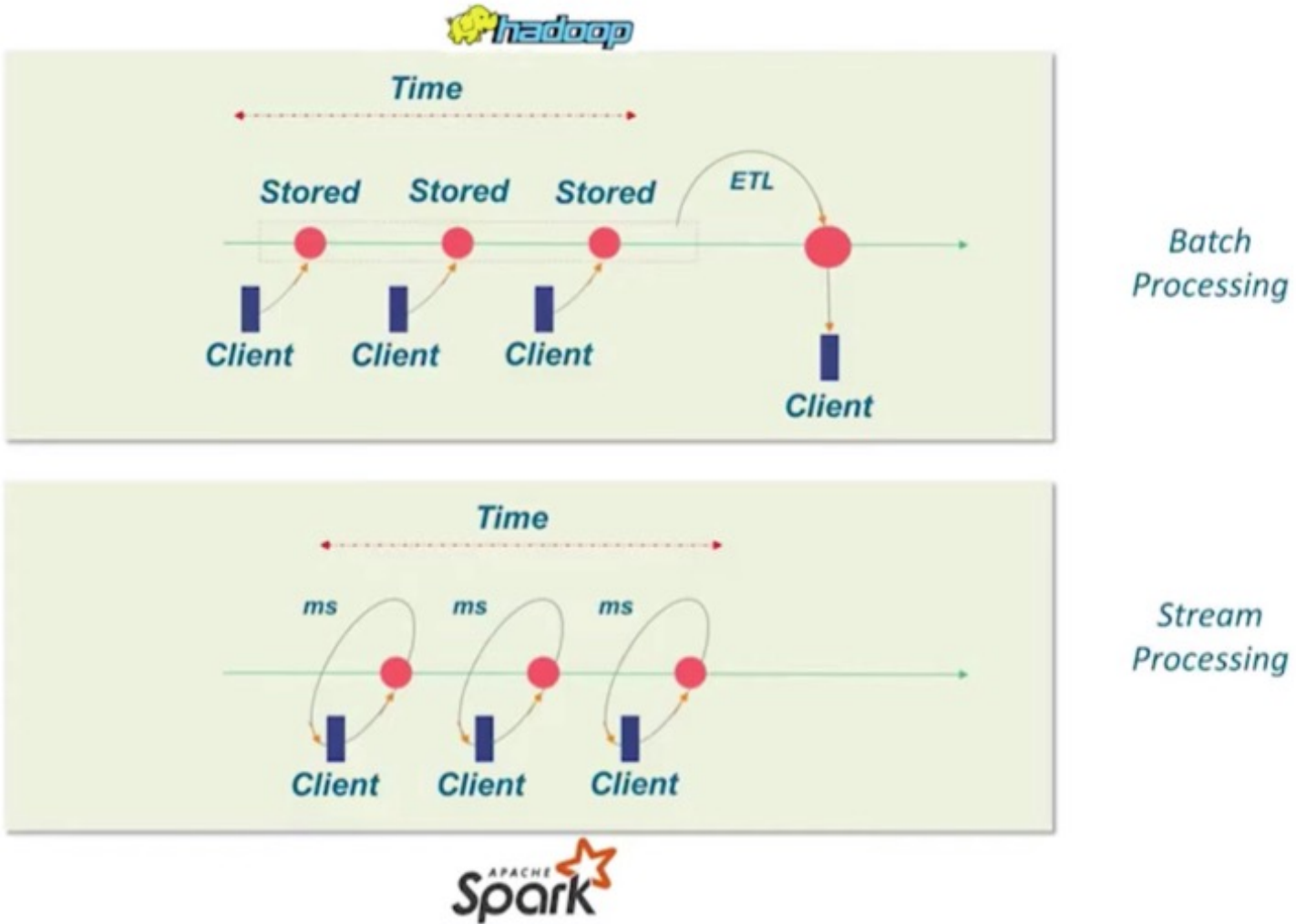
Real-time data analysis



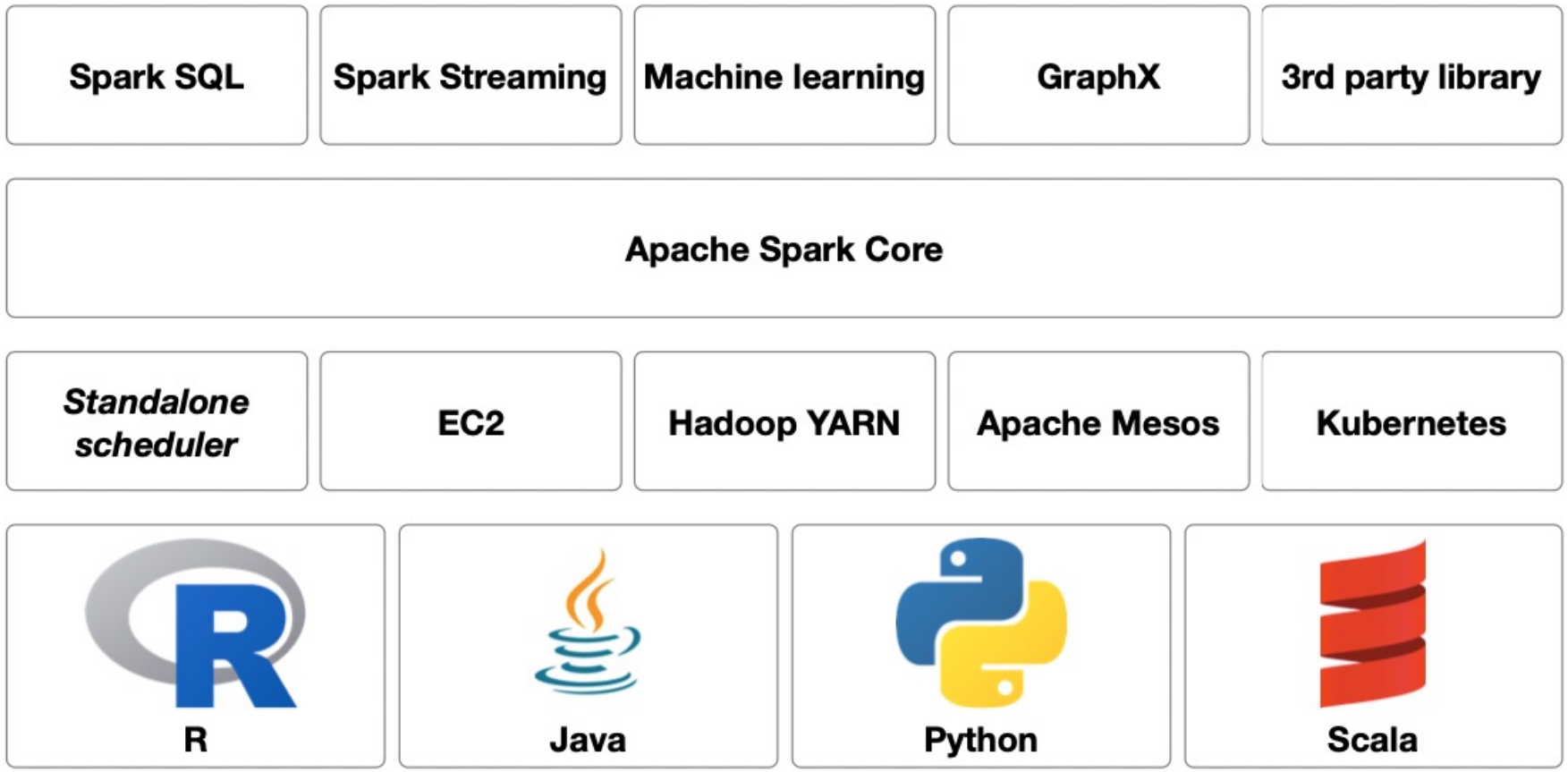
# Hadoop MapReduce vs. Apache Spark

# Hadoop MapReduce vs. Apache Spark

# Hadoop MapReduce vs. Apache Spark



# Apache Eco-System

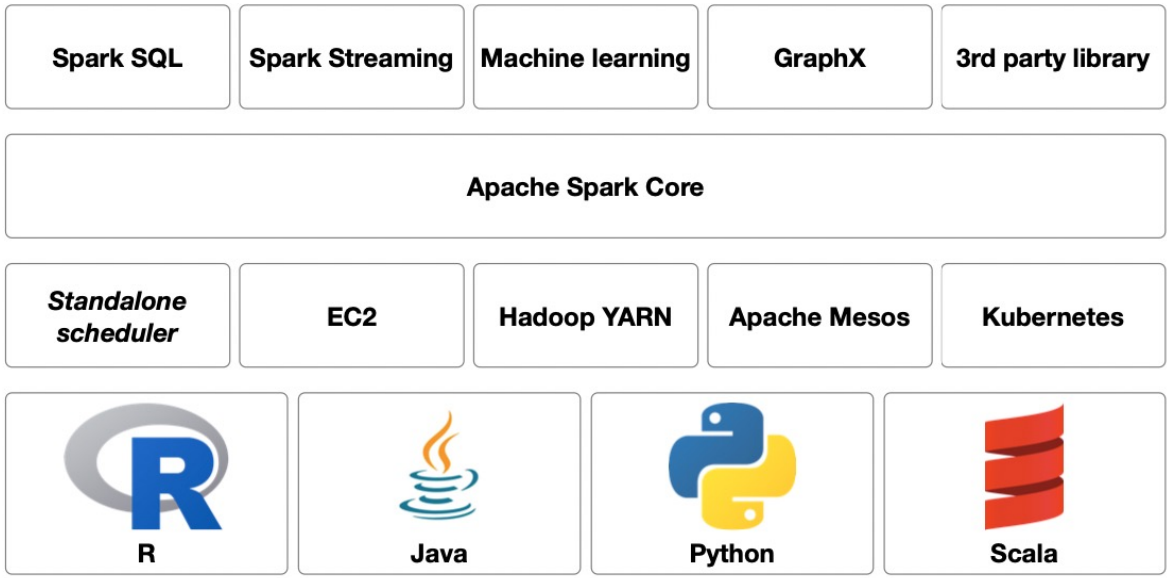


# Spark Core functionalities

Core functionalities
- task scheduling
- memory management
- fault recovery
- storage systems interaction

etc.

Basic data structure definitions/abstractions
- Resilient Distributed Data sets (RDDs)
  main Spark data structure
- Directed Acyclic Graph (DAG)



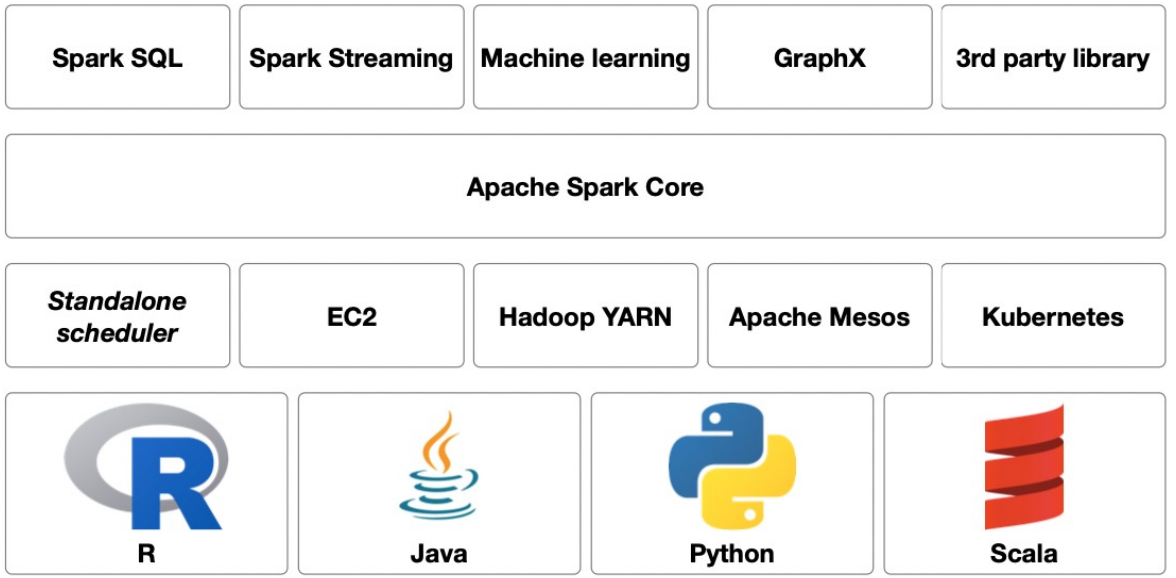
# Ecosystem: Spark SQL

Structured data manipulation
- Data Frames definition

Table-like data representation
- RDDs extension
- Schema definition

SQL queries execution

Native support for schema-based data
- Hive, Paquet, JSON, CSV



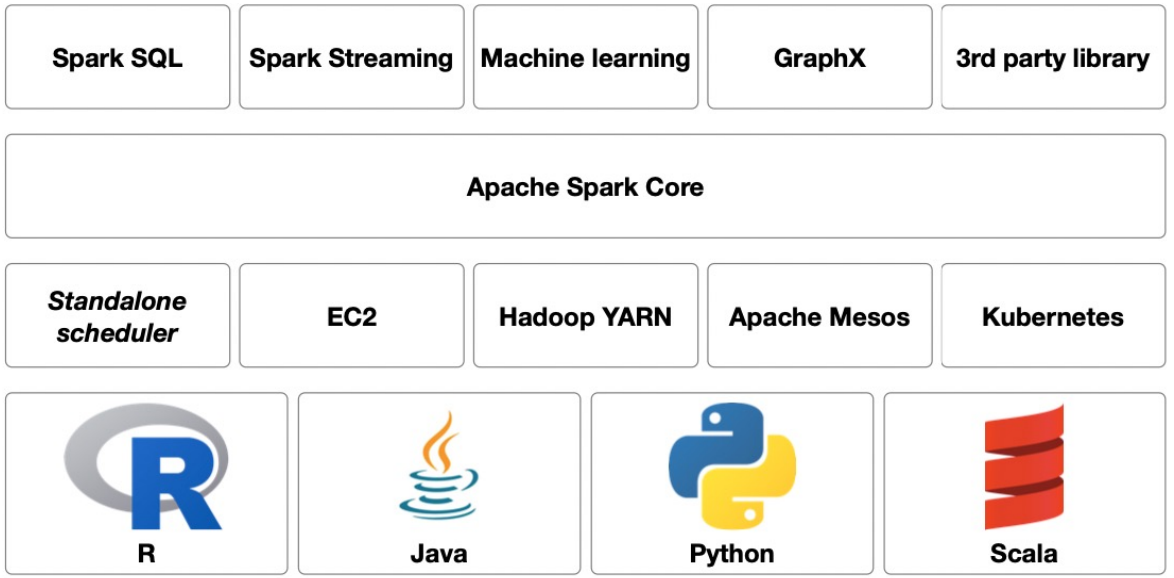
# Ecosystem: Spark Streaming

Data analysis of streaming data
- e.g. tweets, log messages, SCADA

Features of stream processing
- High-throughput
- Fault-tolerant
- End-to-end
- Exactly once

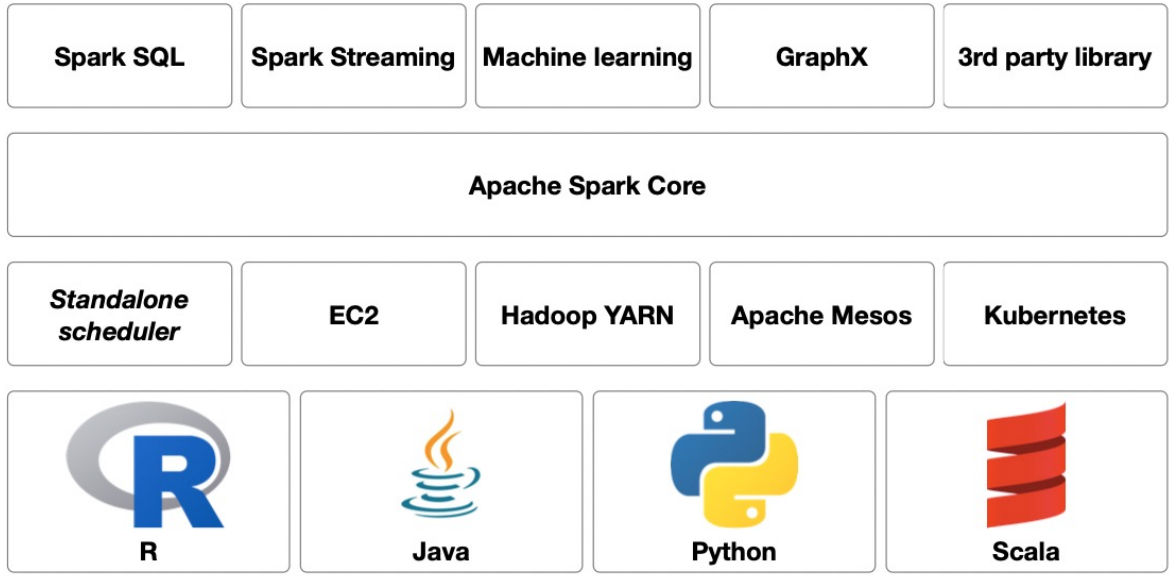
High-level abstraction of a discretized stream
- Stream represented as a sequence of RDD

With Spark 2.3.x and above continuous processing
- End-to-end low latency (< 1ms)




# Ecosystem: Spark MLlib for Machine Learning

- Common ML functionalities
  - 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.
- Two APIs
  - RDD-based API (*spark.mllib package*)
  - Spark 2.0+, DataFrame-based API (*spark.ml package*)
- Methods scale out across the cluster by default



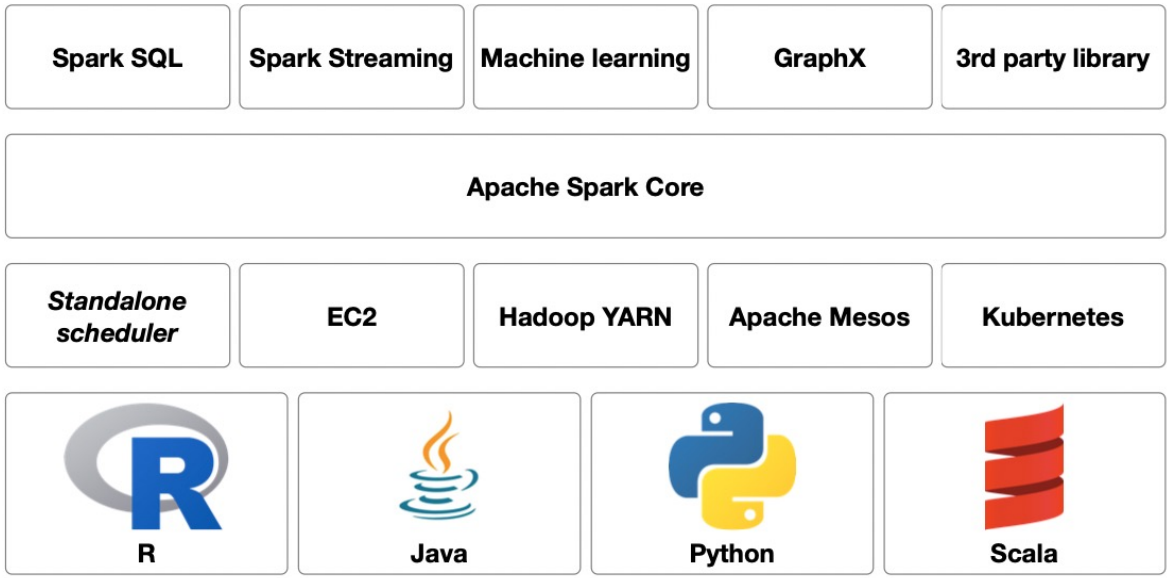
# Ecosystem: GraphX

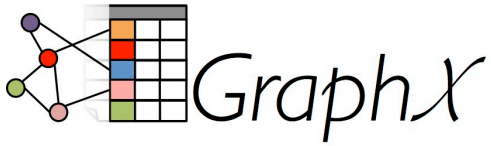
Support for graphs and graph-parallel computation

- Extension of RDDs (Graph)
- Direct multigraph with properties on vertices and edges

Graph computation operators

- subgraph, joinVertices, and aggregateMessages, etc.
- PregelAPI support



# Spark Modes

Spark operates in 4 different modes:

- **Standalone Mode:** Here all processes run within the same JVM process.
- **Standalone Cluster Mode:** In this mode, it uses the Job-Scheduling framework in-built in Spark.
- **Apache Mesos:** In this mode, the work nodes run on various machines, but the driver runs only in the master node.
- **Hadoop YARN:** In this mode, the drivers run inside the application's master node and is handled by YARN on the Cluster.

# Spark Context

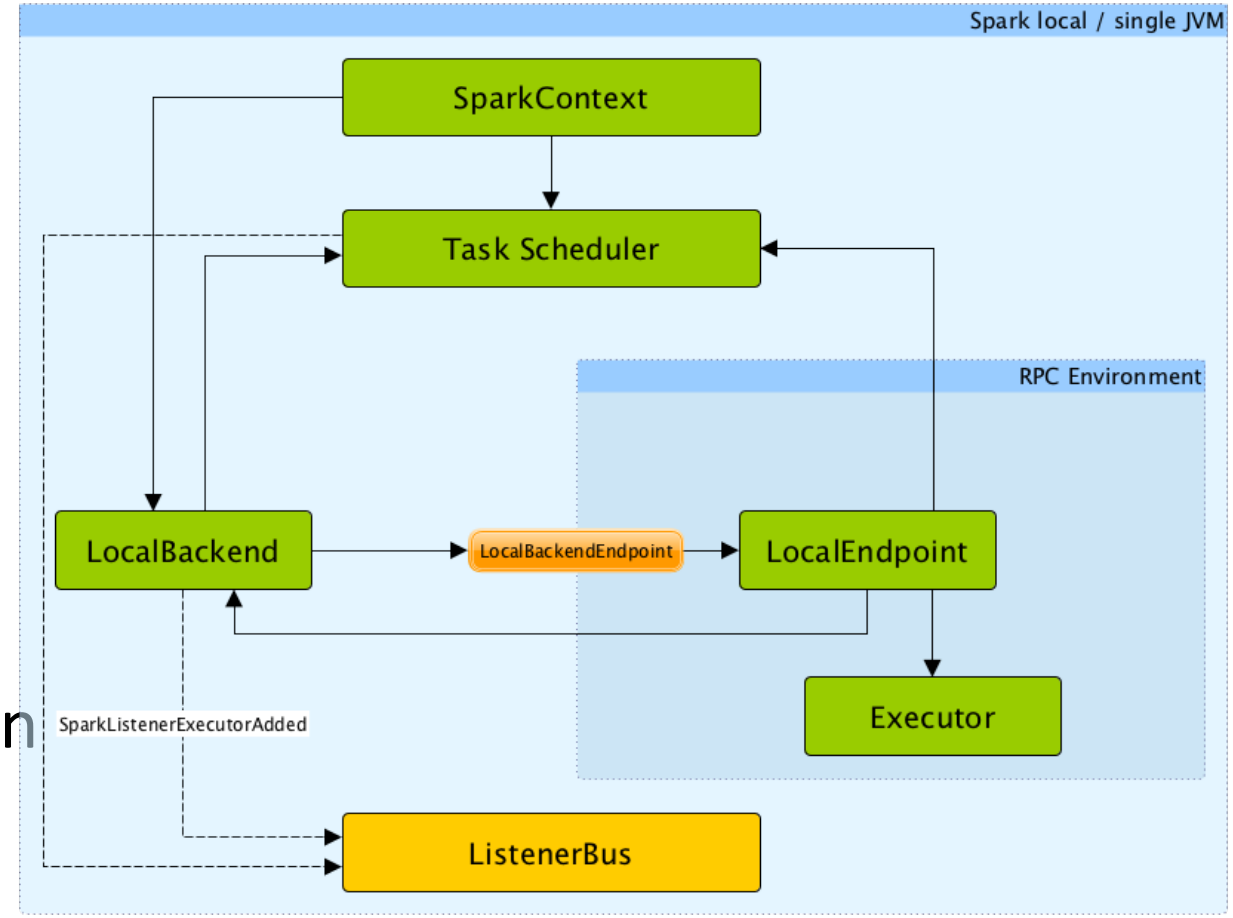
- driver Spark application which communicates the user commands to the Spark Workers
- `SparkContext` is an object, which coordinates with the cluster manager about the resources required for execution and the actual tasks that need to be executed
- Submitting spark applications: Submitting Applications - Spark 3.2.1 Documentation (apache.org)

# Execution modes

- Local mode
  - „Pseudo-cluster“ ad-hoc setup using script
- Cluster mode
  - Running via cluster manager
- Interactive mode
  - Direct manipulation in a shell (*pyspark*, *spark-shell*)

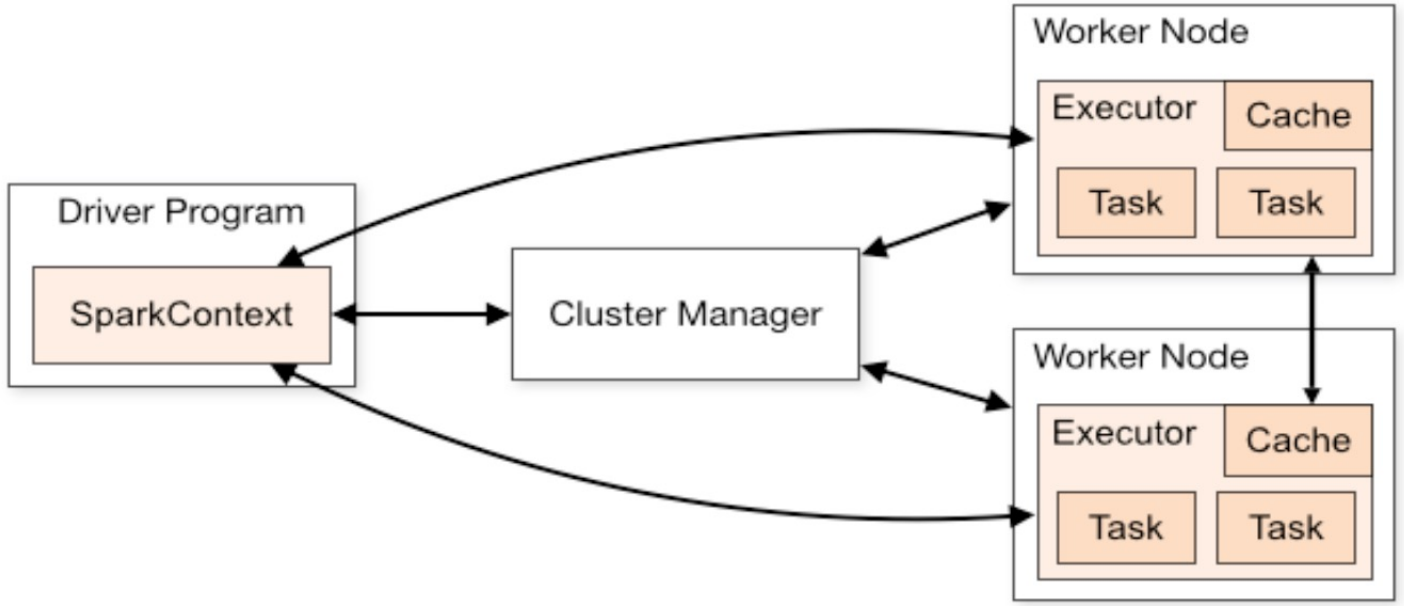
# Spark execution modes Local mode

- Non-distributed single-JVM deployment mode
- Spark library spawns (in a JVM)
  - driver
  - scheduler
  - master
  - executor
- Parallelism is the number of threads defined by a parameter N in a spark master URL
  - *local[N]*

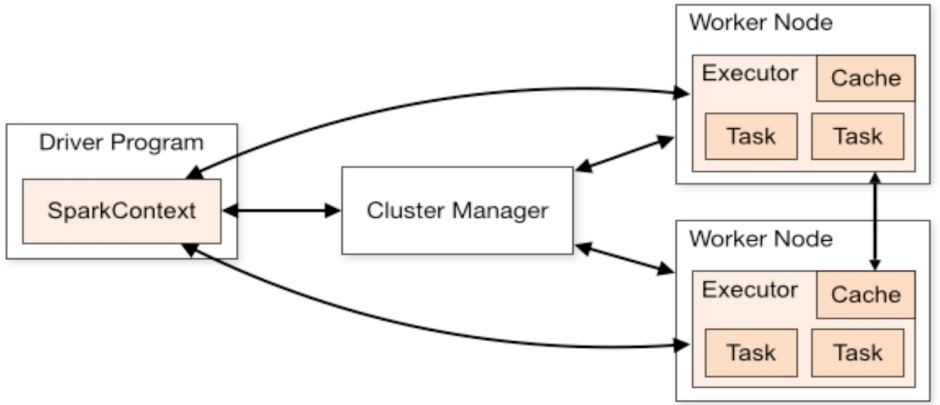


# Spark execution modes Cluster mode

- Deployment on a private cluster
  - Apache Mesos
  - Hadoop YARN
  - Kubernetes
  - Standalone mode, ...

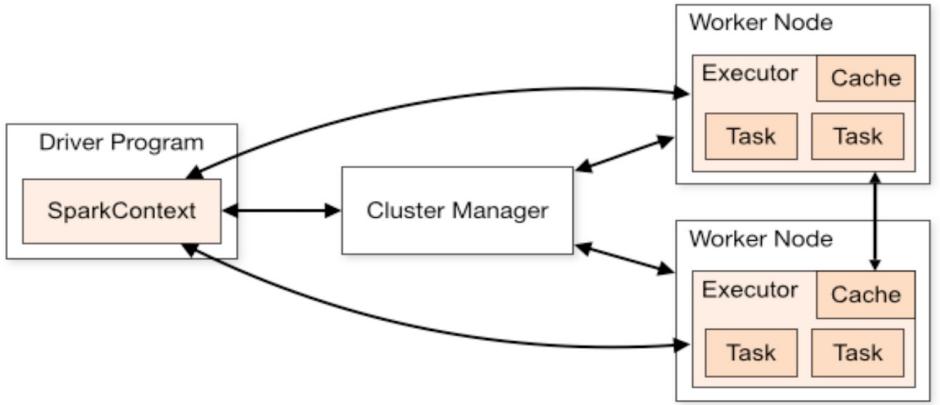


# Spark execution modes Cluster mode



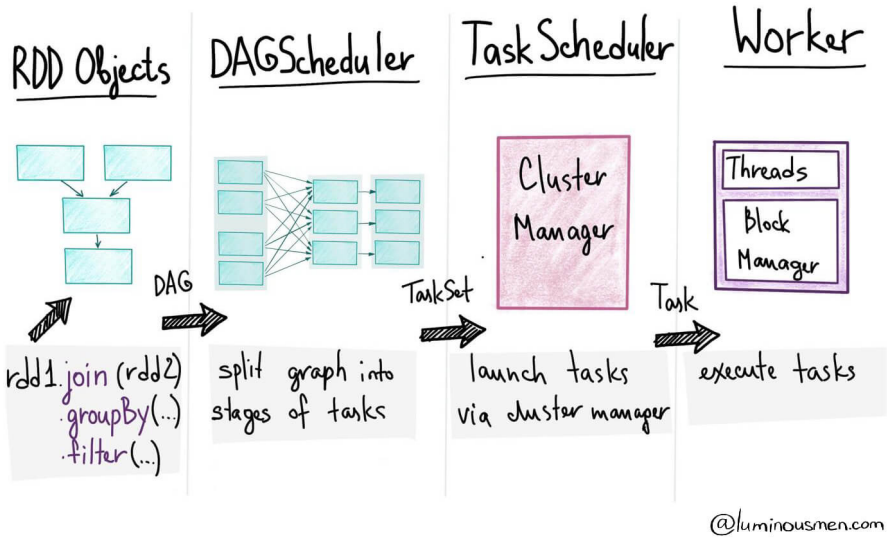

- Components
  - Worker
    - Node in a cluster, managed by an executor
    - Executor manages computation, storage and caching
  - Cluster manager
    - Allocates resources via SparkContext with Driver program
  - Driver program
    - A program holding SparkContext and main code to execute in Spark
    - Sends application code to executors to execute
    - Listens to incoming connections from executors

# Spark execution modes Cluster mode




- Deploy modes (*standalone clusters*)
  - Client mode (default)
    - Driver runs in the same process as client that submits the app
  - Cluster mode
    - Driver launched from a worker process
    - Client process exits immediately after application submission

# Spark execution process




1. Data preparation/import
   - RDDs creation – i.e. parallel dataset with partitions
2. Transformations/actions definition*
   - Creation of tasks (units of work) sent to one executor
   - Job is a set of tasks executed by an action*
3. Creation of a directed acyclic graph (DAG)
   - Contains a graph of RDD operations
   - Defition of stages – set of tasks to be executed in parallel (i.e. at a partition level)
4. Execution of a program (application)

# Module 2

# Module 2

Basic Spark data structures (RDDs)

- RDDs and operations
- RDD transformation and RDD actions

Creating DataFrames

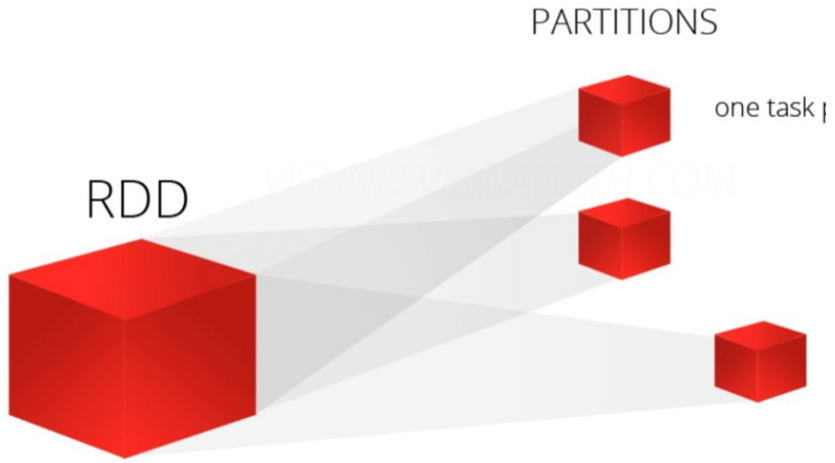
Creating DataSets

Working with Spark API

- Spark DataFrame API
- Structured API (SQL)

# Spark Programming concepts - RDD

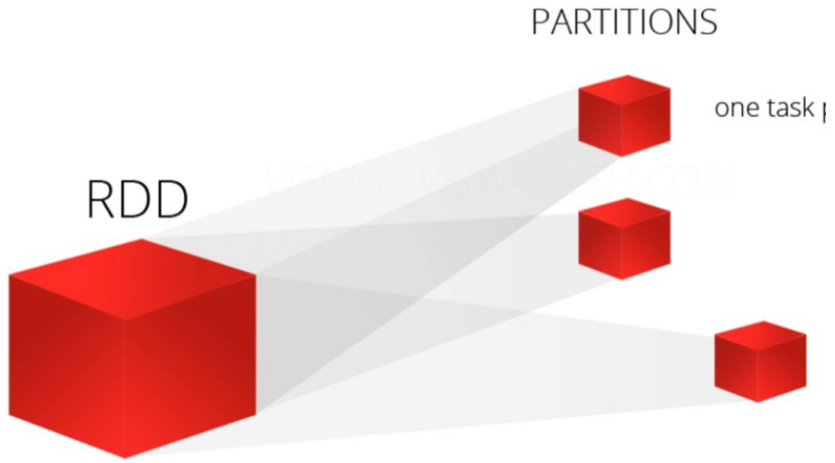
- RDD – Resilient distributed datases
- Basic data representation in Spark
- A distributed collection of items – partitions
  - Enables parallel operations
- RDDs are immutable (”read-only“)
- Fault-tolerant
  - Achieving 100% data preseration, every function creates new partition
- Caching and different storage levels possible
- Supports a set of Spark transformation and actions



# Spark Programming concepts - RDD

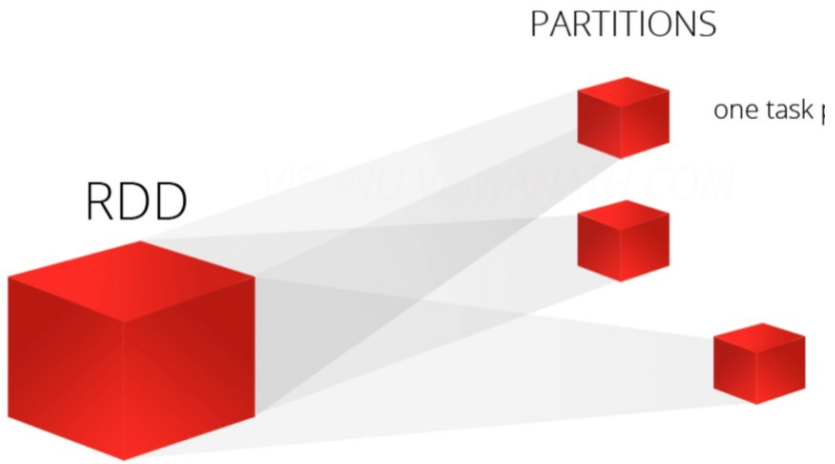
- Computations are expressed using
  - Creation of new RDDs
  - Transforming existing RDDs
  - Operations on RDDs to compute results (actions!)

- Distributes the data within RDDs across nodes (executors) in the clust and parallelizes the calculations

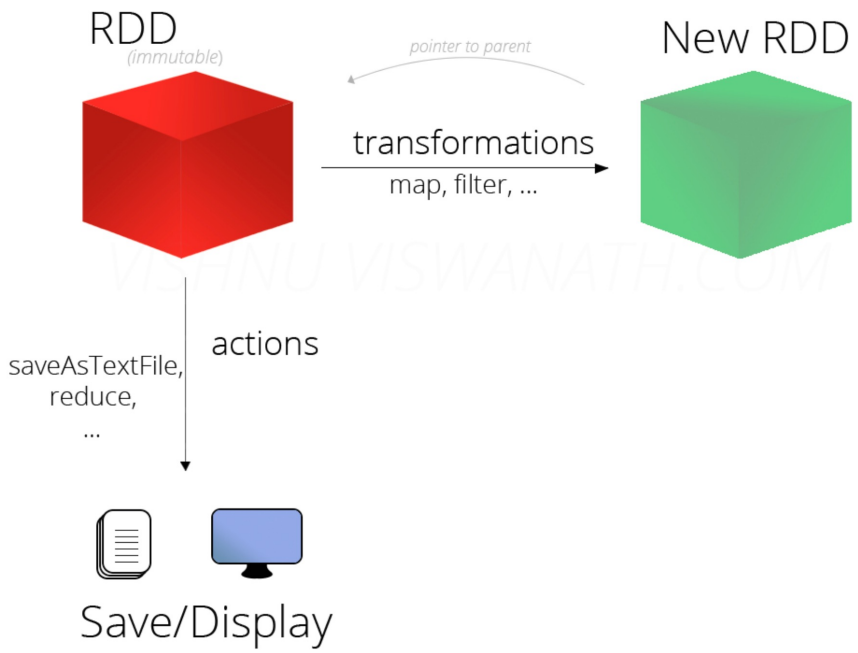


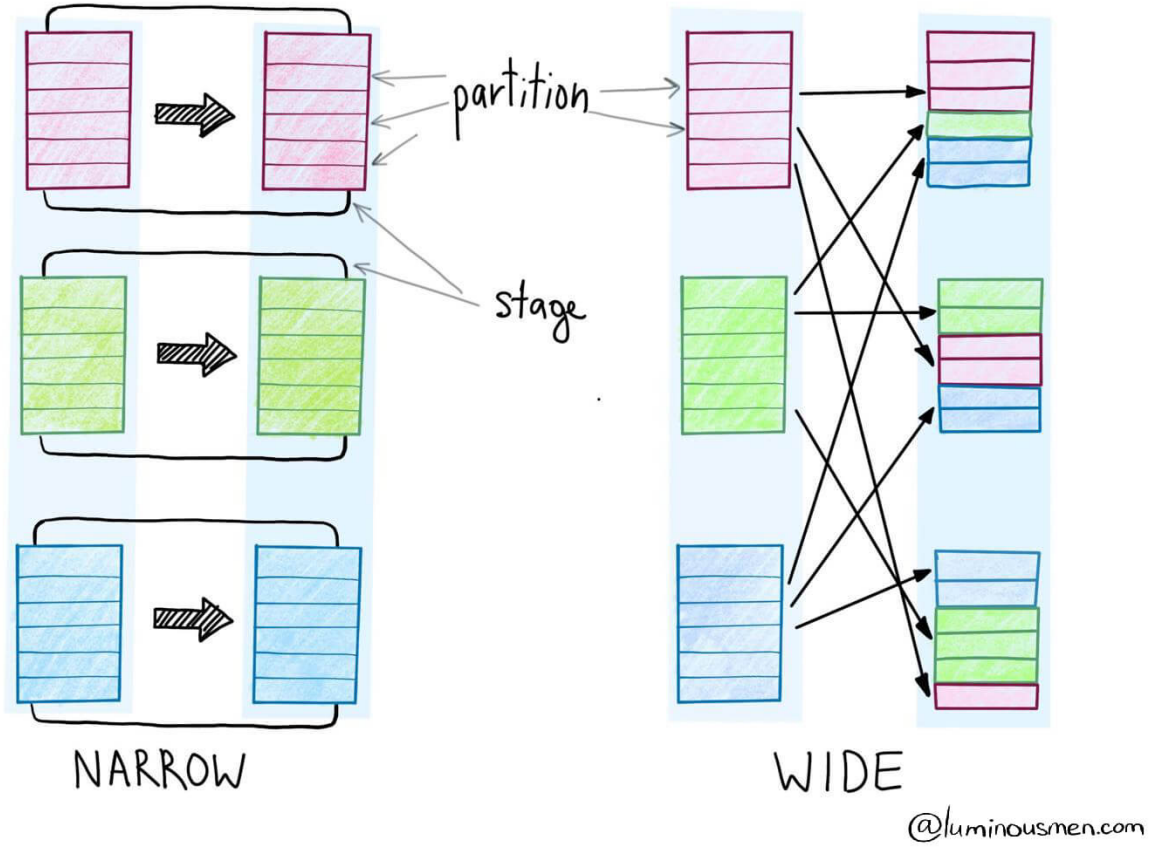
# RDD operations

- RDDs enable following operations
  - **transformations**
    - lazy operations that return a new RDD from input RDDs
    - narrow or wide types
    - examples: map, filter, join, groupByKey...
  - **actions**
    - return a result or write to storage,
      execute transformations
    - examples: count, collect, save



# RDD: Transformation and actions



# RDD Transformation vs. Actions

 = easy  = medium

## Essential Core & Intermediate Spark Operations


**General**

- map
- filter
- flatMap
- mapPartitions
- mapPartitionsWithIndex
- groupBy
- sortBy

**Math / Statistical**

- sample
- randomSplit

**Set Theory / Relational**

- union
- intersection
- subtract
- distinct
- cartesian
- zip

**Data Structure / I/O**

- keyBy
- zipWithIndex
- zipWithUniqueID
- zipPartitions
- coalesce
- repartition
- repartitionAndSortWithinPartitions
- pipe

---


- reduce
- collect
- aggregate
- fold
- first
- take
- forEach
- top
- treeAggregate
- treeReduce
- forEachPartition
- collectAsMap

- count
- takeSample
- max
- min
- sum
- histogram
- mean
- variance
- stdev
- sampleVariance
- countApprox
- countApproxDistinct

- takeOrdered

- saveAsTextFile
- saveAsSequenceFile
- saveAsObjectFile
- saveAsHadoopDataset
- saveAsHadoopFile
- saveAsNewAPIHadoopDataset
- saveAsNewAPIHadoopFile

Source: URL

# RDD in Spark

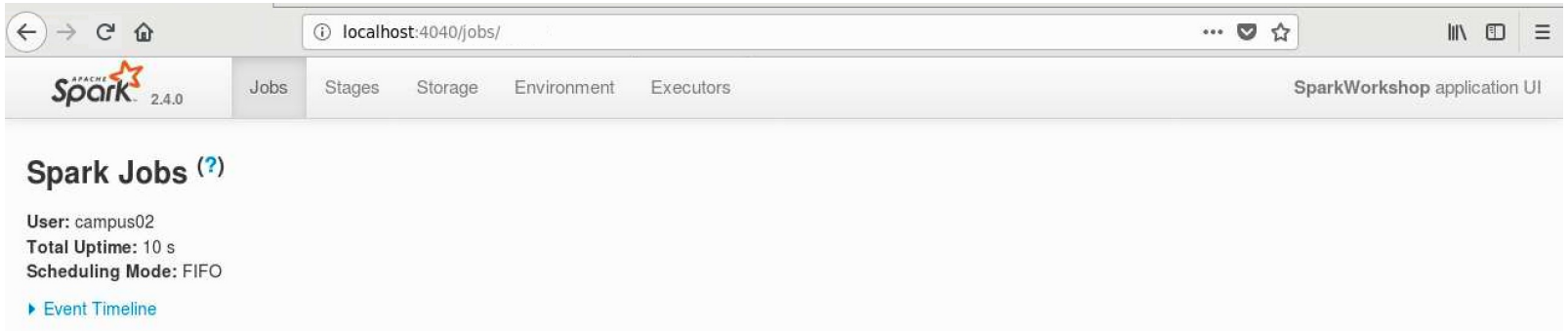
- Continue with notebooks
- Explore Spark Web UI

# RDD in Spark and explore Web UI

We will use pySpark library interactively

```
import pyspark
sc = pyspark.SparkContext(appName=,‘SparkWorkshopSQLBits', master='local[1]')
```

```
2020-09-15 16:28:01 WARN  NativeCodeLoader:62 - Unable to load native-hadoop library for your platform... using builtin-java clas
ses where applicable
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
2020-09-15 16:28:02 WARN  Utils:66 - Service 'SparkUI' could not bind on port 4040. Attempting port 4041.
>>>
```



# RDD in Spark and explore Web UI

## Creation of RDDs

- From a collection

```
rdd1 = sc.parallelize([('John', 23), ('Mark', 11), ('Jenna', 44), ('Sandra', 61)])
```

- From a file

```
rdd2 = sc.textFile('data/IMDB Dataset.csv‘)
```

## Basic transformations map(), filter(), flatMap()

```
older = rdd1.filter(lambda x: x[1] > 18)
anonymized = older.map(lambda x: (x[0][0], x[1]))

birthdays = rdd1.map(lambda x: list(range(1, x[1]+1)))
birthdays = rdd1.flatMap(lambda x: list(range(1, x[1]+1)))

birthdays.map(lambda x: (x, 1)).groupByKey() \
     .mapValues(lambda vals: len(list(vals))).sortByKey()
```

# RDD in Spark and explore Web UI

- Further actions, transformations

```
rdd2.take(2)

def organize(line):
    data = line.split('",')
    data = data if len(data) == 2 else line.split(',')
    return (data[1], data[0][1:51] + ' ...')
movies = rdd2.filter(lambda x: x !=
'review,sentiment').map(organize)

movies.count() // 50.000
movies = movies.filter(lambda x: x[0] in ['positive', 'negative'])
movies.count() // 45.936
movieCounts = movies.groupByKey().map(lambda x: (x[0], len(x[1])))
```

# RDD in Spark and explore Web UI

## Caching

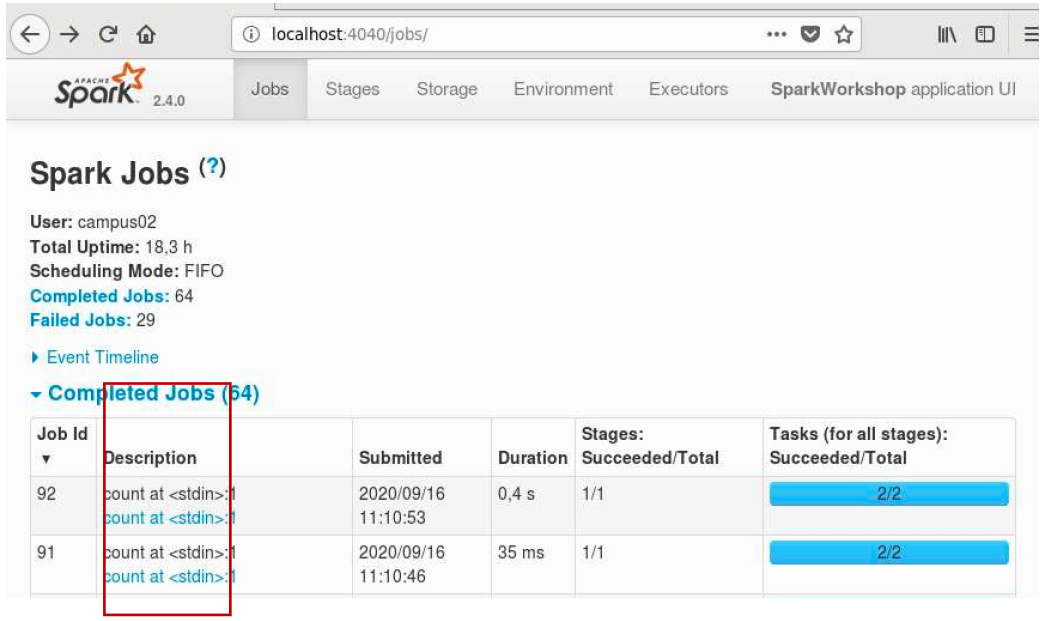
```
movies.take(2)
```


```
posReviews = movies.filter(lambda x: x[0] ==
'positive').map(lambda x: x[1])
negReviews = movies.filter(lambda x: x[0] ==
'negative').map(lambda x: x[1])

posReviews.cache().collect()
```

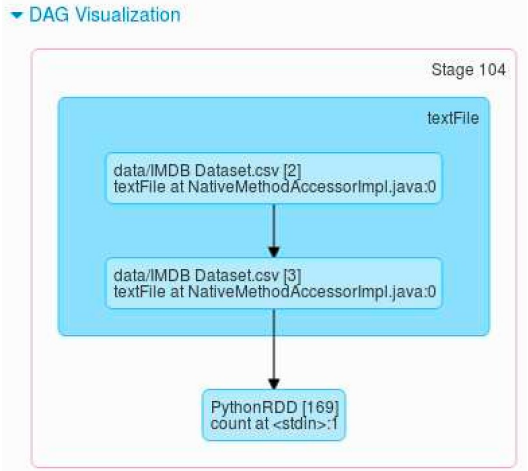
RDD in Spark and explore Web UI

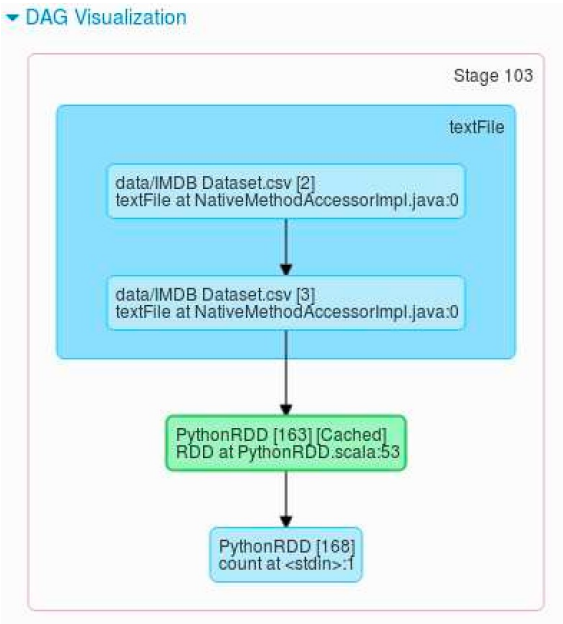
Caching

```
posReviews.filter(lambda x: 'good' in
x).count() //605
negReviews.filter(lambda x: 'bad' in x).count()
//788
```



# RDD in Spark and explore Web UI




**Tasks (2)**

| Index ▲ | ID | Attempt | Status | Locality Level | Executor ID | Host | Launch Time | Duration | GC Time | Input Size / Records | Errors |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 138 | 0 | SUCCESS | PROCESS_LOCAL | driver | localhost | 2020/09/16 11:10:53 | 0,2 s | 4 ms | 32.1 MB / 25311 | |
| 1 | 139 | 0 | SUCCESS | PROCESS_LOCAL | driver | localhost | 2020/09/16 11:10:53 | 0,2 s | 4 ms | 31.1 MB / 24690 | |

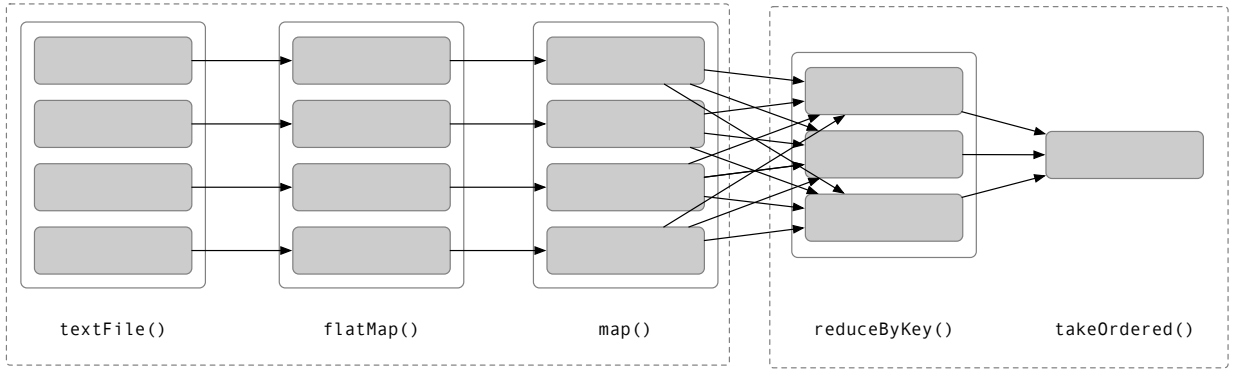
**Tasks (2)**

| Index ▲ | ID | Attempt | Status | Locality Level | Executor ID | Host | Launch Time | Duration | GC Time | Input Size / Records | Errors |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 136 | 0 | SUCCESS | PROCESS_LOCAL | driver | localhost | 2020/09/16 11:10:46 | 18 ms | | 469.2 KB / 16 | |
| 1 | 137 | 0 | SUCCESS | PROCESS_LOCAL | driver | localhost | 2020/09/16 11:10:46 | 12 ms | | 453.7 KB / 16 | |

# RDD in Spark and explore Web UI

## DAG exploration

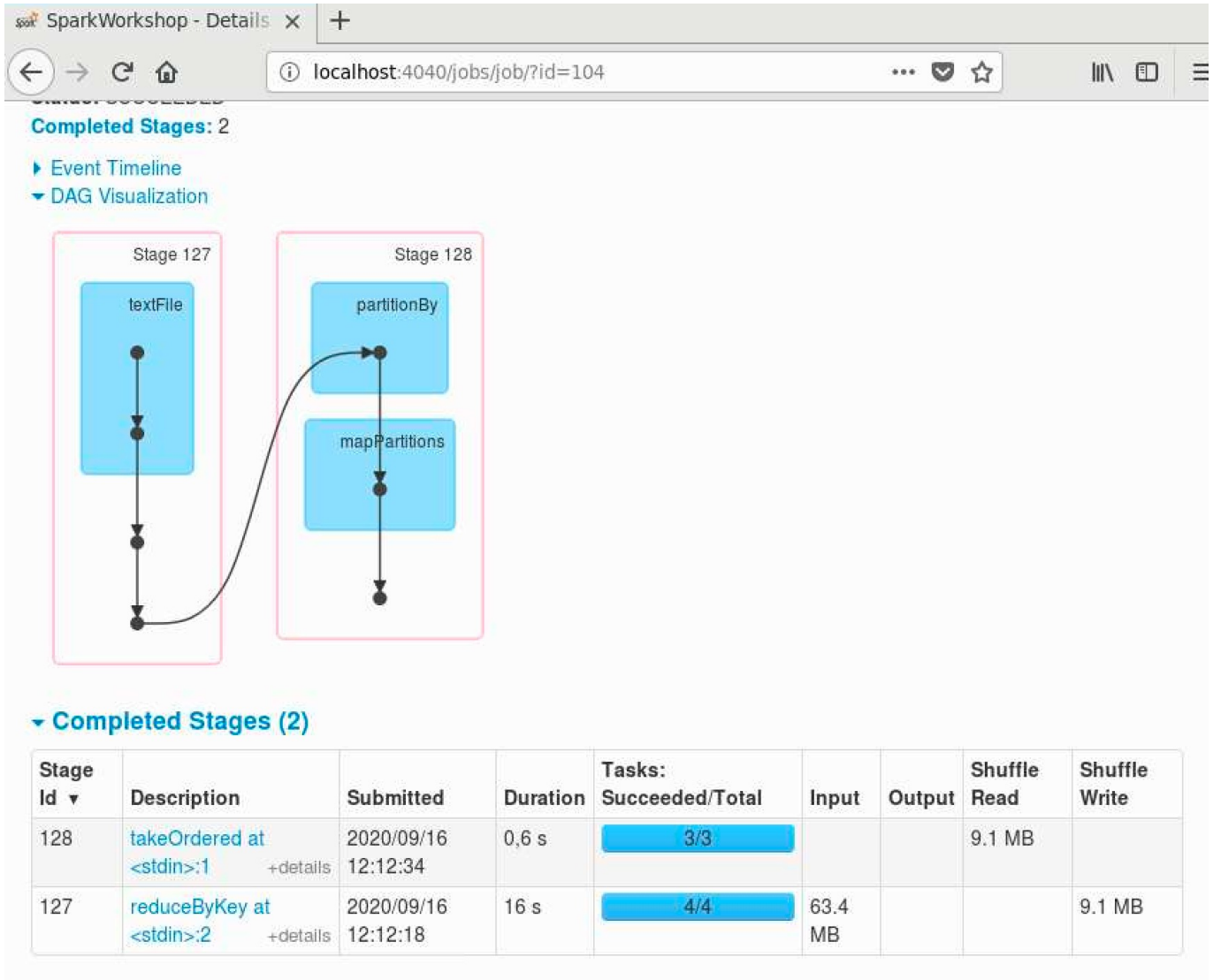
```
def splitLine(line):
    return line.replace(',', ' ').replace('"', ' ').replace('.', '
').split()


rdd2 = sc.textFile('data/IMDB Dataset.csv', 4)
wordCounts = rdd2.flatMap(splitLine).map(lambda word: (word, 1)). \
                    reduceByKey(lambda a,b: a+b, 3)
wordCounts.takeOrdered(10, key = lambda x: -x[1])
```



# RDD in Spark and explore Web UI

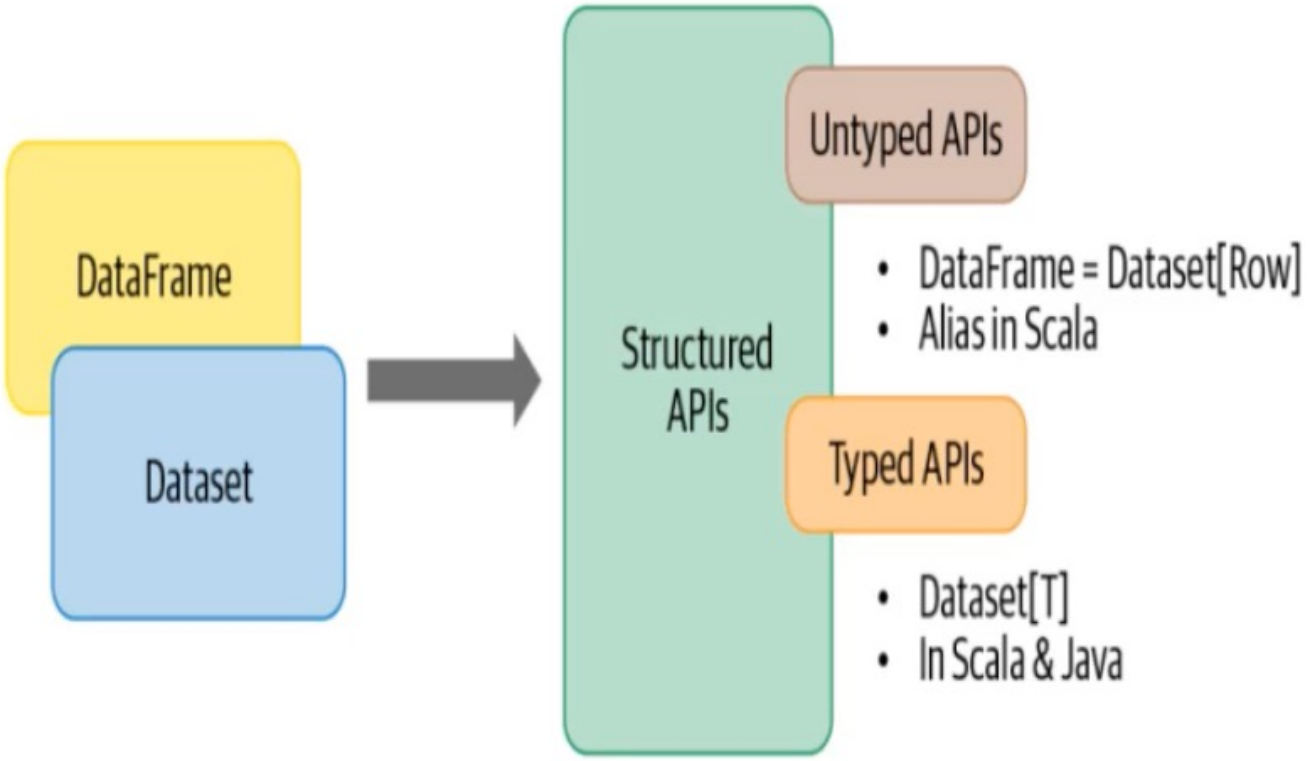
DAG exploration

(admin console  result)



# Structured API

Stuctured API – to address faster adoption and ease of use in comparison with RDD API



# RDD API

```
from pyspark.sql import SparkSession
spark = (SparkSession.builder.appName("SampleRDDAPI").getOrCreate())
```

Command took 0.04 seconds -- by tomaz.kastrun@gmail.com at 06/03/2022, 10:06:59 on CL1_DB_Standard

Cmd 87

```
dataRDD = spark.sparkContext.parallelize([("Person",20),("Dog",2),("Person",34),("Person",63), ("Dog", 6)])
```

Command took 0.03 seconds -- by tomaz.kastrun@gmail.com at 06/03/2022, 10:17:04 on CL1_DB_Standard

Cmd 88

```
# Use map and reduceByKey Transformation with lamba function
# to aggreate and compute an average
agesRDD = (dataRDD
           .map(lambda x: (x[0], (x[1], 1)))
            .reduceByKey(lambda x,y: (x[0] + y[0], x[1] + y[1]))
            .map(lambda x: (x[0], x[1][0]/x[1][1]))
           )
```

Command took 0.05 seconds -- by tomaz.kastrun@gmail.com at 06/03/2022, 10:17:06 on CL1_DB_Standard

Cmd 89

```
for data in agesRDD.collect():
  print(data)
```

▸ (1) Spark Jobs

```
('Person', 39.0)
('Dog', 4.0)
```

# RDD API vs SQL

```
1  from pyspark.sql import SparkSession
2  spark = (SparkSession.builder.appName("SampleRDDAPI").getOrCreate())
```

Command took 0.04 seconds -- by tomaz.kastrun@gmail.com at 06/03/2022, 10:06:59 on CL1_DB_Standard

Cmd 87

```
1  dataRDD = spark.sparkContext.parallelize([("Person",20),("Dog",2),("Person",34),("Person",63), ("Dog", 6)])
```

Command took 0.03 seconds -- by tomaz.kastrun@gmail.com at 06/03/2022, 10:17:04 on CL1_DB_Standard

Cmd 88

```
1  # Use map and reduceByKey Transformation with lamba function
2  # to aggreate and compute an average
3  agesRDD = (dataRDD
4            .map(lambda x: (x[0], (x[1], 1)))
5             .reduceByKey(lambda x,y: (x[0] + y[0], x[1] + y[1]))
6             .map(lambda x: (x[0], x[1][0]/x[1][1]))
7            )
```

Command took 0.05 seconds -- by tomaz.kastrun@gmail.com at 06/03/2022, 10:17:06 on CL1_DB_Standard

Cmd 89

```
1  for data in agesRDD.collect():
2    print(data)
```

▸ (1) Spark Jobs

```
('Person', 39.0)
('Dog', 4.0)
```

```
1
2   SELECT
3      AVG(Age) AS Average_age
4      ,typeBinary
5    FROM
6      dbo.RDDAPITable
7    GROUP BY typeBinary
8
9
```

# From RDD API to DataFrame and Dataset API

The dataset API is a collection of strongly typed JVM (Java Virtual Machine) data object in Scala and classes in Java

## Typed:

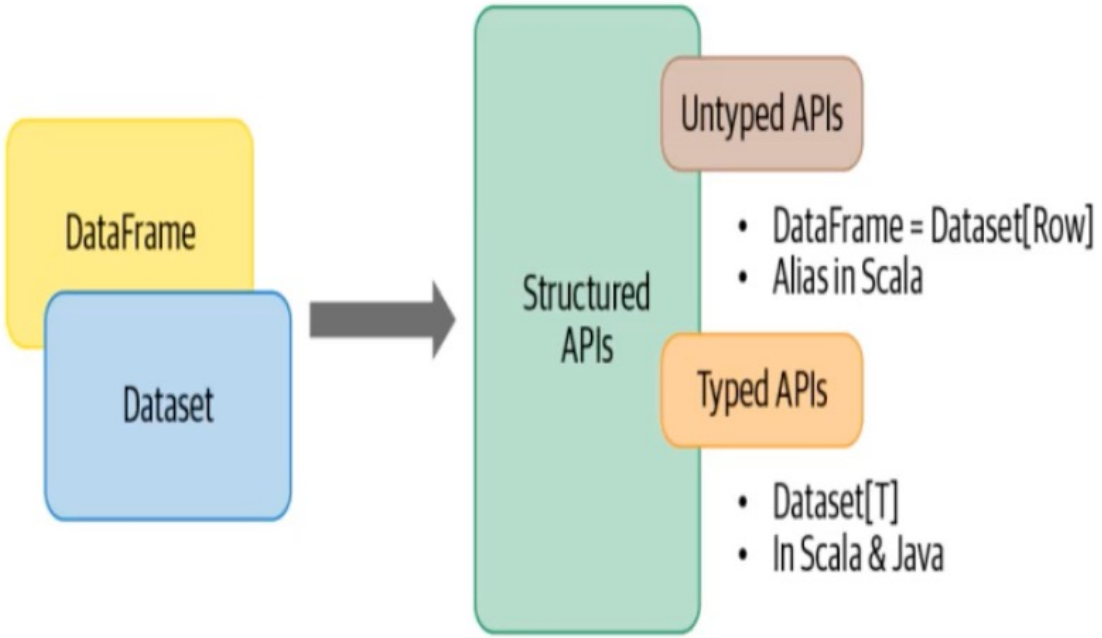
Float avg_age = 23.4

avg_age = “SQLBits” // will return error

## Untyped:

avg_age = 23.4

avg_age = “SQLBits” // will NOT(!) return error




The DataFrame is an Untyped API that has support for Python and Scala

Data Engineers prefere JAVA and Scala due to typed nature, because it provides better data integratity.

# RDD API

```
1  from pyspark.sql import SparkSession
2  spark = (SparkSession.builder.appName("SampleRDDAPI").getOrCreate())
```

Command took 0.04 seconds -- by tomaz.kastrun@gmail.com at 06/03/2022, 10:06:59 on CL1_DB_Standard

Cmd 87

```
1  dataRDD = spark.sparkContext.parallelize([("Person",20),("Dog",2),("Person",34),("Person",63), ("Dog", 6)])
```

Command took 0.03 seconds -- by tomaz.kastrun@gmail.com at 06/03/2022, 10:17:04 on CL1_DB_Standard

Cmd 88

```
1  # Use map and reduceByKey Transformation with lamba function
2  # to aggreate and compute an average
3  agesRDD = (dataRDD
4             .map(lambda x: (x[0], (x[1], 1)))
5              .reduceByKey(lambda x,y: (x[0] + y[0], x[1] + y[1]))
6              .map(lambda x: (x[0], x[1][0]/x[1][1]))
7            )
```

Command took 0.05 seconds -- by tomaz.kastrun@gmail.com at 06/03/2022, 10:17:06 on CL1_DB_Standard

Cmd 89

```
1  for data in agesRDD.collect():
2    print(data)
```

▸ (1) Spark Jobs

```
('Person', 39.0)
('Dog', 4.0)
```

# DataFrame API

Cmd 90

```
1  #Create a DataFrame
2  data_df = spark.createDataFrame([("Person",20),("Dog",2),("Person",34),("Person",63), ("Dog", 6)],["type", "age"])
```

▸ data_df: pyspark.sql.dataframe.DataFrame = [type: string, age: long]

Command took 0.06 seconds -- by tomaz.kastrun@gmail.com at 06/03/2022, 10:32:00 on CL1_DB_Standard

Cmd 91

```
1  avg_df = data_df.groupby("type").avg("age")
2  avg_df.show()
```

▾ avg_df: pyspark.sql.dataframe.DataFrame

```
type: string
avg(age): double
```

```
+------+--------+
|  type|avg(age)|
+------+--------+
|Person|    39.0|
|   Dog|     4.0|
+------+--------+
```

Command took 0.88 seconds -- by tomaz.kastrun@gmail.com at 06/03/2022, 10:35:54 on CL1_DB_Standard

Cmd 92

# DataFrame API

- The DataFrame accounts for readable „English“ like code which resembles SQL queries
- The DataFrame allows you to have a structured presenttion of your data into rows and columns like database tables
- The DataFrame allows you to assign schema to address potential the data integrity issues

Cmd 90

```
1  #Create a DataFrame
2  data_df = spark.createDataFrame([("Person",20),("Dog",2),("Person",34),("Person",63), ("Dog", 6)],["type
```

▸ data_df: pyspark.sql.dataframe.DataFrame = [type: string, age: long]

Command took 0.06 seconds -- by tomaz.kastrun@gmail.com at 06/03/2022, 10:32:00 on CL1_DB_Standard

```
1  avg_df = data_df.groupby("type").avg("age")
2  avg_df.show()
```

▸ (2) Spark Jobs

▾ avg_df: pyspark.sql.dataframe.DataFrame

```
 type: string
 avg(age): double
```

```
+------+--------+
|  type|avg(age)|
+------+--------+
|Person|    39.0|
|   Dog|     4.0|
+------+--------+
```

Command took 0.88 seconds -- by tomaz.kastrun@gmail.com at 06/03/2022, 10:35:54 on CL1_DB_Standard

Cmd 92

# Working with Spark API

- Spark DataFrame API
- Structured API (SQL)

Continue with Notebooks!

# Module 3

Designing and building pipelines, moving data and data models with Spark

# Module 3

Data Pipelines and ETL/ELT

- ETL Query
- Data Source support
- Schema inference
- Data Quality
- Transformation

Python for ETL and performance

Moving and copying data

Creating data models

# What is pipeline

**Data pipelines are set of instructions for data movement from source to sink. It is commonly known in data science community and also between data engineers for ETL or ELT.**

Pipeline is understood as a:

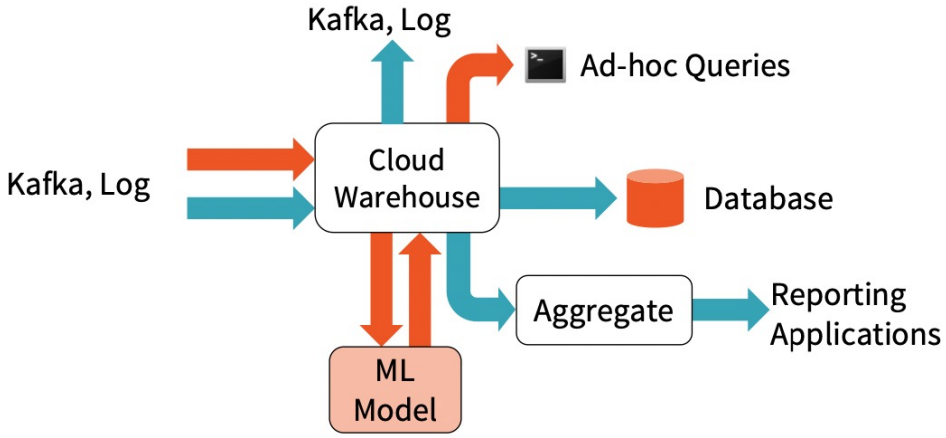
- sequence of transformations on data
- Source data is typically semi-structured/unstructured (JSON, CSV etc.) and structured (JDBC, Parquet, ORC, the other Hive-serde tables)
- Output data is integrated, structured and curated.
- Data at the end of the pipeline is ready for further data processing, analysis and reporting.

# Daily scenario of data pipeline

**Sequence** of transformation on data

**Source**: semi-structured/unstructured (JSON, CSV etc.) and structured (JDBC, Parquet, ORC, the other Hive-serve tables)

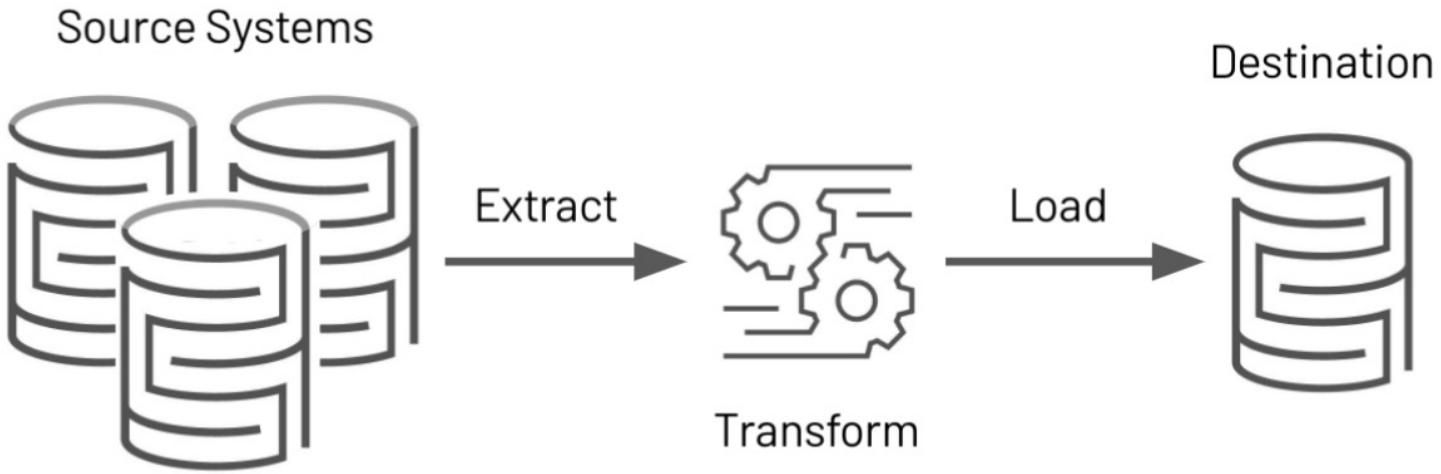
**Sink**: Data is integrated, structured and curated and ready for consumption.



# ETL is the *first Step* in a Data Pipeline

**Goal**: is to clean or curate the data.

- Retrieve data from sources (**E**XTRACT)
- Transform data into a consumable format (**T**RANSFORM)
- Transmit data to downstream consumers (**L**OAD)



# An ETL Query in Spark

```
spark.read.json("/source/path")
        .filter(...)
        .agg(...)
        .write.mode("append")
        .parquet("/output/path")
```

EXTRACT

TRANSFORM

LOAD

# An ETL Query in Spark

```
val csvTable = spark.read.csv("/source/path")
val jdbcTable = spark.read.format("jdbc")
      .option("url", "jdbc:postgresql:...")
      .option("dbtable", "DBO. SALES")
      .load()
csvTable
      .join(jdbcTable, Seq("shop_id"), "inner")
      .filter("region_id <= 14")
      .write
      .mode("overwrite") .format("parquet")
      .saveAsTable("outputTableName")
```

EXTRACT

TRANSFORM

LOAD

# Why is ETL process important!

- Matching different schemas
- Matching different data types
- Checking for data issues (corrupt files)
- Checking for changes in schema

# Why is ETL process hard?

- Scalability
- Continuous ETL and ELT
- Checking for data issues (corrupt files)

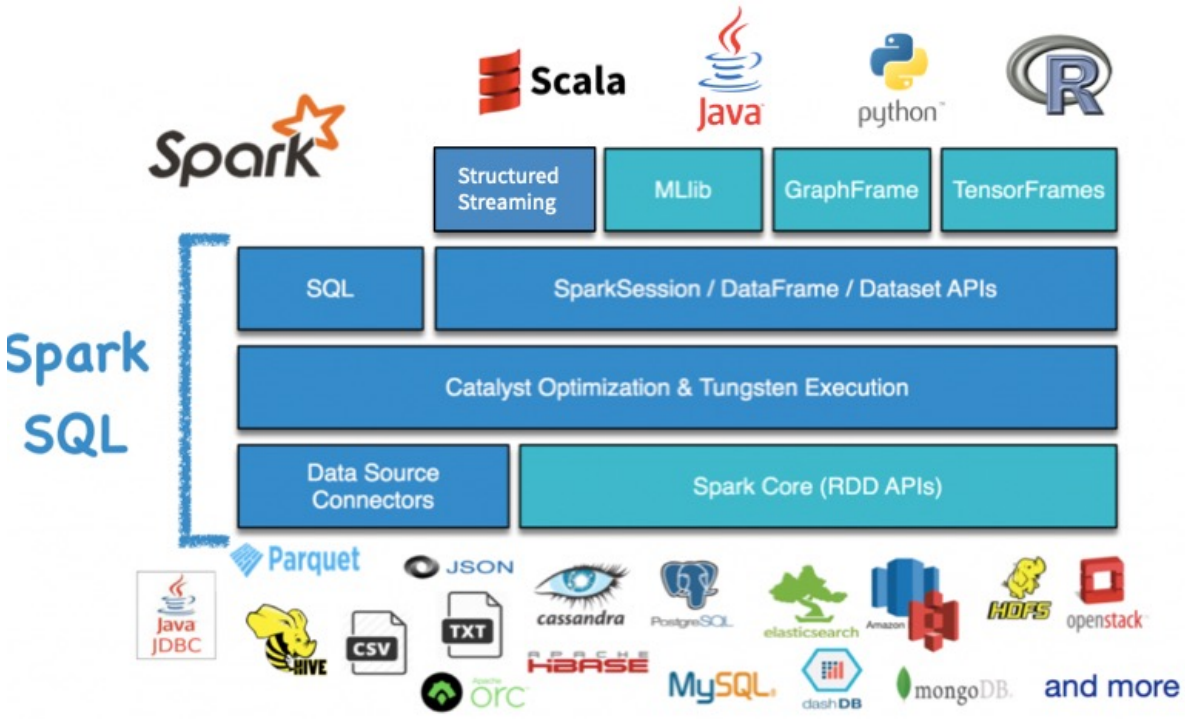
# ETL must have and must be!

- Error-prone
- Fully Logged
- ACID “in all aspects”
- Always “fast” and never “too slow”
- Cheap and low maintenance
- Never ”too complex” and easy to “scale”
- Fully documented

# This is why ETL is important

*Nobody wants messy and complex data ☺*

Spark SQL's flexible APIs, support for a wide variety of datasources, build-in support for structured streaming, state of art catalyst optimizer and tungsten execution engine make it a great framework for building end-to-end ETL pipelines.



# Data Source Supports

Built-in connectors in Spark:

- JSON, CSV, Text, Hive, Parquet, ORC, JDBC

Third-party data source connectors:

- https://spark-packages.org

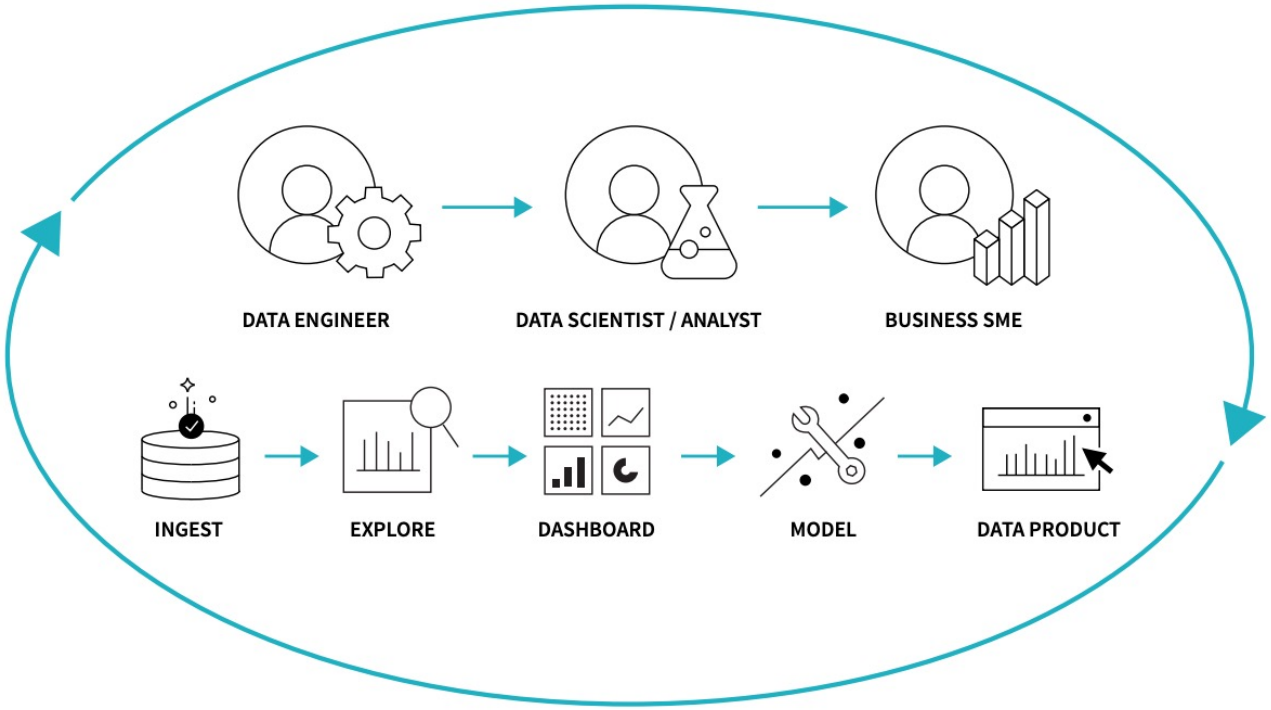
- Define your own data source connectors by Data Source APIs
  - Ref link: https://youtu.be/uxuLRiNoDio

# Module 4

Data and process orchestration

# Data and processes

- **DATA INGESTION AND EXPLORATION**
- TRAINING THE MODEL
- DEPLOY TO PRODUCTION

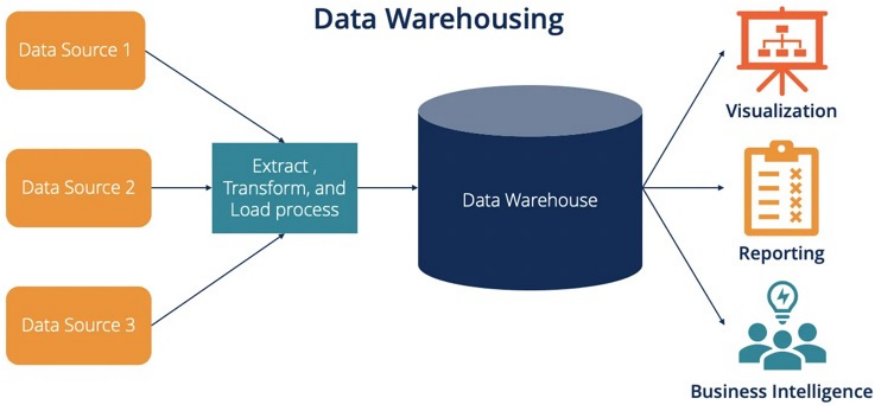


# Storage patterns

- Data Warehouse
- Data Lake
- Data Lakehouse

# Data warehouse

- Unified data repository for storing large amounts of information from multiple sources
- Combines relational datasets from business apps, ERP, CRM, FICO
- Transactional data and master data
- Processes: extracting, cleaning, matching, transforming, storing, versioning, consuming




Data warehousing. Image source: https://corporatefinanceinstitute.com/

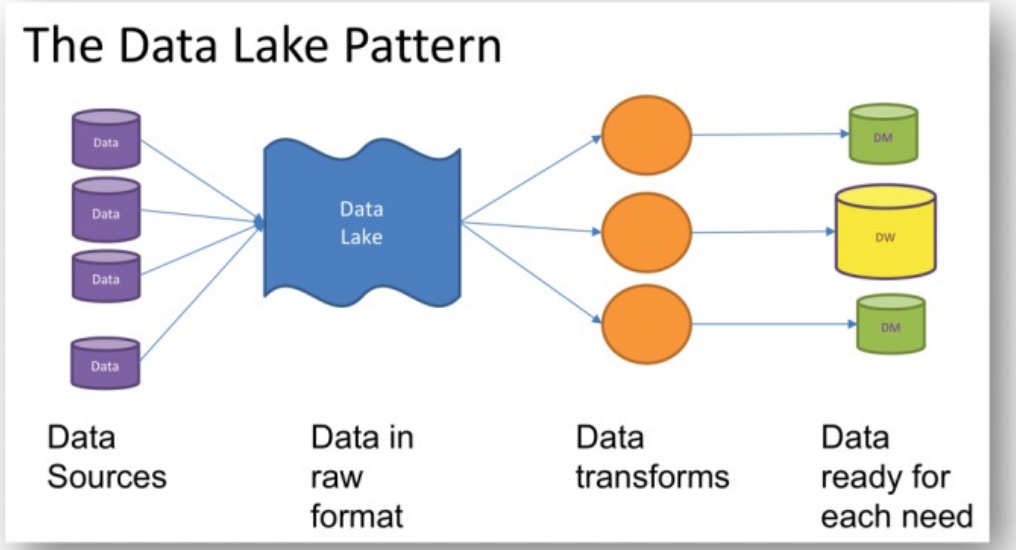
# Data Warehouse PRO  Data Warehouse CON

- Data canonization / standardization
- Improved data quality, consistency
- Improves decision-making business process and analytics

- Lack of flexibility
- “Struggles” with semi- and un-structured data
- High OPEX (implementation, CI/CD, maintenance)

# Data Lake

- Centralized and highly flexible storage repository
- For structured, unstructured data; original / raw format
- Durable, cost-effective
- Schema on-read
- ETL for purposes and “ad-hoc”
- Machine learning, Shallow/Deep
- IoT devices, streaming data




The data lake pattern. Image source: datakitchen.io

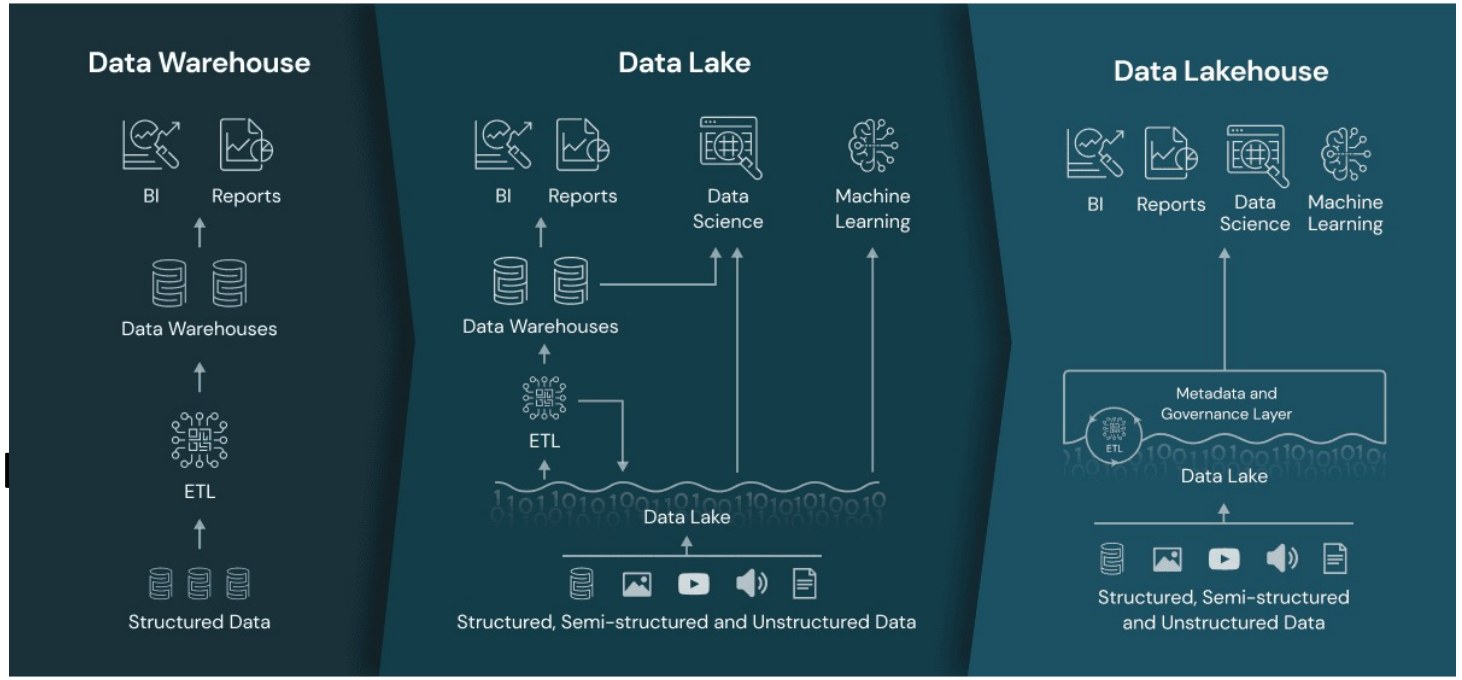
# Data Lake PRO

- Data consolidation (structured + unstructured)
- Improved flexibility
- Cost savings (object store)
- Improvedd and support for variety analytics, shallow and deep learning (machine learning)

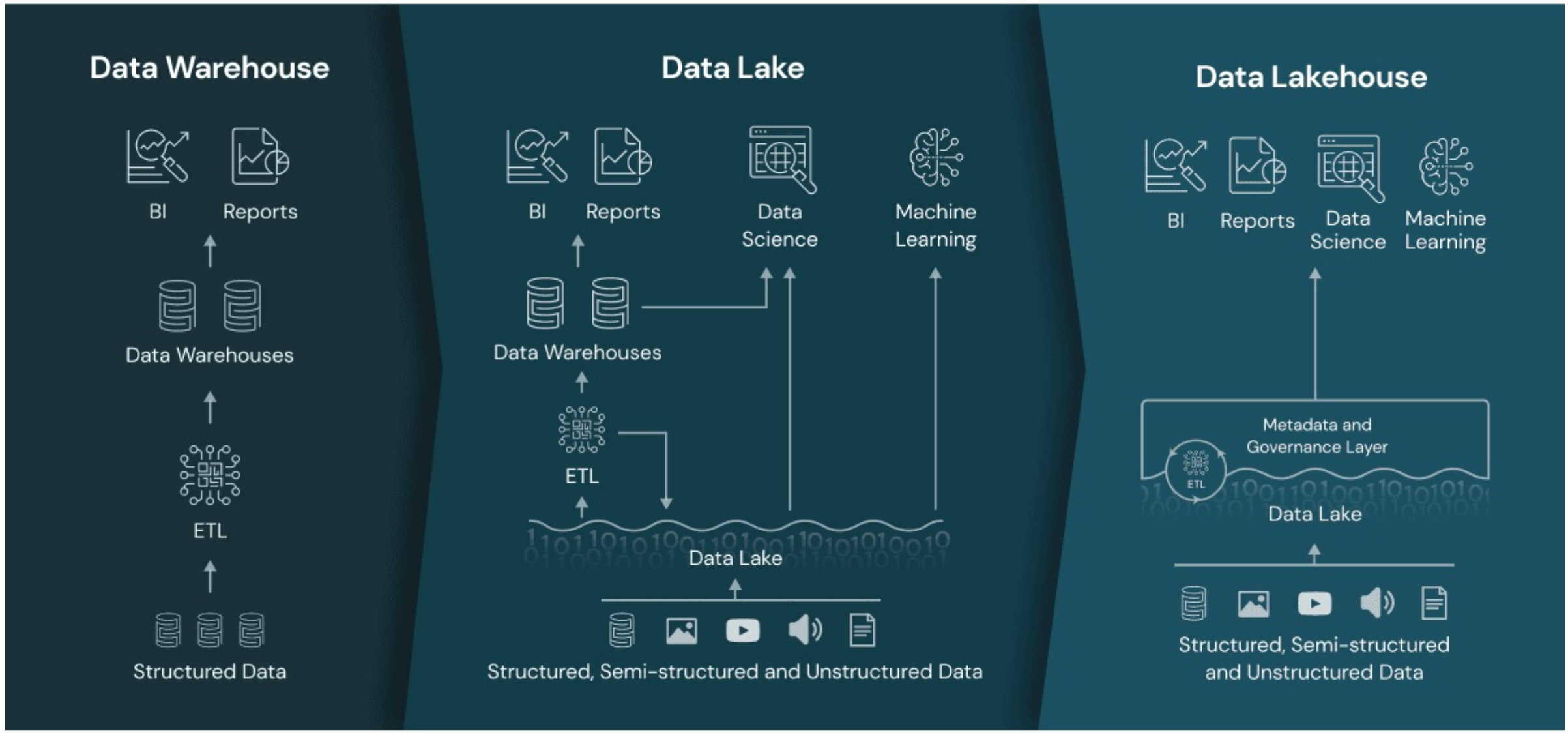
# Data Lake CON

- Lack of management
- Enforcing data reliability
- Uniform security model (data formats)

# Data Lakehouse




- Scalable data lakes
- Organized data managemer
- Organized ACID for:
  - Transactional data (DWH)
  - Improved analytics (DWH)
  - Improving ML, DL (DWH, Data Lake)
- Metadata layer for data lakes
- Improved Query engine
- Optimized acces for data science


Data Warehouse
BI
Reports
Data Warehouses
ETL
Structured Data
Data Lake
BI
Reports
Data Science
Machine Learning
Data Warehouses
ETL
Data Lake
Structured, Semi-structured and Unstructured Data
Data Lakehouse
BI
Reports
Data Science
Machine Learning
Metadata and Governance Layer
ETL
Data Lake
Structured, Semi-structured and Unstructured Data

# Data lakehouse PRO

- Reduced data reduncency
- Improved flexibility
- Cost effectiveness (combination of DWH and Data Lake)
- Improved data governance, data management, and security
- Support for variety of workloads (analytics, open data formats (API, parquet))
- Straightforward Machine Learnign

# Data lakehouse CON

- Relatively new (concept)
- Improved, yet immature technology
- Offerings (Databricks)

# *Recap*: DWH vs. DL vs. DLH

| | Data Warehouse | Data Lake | Data Lakehouse |
|---|---|---|---|
| **Storage Data Type** | Works well with structured data | Works well with semi-structured and unstructured data | Can handle structured, semi-structured, and unstructured data |
| **Purpose** | Optimal for data analytics and business intelligence (BI) use-cases | Suitable for machine learning (ML) and artificial intelligence (AI) workloads | Suitable for both data analytics and machine learning workloads |
| **Cost** | Storage is costly and time-consuming | Storage is cost-effective, fast, and flexible | Storage is cost-effective, fast, and flexible |
| **ACID Compliance** | Records data in an ACID-compliant manner to ensure the highest levels of integrity | Non-ACID compliance: updates and deletes are complex operations | ACID-compliant to ensure consistency as multiple parties concurrently read or write data |

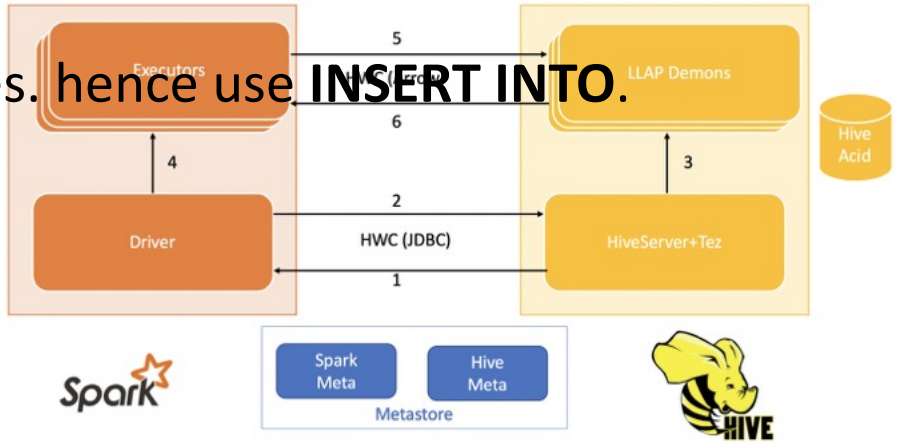
# Achieving ACID in Spark

- ACID – Atomicity, Consistency, Isolation, Durability
- Providing data in data lake to be consistent and durable
- Spark Hive tables
- Delta lake (Databricks)
- HUDI (Hadoop Upsert Delete and Incremental) by Uber/Apache

| | | | |
|---|---|---|---|
| **ACID Compliance** | Records data in an ACID-compliant manner to ensure the highest levels of integrity | Non-ACID compliance: updates and deletes are complex operations | ACID-compliant to ensure consistency as multiple parties concurrently read or write data |

# ACID limitations on Hive

- Hive tables should be created with **TRANSACTIONAL** table property.
- Hive supports ACID transactions on tables that store **ORC** file format.
- Enable ACID support by setting transaction manager to **DbTxnManager**
- Annoyances:
  - Transaction tables cannot be accessed from the non-ACID Transaction Manager (**DummyTxnManager**) session.
  - **External tables** cannot be created to support ACID since the changes on external tables are beyond Hive control.
  - **LOAD** is not supported on ACID transactional Tables. hence use **INSERT INTO**.



# ACID on Data Lake (files)

- ETL frameworks / ELT pipelines
- Assuring zones (architecture)
- Logging, storing, updating
- Delta Lake (Databricks)

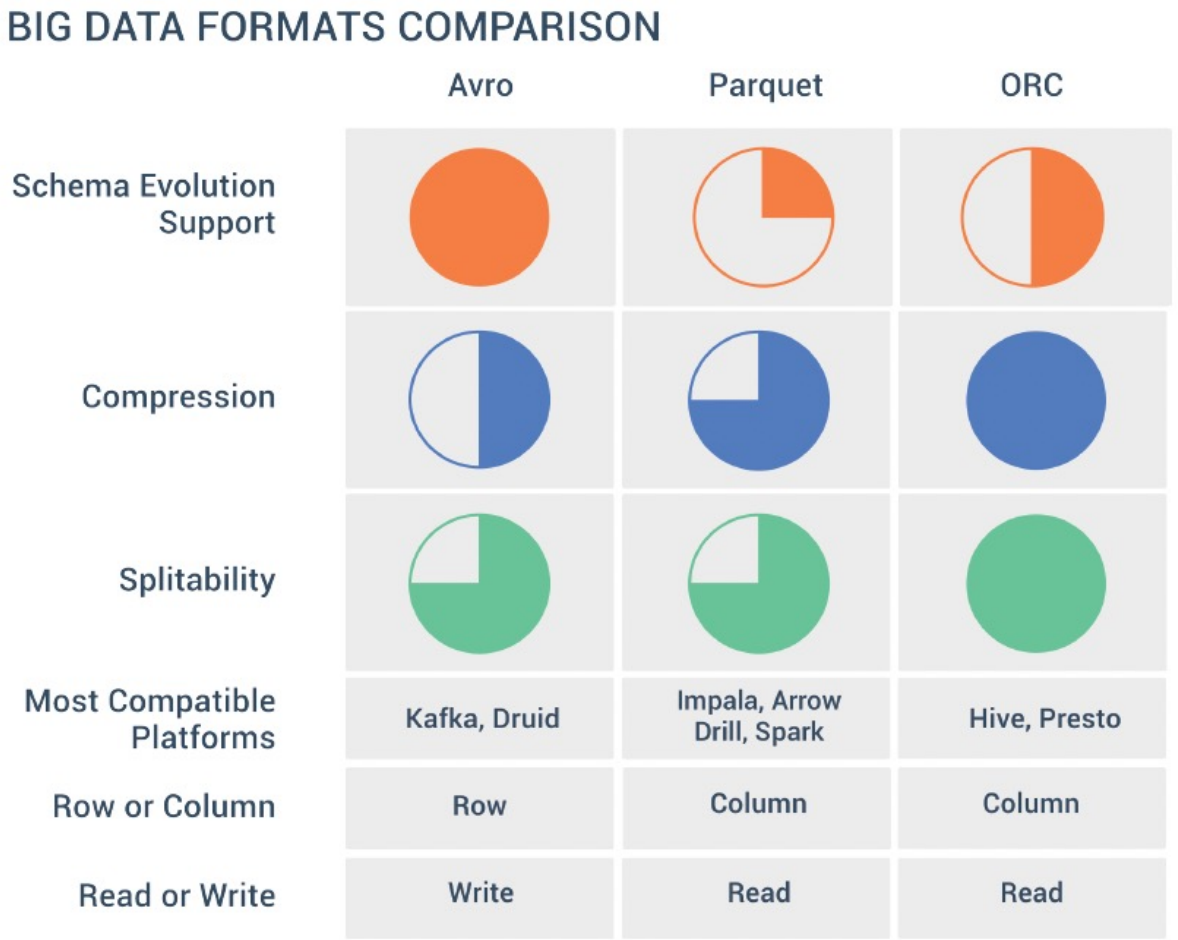
# File Types in supported in Spark

- .CSV – delimited text file using comma, semicolon to separate values.
- .TXT – delimited text file.
- .JSON – text file that stores simple data structures and objects. Standard for data interchange
- .Parquet – column storage file format used by hadoop systes, such as Pig, Spark, and Hive. It has binary presentation, is cross platform.
- .AVRO – open source data serialization system by Apache Hadoop. stores serialized data in binary format and schema in JSON format.
- .ORC – open source columnar data storage format. Similar to Parquet. Is cross platform
- … and others

ORC – created by HortonWorks (feb 2013)

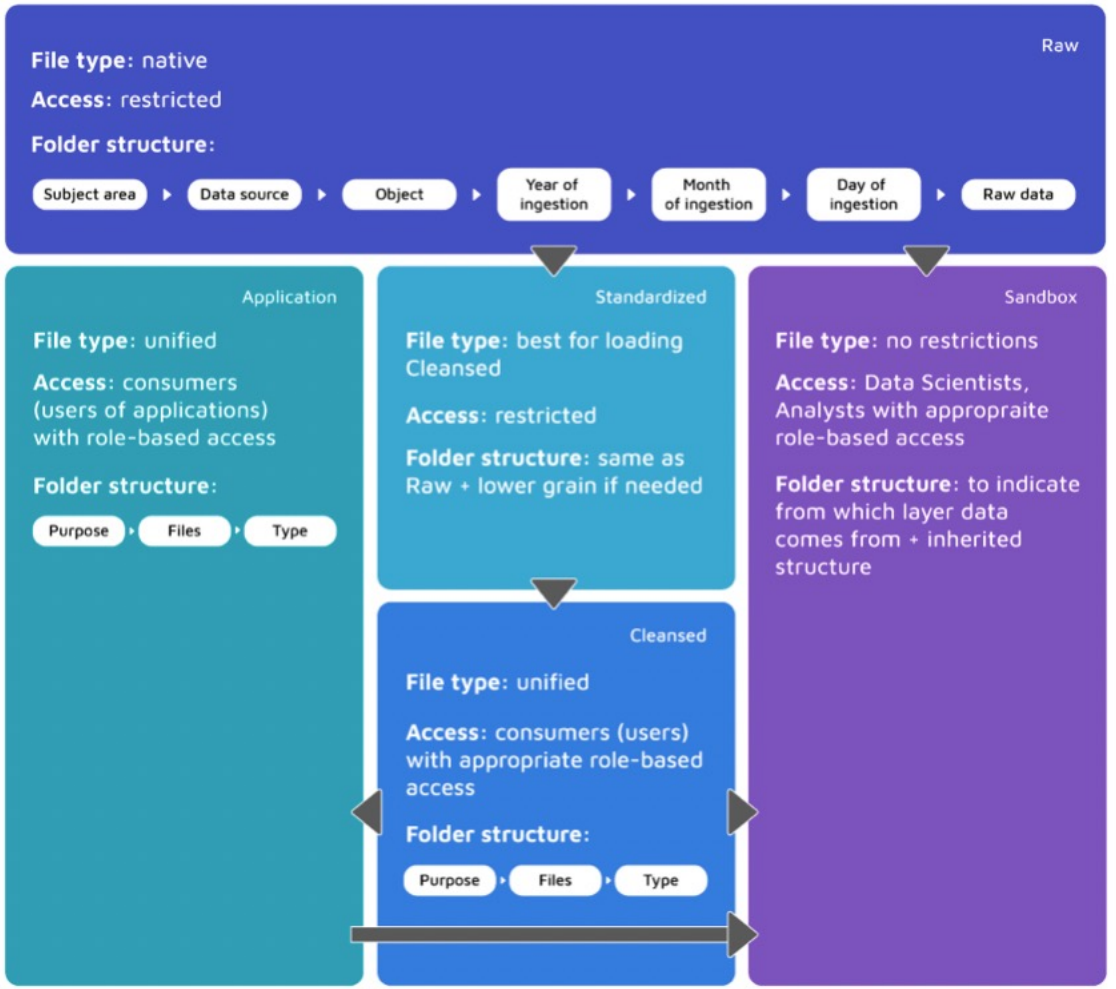
Apache Parquet – Created by Cloudera and twitter (may 2013)

# Avro vs. Parquet vs. ORC



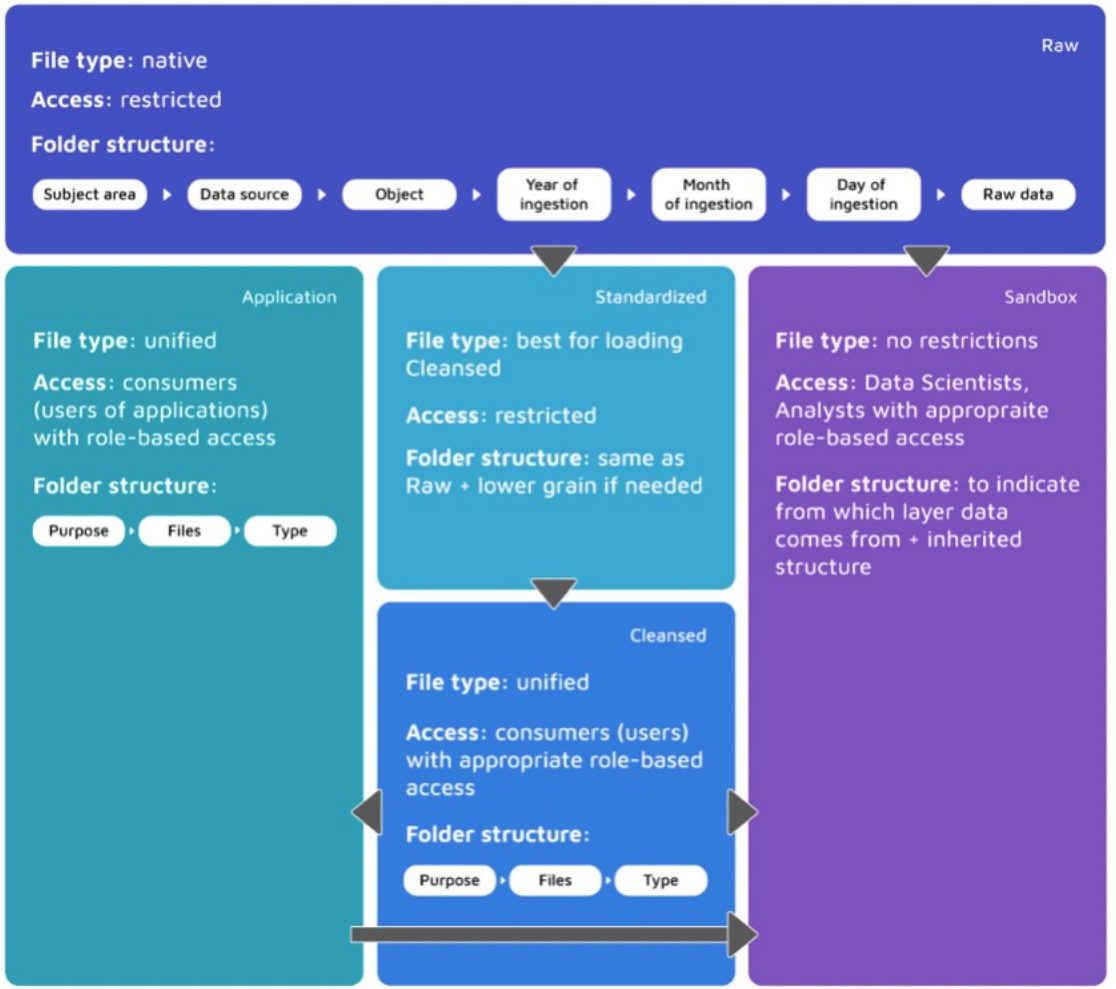
# Organizing data Lake layers

- Data Lake is a “single” repository
- Separating them into layers
- Architecture of layers / zones
- Each layer represents additional data manipulation
- Layers:
  - Raw
  - Standardized
  - Cleaned
  - Application
  - Sandbox
- Alternatives:
  - Raw > Curated > Enriched
    - Sandbox, Machine Learning
  - Ingest > Curated > Conformed > Production
    - Sandbox, Personal, Reporting, Machine Learning
  - Bronze > Silver > Gold
    - Sanbox, Work, Machine Learning



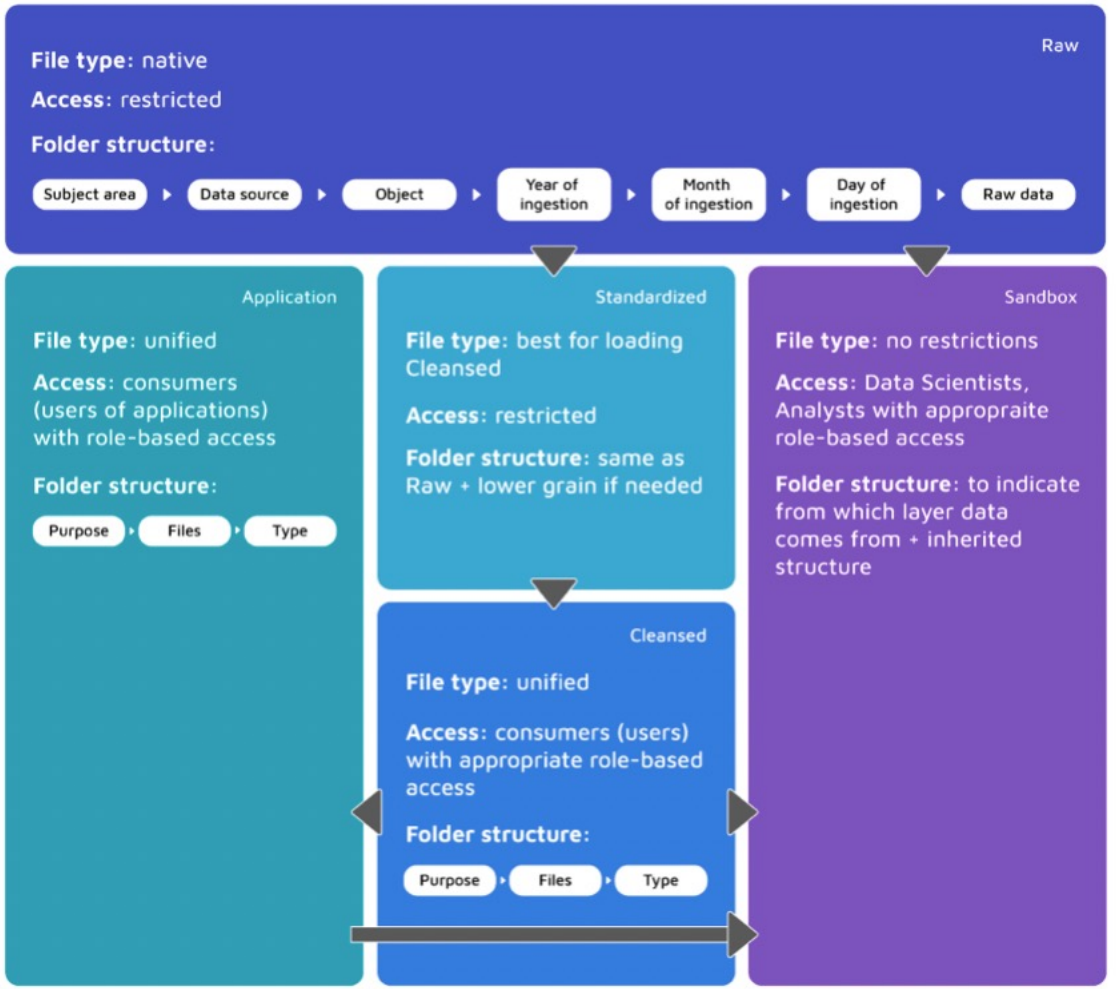
# Organizing data Lake **folders**

- Folder structure within Layer
- Folder by purpose:
  - Area
  - Source
  - Object
  - >>**Key**<<
  - Raw data
- Folders by structure
  - Purpose
  - Files
  - Type

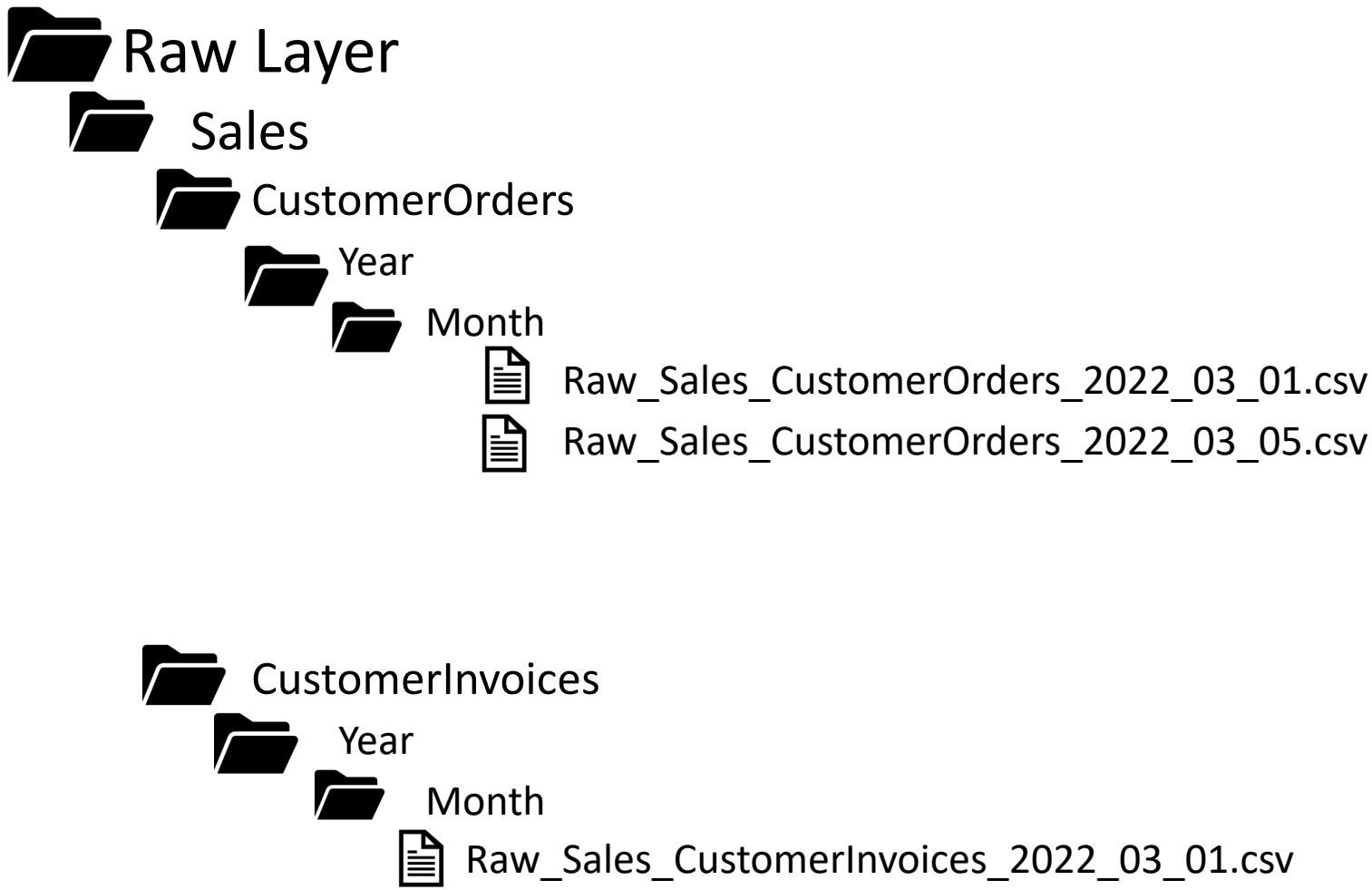


# Organizing data Lake - **Keys**

- Keys are considered as “index”
- Keys help data organized
- Implement keys based on search
- Folder structure:
  - Year=2022
  - Year=2022&Month=03
  - GeoAreaID=100&Year=2022



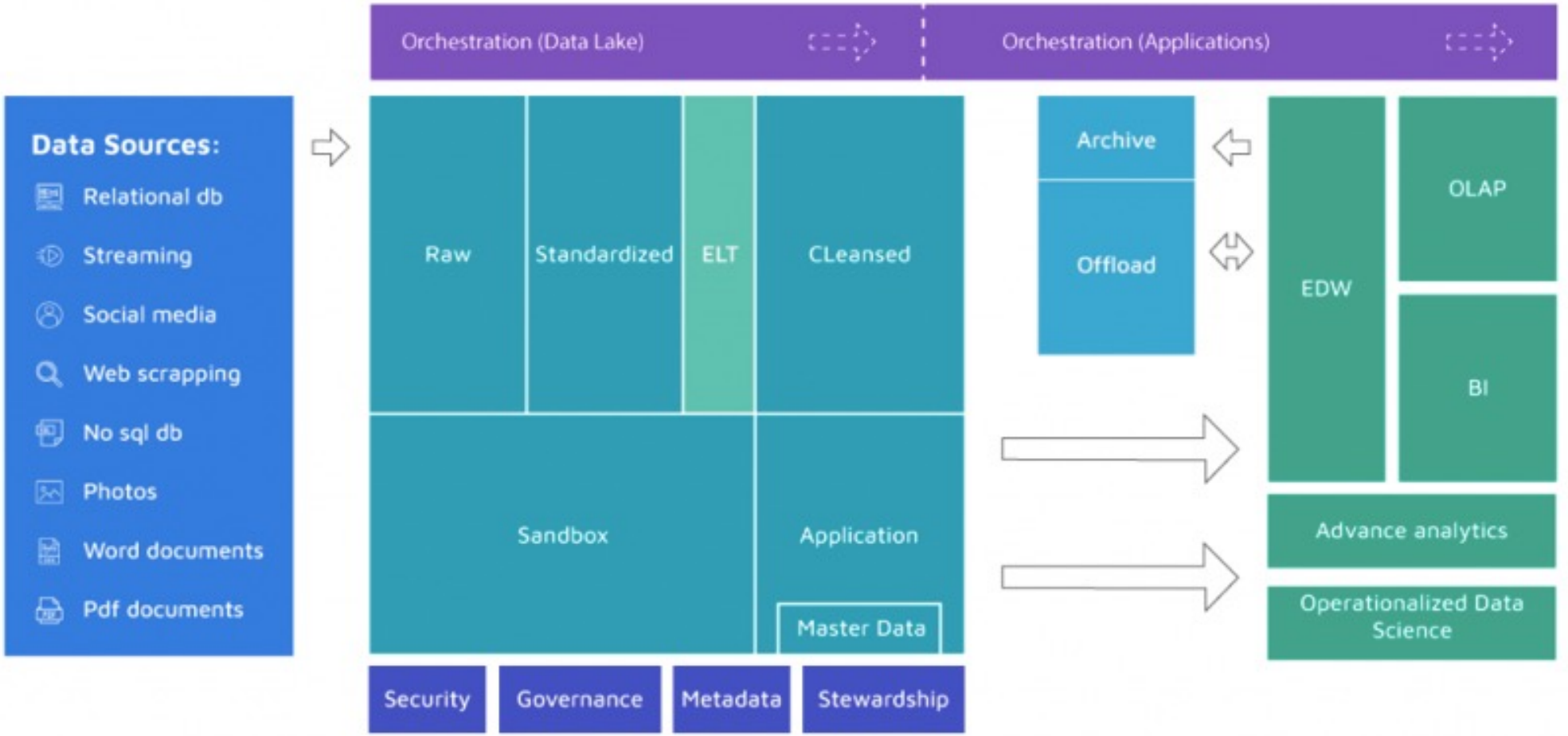
# Files organization



# Structure, type, access, granulation

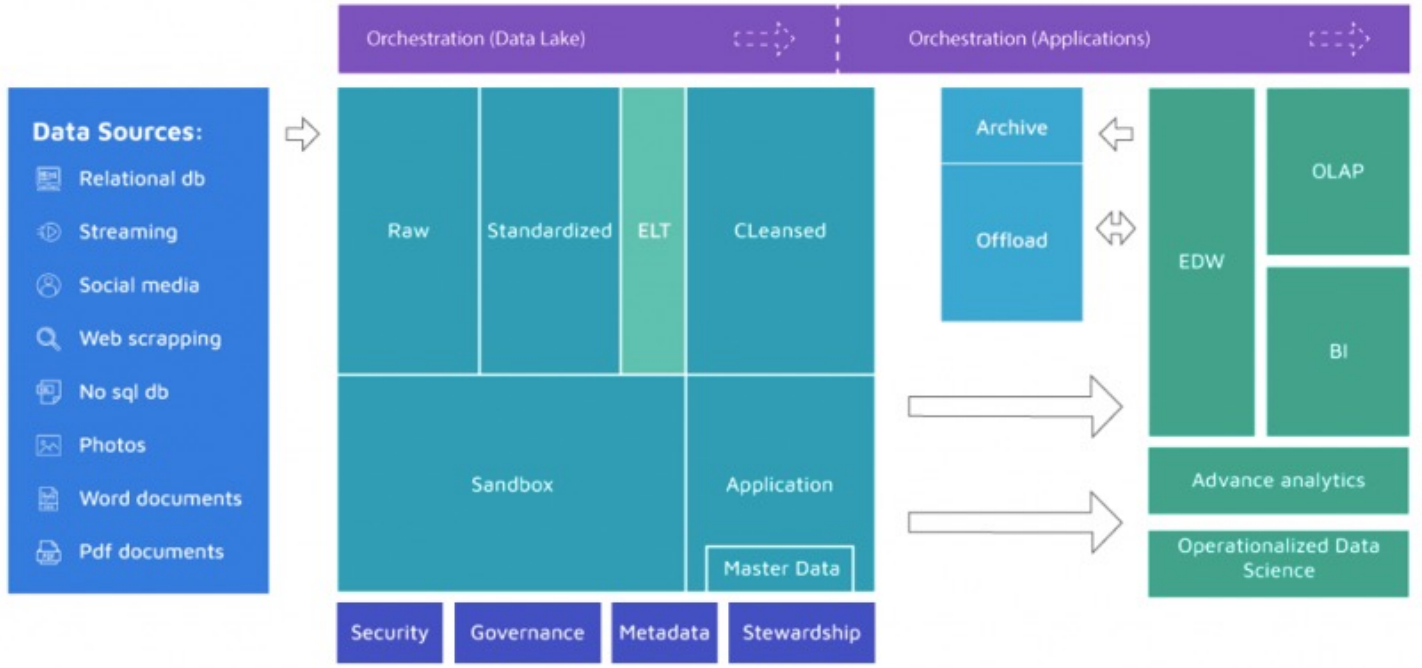
| Folder structure | Data Types | Access control | Granulation (e.g.: strem data) |
| --- | --- | --- | --- |
| Raw layer | Original format | Data Engineers | As is |
| Standardized layer | Files aggregated, still original format | Data Engineers, Data Scientists | Aggregate to higher level (e.g. from sec -> min) |
| Cleaned layer | Sparsed (faster) format | Data Engineers, Data Scientists, Data Analysts | Another abstration layer above (e.g.: from sec -> hour) |
| Application layer | Custom to application | Data Engineers, Business people, Data Analysts | Higher granulation (e.g. Day, month, custom) |

# Data Lake Architecture



# Data Lake Components

- Security
- Governance
- Metadata
- Stewardship
- Master data
- Archiving
- Orchestration



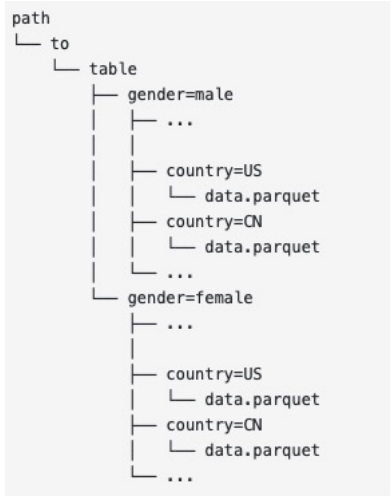
# Architecture designs

Architecture designs can be influenced by:

- Data load patterns (hot/cold, real-time, stream, increment/full)
- Time partitioning
- Data type (relational, time series, blob)
- Subject areas and sources
- Security and roles
- Retention policies (temporary, permanent, time-fixed)
- Business impact (Core, Critical, High, Medium, Low)
- SLA and regional regulations
- GDPR and confidential classifications (public use, internal use, financial, Government, supplier/partner confidential)

# Parquet:  Table partitioning

- Parqueta is a columnar format
- Spark SQL API provides support for *reading* and *writing* Parquet file
- Parquet preserves the schema of the original data.
- When readin,  all columns are automatically converted to be nullable for compatibility reasons.
- Partition discovery is optimization support



# Working with Parquet files (Spark + Python)

- Appending
- Overwriting
- Compling
- Caching
- Reading / writing

Continue with Notebooks

# Moving data around using fs / dbutils and DBFS API 2.0

- Python:

```
import os
os.listdir('/dbfs/tmp')
dbutils.fs.ls("/mnt/mymount")
df = spark.read.text("dbfs:/mymount/my_file.txt")
```

- Bash

```
%fs ls /tmp/
%fs mkdirs /tmp/my_cloud_dir
%fs cp /tmp/test_dbfs.txt /tmp/file_b.txt
```

# Moving data around using CLI / PS  / Spark SQL CLI

- CLI:

mkdirs /tmp/my_cloud_dir

cp /tmp/test_dbfs.txt /tmp/file_b.txt

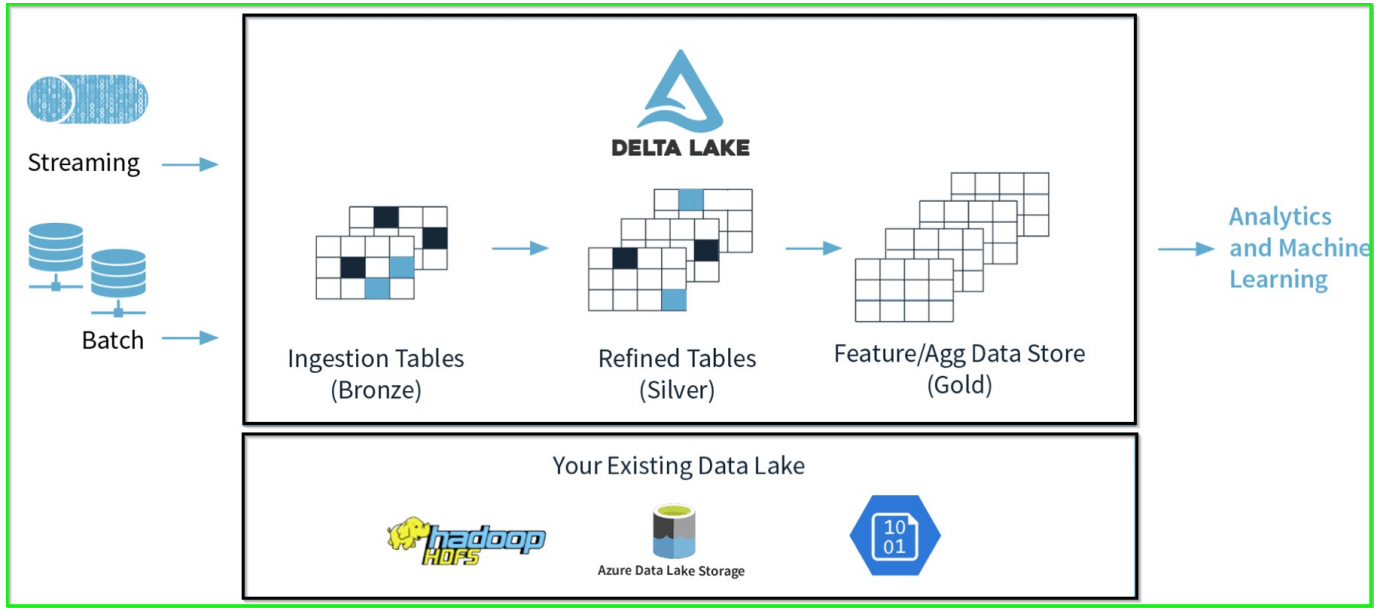
- PS
- Spark SQL CLI
  - ./bin/spark-sql

# Exception handling and exception coding

# Logging and working with logs

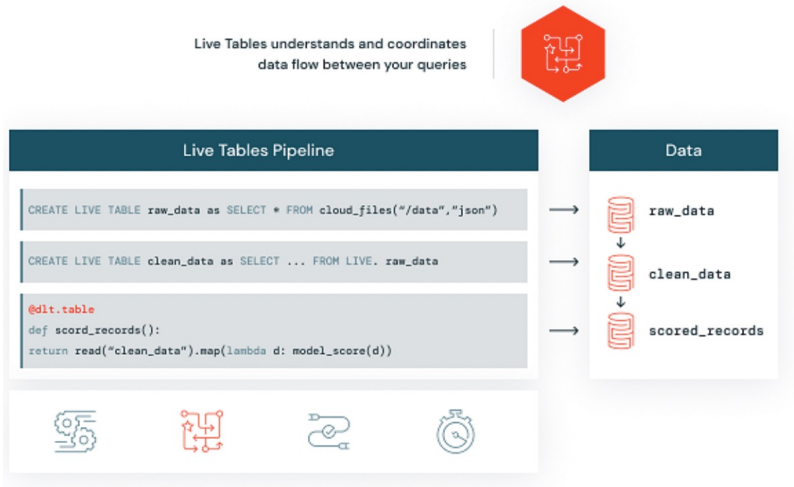
# Delta lake

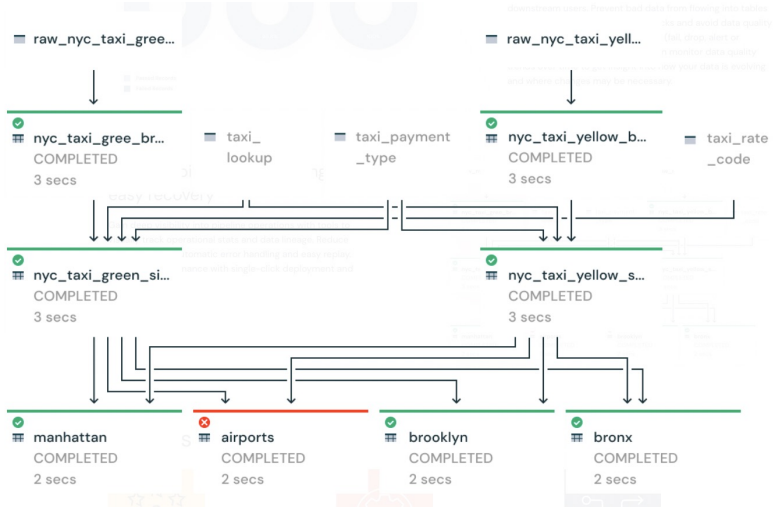
- Open source data storage layer that unifies ACID transactions, scalable metadata management and batch and/or streaming data processing. It also offers time travel and CDC, SCD, streaming upsers operations
- Sits on top of your existing Data Lake in tandem with Apache Spark APIs.



# Delta Live tables (Databricks)

- DLT helps data engineers to build and manage reliable data pipelines for better data quality.
- DLT simplifies ETL development and management with declarative pipeline development, automatic data testing and improved visibility for monitoring and troubleshooting



# Module 5

Spark Streaming

# Spark Streaming

- Spark Streaming or Structured Streaming is a **scalable** and **fault-tolerant**, end-to-end stream processing engine.
- Extention of the core Spark API.
- Data can be processes from many sources (Kafka, Fume, Kinesis, Twitter, TCP Sockets,…)
- Data can be pushed out to filesystem, databases, dashboards

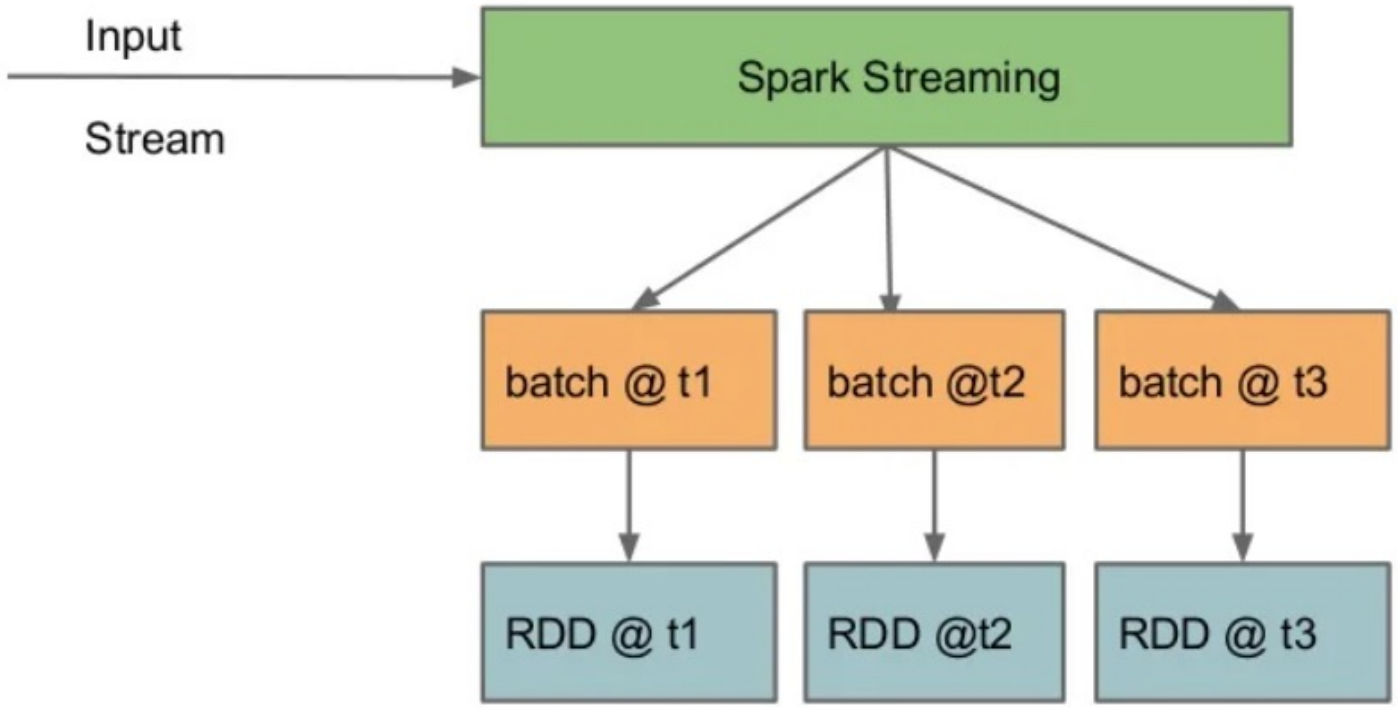
# How it works:

- Spark Streaming receives **live input** data streams and divides the data into batches
- Batches are then processed by the **Spark engine** to generate the final stream of results in batches.

# DStreams

- Discretized Stream or Dstream is represented as contionus stream of data.
- It can be Kafka,  Kinesis or high-level operations
- Dstream is internally represented as a sequence of RDDs
- Supports Java, Scala, Python, also R. (both Py and R with some limitations)

# DStreams

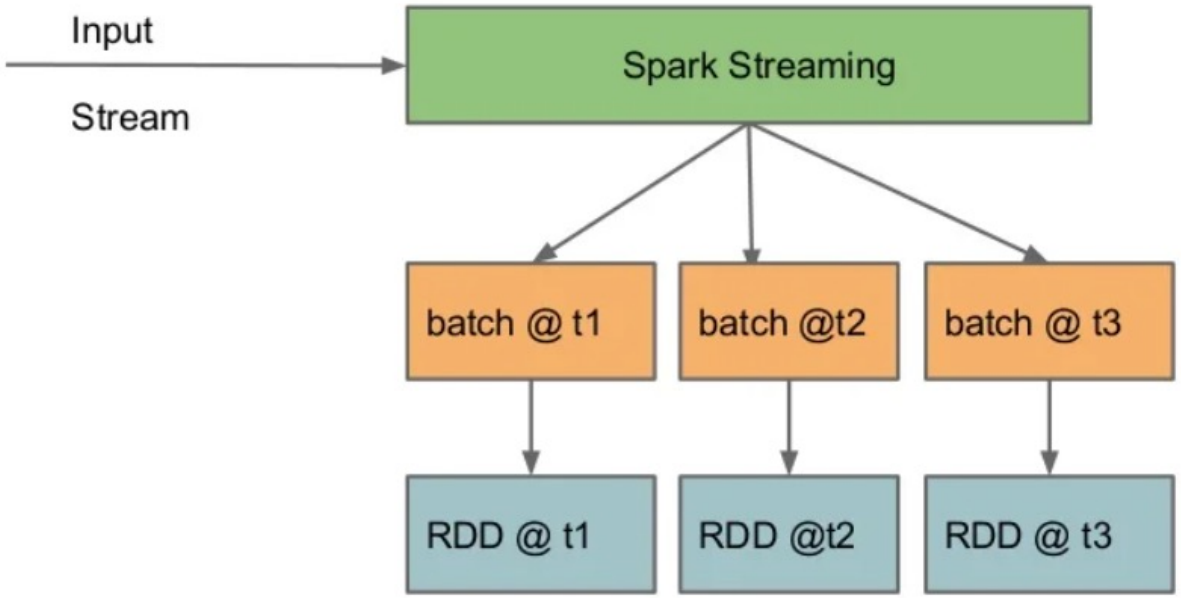

Input
Stream
Spark Streaming
batch @ t1
batch @t2
batch @ t3
RDD @ t1
RDD @t2
RDD @ t3

# Dstreams to RDD

- Each batch of Dstream is represented as RDD underneath
- RDD files are replicated in cluster for fault tolerance
- Every Dstream operation results in RDD Transformation
- API access is available
- Dstream can be stream and batch processing

# Dstream transformation

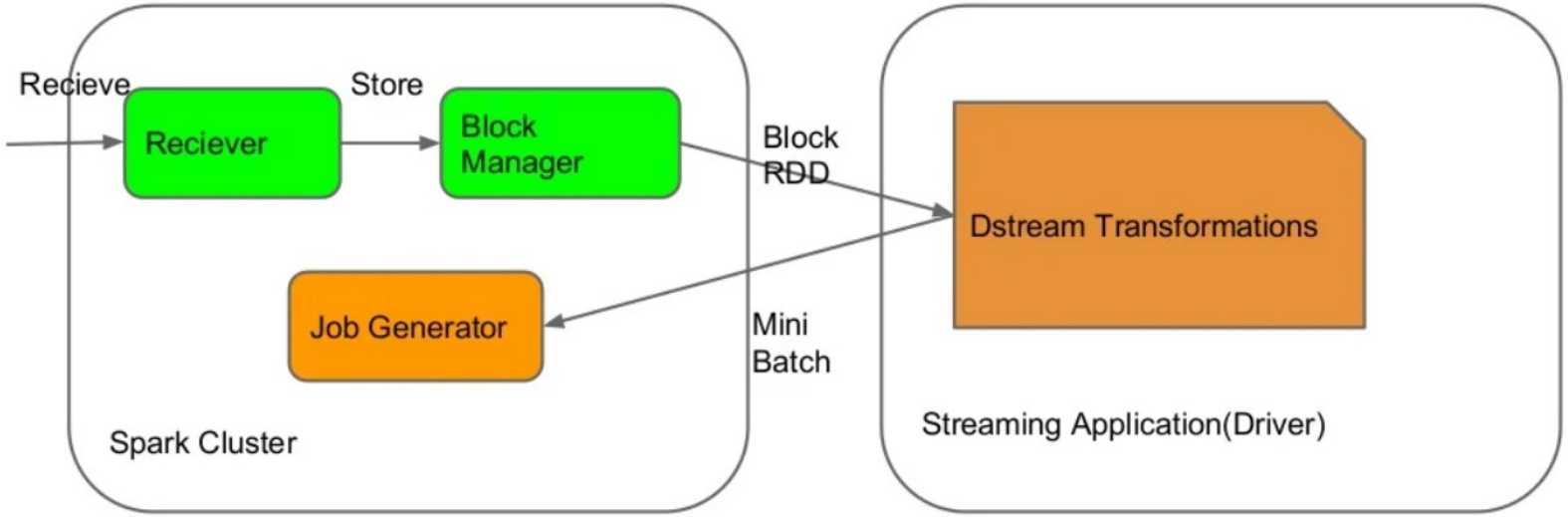
```
val ssc = new
StreamingContext(args(0),
"wordcount", Seconds(5))

val lines = ssc.
socketTextStream
("localhost",50050)

val words = lines.flatMap(_.
split(" "))
```



# Socket example with file stream

- File stream allows to track new files in a given directory (on HDFS)
- When new file appears, Spark Stream will pick it up
- It will pick only new files, modification of existing files will not be considered
- Tracks using file

creation time



# Auto Loader

- Incrementally processes new data files that arrive in storage
- Structured streaming source is called **cloudFiles**
- Any new files that arrive in storage are automatically processed
- It uses COPY INTO command

| Cloud Storage | Directory Listing | File Notifications |
|---|---|---|
| AWS S3 | All versions | All versions |
| ADLS Gen2 | All versions | All versions |
| GCS | All versions | Databricks Runtime 9.1 and above |
| Azure Blob Storage | All versions | All versions |
| ADLS Gen1 | Databricks Runtime 7.3 and above | Unsupported |
| DBFS | All versions | For mount points only. |

# Benefits of Auto Loader over Apache Spark

- Apache Spark can read files incrementally using ***spark.readStream.format(fileFormat).load(directory).***
- Auto Loader provides benefits:
  - Scalable; can discover billion of files efficiently and backfills are performed asynchronously to avoid wasting compute resources
  - Performance; discovering files with AL scales with the number of files that are being ingested instead of number of directories that the files may land in
  - Schema inference; automatically detects schema drifts and nofity you
  - Cost; native cloud API to get to existing storage. File notification mode help reduce cloud costs by avoiding directory listing altogether

# Module 6

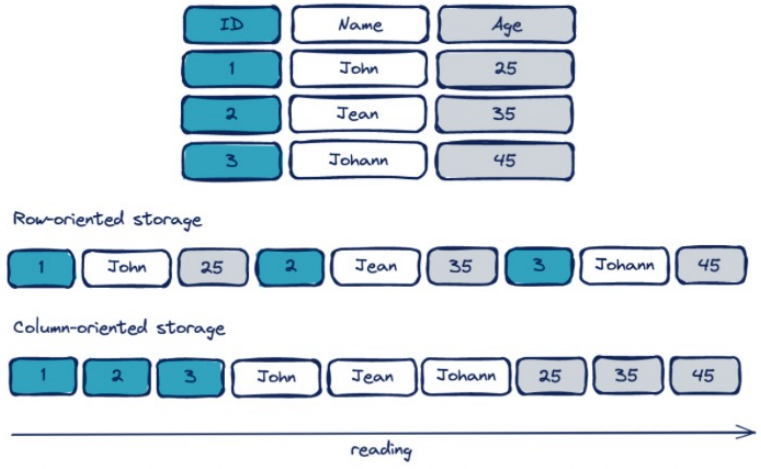
Comparison Scala Python, Full scale demo, Recommendations, Troubleshooting

# Optimization recommendations for Spark jobs

## 1 Use Apache Parquet file format

Achieves great performance when performing queries (*transformations*) on a subset of columns and on a large *dataframe*. This is because it **loads only the data associated with the required columns into memory**.

Parquet also improves reading / writing of the binary files in data lake, because it has compressed schema and encoding for each column.

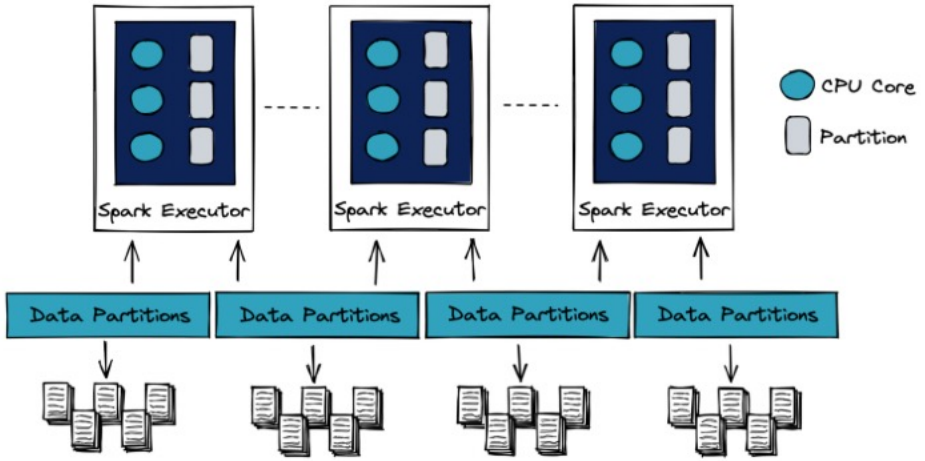


# Optimization recommendations for Spark jobs

## 2 Maximize parallelism in Spark

The more we facilitate the division into tasks, faster they will be completed and therefore optimizing means, reading and processing as much data as possible in parallel. So, splitting a dataset **into several partitions**.

Set paramter ***spark.sql.files.maxPartitionBytes*** in accordance with your HDFS I/O system. Default is 128MB but it can be set to smaller value



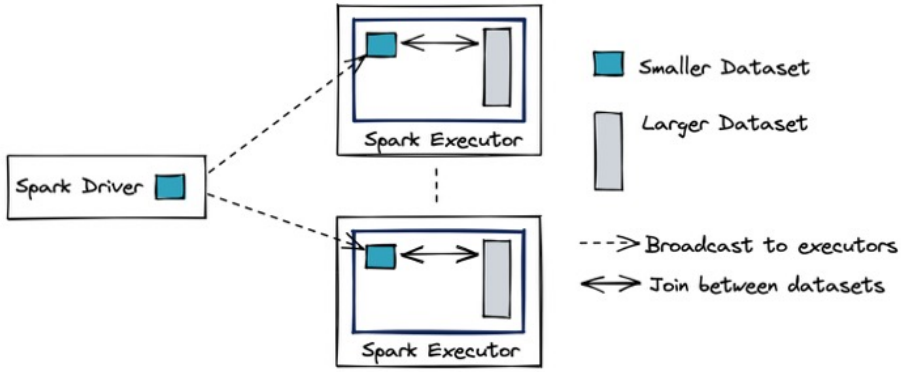
# Optimization recommendations for Spark jobs

## 3 Use Broadcast Hash Join (BHJ)

Joining dataset is a common operation. In distributed context, network latency can cause slow query processing. Spark SQL offers many, BHJ is suitable when one of the merged dataframe is small to be duplicated in memory on all executing nodes (broadcast operation)

By duplicating smaller dataframes, the join no longer requires significant data exchange in the cluster, apart from the broadcast of this table.

Set parameter: ***spark.sql.autoBroadcastHashJoin***



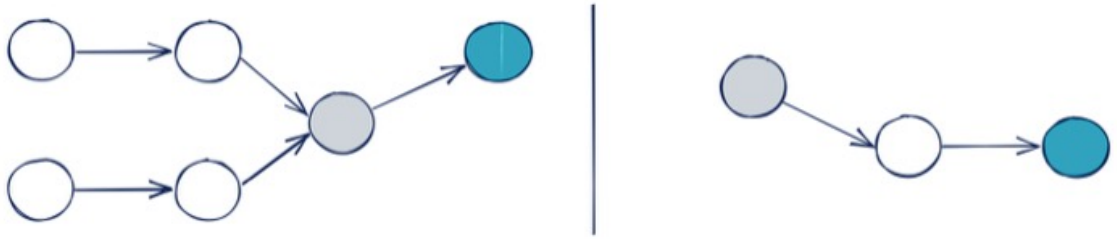
# Optimization recommendations for Spark jobs

## 4 Cache intermediate results!!!

Caching is performed when spark action is run.

Spark uses lazy evaluation and DAG to describe the job how RDD concept is set. This offers possibility to quickly recalculate the step before an action is necessary! To take full advantage of this feature, is it wise to **store expensive intermediate results,** when several operations follow this operation in downstream of DAG.

When caching, the result is stored
On disk and/or memory.



# Optimization recommendations for Spark jobs

## 5 Beware of UDF in non-native Scala

Spark API, especially Python, R and SQL can have problems converting user defined functions to native Java.

Check for user create functions and use functions on Spark core to achieve better performance.

# Optimization recommendations for Spark jobs

## 6 Manage the memory of the executor nodes

By default spark.memory.fraction is set to 0.6; meaning 60% of the memory is allocated for execution and 40% for the storage, once the reserved memory is removed. Beware of out of memory (OOM) errors.

this can be modifies using two parameters:

- ***spark.executor.memory***
- ***spark.memory.fraction***

# Optimization recommendations for Spark jobs

## 7 Easy on shuffle operations

When a wide transformation is created (e.g.: groupBy(), join()), where will be shuffle partition created that can cause network traffic and read/write disk resources.

Parameter: ***spark.sql.shuffle.partitions*** with default 200 partition which can be too high for some operations. Adjust this parameter to size of data (smaller datasets, smaller number) and to number of CPU cores per cluster (partitions <= number of CPU cores).  SSD disks improve this issue significantly.

# Documentation, learning

- ApacheCon conference (ApacheCon | Home)
  - Apache Flink conference
  - Airflow summit
- Spark + AI Summit
- Data + AI Summit (Databricks conference)
- Books:
  - O'Reilly
  - Packt Publishing
  - HB publishing
- Community:
  - Stackoverflow
  - Meetup: Spark, Hadoop, IBM

# Tools and providers for Apache Spark

- Complete apache Family
- Cloud providers: Azure, AWS, GC, Oracle
- ETL, ELT tools