

Chapter 2 : Big data - No sql

MapReduce vs Apache Spark vs Apache Flink

Technology	MapReduce	Apache Spark	Apache Flink
Execution Models	Batch processing	Batch + Near real-time processing	Batch + Real-time processing
Interactive Analysis	No support for interactive analysis	Support interactive analysis	Support interactive analysis
Iterative Processing	Not suitable for iterative processing	Suitable for iterative processing	Native iterative processing



1. Execution Models

- **MapReduce:**

Designed for batch processing only. Each job runs independently and reads/writes from disk.

Example: Running a daily report on website traffic using historical logs.

- **Spark:**

Supports batch processing and near real-time streaming using Spark Streaming.

Example: Monitoring user clickstream data with ~5 second delay.

- **Flink:**

Designed for true real-time streaming and batch as a special case.
Example: Fraud detection in online transactions, reacting instantly.

2. Interactive Analysis

- MapReduce:

Not suitable for quick data exploration. Every job must be submitted, compiled, and run.

Example: Wanting to run a quick SQL query takes several minutes.

- Spark:

Offers Spark Shell and Spark SQL for fast, interactive queries.

Example: A data scientist quickly exploring sales data using Spark SQL.

- Flink:

Also supports interactive analysis with fast response times.

Example: Adjusting live data dashboards with near-zero delay.

3. Iterative Processing

- MapReduce:

Poor for iterative algorithms (e.g., machine learning), because each iteration writes to disk.

Example: Running PageRank would be very slow.

- Spark:

Efficient for iterations using RDDs and in-memory caching.

Example: Training a machine learning model with MLlib.

- Flink:

Has native support for iterations with low latency. Ideal for algorithms that loop many times.

Example: Real-time graph processing or complex event detection.

Stream Processing

Streaming systems such as Apache Storm, Spark Streaming, and Apache Samza enable real-time, distributed computation on unbounded streams of data. These systems can continuously process incoming data and emit results either to Hadoop storage (HDFS) or to external systems for further analysis or action.

Search

The Solr search platform can be deployed on a Hadoop cluster, allowing it to:

- Index documents as they are added to HDFS
- Serve search queries using indexes stored within HDFS

Another widely used tool is Elasticsearch, which is also leveraged for information retrieval in distributed systems, often providing fast, scalable full-text search capabilities.

Benefits of Stream Processing

1. Low Latency

Stream processing enables low-latency data processing, which is essential when your application needs to respond quickly—on a timescale of minutes, seconds, or even milliseconds.

To achieve this, streaming systems keep state in memory, allowing them to deliver results in real time or near real time with acceptable performance.

2. Efficiency through Incremental Updates

Stream processing is often more efficient than running repeated batch jobs. Instead of reprocessing the entire dataset, a streaming system can incrementally update the result as new data arrives.

This makes it ideal for use cases like real-time analytics, monitoring dashboards, and event-driven applications.

Why Do We Still Study MapReduce?

Despite the emergence of more modern data processing frameworks on top of Hadoop, MapReduce still holds value, particularly for batch processing tasks. Understanding how MapReduce works is important because it introduces several fundamental concepts that are applicable across many distributed data processing systems.

These concepts include:

- The idea of input formats
- How a dataset is split into chunks for parallel processing
- The map and reduce functions as a model of computation

Studying MapReduce provides a strong foundation for understanding newer technologies like Apache Spark and Flink, which often build upon or extend these core ideas.

Hadoop vs Grid Computing – Summary

- High-Performance Computing (HPC) and Grid Computing have been used for large-scale data processing for many years, often relying on low-level APIs like MPI (Message Passing Interface).
 - In Grid Computing / HPC:
 - Work is distributed across a cluster of machines that access a shared file system via a Storage Area Network (SAN).
 - This setup works well for compute-intensive tasks, but struggles with data-intensive workloads due to network bandwidth bottlenecks.
 - Programming is low-level, requiring developers to handle task failure, checkpoints, and recovery manually.
 - Resources are usually expensive and infrastructure is complex to manage.
 - In Hadoop:
 - Designed specifically for data-intensive processing.
 - Uses high-level programming models like MapReduce, which automatically handles failures and reschedules failed tasks.

- Emphasizes data locality – moving computation to where the data is stored – reducing network traffic and improving performance.
 - Models network topology to minimize data movement and conserve bandwidth, making it more efficient in large-scale environments.
-

Key Differences

- Programming Level:
 - Grid: Low-level
 - Hadoop: High-level (MapReduce)
 - Fault Tolerance:
 - Grid: Manual recovery
 - Hadoop: Automatic
 - Cost & Infrastructure:
 - Grid: High cost
 - Hadoop: Commodity hardware
 - Data Locality:
 - Grid: Data moved to compute
 - Hadoop: Compute moved to data
-

Example Use Case

- Grid Computing:

Running a climate simulation across thousands of CPU cores in a research lab.

- Hadoop:

Processing hundreds of terabytes of log files daily to generate analytics reports in a tech company.

--MapReduce vs traditional database

	Traditional RDBMS	MapReduce
Data size	Gigabytes	Petabytes
Access	Interactive and batch	Batch
Updates	Read and write many times	Write once, read many times
Transactions	ACID	None
Structure	Schema-on-write	Schema-on-read
Integrity	High	Low
Scaling	Nonlinear	Linear

Hadoop Framework Components

1. MapReduce

A parallel processing engine designed for handling large-scale data sets. It implements the MapReduce programming model, allowing distributed computation across multiple nodes efficiently.

2. YARN (Yet Another Resource Negotiator)

A resource management and job scheduling framework within Hadoop.

YARN manages cluster resources and coordinates the execution of various applications running on the Hadoop ecosystem.

3. HDFS (Hadoop Distributed File System)

A high-performance, distributed file system optimized for storing and processing very large files across a cluster of machines. HDFS ensures data reliability and fault tolerance through replication.

MapReduce Framework

- MapReduce operates by dividing the data processing task into two main phases:

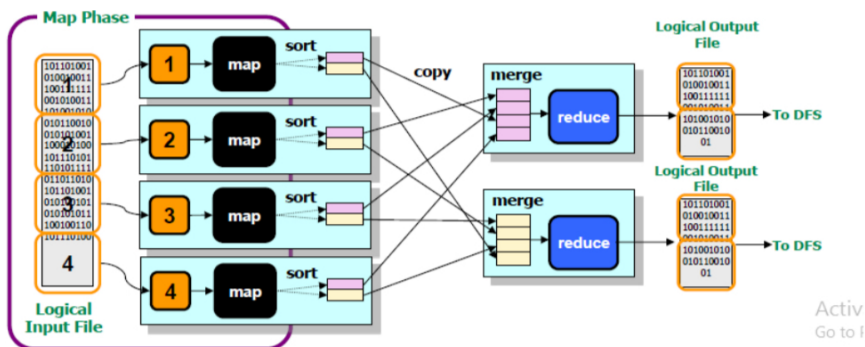
Map Phase and Reduce Phase.

Each phase works with key-value pairs as both input and output, and the data types for these pairs are defined by the programmer.

- The programmer must implement two core functions:
 - Map Function – processes input data and produces intermediate key-value pairs.
 - Reduce Function – aggregates and processes the intermediate data to produce final results.
- The Mapper class is a generic type that uses four formal type parameters, which define:
 - Input Key Type
 - Input Value Type
 - Output Key Type
 - Output Value Type

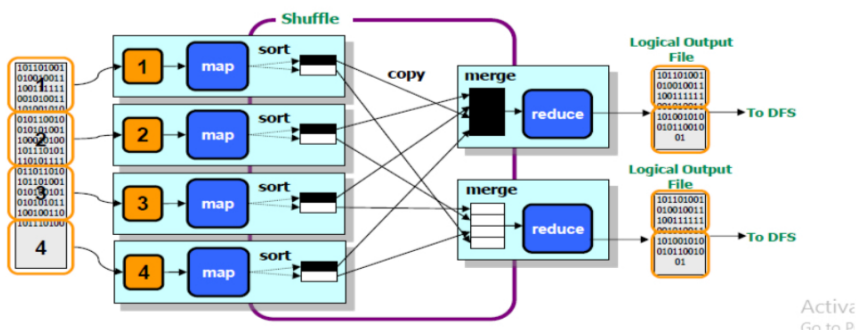
Mappers:

- Mappers
 - Small program (typically), distributed across the cluster, local to data
 - Handed a *portion* of the input data (called a split)
 - Each mapper parses, filters, or transforms its input
 - Produces grouped $\langle \text{key}, \text{value} \rangle$ pairs



Shuffle:

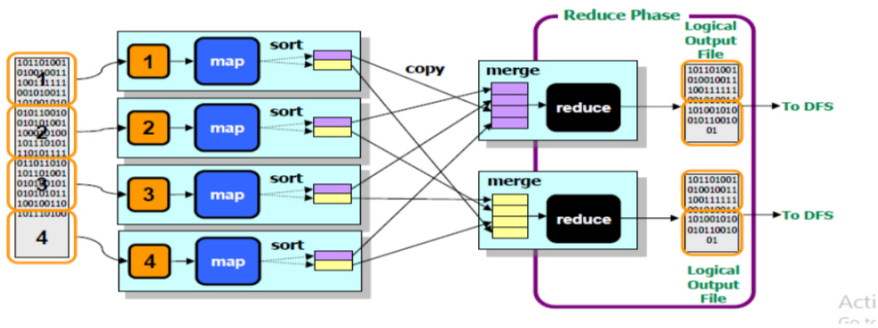
- The output of each mapper is locally grouped together by **key**
- One node is chosen to process data for each unique **key**
- All of this movement (shuffle) of data is transparently orchestrated by MapReduce



reducers:

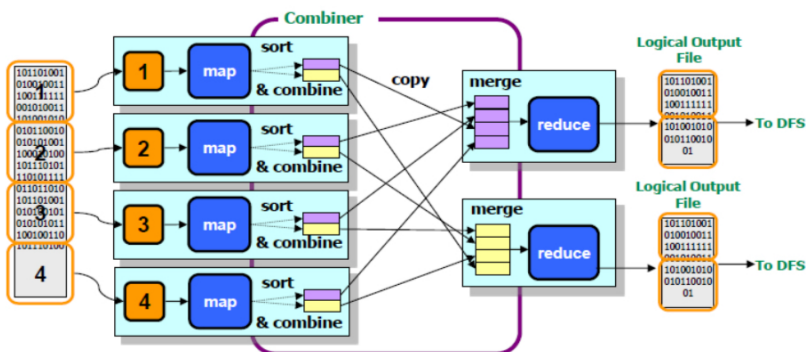
- Reducers

- Small programs (typically) that aggregate all of the values for the key that they are responsible for
- Each reducer writes output to its own file

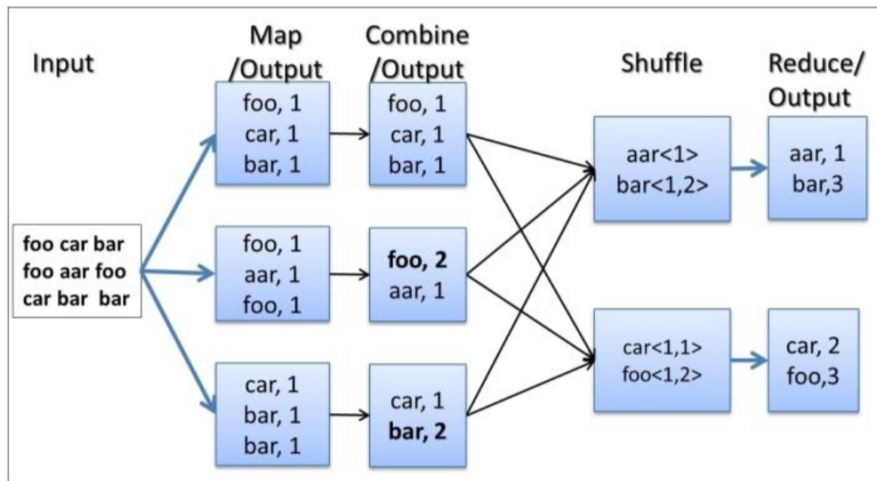


Combiners:

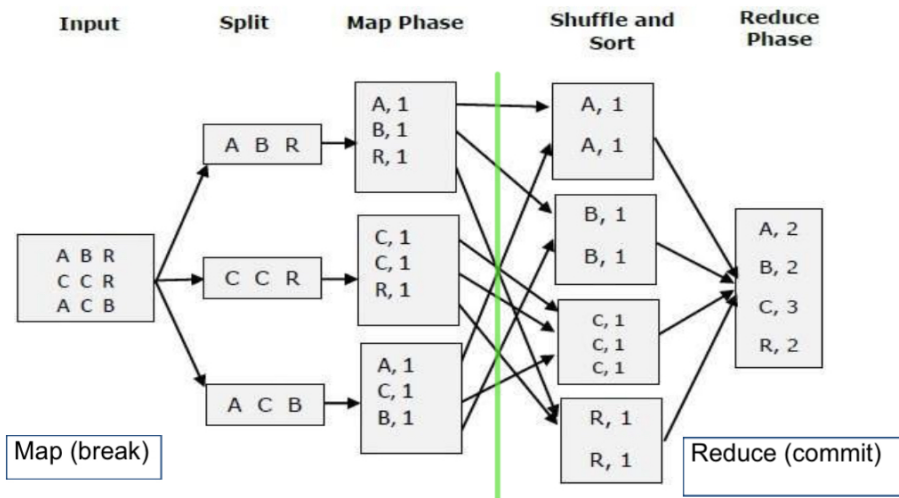
- The data that will go to each reduce node is sorted and merged before going to the reduce node, pre-doing some of the work of the receiving reduce node in order to minimize network traffic between map and reduce nodes.



word count example
with combiner:



word count example
without combiner:



Input Splits in Hadoop MapReduce

Hadoop divides the input data of a MapReduce job into fixed-size units called input splits (or simply splits). For each split, Hadoop creates a

separate map task, which executes the user-defined map function on each record within that split.

Having a larger number of smaller splits improves parallelism and load balancing. This is because smaller splits allow faster machines in the cluster to process more tasks over time, while slower machines handle fewer. As a result, overall job performance is optimized, and cluster resources are better utilized, good split size tends to be the size of an HDFS block, which is 128 MB by default

Data locality:

Hadoop does its best to run the map task on a node

If the split spanned two blocks, it would be unlikely that any HDFS node stored both blocks, so some of the split would have to be transferred across

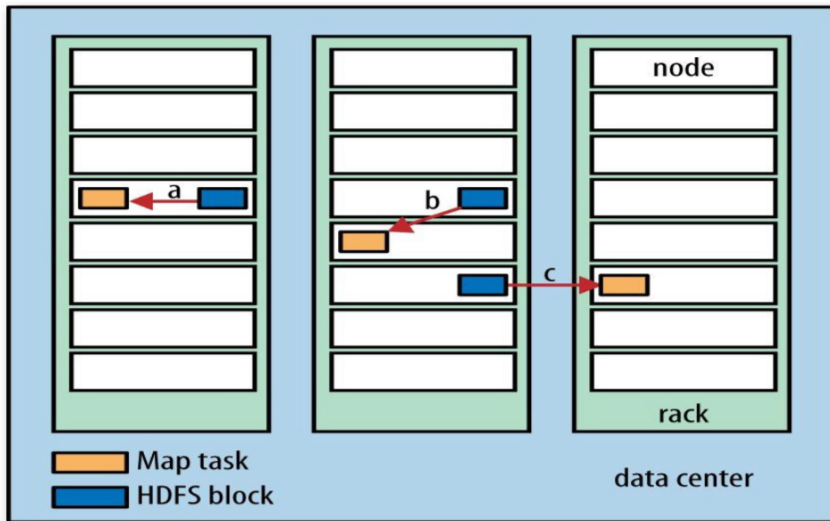
the network to the node running the map task, which is clearly less efficient

than running the whole map task using local data

- Map tasks write their output to the local disk, not to HDFS. Why is this?





Map output is intermediate output: it's processed by reduce tasks to produce the final output, and once, Reduce tasks don't have the

advantage of data locality; the input to a single reduce task is normally the output from all mappers



There are three main Java classes provided in Hadoop to read data in MapReduce:

1. InputSplitter
2. RecordReader
3. InputFormat

InputFormat	Description	Key	Value	Fil type
TextInputFormat 	Default format; reads lines of text files	The byte offset of the line	The line contents	Text
KeyValueInputFormat 	Parses lines into (K, V) pairs	Everything up to the first tab character	The remainder of the line	Text
NLineInputFormat 	mappers receives a fixed number of lines of input	The byte offset of the line	The line contents	Text
SequenceFileInputFormat 	A Hadoop-specific high-performance binary format	user-defined	user-defined	Binary

Why does Hadoop use classes such as `Writable` and `Text` instead of `int` and `string` ?

because java `Serializable` is too big or too heavy for Hadoop, `Writable` can

serialize the Hadoop Object in a very light way

Why & Where Hadoop Is Used / Not Used

What Hadoop Is Good For

1. Processing Massive Amounts of Data

Hadoop excels at handling huge datasets by distributing the workload across many machines (parallel processing).

2. Handling Diverse Data Types

It can work with structured, semi-structured, and unstructured data, making it suitable for a wide range of applications.

3. Cost Efficiency

Runs on inexpensive, commodity hardware, which reduces infrastructure costs compared to traditional systems.

✗ What Hadoop Is Not Good For

1. Transaction Processing

Not suitable for real-time, random-access operations like those required in banking or inventory systems.

2. Non-Parallel Workloads

If a task cannot be parallelized, Hadoop offers no advantage and may introduce unnecessary overhead.

3. Low-Latency Requirements

Hadoop is not designed for applications that need instant response times (e.g., real-time dashboards or user-facing APIs).

4. Processing Many Small Files

Hadoop performs poorly with a large number of small files, as it's optimized for large, continuous datasets.

5. Compute-Intensive Tasks with Minimal Data

Tasks that involve heavy computation but only small amounts of data (e.g., simulations, deep learning training) are better handled by high-performance computing (HPC) or GPU-based systems.

Hadoop MapReduce Techniques

Hadoop MapReduce supports two important techniques to optimize job

execution and data sharing:

1. Using the Job Configuration

This technique involves setting configuration parameters in the job setup to control how the MapReduce job runs. For example, you can specify input/output paths, memory settings, or custom parameters that the mapper and reducer use.

2. Using Distributed Cache

Distributed Cache allows you to distribute read-only data files (like lookup tables or static datasets) to all nodes running map or reduce tasks. These files are cached locally on each node, improving access speed and reducing network traffic.

Cust ID	First Name	Last Name	Age	Profession
4000001	Kristina	Chung	55	Pilot
4000002	Paige	Chen	74	Teacher
4000003	Sherri	Melton	34	Firefighter
4000004	Gretchen	Hill	66	Engineer
*****	*****	*****	*****	*****

Trans ID	Date	Cust ID	Amount	Game Type	Equipment	City	State	Mode
0000000	06-26-2011	4000001	40.33	Exercise & Fitness	Cardio Machine Accessories	Clarksville	Tennessee	credit
0000001	05-05-2011	4000002	198.44	Exercise & Fitness	Weightlifting Gloves	Long Beach	California	credit
0000002	06-17-2011	4000002	5.58	Exercise & Fitness	Weightlifting Machine Accessories	Anaheim	California	credit
0000003	06-14-2011	4000003	198.19	Gymnastics	Gymnastics Rings	Milwaukee	Wisconsin	credit
0000004	12-28-2011	4000002	98.81	Team Sports	Field Hockey	Nashville	Tennessee	credit
0000005	02-14-2011	4000004	193.63	Outdoor Recreation	Camping & Backpacking & Hiking	Chicago	Illinois	credit
0000006	10-17-2011	4000005	27.89	Puzzles	Jigsaw Puzzles	Charleston	South Carolina	credit
*****	*****	*****	*****	*****	*****	*****	*****	*****

Join in MapReduce

In MapReduce, a join operation combines related records from two or more datasets based on a common key. Since MapReduce processes data in parallel, implementing joins requires careful design.

There are several types of joins in MapReduce:

1. Reduce-Side Join

Both datasets are sent to the reducer, where records with the same key are combined. This is the most common and flexible approach but can be expensive due to data shuffling.

Example: Joining customer data with their orders by customer ID.

2. Map-Side Join

Both datasets are pre-sorted and partitioned identically so the join can be performed during the map phase without data shuffling. This is more efficient but requires preprocessing.

Example: Joining a large dataset with a smaller, static dataset loaded in

memory.

3. Semi-Join and Other Variants

Optimizations that reduce data transfer by filtering records before joining.

Output from mapper1 as *Example*: [4000001, cname Kristina], [4000002, cname paige].

Output from mapper2 as *Example*: [4000001, camount 40.33], [4000002, camount 198.44].

[After shuffling and sorting step](#)

{4000001 – [(cname kristina), (camount 40.33), (camount 47.05),...]};

Differences Between SQL, NoSQL

1. SQL (Structured Query Language)

- Type: Relational databases (e.g., MySQL, PostgreSQL, Oracle).
- Data Model: Organized into tables with rows and columns; supports relationships between tables.
- Query Language: Uses SQL for complex queries, joins, aggregations, and transactions.
- Strengths: Ideal for structured data requiring strong consistency and ACID transactions.
- Example: A customer order management system with clear relationships between customers, orders, and invoices.

2. NoSQL (Not Only SQL)

- **Type:** Non-relational databases, including document stores, key-value stores, column-family stores, and graph databases (e.g., MongoDB, Cassandra, Redis).
- **Data Model:** Flexible and schema-less, does not rely on tables.
- **Query Language:** Varies by database type; often uses APIs or custom query languages.
- **Strengths:** Designed for large-scale, distributed data with high scalability and flexibility; handles unstructured or semi-structured data well.
- **Example:** A social media app storing diverse and evolving user data without fixed schemas.

1. Document Database

Structure:

- Data is organized into collections, each containing multiple documents.
- A document is a set of key-value pairs (often in JSON or BSON format). Each document has a unique key (e.g., `_id` in MongoDB) and can contain nested or complex data structures (arrays, sub-documents, etc.).
- Collections do not enforce a rigid schema: documents within the same collection can have different fields.

Use Cases:

- **Content Management Systems & Blogging Platforms:** Flexible

schema allows storing posts, comments, user information, and related metadata all in one place.

- **E-Commerce Platforms:** Product catalogs with variable attributes (size, color, specs) can be stored as documents without having to alter a fixed table schema.
- **Web Applications:** Fast lookups and updates for user profiles, session data, shopping carts, and dynamic forms.
- **Real-Time Analytics & Reporting:** Aggregations and map-reduce operations over documents allow quick computation of counters, trends, and dashboards.

Example:

MongoDB stores customer profiles (name, address, purchase history) in a single document, while a product page might include nested arrays of reviews and ratings—making it simple to query or update any part of that structure without performing expensive joins.

2. Column Database

Structure:

- Data is organized into a keyspace, which contains one or more column families.
- A column family groups together a set of rows.
- Each row has a unique row key.
- Within each row, there can be a varying number of named

columns; each column is a key-value pair.

- Because related columns that are often accessed together live in the same column family, reads of large, sparse datasets can be highly efficient.

Use Cases:

- **Write-Heavy Workloads:** Systems that perform massive write operations (e.g., log collection, sensor data ingestion) benefit from fast sequential writes to disk.
- **Online Analytical Processing (OLAP):** Aggregating large datasets (e.g., time-series metrics, clickstream data) with high throughput.
- **Real-Time & Web Analytics:** Generating counters, dashboards, and reports on large volumes of events.
- **Scalable, High-Availability Storage:** Distributed across many nodes, column databases handle node failures gracefully while maintaining availability.

Example:

Cassandra stores IoT sensor readings in a table where each row key is a device ID, and each column name is a timestamp. Since each device can generate thousands of readings per day, storing them as columns under that row key lets you quickly retrieve all readings for a given device over a time range.

3. Key-Value Database

Structure:

- The simplest NoSQL model: data is stored as pairs of key → value.
- The key is an atomic identifier used to retrieve its associated value, which can be anything from a simple string to a complex object.
- No schema enforcement or indexing on fields within the value—only the top-level key is indexed.

Use Cases:

- **Session Management:** Storing session tokens, user preferences, or temporary state for web applications (fast reads/writes keyed by session ID).
- **Caching Layers:** In-memory key-value stores (e.g., Redis) act as a cache in front of a relational database, reducing load and improving response times.
- **Shopping Cart / Wish List:** Quickly storing each user's cart contents under a single key for instant retrieval and updates.
- **Feature Flags & Configuration:** Managing application-wide settings in a distributed system, allowing immediate rollouts or rollbacks keyed by feature names.

Example:

Redis can store a user's shopping cart as a hash under key `cart:<userID>`; each field in that hash is a product ID mapped to quantity. Retrieving or updating a user's cart requires only one key

lookup.

4. Graph Database

Structure:

- Data is modeled as nodes (entities) and edges (relationships).
- Nodes are analogous to rows in an RDBMS table but can hold multiple properties (key-value attributes).
- Edges explicitly represent relationships between nodes and can also carry properties.
- Labels categorize nodes (e.g., "Person," "Product"), and properties attach data to nodes or edges (e.g., name, age, timestamp).
- Because relationships are first-class citizens, traversing connections is very fast—no need for cost-intensive joins at query time.

Use Cases:

- **Social Networking:** Modeling users as nodes and friendships or follows as edges; finding mutual friends, recommendations, or degrees of separation is highly efficient.
- **Recommendation Engines:** Traversing purchased-together or viewed-together patterns to suggest products based on a customer's preferences.
- **Fraud Detection & Security:** Tracking financial transactions as nodes, with relationships representing transfers; quickly identifying suspicious loops or chains.

- **Network & Infrastructure Management:** Representing network devices and their connections, allowing fast impact analysis when a switch or router fails.
- **Bioinformatics & Knowledge Graphs:** Storing genes, proteins, or concepts as nodes with edges denoting interactions or relationships for complex queries.

Example:

Neo4j maintains a "Person" node for each user and "FOLLOWS" edges between them. To recommend new connections, a query can traverse two hops away—efficiently finding "friends of friends" without expensive SQL joins.

Graph Database :

Live Demo: LinkedIn-style Professional Graph

Step 1: Create Nodes (People, Companies, Skills)

```
CREATE (:Person {name: 'Nour', title: 'Software Engineer'});
```

```
CREATE (:Person {name: 'Mostafa', title: 'Data Analyst'});
```

```
CREATE (:Person {name: 'Salma', title: 'UX Designer'});
```

```
CREATE (:Company {name: 'TechCorp'});
```

```
CREATE (:Company {name: 'DataSolutions'});
```



```
CREATE (:Skill {name: 'Python'});  
CREATE (:Skill {name: 'Data Visualization'});  
CREATE (:Skill {name: 'UI/UX Design'});
```

Step 2: Create Relationships

-- People connected professionally

```
MATCH (a:Person {name: 'Nour'}), (b:Person {name: 'Mostafa'})  
CREATE (a)-[:CONNECTED]->(b), (b)-[:CONNECTED]->(a);
```

```
MATCH (a:Person {name: 'Nour'}), (b:Person {name: 'Salma'})  
CREATE (a)-[:CONNECTED]->(b), (b)-[:CONNECTED]->(a);
```

-- People work at companies

```
MATCH (p:Person {name: 'Nour'}), (c:Company {name: 'TechCorp'})  
CREATE (p)-[:WORKS_AT]->(c);
```

```
MATCH (p:Person {name: 'Mostafa'}), (c:Company {name:  
'DataSolutions'})  
CREATE (p)-[:WORKS_AT]->(c);
```

```
MATCH (p:Person {name: 'Salma'}), (c:Company {name: 'TechCorp'})  
CREATE (p)-[:WORKS_AT]->(c);
```

-- People have skills

```
MATCH (p:Person {name: 'Nour'}), (s:Skill {name: 'Python'})
```

```
CREATE (p)-[:HAS_SKILL]->(s);
```

```
MATCH (p:Person {name: 'Mostafa'}), (s:Skill {name: 'Data  
Visualization'})
```

```
CREATE (p)-[:HAS_SKILL]->(s);
```

```
MATCH (p:Person {name: 'Salma'}), (s:Skill {name: 'UI/UX Design'})
```

```
CREATE (p)-[:HAS_SKILL]->(s);
```



Step 3: Query the Graph

```
-- Find all connections of Nour
```

```
MATCH (:Person {name: 'Nour'})-[:CONNECTED]->(colleague)
```

```
RETURN colleague.name AS ConnectionOfNour;
```

```
-- Find employees of TechCorp
```

```
MATCH (:Company {name: 'TechCorp'})<-[:WORKS_AT]-(e:Person)
```

```
RETURN e.name AS Employee;
```

```
-- List skills of each person
```

```
MATCH (p:Person)-[:HAS_SKILL]->(s:Skill)
```

```
RETURN p.name AS Person, s.name AS Skill;
```

```
-- Show professionals with same skills
```

```
MATCH (a:Person)-[:HAS_SKILL]->(s:Skill)<-[:HAS_SKILL]-(b:Person)
```

```
WHERE a.name <> b.name
```

```
RETURN DISTINCT a.name, s.name AS SharedSkill, b.name;
```

```
-- Full graph
```

```
MATCH (n)-[r]->(m)
```

```
RETURN n, r, m;
```



Optional: Delete the graph to reset

```
-----
```

```
MATCH (n) DETACH DELETE n;
```

```
-----
```

Using MongoDB with Python

1. Setting Up MongoDB with Python

- Install PyMongo:

```
pip install pymongo
```

- Connect to MongoDB:

```
from pymongo import MongoClient
```

```
client = MongoClient("mongodb://localhost:27017/")
```

```
db = client["mydatabase"]
```

```
collection = db["mycollection"]
```

2. Basic CRUD Operations

- Insert:

```
collection.insert_one({"name": "Alice", "age": 25})
```

```
collection.insert_many([{"name": "Bob", "age": 30}, {"name": "Charlie", "age": 35}])
```

- Find:

```
collection.find_one({"name": "Alice"})
```

```
for doc in collection.find(): print(doc)
```

- Update:

```
collection.update_one({"name": "Alice"}, {"$set": {"age": 26}})-
```

Delete:

```
collection.delete_one({"name": "Alice"})
```

3. Querying with Filters and Projections

```
for doc in collection.find({"age": {"$gt": 25}}, {"_id": 0, "name": 1, "age": 1}):
```

```
print(doc)
```

4. Aggregation Framework

- Group by City:

```
pipeline = [{"$group": {"_id": "$city", "count": {"$sum": 1}}}]
```

```
collection.aggregate(pipeline)
```

- Average Age per City:

```
pipeline = [{"$group": {"_id": "$city", "average_age": {"$avg": "$age"}}}]
```

- Lookup (Join):

```
"user_orders"}}]
```

```
pipeline = [{"$lookup": {"from": "orders", "localField": "user_id",  
"foreignField": "user_id", "as":
```

5. Indexes

```
collection.create_index("name")
```

```
collection.create_index([("name", 1), ("age", -1)])
```

6. Embedded Documents & Arrays

```
collection.insert_one({"name":  
"Sara", "address": {"city": "Cairo"}})
```

```
collection.find_one({"address.city": "Cairo"})
```

```
collection.insert_one({"name": "Ahmed", "skills": ["Python",  
"MongoDB"]})
```

```
collection.update_one({"name": "Ahmed"}, {"$push": {"skills":  
"Django"}})
```

7. Bulk Operations

```
from pymongo import InsertOne, DeleteOne, ReplaceOne
```

```
requests = [
```

```
InsertOne({"name": "Ali"}),
```

```
DeleteOne({"name": "Charlie"}),
```

```
ReplaceOne({"name": "Bob"}, {"name": "Bob", "age": 40})
```

```
]
```

```
collection.bulk_write(requests)
```

8. Error Handling

```
from pymongo.errors import ConnectionFailure
```

```
try:
```

```
    client.admin.command('ping')
```

```
except ConnectionFailure:
```

```
    print("Failed to connect")
```

9. Use Cases- User profiles

- Real-time dashboards
- Session storage

complete in vs code creat this project

Using Python and the PyMongo library, connect to a MongoDB instance and perform the following tasks:

1. Create Collections & Insert Data
 - Create at least two collections.
 - Insert at least three documents in each collection.
 - Use multiple field types such as arrays, embedded documents, Booleans, etc.
2. Update Documents

- Perform updates using at least two different update operators (e.g., \$set, \$push, \$inc).

3. Delete Documents

- Delete at least one document from each collection using a filter condition.

4. Find Queries – Logical Operators

- Use logical operators like \$or, \$and, or \$nor in a query.
- Use operators like \$in, \$all, or \$size in a query.
- Use \$expr or comparison operators to compare fields or values within documents.

5. Create Indexes

- Create one single-field index.
- Create one compound index.
- Create one unique index.

6. Array Aggregation

- For each document that contains an array field called grades, calculate the sum of all array elements and store the result in a new field called totalGrade.