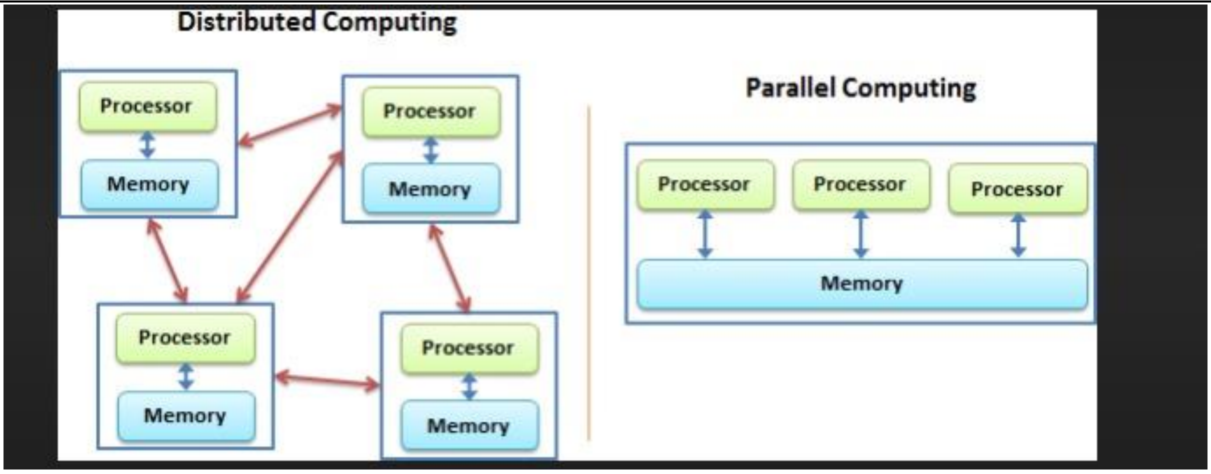
# DATA ENGINEERING BY TRISHANSH

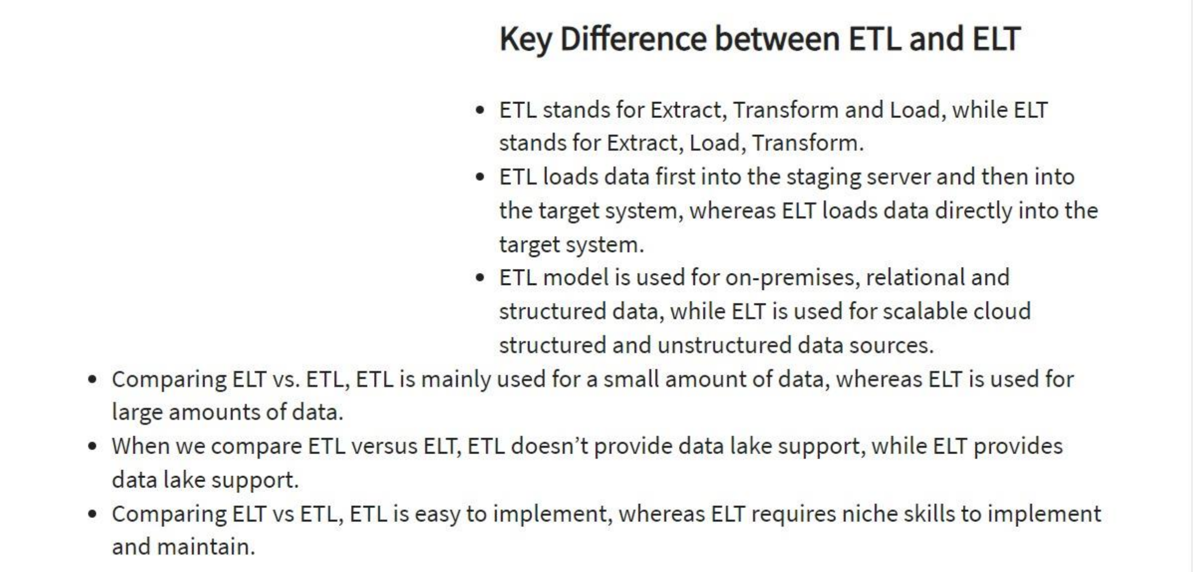
**Distributed computing** is a model in which multiple interconnected computers (nodes or servers) work together to solve a computational problem. Instead of relying on a single machine to perform all tasks, distributed computing breaks down tasks into smaller units and distributes them across multiple computers, enabling parallel processing and more efficient use of resources.

Hadoop , spark , kafka

**Parallel computing** is a model in which multiple processors or cores within a single computer or a closely connected system work together to solve a computational problem. Instead of relying on a single processor to perform all tasks sequentially, parallel computing divides tasks into smaller units and distributes them across multiple processors, enabling concurrent execution and faster processing.

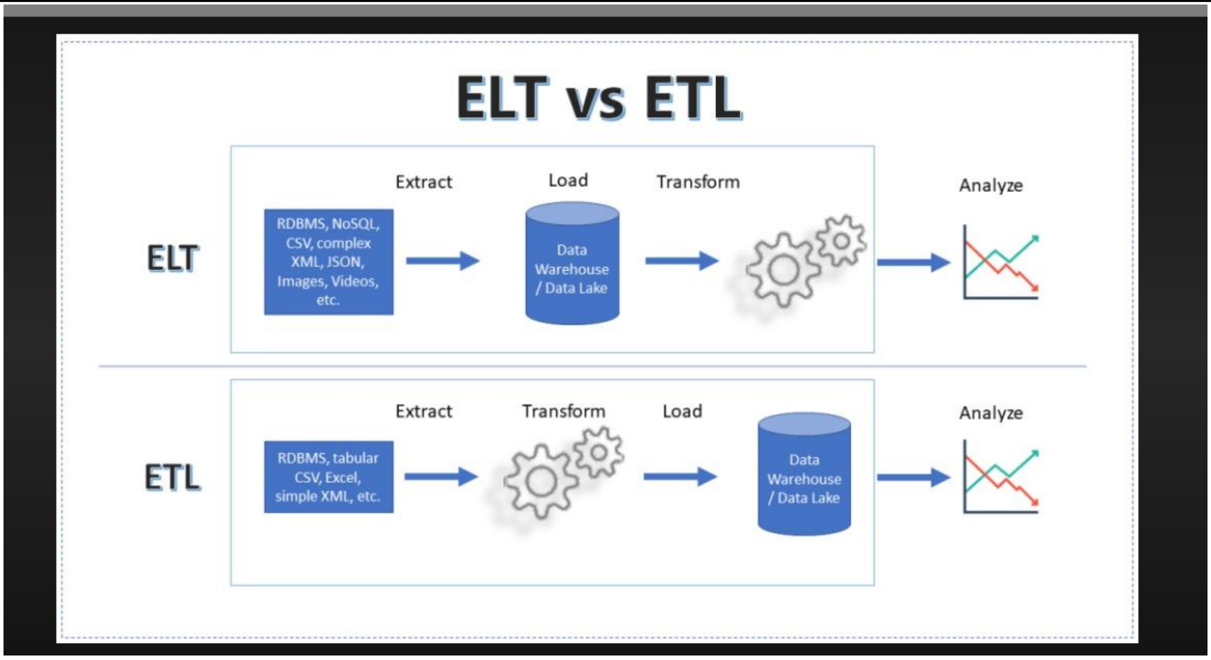
Dask , flink , tenserflow and pytorch





## Key Difference between ETL and ELT

- ETL stands for Extract, Transform and Load, while ELT stands for Extract, Load, Transform.
- ETL loads data first into the staging server and then into the target system, whereas ELT loads data directly into the target system.
- ETL model is used for on-premises, relational and structured data, while ELT is used for scalable cloud structured and unstructured data sources.
- Comparing ELT vs. ETL, ETL is mainly used for a small amount of data, whereas ELT is used for large amounts of data.
- When we compare ETL versus ELT, ETL doesn't provide data lake support, while ELT provides data lake support.
- Comparing ELT vs ETL, ETL is easy to implement, whereas ELT requires niche skills to implement and maintain.



## ELT vs ETL

ELT

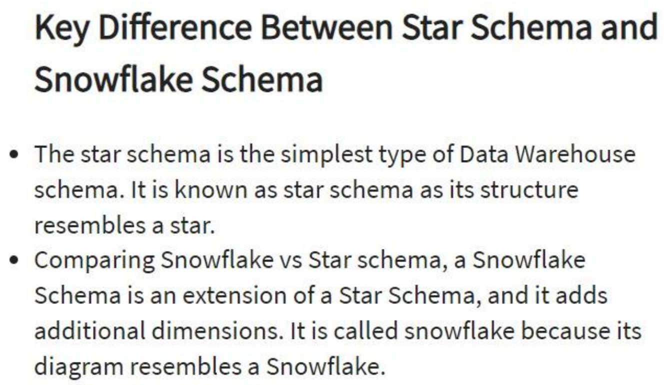
Extract → Load → Transform → Analyze

RDBMS, NoSQL, CSV, complex XML, JSON, Images, Videos, etc. → Data Warehouse / Data Lake → Transform → Analyze

ETL

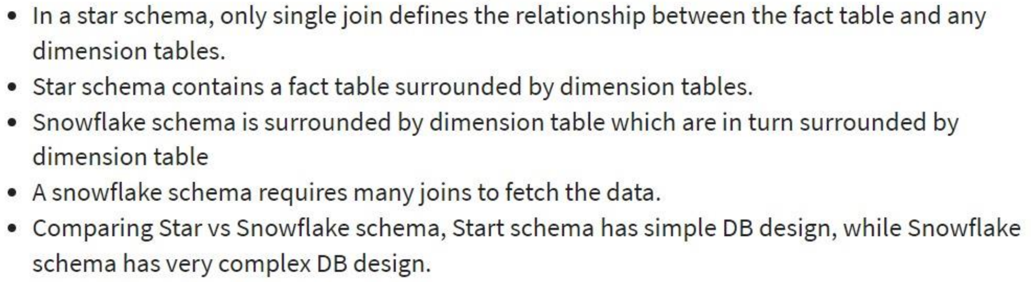
Extract → Transform → Load → Analyze

RDBMS, tabular CSV, Excel, simple XML, etc. → Transform → Data Warehouse / Data Lake → Analyze

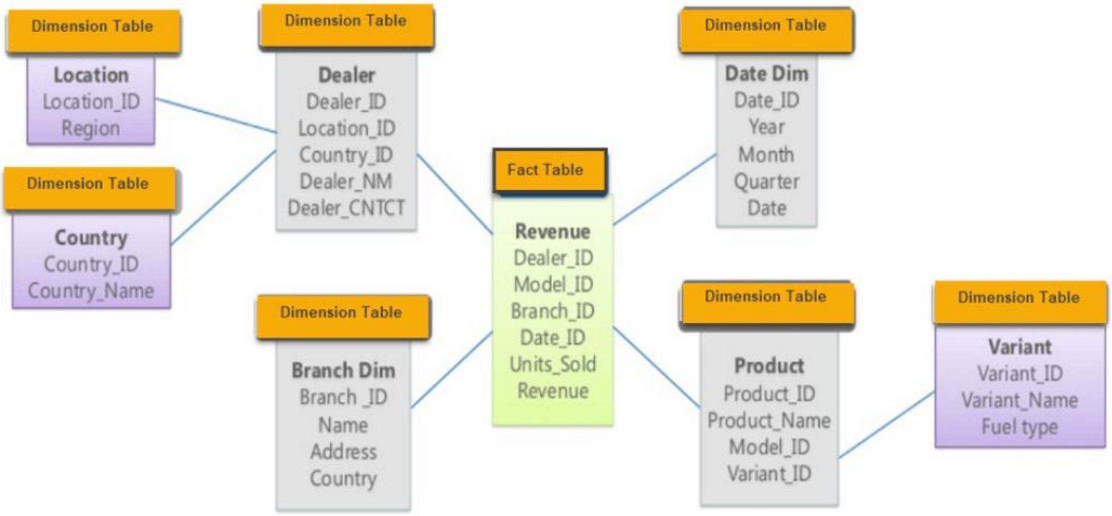


## Key Difference Between Star Schema and Snowflake Schema

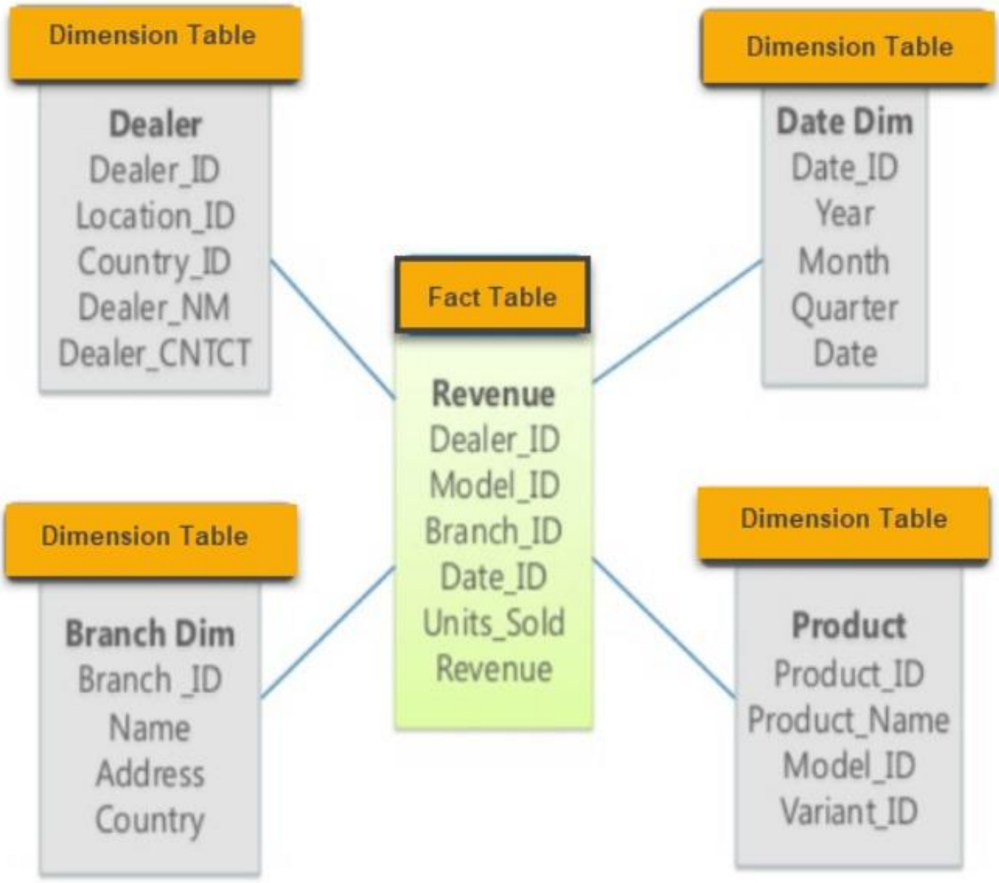
- The star schema is the simplest type of Data Warehouse schema. It is known as star schema as its structure resembles a star.
- Comparing Snowflake vs Star schema, a Snowflake Schema is an extension of a Star Schema, and it adds additional dimensions. It is called snowflake because its diagram resembles a Snowflake.



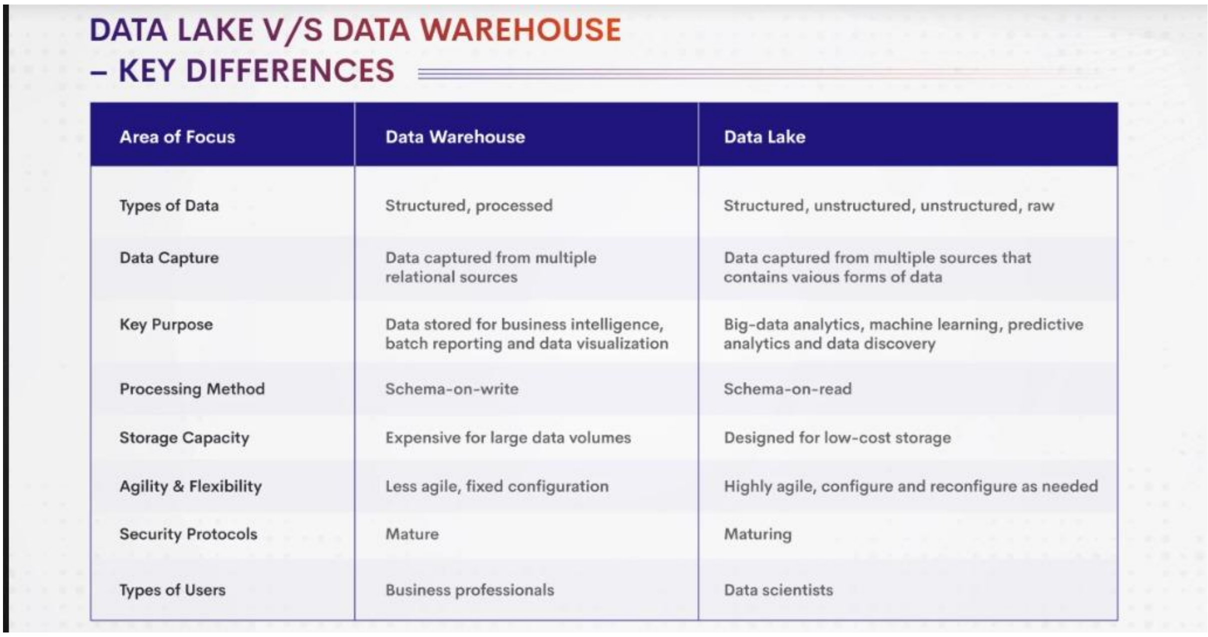
- In a star schema, only single join defines the relationship between the fact table and any dimension tables.
- Star schema contains a fact table surrounded by dimension tables.
- Snowflake schema is surrounded by dimension table which are in turn surrounded by dimension table
- A snowflake schema requires many joins to fetch the data.
- Comparing Star vs Snowflake schema, Start schema has simple DB design, while Snowflake schema has very complex DB design.



Example of Snowflake Schema

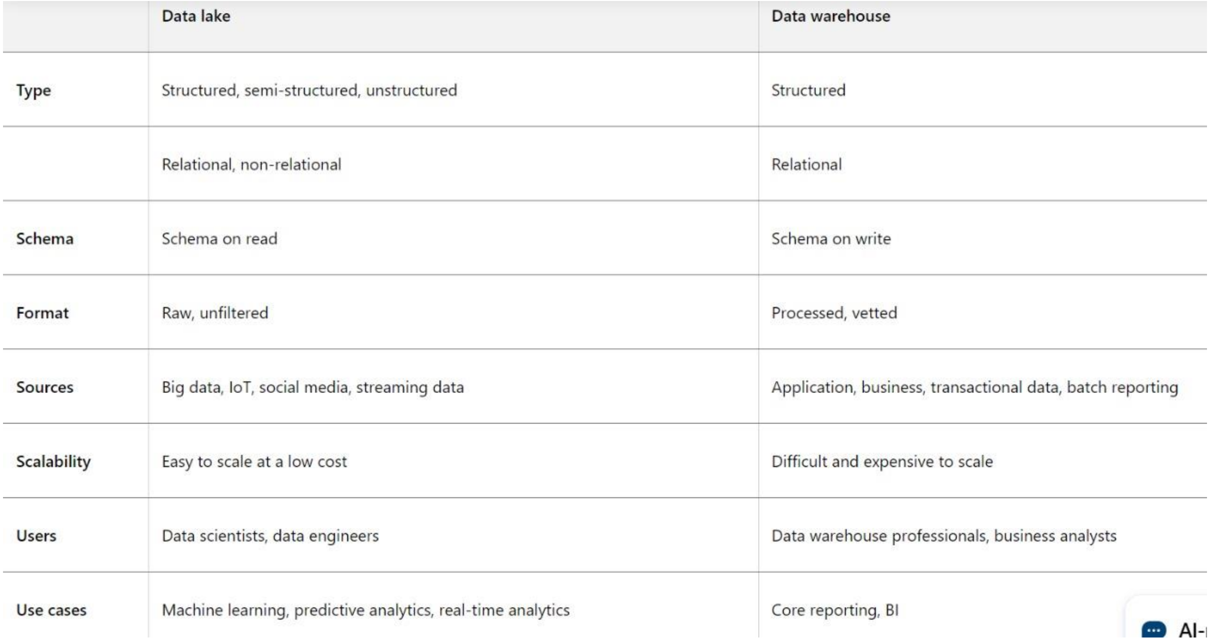


# DATA ENGINEERING BY TRISHANSH

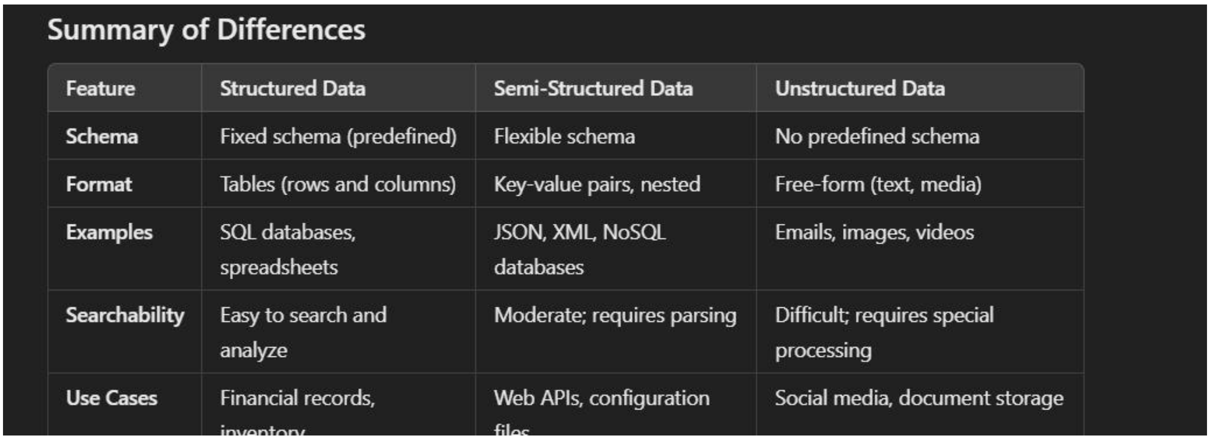


## DATA LAKE V/S DATA WAREHOUSE – KEY DIFFERENCES

| Area of Focus | Data Warehouse | Data Lake |
|---|---|---|
| Types of Data | Structured, processed | Structured, unstructured, unstructured, raw |
| Data Capture | Data captured from multiple relational sources | Data captured from multiple sources that contains vaious forms of data |
| Key Purpose | Data stored for business intelligence, batch reporting and data visualization | Big-data analytics, machine learning, predictive analytics and data discovery |
| Processing Method | Schema-on-write | Schema-on-read |
| Storage Capacity | Expensive for large data volumes | Designed for low-cost storage |
| Agility & Flexibility | Less agile, fixed configuration | Highly agile, configure and reconfigure as needed |
| Security Protocols | Mature | Maturing |
| Types of Users | Business professionals | Data scientists |



| | Data lake | Data warehouse |
|---|---|---|
| Type | Structured, semi-structured, unstructured | Structured |
| | Relational, non-relational | Relational |
| Schema | Schema on read | Schema on write |
| Format | Raw, unfiltered | Processed, vetted |
| Sources | Big data, IoT, social media, streaming data | Application, business, transactional data, batch reporting |
| Scalability | Easy to scale at a low cost | Difficult and expensive to scale |
| Users | Data scientists, data engineers | Data warehouse professionals, business analysts |
| Use cases | Machine learning, predictive analytics, real-time analytics | Core reporting, BI |

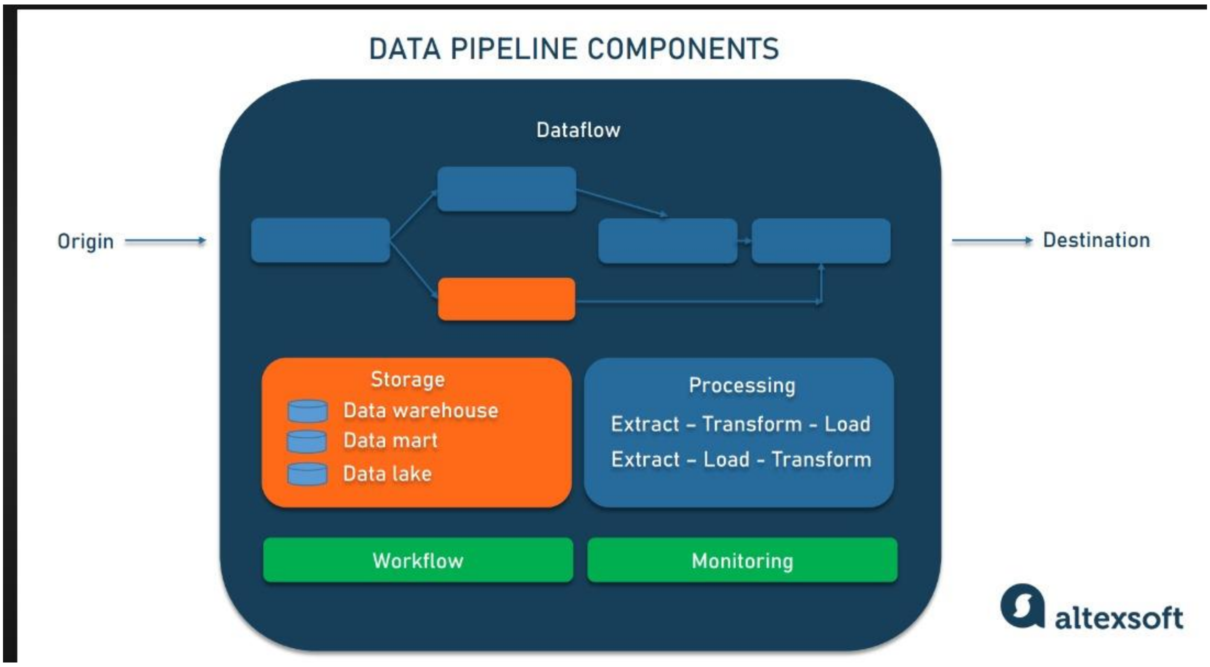


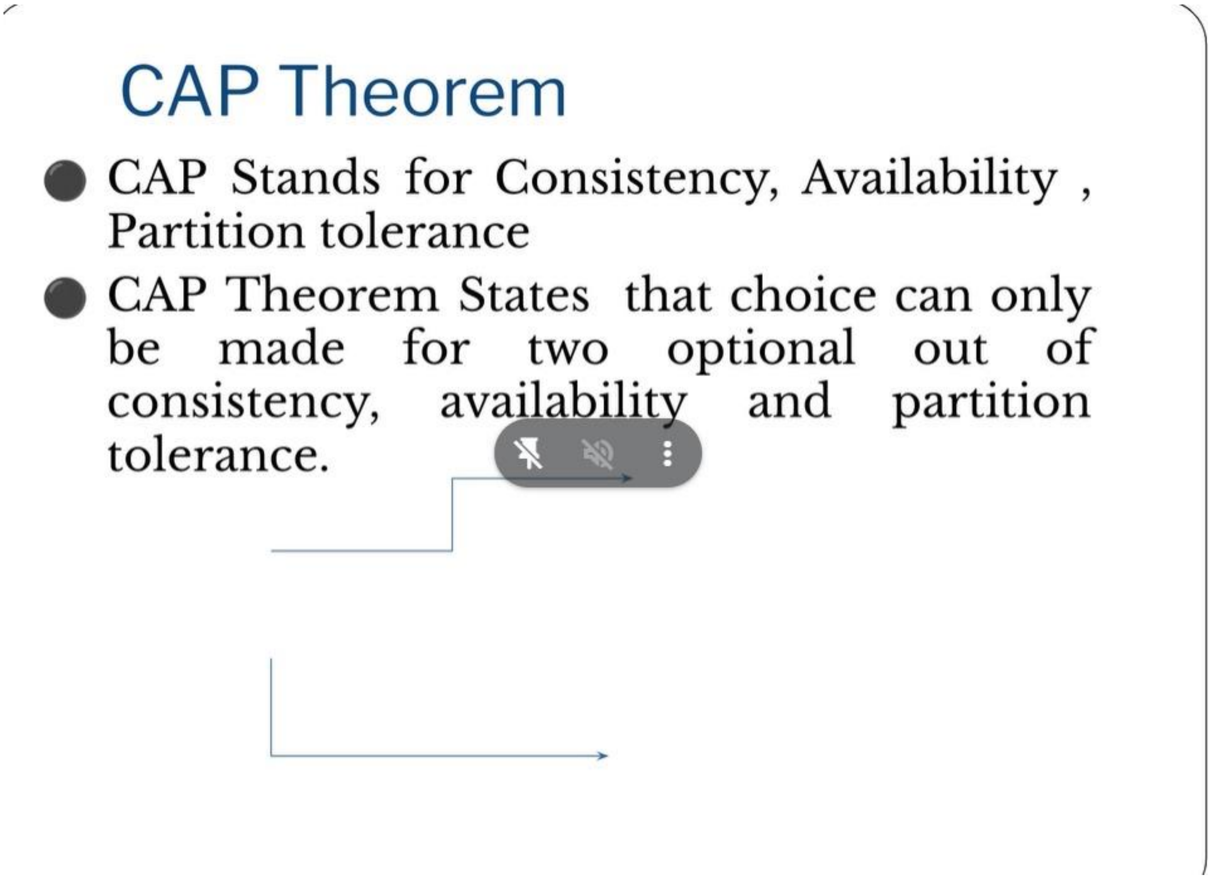
## Summary of Differences

| Feature | Structured Data | Semi-Structured Data | Unstructured Data |
|---|---|---|---|
| Schema | Fixed schema (predefined) | Flexible schema | No predefined schema |
| Format | Tables (rows and columns) | Key-value pairs, nested | Free-form (text, media) |
| Examples | SQL databases, spreadsheets | JSON, XML, NoSQL databases | Emails, images, videos |
| Searchability | Easy to search and analyze | Moderate; requires parsing | Difficult; requires special processing |
| Use Cases | Financial records, inventory | Web APIs, configuration files | Social media, document storage |

# DATA ENGINEERING BY TRISHANSH

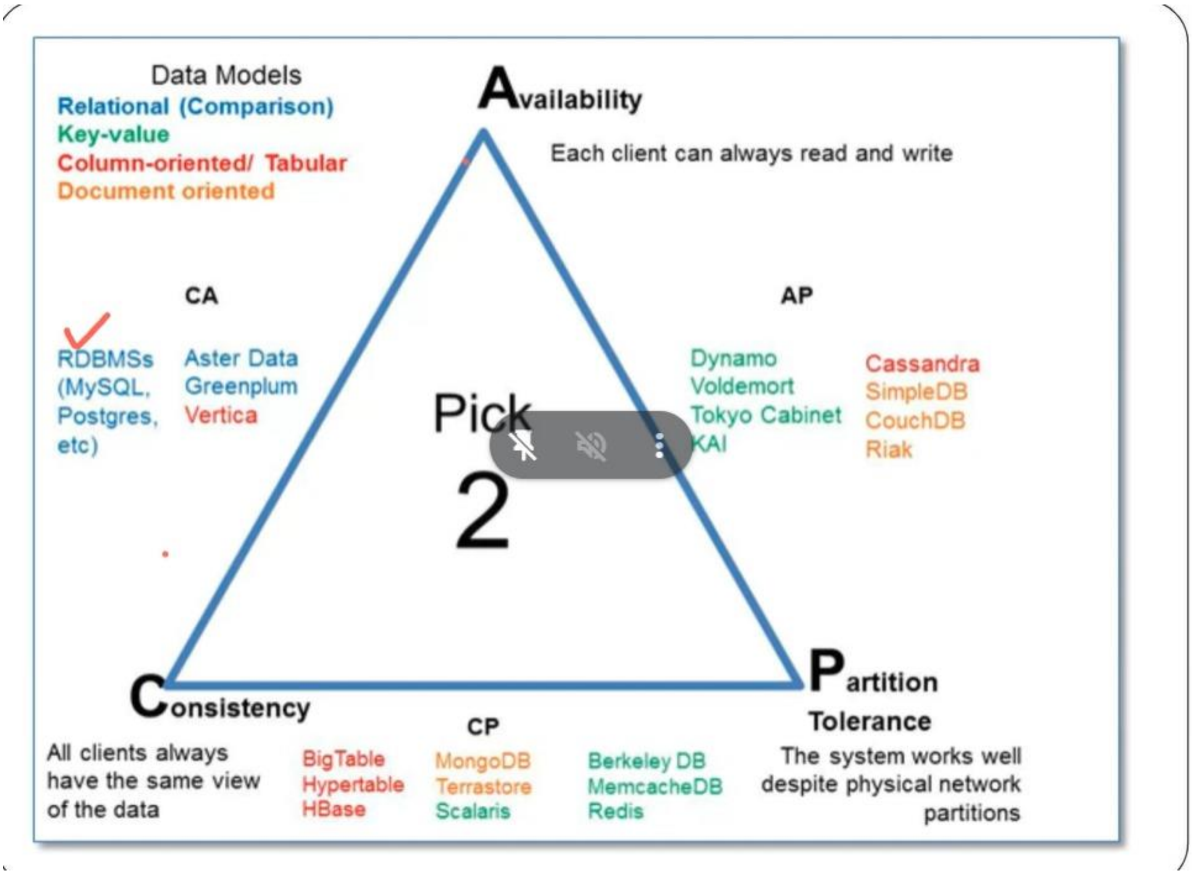
**Data pipeline** is a series of processes that automatically move data from one system to another. It typically involves collecting data from various sources, transforming it into a usable format, and then loading it into a destination (like a database or data warehouse) for analysis or further use. The goal is to ensure data flows smoothly and efficiently, enabling better insights and decision-making.





## CAP Theorem

- CAP Stands for Consistency, Availability , Partition tolerance
- CAP Theorem States that choice can only be made for two optional out of consistency, availability and partition tolerance.



Data Models

Relational (Comparison)

Key-value

Column-oriented/ Tabular

Document oriented

**A**vailability

Each client can always read and write

CA

RDBMSs (MySQL, Postgres, etc)

Aster Data

Greenplum

Vertica

AP

Dynamo

Voldemort

Tokyo Cabinet

KAI

Cassandra

SimpleDB

CouchDB

Riak

Pick 2

**C**onsistency

All clients always have the same view of the data

BigTable

Hypertable

HBase

CP

MongoDB

Terrastore

Scalaris

Berkeley DB

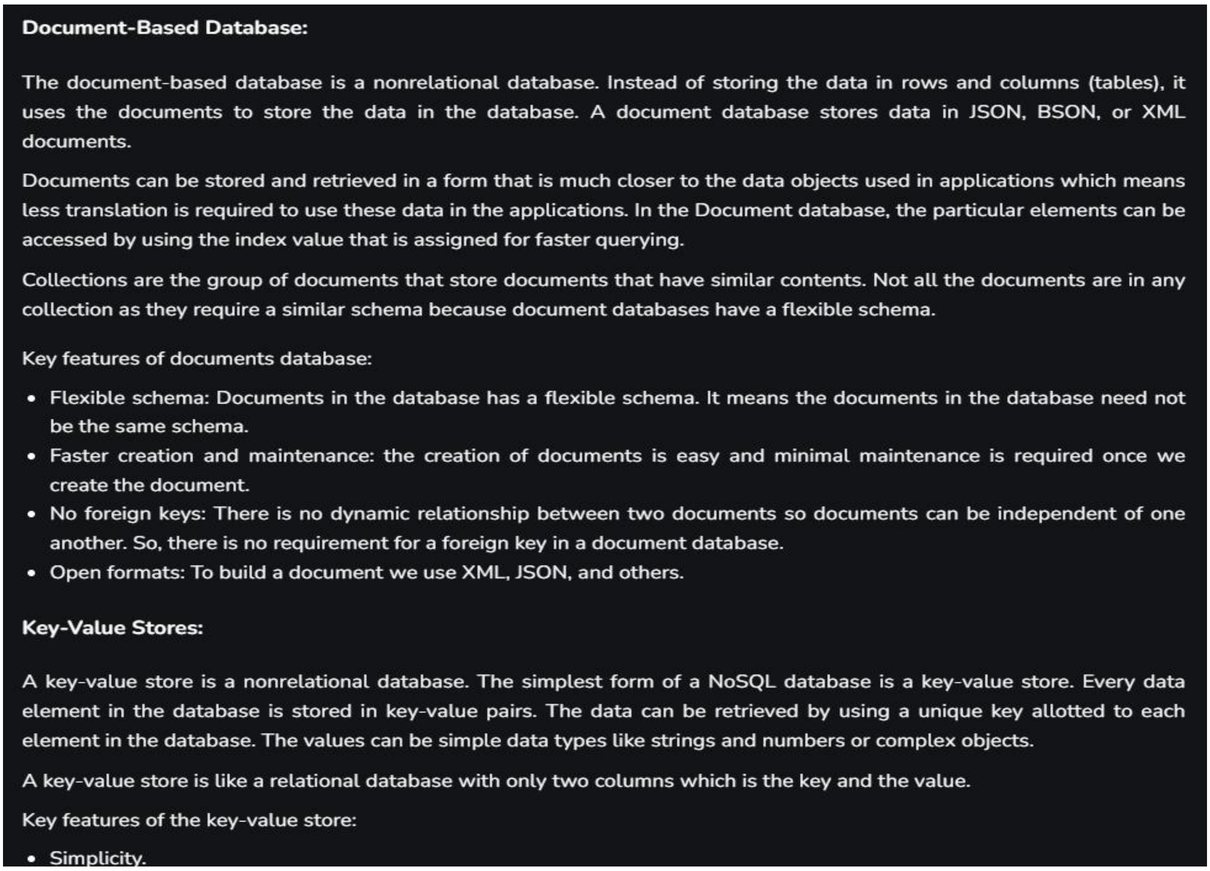
MemcacheDB

Redis

**P**artition Tolerance

The system works well despite physical network partitions

## NO SQL Databases



**Document-Based Database:**

The document-based database is a nonrelational database. Instead of storing the data in rows and columns (tables), it uses the documents to store the data in the database. A document database stores data in JSON, BSON, or XML documents.

Documents can be stored and retrieved in a form that is much closer to the data objects used in applications which means less translation is required to use these data in the applications. In the Document database, the particular elements can be accessed by using the index value that is assigned for faster querying.

Collections are the group of documents that store documents that have similar contents. Not all the documents are in any collection as they require a similar schema because document databases have a flexible schema.

Key features of documents database:

- Flexible schema: Documents in the database has a flexible schema. It means the documents in the database need not be the same schema.
- Faster creation and maintenance: the creation of documents is easy and minimal maintenance is required once we create the document.
- No foreign keys: There is no dynamic relationship between two documents so documents can be independent of one another. So, there is no requirement for a foreign key in a document database.
- Open formats: To build a document we use XML, JSON, and others.

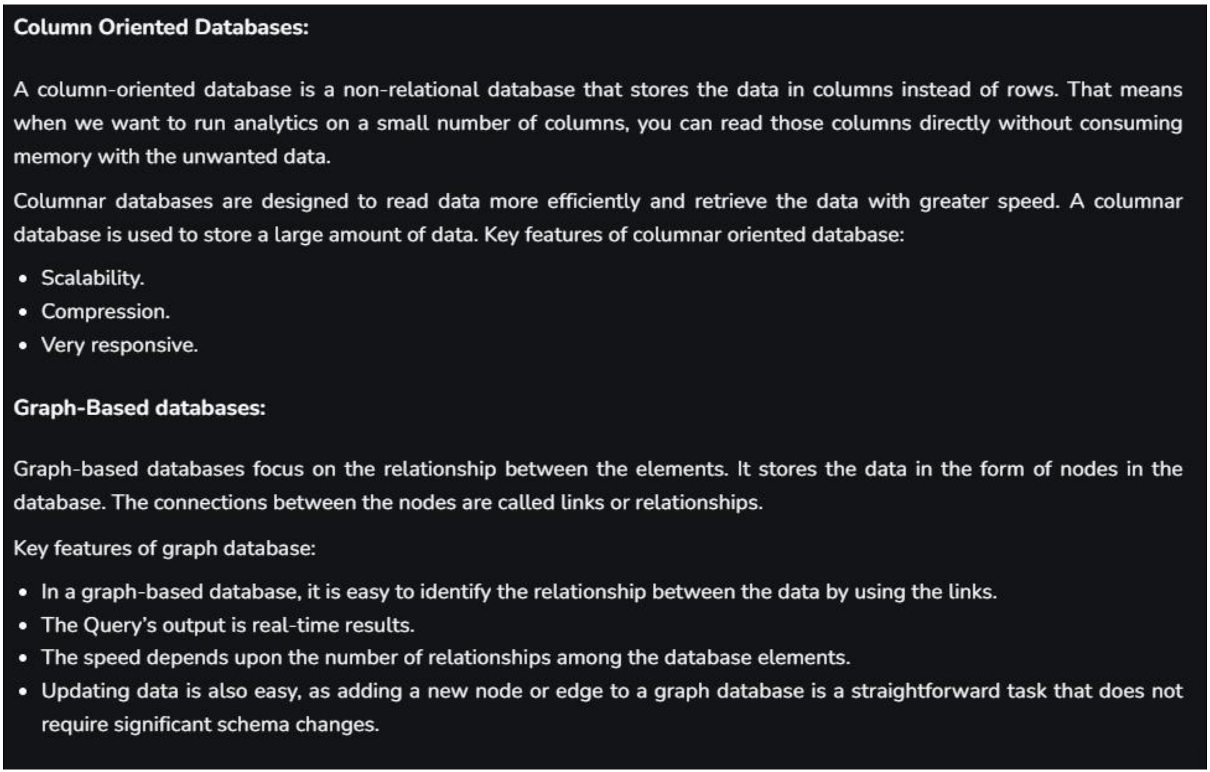
**Key-Value Stores:**

A key-value store is a nonrelational database. The simplest form of a NoSQL database is a key-value store. Every data element in the database is stored in key-value pairs. The data can be retrieved by using a unique key allotted to each element in the database. The values can be simple data types like strings and numbers or complex objects.

A key-value store is like a relational database with only two columns which is the key and the value.

Key features of the key-value store:

- Simplicity.



**Column Oriented Databases:**

A column-oriented database is a non-relational database that stores the data in columns instead of rows. That means when we want to run analytics on a small number of columns, you can read those columns directly without consuming memory with the unwanted data.

Columnar databases are designed to read data more efficiently and retrieve the data with greater speed. A columnar database is used to store a large amount of data. Key features of columnar oriented database:

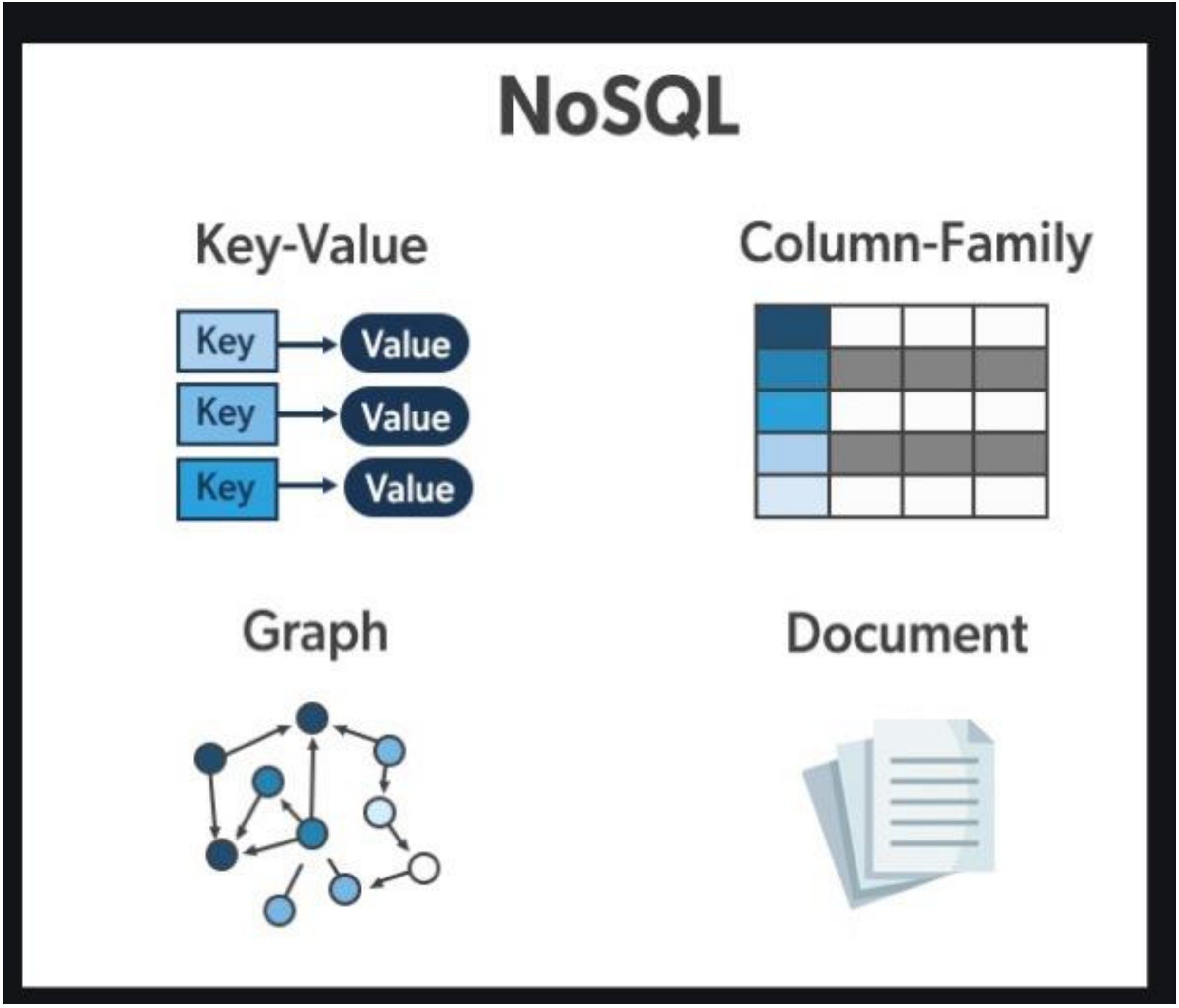
- Scalability.
- Compression.
- Very responsive.

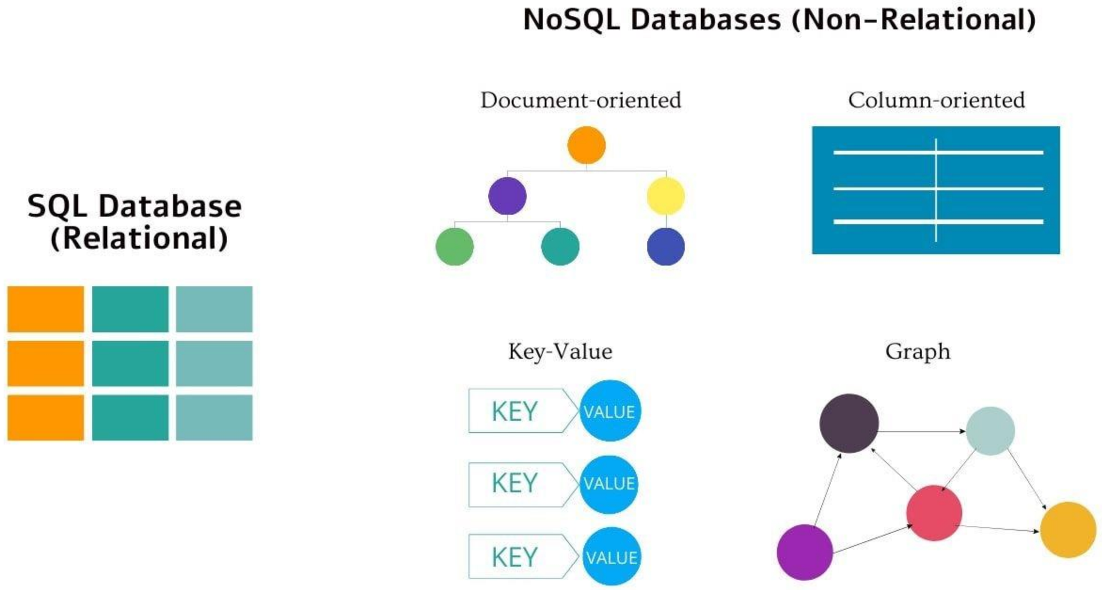
**Graph-Based databases:**

Graph-based databases focus on the relationship between the elements. It stores the data in the form of nodes in the database. The connections between the nodes are called links or relationships.

Key features of graph database:

- In a graph-based database, it is easy to identify the relationship between the data by using the links.
- The Query's output is real-time results.
- The speed depends upon the number of relationships among the database elements.
- Updating data is also easy, as adding a new node or edge to a graph database is a straightforward task that does not require significant schema changes.





*Scheduling in data engineering* refers to the process of automating the execution of data pipelines, jobs, or tasks at specific times or in response to certain events. It ensures that data is processed, transformed, and delivered consistently, without manual intervention. Scheduling is a critical component in managing the *ETL (Extract, Transform, Load)* processes, data pipelines, and workflows in data engineering.

### Key Concepts in Scheduling:

1. *Time-based Scheduling*:

- Tasks or jobs are triggered at specific times, such as daily, hourly, or weekly. This is useful for recurring processes like refreshing data warehouses or generating reports.

- *Example*: A pipeline that loads sales data every day at midnight.

2. *Event-based Scheduling*:

- Tasks are triggered by specific events or conditions, such as the arrival of new data, completion of another task, or changes in a data source.

- *Example*: Triggering a data transformation process once a new file is uploaded to a data lake.

3. *Batch Scheduling*:

- Used when tasks need to be executed in bulk or groups. Batch scheduling is common in data processing tasks where large volumes of data are processed at predefined intervals.

- *Example*: Running a data cleaning process on an entire dataset at the end of the day.

4. *Real-time Scheduling*: - Involves continuous or near-real-time execution of tasks to handle streaming data. Real-time scheduling is used when immediate action is required after a data event.

- *Example*: Processing incoming transaction data from a financial system as it arrives.

### Common Tools for Scheduling in Data Engineering:

1. *Apache Airflow*:

- A popular open-source tool for orchestrating complex workflows. It allows users to define data workflows as Directed Acyclic Graphs (DAGs) and supports both time-based and event-driven scheduling.

- *Use Case*: Scheduling and monitoring complex ETL jobs and data pipelines.

2. *Cron Jobs*:

- A time-based job scheduler in Unix-like systems. Cron is simple and effective for scheduling recurring tasks based on a specific time.

- *Use Case*: Automating the execution of scripts or data processing jobs at regular intervals (e.g., hourly, daily).

3. *Kubernetes CronJobs*:

- A feature in Kubernetes that allows you to run jobs on a schedule. Similar to traditional cron jobs but designed to work in containerized environments.

- *Use Case*: Scheduling jobs in containerized data applications.

4. *Luigi*:

- A Python-based framework for building complex pipelines, with built-in scheduling capabilities. Luigi is designed for long-running data pipelines.

- *Use Case*: Managing dependencies and scheduling multi-step data processes.

5. *Dagster*:

- A modern data orchestrator tool that allows scheduling, monitoring, and tracking of data workflows, with a strong focus on data quality and lineage.

- *Use Case*: Scheduling and orchestrating data pipelines while tracking data flow and transformations.

6. *Apache NiFi*:

- A data integration tool that automates the flow of data between systems. NiFi enables real-time scheduling of data flows.

- *Use Case*: Automating real-time data ingestion and movement between different data systems

7. *Prefect*:

- A workflow orchestration tool designed to handle data workflows in modern cloud-based environments. It offers flexible scheduling and monitoring features.

- *Use Case*: Scheduling cloud-based ETL tasks and managing data pipeline dependencies.

8. *Oozie*: - A workflow scheduling system for managing Hadoop jobs. Oozie supports both time-based and event-based scheduling. *Use Case*: Scheduling big data processing jobs within a Hadoop ecosystem.

---

### ***What is RDD in Spark?***

*RDD (Resilient Distributed Dataset)* is a fundamental data structure in *Apache Spark*. It's a distributed collection of data that is fault-tolerant, meaning it can recover from failures. RDDs can be created by:

- Loading external data (e.g., from HDFS, local storage).

- Transforming existing RDDs through operations like map() or filter().

### *Why Do We Need RDD?*

- *Fault Tolerance*: If part of the data is lost, RDD can rebuild it automatically using lineage (the sequence of operations used to create the RDD).

- *Distributed*: RDD splits data across multiple machines, allowing Spark to process large datasets in parallel.

- *In-memory Computation*: RDD keeps data in memory (RAM), which speeds up processing compared to reading from disk.

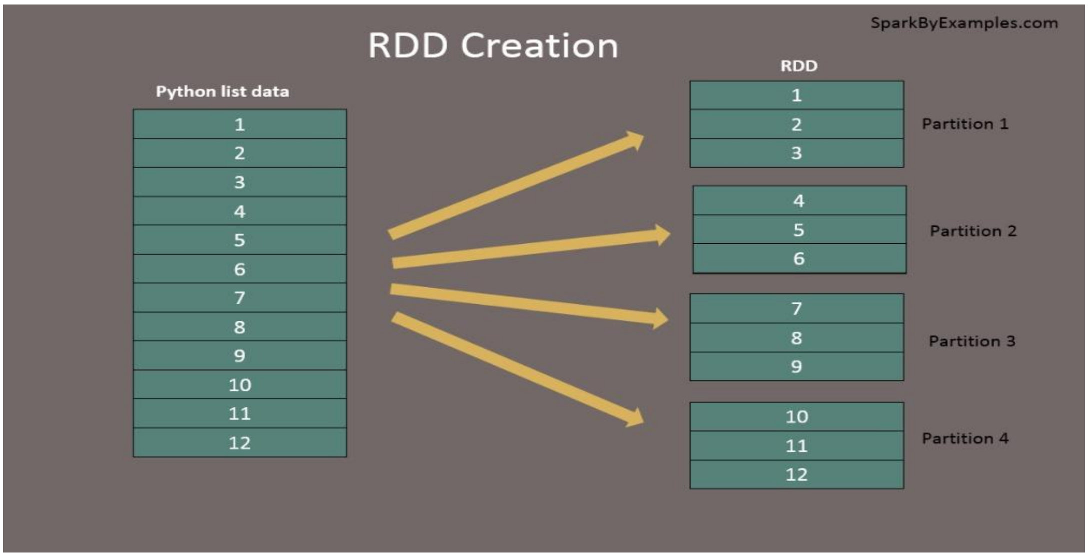
### *Why is Spark Faster than Hadoop?*

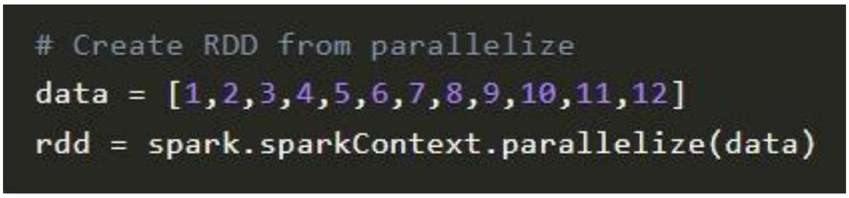
1. *In-memory Computation: Spark performs most of its operations in memory (RAM), while Hadoop reads and writes data to disk. This makes Spark **much faster* for iterative tasks where data is used repeatedly.

2. *Efficient DAG Execution: Spark uses **Directed Acyclic Graph (DAG)* execution for better task scheduling and optimization, making it smarter in managing dependencies between tasks.

3. *Lazy Evaluation*: Spark only executes computations when an action (like count() or collect()) is called, allowing it to optimize the entire computation process.

In short, Spark is faster because it keeps data in memory, optimizes its execution, and avoids writing data to disk as often as Hadoop.





```
# Create RDD from parallelize
data = [1,2,3,4,5,6,7,8,9,10,11,12]
rdd = spark.sparkContext.parallelize(data)
```

### **What is Hadoop?**

Hadoop is an open-source framework that lets you store and process large amounts of data across many computers. It allows you to work with big data (huge datasets) by splitting the data and distributing it across multiple machines (called nodes), making the whole process faster and scalable.

### Hadoop Architecture (in simple words):

Hadoop has two main parts:

1. HDFS (Hadoop Distributed File System): This is the **storage system. It stores data across different machines (nodes) by breaking it into smaller chunks called **blocks. It also makes copies (replicas) of these blocks to ensure that even if one machine fails, the data is safe.

   - NameNode (Master): Keeps track of where all the pieces of the data are stored.

   - DataNodes (Slaves): Store the actual data blocks.

2. MapReduce: This is the **processing system. It processes the data stored in HDFS by breaking down the work into smaller tasks that can be handled by different machines.

   - JobTracker (Master): Manages and assigns jobs to the machines.

   - TaskTrackers (Slaves): Perform the actual data processing (tasks) on the data stored in the DataNodes.

### How Hadoop Works (In Short):

1. Data is split into smaller parts (blocks) and stored across many machines using HDFS.

2. When you want to process this data, MapReduce breaks the job into smaller tasks and runs them on different machines where the data is stored.

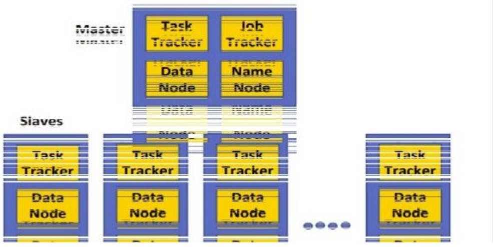
3. The results are combined to give the final output.

### Why Hadoop is Useful:

- Handles large data: It can process big data by distributing the workload across many computers.

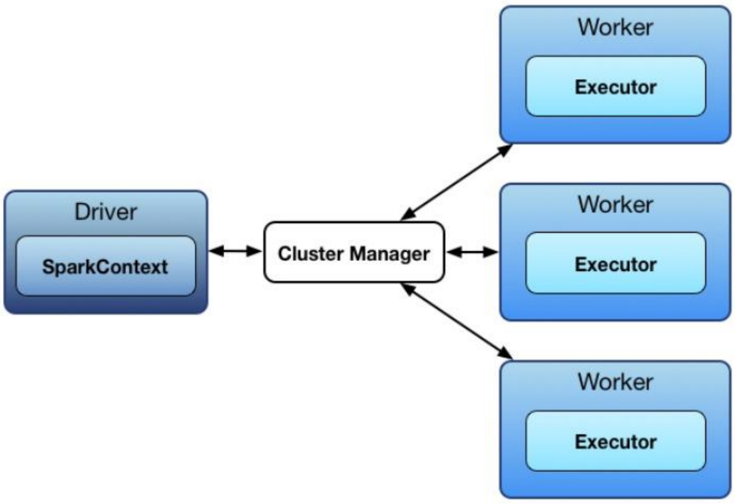
- Fault tolerance: If one machine fails, Hadoop keeps running because it stores multiple copies of the data.

- Scalable: You can add more computers to handle more data or work without changing the system.



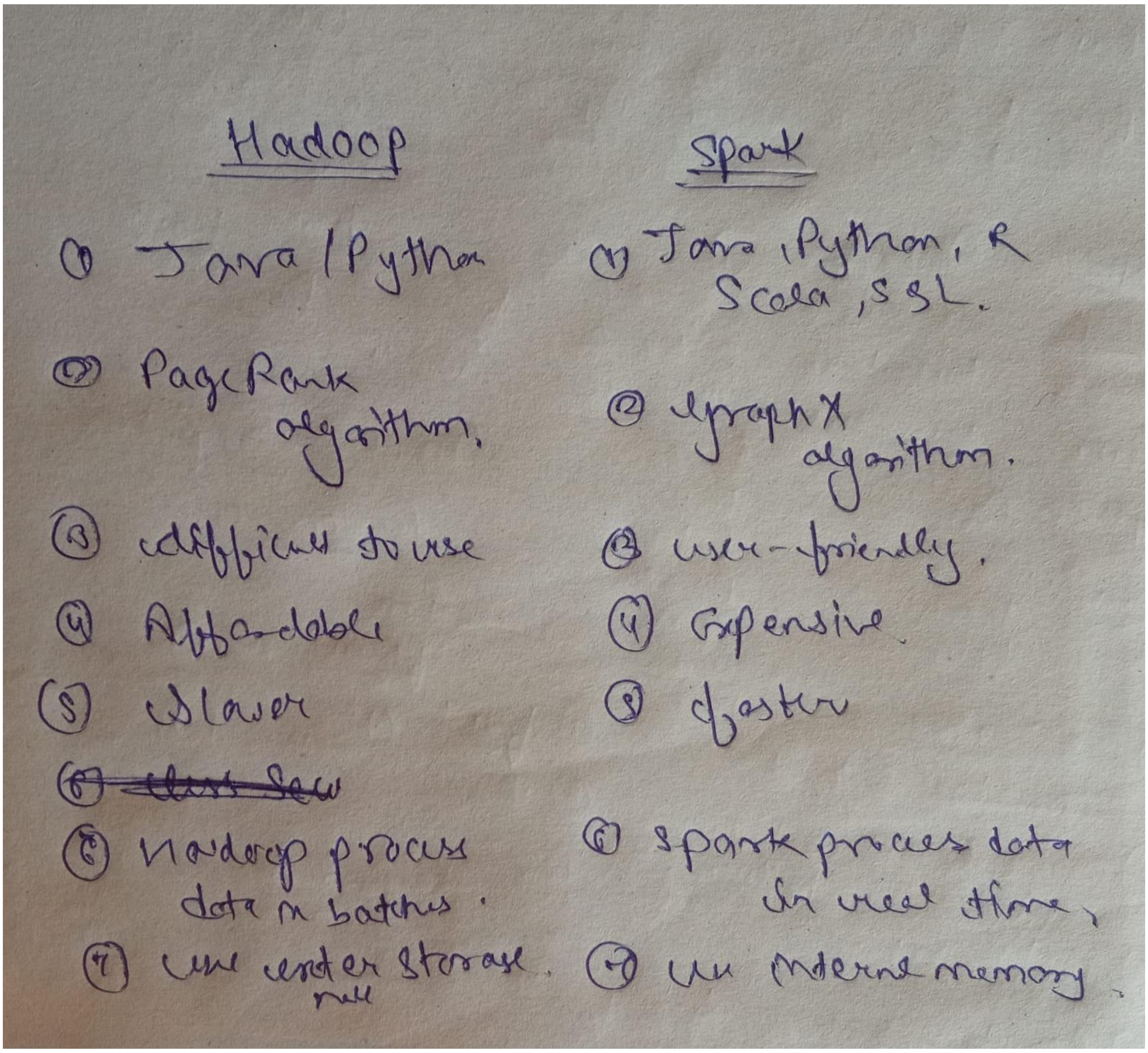
# DATA ENGINEERING BY TRISHANSH

**Spark Archietecture**



---

**HADOOP v/s SPARK**



| Hadoop | Spark |
|---|---|
| ① Java/Python | ① Java, Python, R Scala, SQL. |
| ② PageRank algorithm. | ② GraphX algorithm. |
| ③ difficult to use | ③ user-friendly. |
| ④ Affordable | ④ Expensive. |
| ⑤ Slower | ⑤ faster |
| ~~⑥ [illegible]~~ | |
| ⑥ Hadoop process data in batches. | ⑥ spark process data in real time, |
| ⑦ use outer storage. [illegible] | ⑦ use internal memory. |

**Horizontal and vertical scaling**

### Vertical Scaling (Scaling Up):

Vertical scaling involves increasing the capacity of a single machine by adding more power (CPU, RAM, storage) to handle larger workloads.

- How it works: You enhance the existing server's performance by upgrading hardware resources like memory, processors, or storage.

- Use case: Useful when the current system can handle more load by just increasing its resources.

- Advantages:

  - Easier to implement (no need to modify existing software architecture).

  - Ideal for applications that need higher performance on one server.

- Disadvantages:

  - Limited by the physical capacity of the server.

  - More expensive as hardware upgrades can become costly.

### Horizontal Scaling (Scaling Out):

Horizontal scaling involves adding more machines (nodes) to a system to distribute the load across multiple servers.

- How it works: Instead of upgrading one machine, you add multiple machines to your system. The workload is split among these machines.

- Use case: Common in distributed systems and cloud environments, like Hadoop or Spark clusters, where data is processed in parallel across many nodes.

- Advantages:

  - Can scale infinitely by adding more machines.

  - More fault-tolerant (if one machine fails, others continue working).

- Disadvantages:

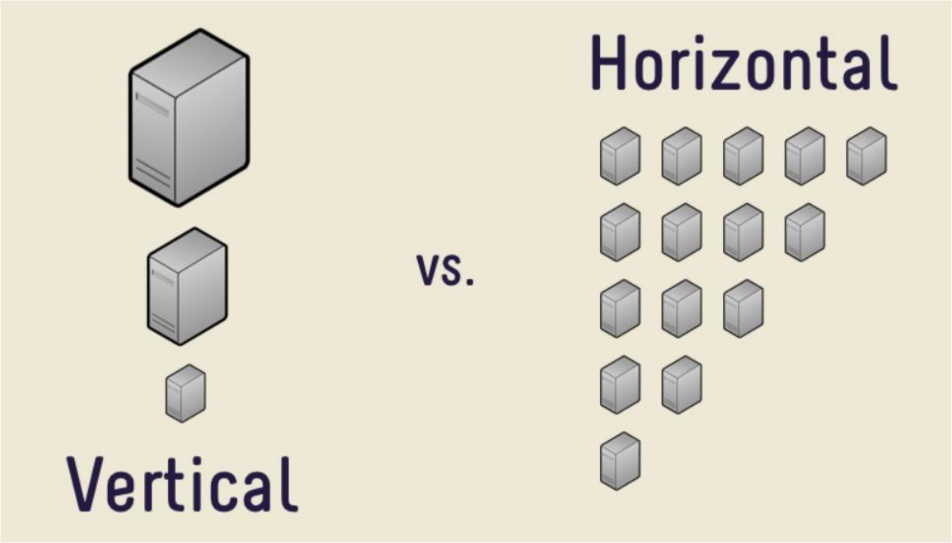
  - Requires more complex architecture and data distribution strategies.

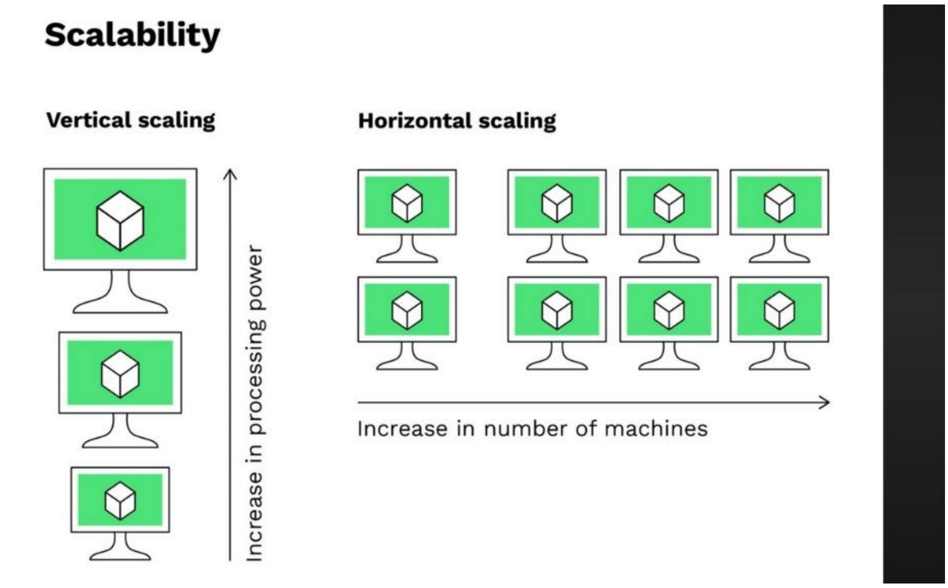
  - Managing and synchronizing multiple nodes can be more challenging.

### Summary in Data Engineering:

- Vertical Scaling is often suitable for monolithic applications that need more power in a single node but can’t handle distributed processing.

- Horizontal Scaling is ideal for Big Data systems, like Hadoop or Spark, which distribute tasks across many machines for parallel processing, making it scalable and cost-effective.





**Apache Airflow Architecture** (in simple words):

Apache Airflow is an open-source tool that helps manage and automate workflows or data pipelines. It schedules, monitors, and executes tasks in an organized way.

Key Components:

DAG (Directed Acyclic Graph):

This is the blueprint or map of the entire workflow. It shows how tasks are connected and their order of execution (i.e., which task runs first, next, and so on).

DAGs ensure that there are no loops (i.e., once a task is done, it doesn’t repeat unnecessarily).

Tasks:

Tasks are the individual units of work in a DAG. Each task could be something like moving data, transforming data, or running a machine learning model.

Airflow executes these tasks one by one or in parallel, depending on how you design the DAG.

Scheduler:

The scheduler is responsible for looking at all the DAGs and determining when to run each task.

It checks the DAG and starts tasks when they are supposed to run based on time (e.g., hourly, daily) or when dependencies are satisfied.

Executor:

The executor is in charge of actually running the tasks. Different executors handle this in different ways (e.g., running tasks on your machine or distributing them across a cluster).

Popular executors include the LocalExecutor (for local tasks) and CeleryExecutor (for distributed tasks across multiple machines).

Metadata Database:

Airflow keeps track of everything (DAGs, task states, logs) in a metadata database (usually a SQL database). This database stores the status of tasks (e.g., success, failure, or running).

Web Interface:

The Airflow web interface provides a user-friendly way to monitor DAGs and tasks. You can see which tasks have succeeded or failed and trigger workflows manually if needed.

Workers:

Workers are the machines that actually run the tasks. When a task needs to be executed, the executor assigns it to a worker machine. The worker does the job and reports back whether it succeeded or failed.

Imagine you have a workflow to:

Extract data from an API.

Clean the data.

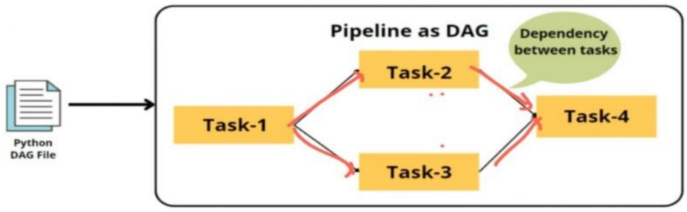
Load it into a database.

With Airflow:

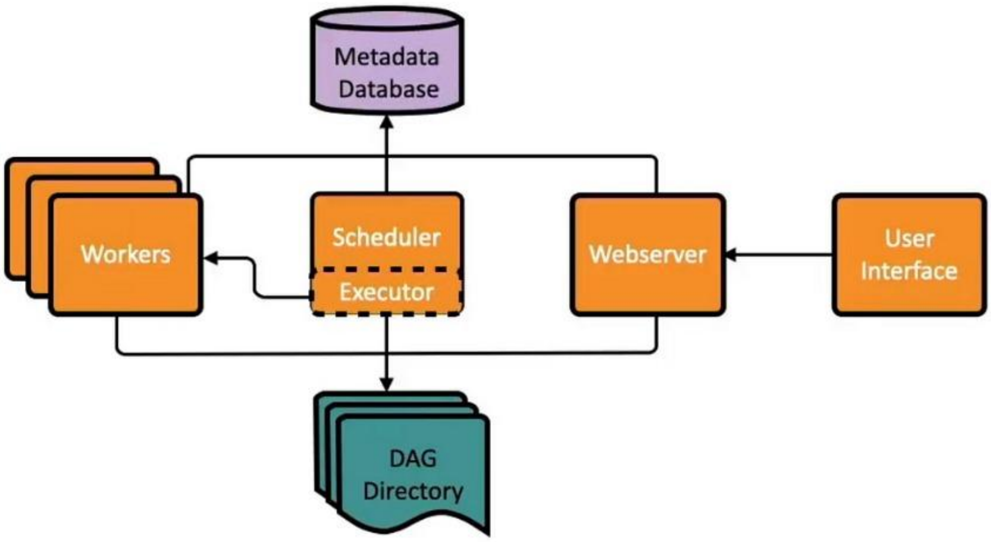
You define a DAG where Task 1 is "Extract", Task 2 is "Clean", and Task 3 is "Load".

Airflow's scheduler will see that Task 1 needs to run, send it to a worker, and once it's done, move on to Task 2 and then Task 3.

You can watch all of this happen on the web interface



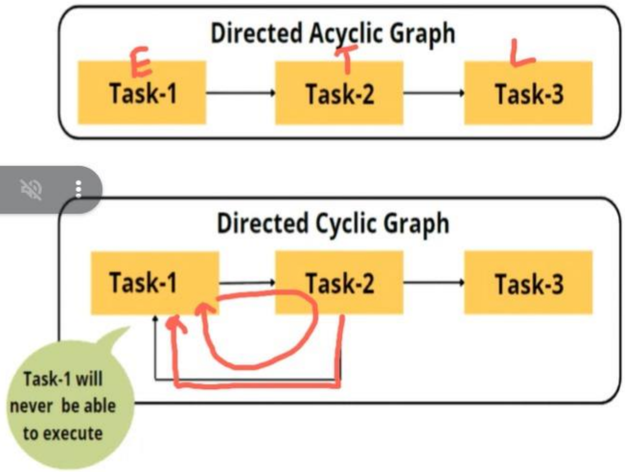
## Airflow Architecture



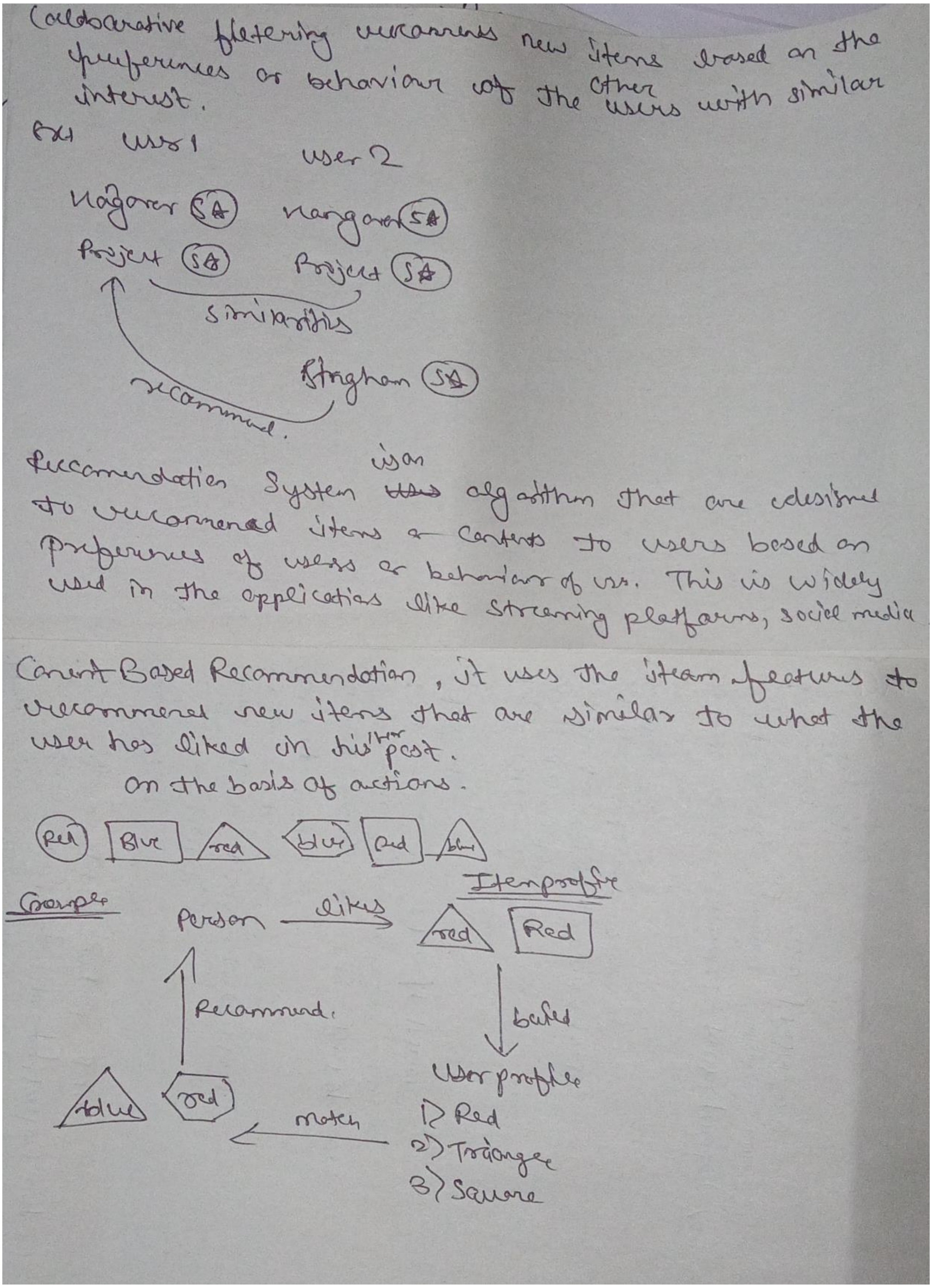
## Why Directed Acyclic Graphs?

Directed Acyclic Graph (DAG) comprises directed edges, nodes, and no loop or cycles. Acyclic means there are no circular dependencies in the graph between tasks.

Circular dependency creates a problem in task execution. for example, if task-1 depends upon task-2 and task-2 depends upon task-1 then this situation will cause deadlock and leads to logical inconsistency.



**Content based , collaborative and Recommendation systems**



Collaborative filtering recommends new items based on the preferences or behaviour of the other users with similar interest.

Ex. user 1 user 2

Nagarer (5★) Nangarer (5★)

Project (5★) Project (5★)

similarities

Stngham (5★)

recommend.

Reccomendation System ~~uses~~ is an algorithm that are designed to recommend items & contents to users based on preferences of users or behaviour of usr. This is widely used in the applications like streaming platforms, social media.

Conent Based Recommendation, it uses the item features to recommend new items that are similar to what the user has liked in his/her past.

On the basis of actions.

Red Blue red blue Red blue

Example

Person likes

Itemprofile

red Red

based

User profile

1) Red

2) Triangle

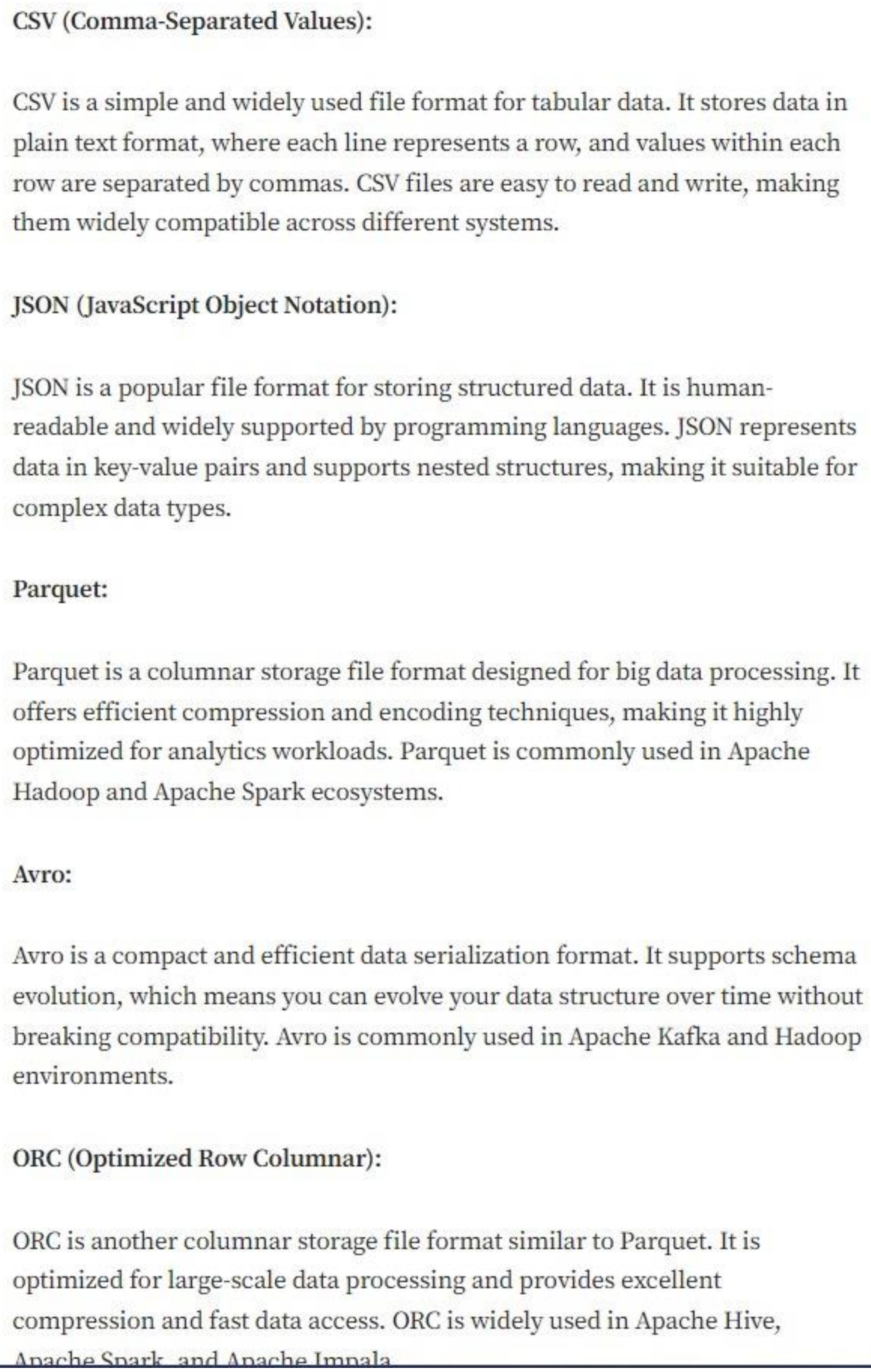
3) Square

match

Recommend.

blue red

## File Formats



**CSV (Comma-Separated Values):**

CSV is a simple and widely used file format for tabular data. It stores data in plain text format, where each line represents a row, and values within each row are separated by commas. CSV files are easy to read and write, making them widely compatible across different systems.

**JSON (JavaScript Object Notation):**

JSON is a popular file format for storing structured data. It is human-readable and widely supported by programming languages. JSON represents data in key-value pairs and supports nested structures, making it suitable for complex data types.

**Parquet:**

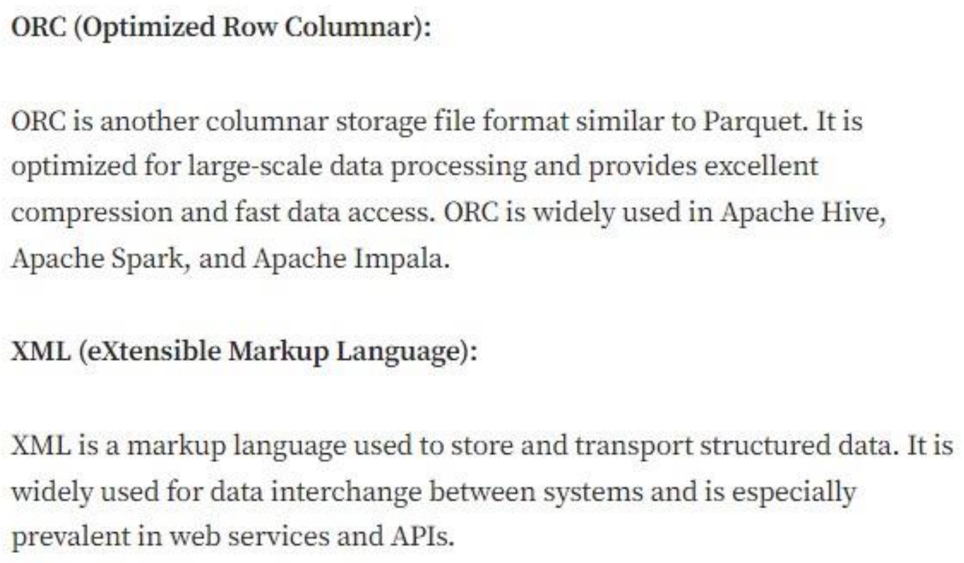
Parquet is a columnar storage file format designed for big data processing. It offers efficient compression and encoding techniques, making it highly optimized for analytics workloads. Parquet is commonly used in Apache Hadoop and Apache Spark ecosystems.

**Avro:**

Avro is a compact and efficient data serialization format. It supports schema evolution, which means you can evolve your data structure over time without breaking compatibility. Avro is commonly used in Apache Kafka and Hadoop environments.

**ORC (Optimized Row Columnar):**

ORC is another columnar storage file format similar to Parquet. It is optimized for large-scale data processing and provides excellent compression and fast data access. ORC is widely used in Apache Hive, Apache Spark, and Apache Impala.



**ORC (Optimized Row Columnar):**

ORC is another columnar storage file format similar to Parquet. It is optimized for large-scale data processing and provides excellent compression and fast data access. ORC is widely used in Apache Hive, Apache Spark, and Apache Impala.

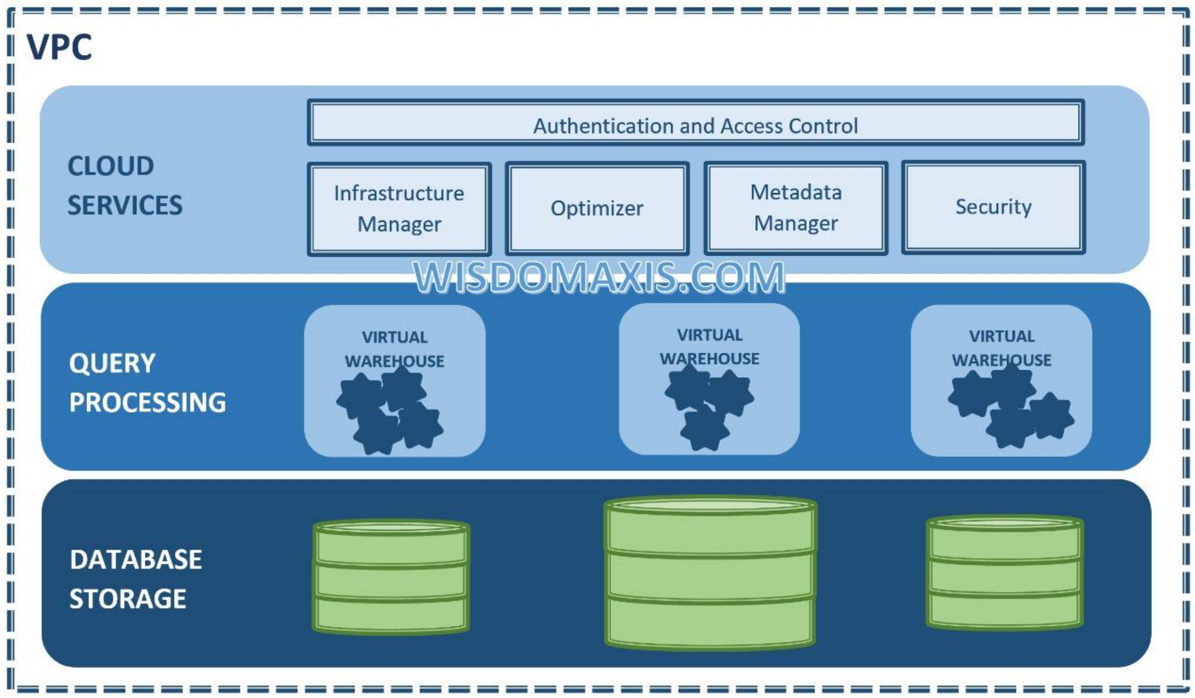
**XML (eXtensible Markup Language):**

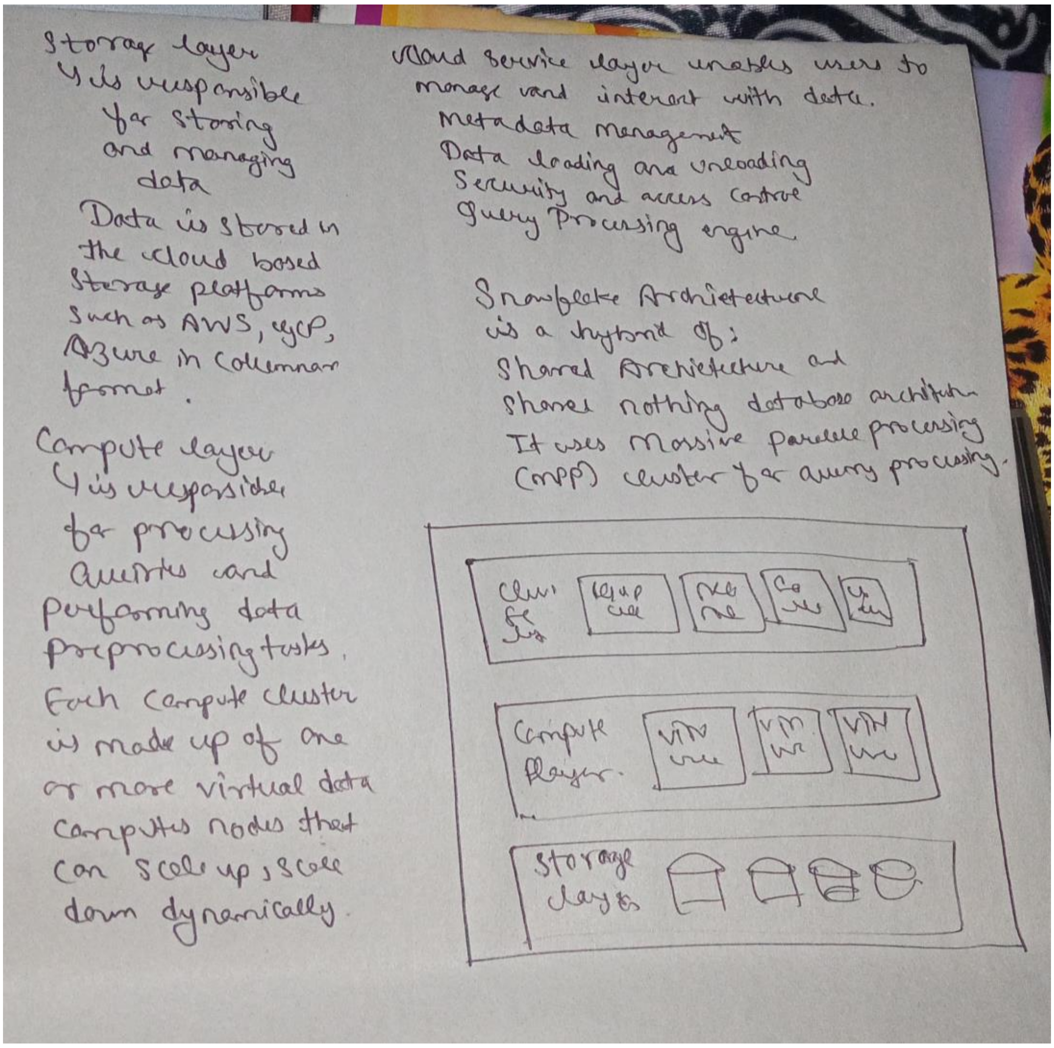
XML is a markup language used to store and transport structured data. It is widely used for data interchange between systems and is especially prevalent in web services and APIs.

---

### Snowflake Archietecture

Snowflake is a cloud-based data warehousing platform designed for storing, processing, and analyzing large volumes of data. It is unique because it separates storage and compute, enabling organizations to scale resources independently based on their needs.





Storage layer
↳ is responsible for storing and managing data
Data is stored in the cloud based storage platforms such as AWS, GCP, Azure in columnar format.

Compute layer
↳ is responsible for processing queries and performing data preprocessing tasks.
Each compute cluster is made up of one or more virtual data computes nodes that can scale up, scale down dynamically.

Cloud service layer enables users to manage and interact with data.
Metadata management
Data loading and unloading
Security and access control
Query Processing engine.

Snowflake Architecture is a hybrid of:
Shared Architecture and
Shared nothing database architecture
It uses Massive parallel processing (MPP) cluster for query processing.

---

**AWS Services**

**AWS (Amazon Web Services) provides several data engineering services to help manage, process, and analyze data effectively. Here's a simple explanation of its key services with examples:**

### 1. *Data Storage*

- *Amazon S3 (Simple Storage Service):* A service to store large amounts of data securely and cost-effectively.

  *Example:* Storing customer order history or images for an e-commerce website.

- *Amazon RDS (Relational Database Service):* For storing structured data in databases like MySQL or PostgreSQL.

  *Example:* Storing product details or user information.

### 2. *Data Processing*

- *AWS Glue:* A tool to clean, transform, and prepare data for analysis.

*Example:* Converting raw sales data into a format suitable for machine learning.

- *Amazon EMR (Elastic MapReduce):* Processes big data using tools like Apache Spark and Hadoop.

*Example:* Analyzing large logs from a website to find user behavior trends.

### 3. *Data Streaming*

- *Amazon Kinesis:* Collects and processes streaming data in real time.

*Example:* Monitoring live stock prices or tracking user activity on a website.

### 4. *Data Warehousing*

- *Amazon Redshift:* A fast, scalable data warehouse for running complex queries and analysis.

*Example:* Analyzing customer purchase patterns to improve marketing strategies.

### 5. *Data Migration*

- *AWS DMS (Database Migration Service):* Helps migrate data from one database to another with minimal downtime.

*Example:* Moving data from an on-premises database to AWS.

### 6. *Data Integration*

- *Amazon Data Pipeline:* Automates the transfer and transformation of data between different AWS services.

*Example:* Moving data from Amazon S3 to Redshift for analysis.

### Example Use Case:

Imagine you run an online store:

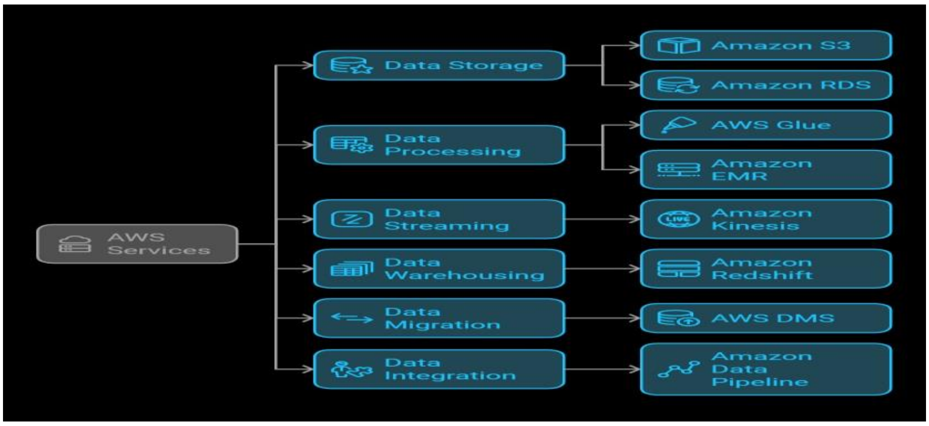
1. *Amazon S3* stores product images and transaction logs.

2. *AWS Glue* cleans and organizes sales data.

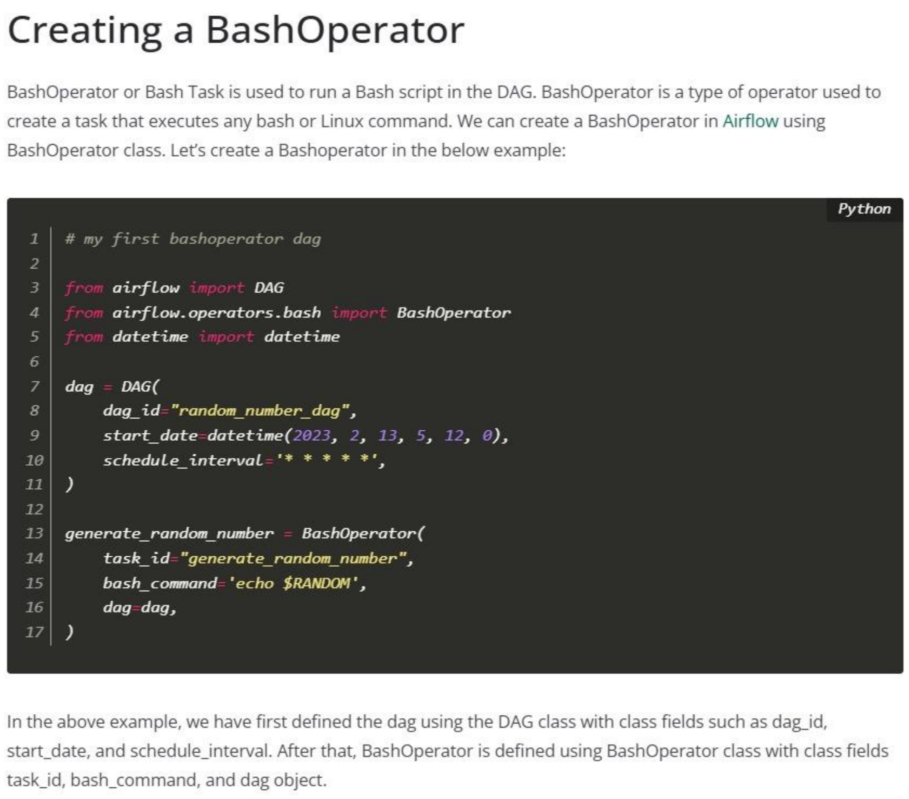
3. *Amazon Redshift* analyzes sales trends for better decision-making.

4. *Amazon Kinesis* monitors real-time user activity during a sale.

AWS data engineering services make it easy to handle large data workflows efficiently!



**Python and Bash Operators**

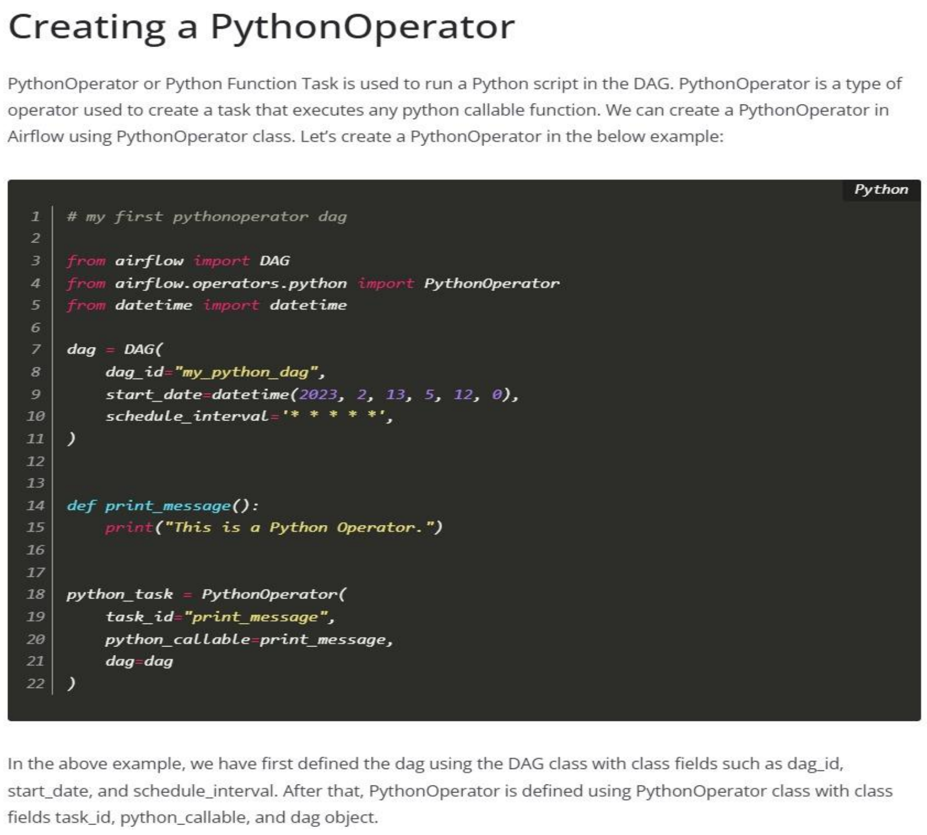


## Creating a BashOperator

BashOperator or Bash Task is used to run a Bash script in the DAG. BashOperator is a type of operator used to create a task that executes any bash or Linux command. We can create a BashOperator in Airflow using BashOperator class. Let's create a Bashoperator in the below example:

```python
# my first bashoperator dag

from airflow import DAG
from airflow.operators.bash import BashOperator
from datetime import datetime

dag = DAG(
    dag_id="random_number_dag",
    start_date=datetime(2023, 2, 13, 5, 12, 0),
    schedule_interval='* * * * *',
)

generate_random_number = BashOperator(
    task_id="generate_random_number",
    bash_command='echo $RANDOM',
    dag=dag,
)
```

In the above example, we have first defined the dag using the DAG class with class fields such as dag_id, start_date, and schedule_interval. After that, BashOperator is defined using BashOperator class with class fields task_id, bash_command, and dag object.



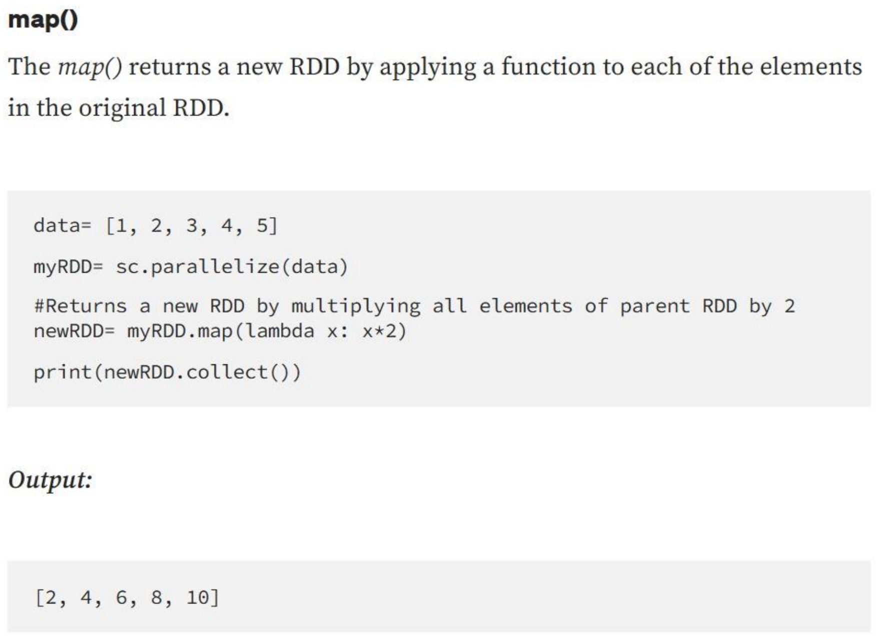
## Creating a PythonOperator

PythonOperator or Python Function Task is used to run a Python script in the DAG. PythonOperator is a type of operator used to create a task that executes any python callable function. We can create a PythonOperator in Airflow using PythonOperator class. Let's create a PythonOperator in the below example:

```python
# my first pythonoperator dag

from airflow import DAG
from airflow.operators.python import PythonOperator
from datetime import datetime

dag = DAG(
    dag_id="my_python_dag",
    start_date=datetime(2023, 2, 13, 5, 12, 0),
    schedule_interval='* * * * *',
)


def print_message():
    print("This is a Python Operator.")


python_task = PythonOperator(
    task_id="print_message",
    python_callable=print_message,
    dag=dag
)
```

In the above example, we have first defined the dag using the DAG class with class fields such as dag_id, start_date, and schedule_interval. After that, PythonOperator is defined using PythonOperator class with class fields task_id, python_callable, and dag object.

**PYSPARK functions**



**map()**

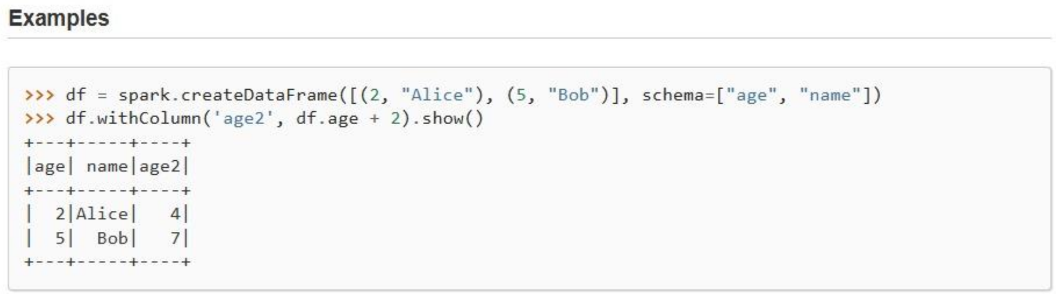
The *map()* returns a new RDD by applying a function to each of the elements in the original RDD.

```
data= [1, 2, 3, 4, 5]

myRDD= sc.parallelize(data)

#Returns a new RDD by multiplying all elements of parent RDD by 2
newRDD= myRDD.map(lambda x: x*2)

print(newRDD.collect())
```

*Output:*

```
[2, 4, 6, 8, 10]
```

**withColumn()**



Examples

```
>>> df = spark.createDataFrame([(2, "Alice"), (5, "Bob")], schema=["age", "name"])
>>> df.withColumn('age2', df.age + 2).show()
+---+-----+----+
|age| name|age2|
+---+-----+----+
|  2|Alice|   4|
|  5|  Bob|   7|
+---+-----+----+
```

countByKey

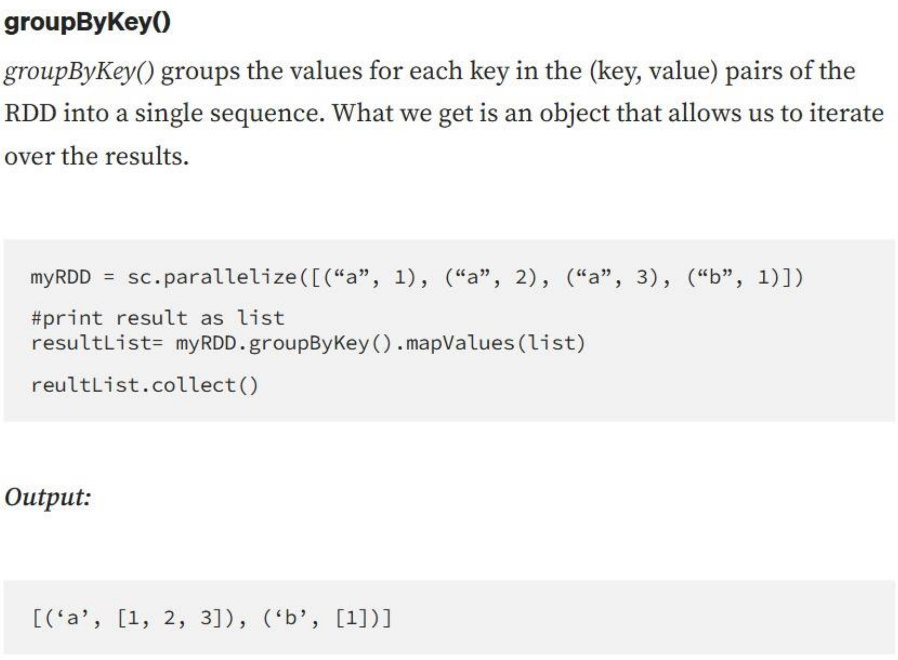
Counts the number of occurrences for each key in an RDD.

sortByKey

Sorts the RDD by key in ascending or descending order.

reduceByKey

Aggregates values for each key using a specified function (e.g., sum, max, etc.).



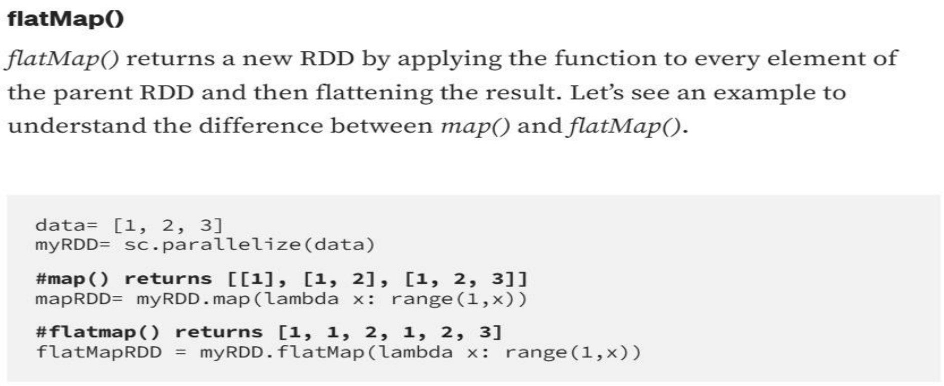
### groupByKey()

*groupByKey()* groups the values for each key in the (key, value) pairs of the RDD into a single sequence. What we get is an object that allows us to iterate over the results.

```
myRDD = sc.parallelize([(“a”, 1), (“a”, 2), (“a”, 3), (“b”, 1)])

#print result as list
resultList= myRDD.groupByKey().mapValues(list)

reultList.collect()
```

***Output:***

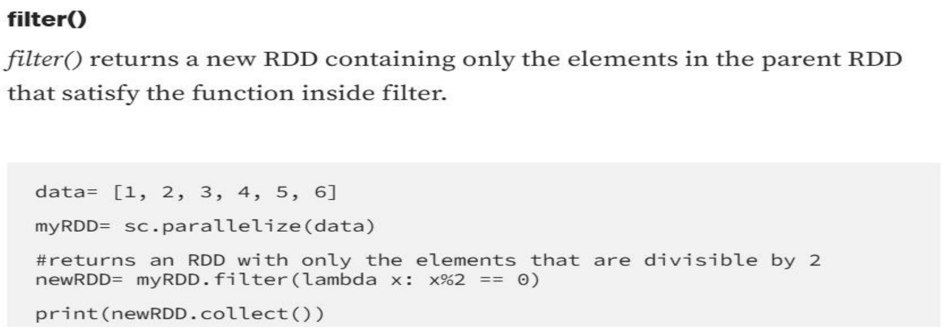
```
[(‘a’, [1, 2, 3]), (‘b’, [1])]
```



### flatMap()

*flatMap()* returns a new RDD by applying the function to every element of the parent RDD and then flattening the result. Let’s see an example to understand the difference between *map()* and *flatMap()*.

```
data= [1, 2, 3]
myRDD= sc.parallelize(data)

#map() returns [[1], [1, 2], [1, 2, 3]]
mapRDD= myRDD.map(lambda x: range(1,x))

#flatmap() returns [1, 1, 2, 1, 2, 3]
flatMapRDD = myRDD.flatMap(lambda x: range(1,x))
```



### filter()

*filter()* returns a new RDD containing only the elements in the parent RDD that satisfy the function inside filter.

```
data= [1, 2, 3, 4, 5, 6]

myRDD= sc.parallelize(data)

#returns an RDD with only the elements that are divisible by 2
newRDD= myRDD.filter(lambda x: x%2 == 0)

print(newRDD.collect())
```