DATA LAKES with SPARK

BIG DATA: You need a distributed system of several servers to be able to work with the data.

Numbers everyone should know

FASTEST

CPU. "Brain of computer" Operation: 0.4ns

Memory/RAM. "ephemeral storage". Memory Reference: 100ns

Storage: SSD/magnetic Disk. Random Read: 16µs

SLOWEST

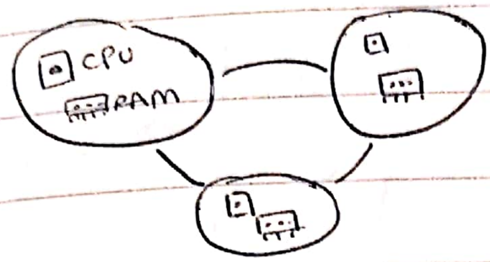
Network: Acces to outside. Round trip for data from EU to US: 150ms

2.5 GHz CPU: 2.5 Billion operations/second.

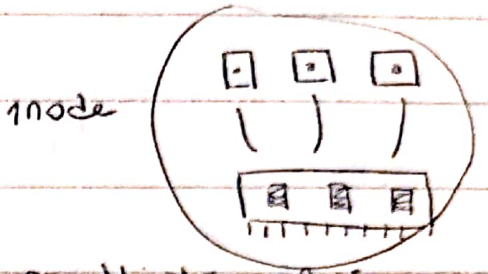
if 1 operation = 8 bytes → CPU can process 20 Billion Bytes/sec.

Memory is 2x faster than storage
But it's more expensive.

Distributed Computing vs. Parallel Computing



Several Nodes accross Network.
Each cpu has itsown memory



multiple CPUs share a single memory.

→ 200x Faster than RAM
→ 15x " than SSD
→ 20x " thannetwork

Hadoop Framework

HDFS - Data Storage

MapReduce - Data Processing

YARN - Resources Manager

Hadoop Commons - libraries + Utilities

+ other Tools on Top.

Pig - SQL for MR

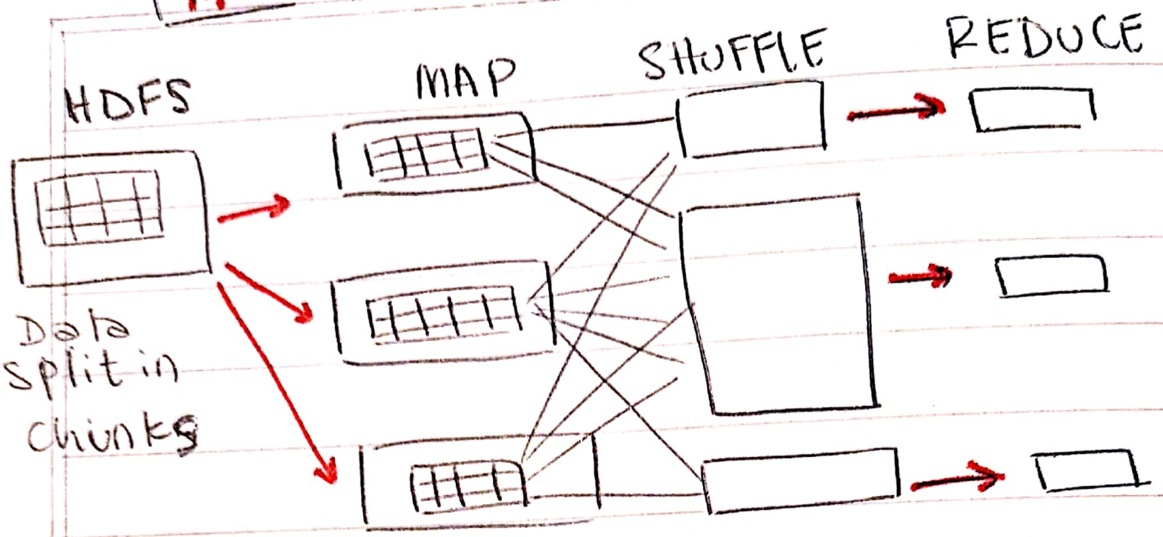
Hive - SQL for MR

SPARK

Storm - StreamigData

Flink - " "

MR



A B B
A C B B
C D A
D E

→

key, values

(A, 1)
(B, 1)
(B, 1)
(A, 1)
(C, 1)
...

→

keys, value.

(A, 1)
(A, 1)
(A, 1)

(B, 1)
(B, 1)
(B, 1)
(B, 1)
...

→

(A, 3)
(B, 4)
(C, 2)
(D, 2)
(E, 1)

DAG: directed Acyclical Graph

Spark builds a step by step directions of what functions + data it will need

→ Once it builds the DAG from the code it checks if there is anything it can procastinate

→ LAZY EVAWATION

IMMUTABLE

Spark does not change of mutate data

MAPS

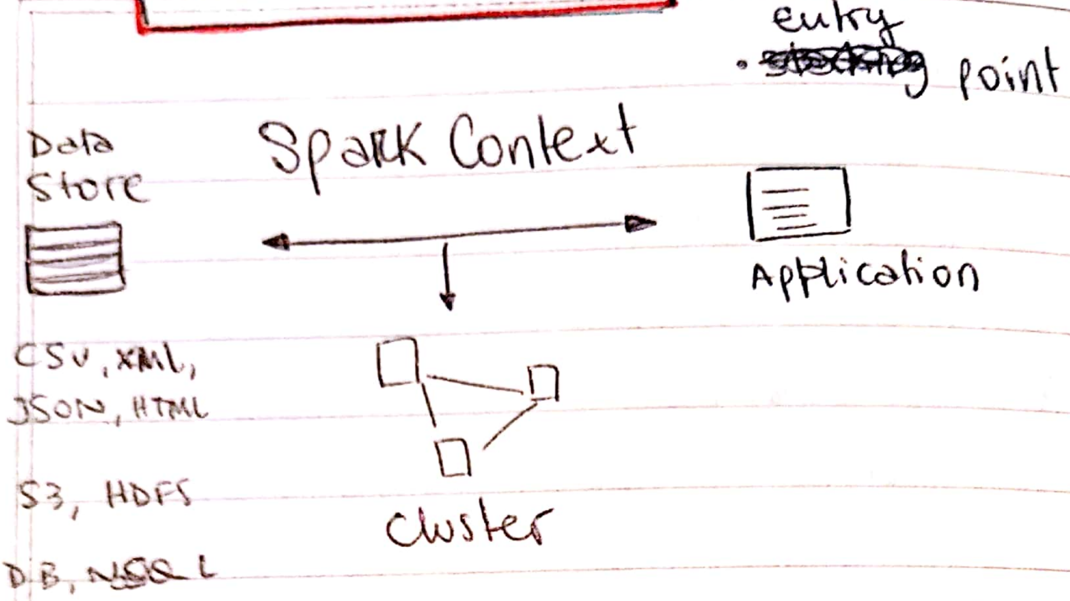
Makes a copy of the original input data and applies a function to it.

example

```
import pyspark
sc= pyspark.sparkContext(appName=" -- ")
distrib_songs= sc.paalellize(songs)
distrib_songs.map(lambda x : x.lower())

distrib_songs.collect()
```

SPARK PROGRAMS



```
   from pyspark import SparkContext, SConf
config = SparkConfig(). set AppName(" ")
      . set master(" IP |local")
SC = SparkContext(conf= conf)
```

TO read DF :

SC SQL equiv.

```
from pyspark sql import SparkSession
spark = SparkSession . builder /
          . appName (" "). config(" ")./
          . get Or Create ()
```

Data wrangling with DF

Read JSON into DF:

```
user_log = spark.read.json(path)
user_log.printSchema()
   "     .describe()
         .show(n=1)
         .take(5)
```

SAVE into csv:

```
 "   .write.save(out-path, format="csv",
                header=True)
user_2 = spark.read.csv(path, header=True)
        .describe('column').show()
        .count()              .dropDuplicates()
        .select()             .sort("column")
        .where(column == value)
        .groupby(    )
```

draw a Plot of your df

```
pd = df.toPandas()
plt.scatter(pd[x], pd[y])
plt.xlim(-1, 24);
 func = udf(lambda x: ..., IntegerType())
df.withColumn("colname", func(col))
```

DATA LAKES

WHY

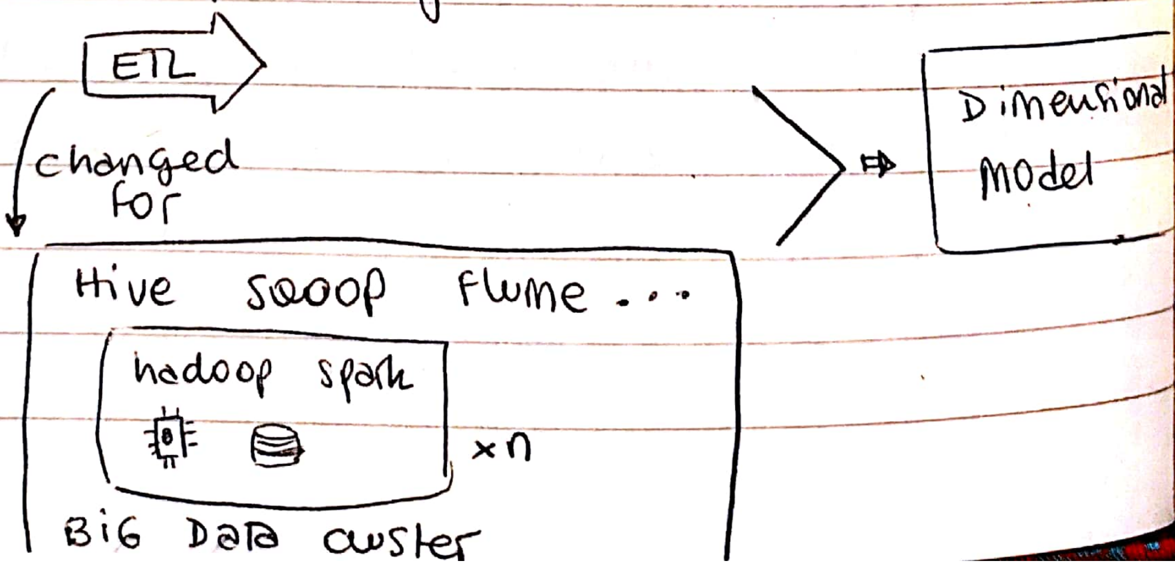
- Abundance of unstruct. data
- Unprecedented data volumes
- Rise of Big Data techs.
- Data analysis - new types (ex. ML, NLP,)
- Emergence of new roles (ex. Data Scientist)

*Evolved from Data warehouse to cope with:

- Variety of data formats and structuring
- Support to agile + ad-hoc data exploration.
- Wider data transformation needed

Big Data tech:

(i) **ETL Offloading** : instead of ETL grid or staging area → same Hw for storage and processing.



(11) schema - on - read → As easy to work with files as with a database, without having to → create a DB
→ insert data into DB.

```
df = spark.read.csv(path,
        inferSchema = "True", header=True,
        sep= ';').
```

To specify a schema:

```
schema = StructType([
        StructField("col-name", IntegerType()),
        ... ])
df = spark.read.csv(path,
        schema= schema, sep=';',
        mode = 'DROPMALFORMED')
```

```
df.createOrReplaceTempView("Table")
spark.sql ("select * from table").show(5)
```

(III) Un Structured support

- Spark can read /write files in
  - Text based formats → Text, JSON, CSV..
  - Binary formats → Avro, Parquet ..
  - Compressed formats → Gzip, Snappy

- Read / write from <> FS:
  - Local file system
  - HDFS
  - S3

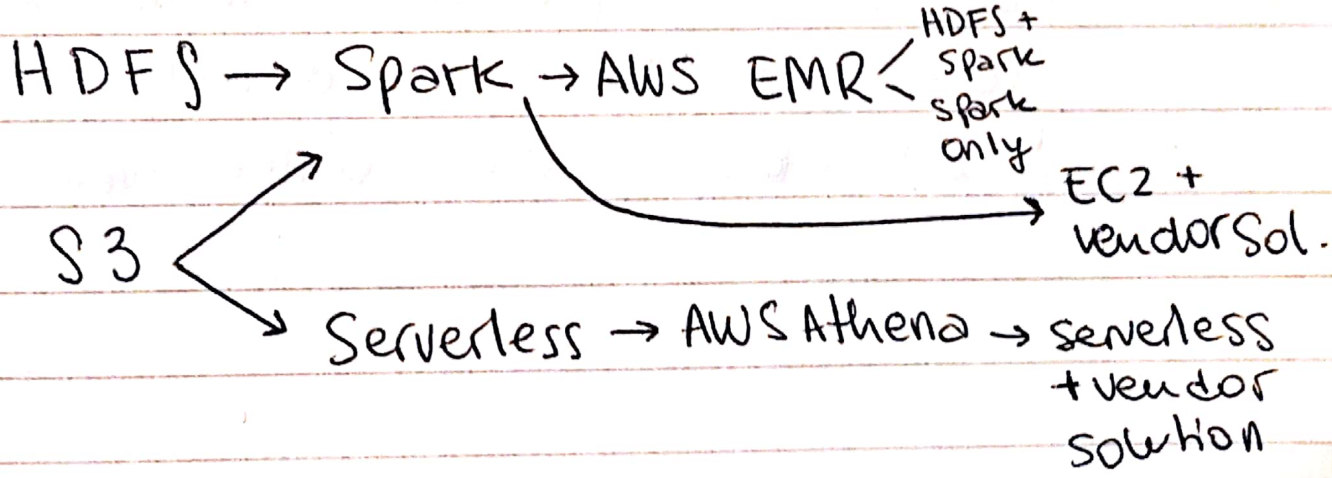
- & Databases:
  - SQL through JDBC
  - NOSQL → MongoDB, Cassandra, Neo4j

    spark . read . format ("jdbc"). ...

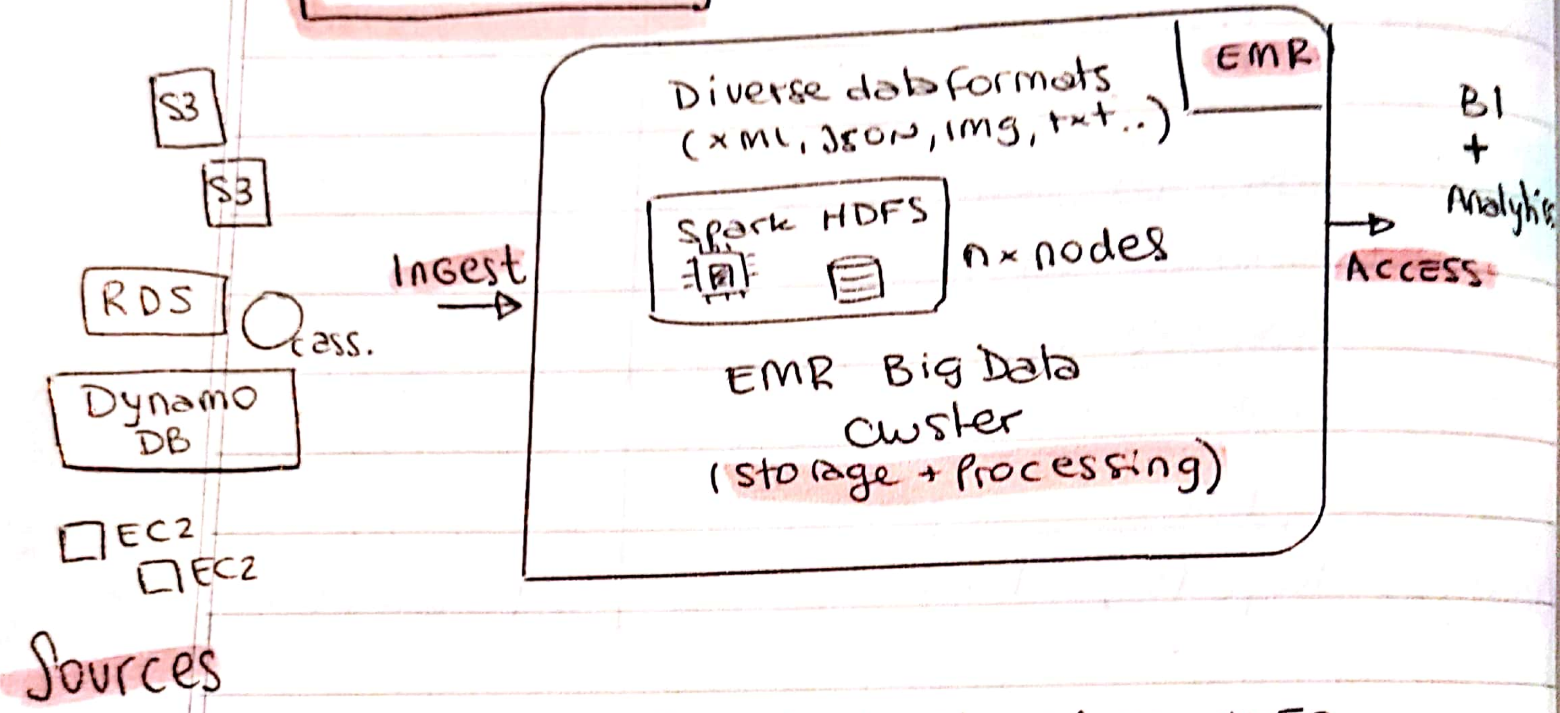
	Data Warehouse	Data Lake
Data Format	Tabular	All formats
Ingestion	ETL	ELT
Data model	Star/snowflake w/ conformed dimensions or DM or OLAP cubes	star, snow, OLAP + ad-hoc reps.
Schema	schema-on-wite	schema-on-read
Tech	Expensive MPP, disks, netws.	Commodity Hw.
Data Q.	High, consistent, clean	mixed, raw, some transfo
users	BA	DS, BA, ML engineers
Analysis	Reports, BI viz.	ML, Graphs, data exploration

DATA LAKE ON AWS

Storage | Processing | AWS-managed solutions | Vendor managed



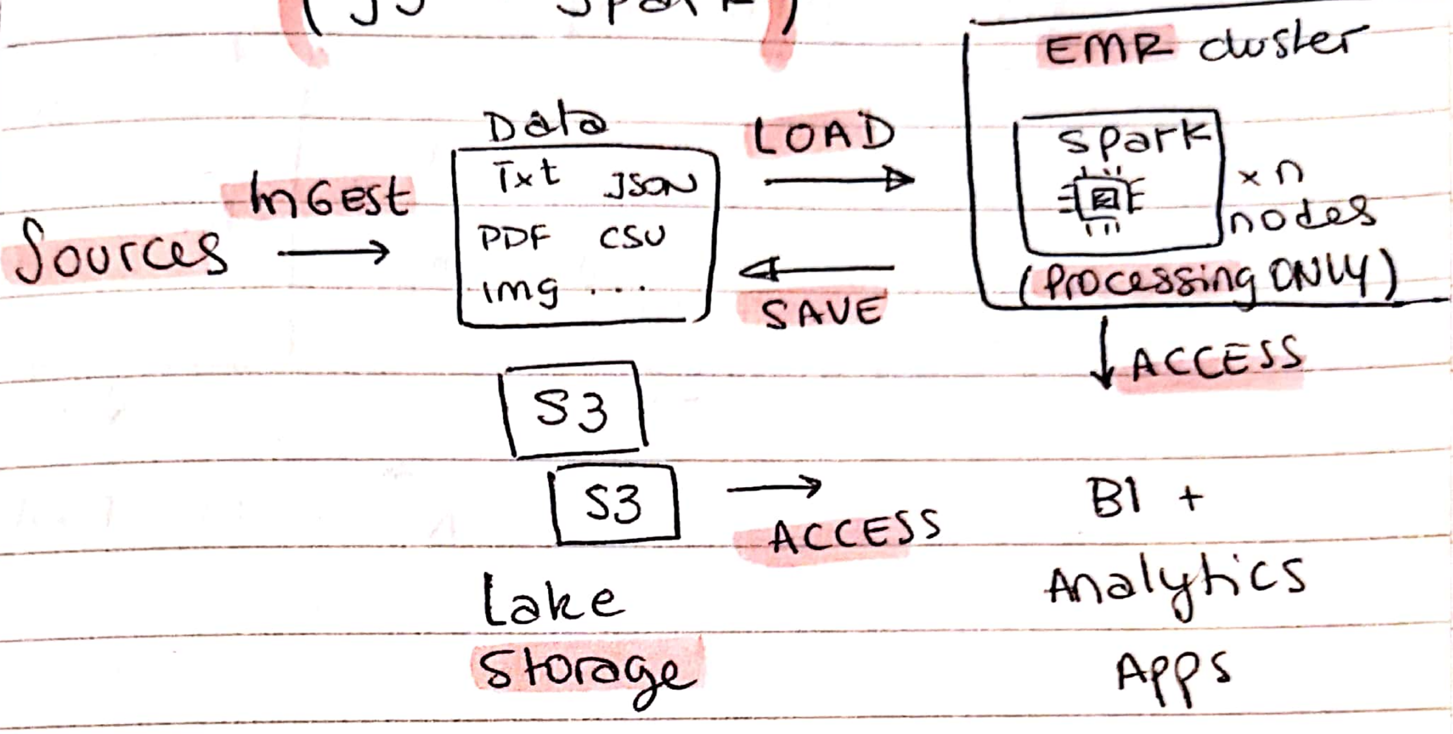
AWS EMR (HDFS + Spark)


S3
S3
RDS
Cass.
Dynamo DB
EC2
EC2
Sources
Ingest
EMR
Diverse data formats (xml, json, img, txt..)
Spark HDFS
n × nodes
EMR Big Data cluster (storage + processing)
Access
BI + Analytics

- Once ingested all data is stored on HDFS, processed on the cluster + accessed from Analytics + BI Apps.
- EMR cluster is created once, it can grow but it cannot be shutdown or else it loses all the stored data.

! Cost → keeping the cluster running -

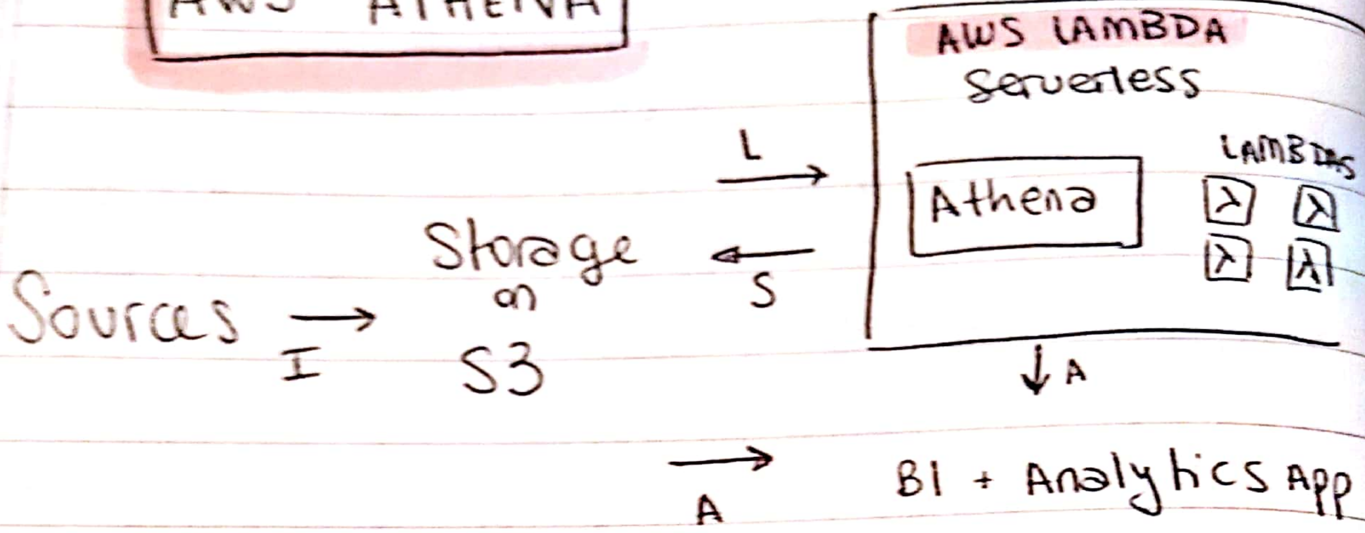
(S3 + Spark)



- NO HDFS in EMR
  - EMR cluster is spun on-demand, shutdown otherwise.

Potentially → less costly, easier to manage, more performant.

AWS ATHENA



LAMBDA → Function as a service
(Pay by execution time)

ATHENA → loads + processes data on serverless lambda resources

- Transparent management of resources