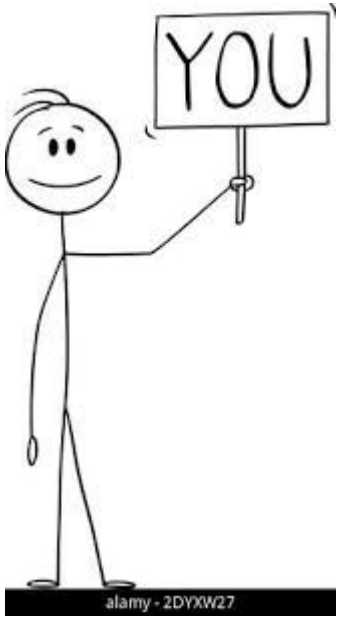
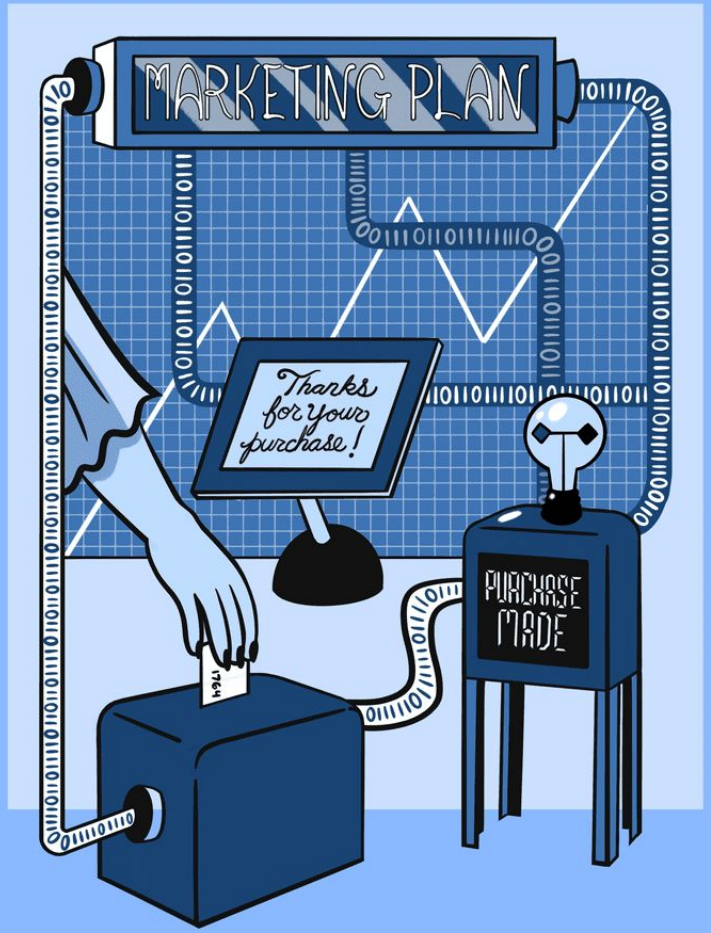
Spark

Inspiration

Lets talk about data



15 GB data



Big Data

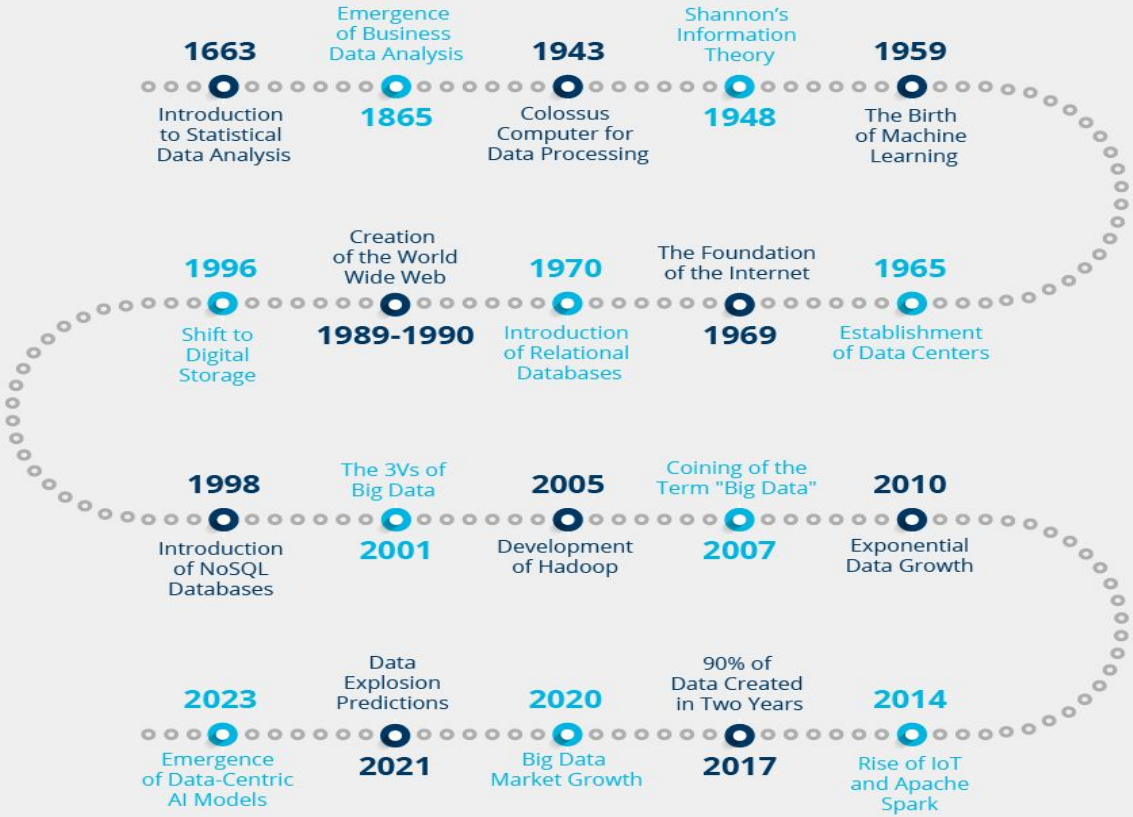
['big 'dā-tə]

Large, diverse sets of information that grow at ever-increasing rates.

Investopedia

VELTRIS
INNOVATE | ACCELERATE | TRANSFORM

The Evolution of Big Data:
Timeline of Crucial Developments



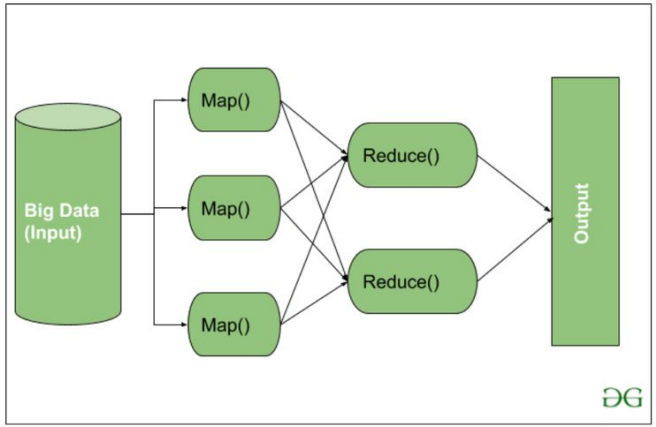
MapReduce

- A **programming model** (created at Google, 2004) for processing huge datasets in parallel across many machines.
- **Two main steps:**
  - **Map:** Break data into chunks → process each chunk independently → emit key-value pairs.
  - **Reduce:** Shuffle data by key → aggregate or combine results.

Example:
Imagine counting words in millions of log files:

- **Map:** Each worker counts words in its chunk → emits `(word, count)`.
- **Reduce:** All counts for the same word are summed → final global counts.

Let companies process **terabytes–petabytes** of data without buying a supercomputer.



Hadoop

Open-source implementation of Google's MapReduce

Core ideas

HDFS (Hadoop Distributed File System):
Stores data across many machines with replication → fault tolerant, scalable.

YARN (Yet Another Resource Negotiator):
Manages cluster resources (decides where tasks run).

MapReduce Engine:
Executes MapReduce jobs on top of HDFS.

MapReduce and Hadoop

Are disk based frameworks: Persists intermediate results to disk – >Data is reloaded from disk with every query

Good reliability , terrible speed

Spark is a memory-optimized data processing engine that can execute operations against Hadoop data (in HDFS) with better performance than MapReduce.

“Spark took the ideas of MapReduce, kept the fault tolerance and scalability, but ditched the disk I/O bottleneck, letting us analyze big data *in memory*, interactively, and much faster”

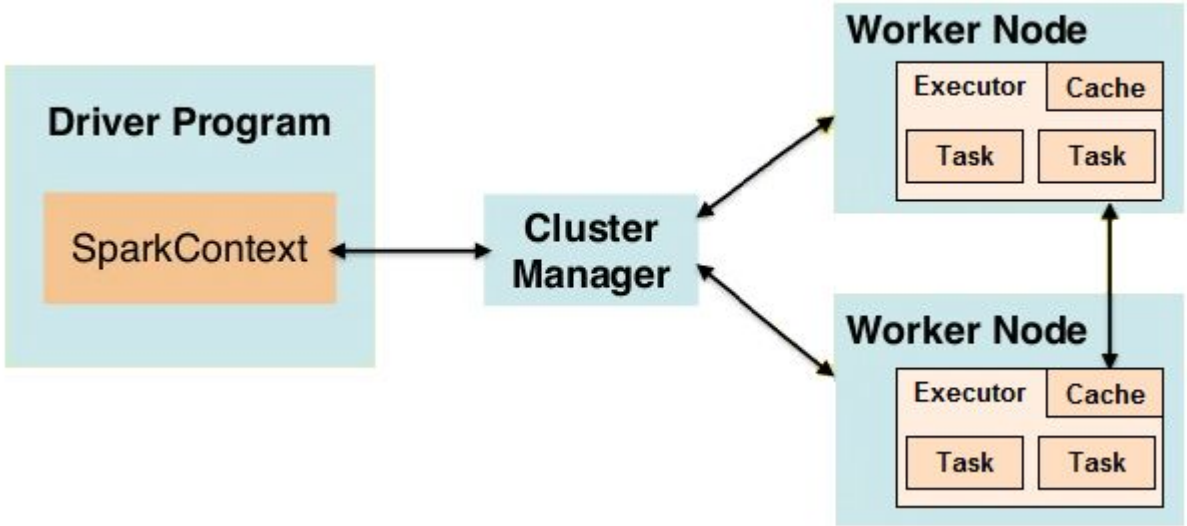
MapReduce	Spark
Writes to disk after every step	**Keeps data in memory** between steps (RDD caching)
Batch-only	Supports **batch, streaming, ML, and graph** workloads
Low-level Java code	**High-level APIs** in Python, Scala, SQL

Spark under the hood

Driver Program: coordinates everything, builds DAG, sends tasks.

Cluster Manager: can be YARN, Kubernetes, Mesos, or Spark Standalone.

Executors: run tasks in parallel on worker nodes, keep data in memory.



Spark Ecosystem

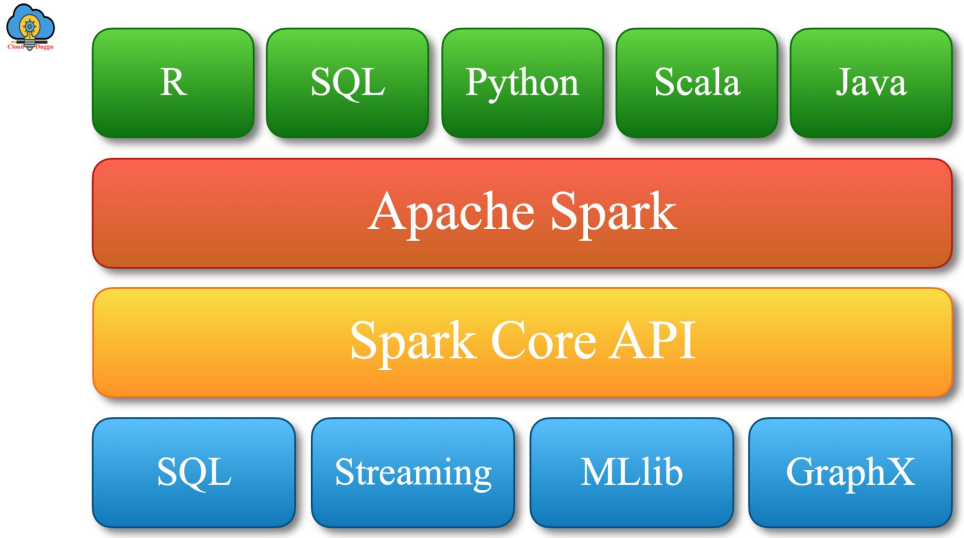
Spark Core (RDD engine) – the foundation

Spark SQL – structured data, DataFrames, SQL

MLlib – machine learning library

GraphX – graph processing

Structured Streaming – real-time processing



Spark in data engineering

Ingestion & ETL: Read raw data from HDFS, S3, GCS, Kafka → transform → write to data lake or warehouse.

Batch & Streaming: Same engine for scheduled pipelines **and** near real-time data processing.

Data Modeling: Build clean, partitioned tables (Bronze/Silver/Gold layers).

Feature Engineering: Prepare datasets for machine learning.

So wait….whats the point of dbt?

dbt is *amazing* for analytics transformations in SQL, but it isn't a compute engine and can't do complex ETL or machine learning. Spark often powers the heavy lifting, and dbt sits on top for the modeling layer

Raw Data → Spark (heavy ETL) → Curated Tables → dbt (SQL modeling/tests) → BI/ML

Why not just use BigQuery / Snowflake / Redshift and stay serverless?

If you need:

SQL-Only Transformations: Straightforward SELECT/JOIN/AGG logic fits perfectly.

Data Size Fits Warehouse Limits: BigQuery happily handles terabytes-scale tables.

Serverless, No Cluster Management: You don’t want to manage compute resources.

Ad-hoc Analytics: Analysts just need to run SQL and get answers.

BQ is fine enough

Complex / Non-SQL Logic

- Custom UDFs, machine learning, NLP, graph processing, advanced feature engineering.
- Not everything maps cleanly to SQL (e.g. iterative algorithms).

Streaming + Batch in One Engine

- Structured Streaming lets you process real-time data *and* historical data with the same code.

Cost & Performance Control

- For very large transformations, controlling partitioning, caching, and cluster sizing can be cheaper than per-query pricing in BigQuery.

Pre-Warehouse Data Processing

- Clean/transform **before** data ever lands in BigQuery (e.g. sensitive data masking, heavy parsing).

Spark landscape

Databricks

- Created by the original Spark creators
- Fully managed **“lakehouse platform”** built around Spark.
- Offers:
    - **Collaborative notebooks**
    - **Delta Lake** for ACID data lake tables
    - MLflow, Unity Catalog, job orchestration
- Good for: End-to-end data engineering + data science in one place.

GCP Dataproc

Managed Spark/Hadoop clusters in GCP.

Spin up on demand, run Spark jobs, auto-terminate.

Good for: Lift-and-shift workloads, ad-hoc batch jobs, cost control (pay-per-second clusters).

AWS EMR (Elastic MapReduce)

- Amazon’s managed Spark/Hadoop/YARN service.
- Similar to Dataproc but AWS-native.

Azure Synapse / HDInsight – Spark on Azure.

Kubernetes + Spark Operator – run Spark jobs in a container-native environment.

Standalone Spark – on-prem or custom cluster setups.

PySpark

PySpark is **the Python API for Apache Spark**. Spark itself is a distributed compute engine (written in Scala/Java). PySpark lets you write Python code that **drives Spark** on a cluster (or locally).

Lazy vs eager: Spark builds a plan; nothing runs until an **action** (`show()`, `count()`, `collect()`, `write`).

Expressions: In Spark you build **column expressions** (not arbitrary Python), e.g., `F.col("x") + 1`.

UDFs: Prefer Spark's built-ins (`F.*`). Python UDFs serialize data and are slower; **pandas UDFs** (a.k.a. vectorized UDFs) via Arrow are better, but still second choice after built-ins.

Nulls & types: Spark has SQL-like null semantics and explicit types.

Joins/window functions: Similar concepts, different API.

Filter->group->aggregate

pandas

```python
import pandas as pd
pdf = pd.DataFrame({"team": ["A","A","B"], "score": [1,5,3]})
out = (pdf[pdf["score"] > 2]
       .groupby("team", as_index=False)["score"].mean())
```

PySpark

```python
from pyspark.sql import functions as F

df = spark.createDataFrame([("A",1),("A",5),("B",3)], ["team","score"])
out = (df.filter(F.col("score") > 2)
       .groupBy("team")
       .agg(F.mean("score").alias("score")))
out.show()
```