DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

NOSQL DATABASE – CS3EL17

PRACTICAL FILE

SESSION

JULY-DECEMBER 2026

SUBMITTED TO:

Prof. AJAJ KHAN

SUBMITTED BY:

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

MEDI CAPS UNIVERSITY, INDORE - 453331, MADHYA PRADESH

INDEX

S.No	List of experiments	Date of Experiment	Date of Submission	Page No	Teacher's Signature
1	Introduction to NoSQL Databases, CAP Theorem				
2	NoSQL vs. RDBMS, Data Types and NoSQL Classification				
3	Case Study Key-Value Databases				
4	MongoDB Installation and configuration in windows				
5	Querying all the documents in JSON Format				
6	How to create and drop a database in mongoDB also Creating Collection in MongoDB on the fly				
7	MongoDB insert, update and delete in a document				
8	Apply Limit(), skip(),sort() method in MongoDB				
9	Case Study of Graph Databases, Column Family Databases				
10	Exploring Search Engines				

EXPERIMENT-1

(CS3EL17): NOSQL DATABASE	
AIM: Introduction to NOSQL Databases, CAP Theorem	Page 1 to 5

OBJECTIVE:-

- Understand the basic concepts and history of NoSQL databases.
- Learn the key features and types of NoSQL databases.
- Understand the Consistency-Availability-Partitioning (CAP) theorem.

INTRODUCTION TO NOSQL:-

NoSQL (Not Only SQL) is a type of database management system designed to handle a wide variety of data models, including key-value, document, column-family, and graph formats. Unlike traditional relational databases, which use structured query language (SQL) and store data in tables with rows and columns, NoSQL databases are designed to handle large volumes of unstructured, semi-structured, or structured data more flexibly and efficiently.

HISTORY OF THE TERM NOSQL:-

1. 1998: Carlo Strozzi's NoSQL Database
   - Carlo Strozzi coined the term "NoSQL" for his lightweight, open-source relational database called Strozzi NoSQL. Unlike traditional databases, it did not use the standard SQL interface but was still relational in nature. Strozzi later argued that the modern NoSQL movement, which focuses on non-relational databases, should have been named "NoREL" (not relational) since it departs entirely from the relational model.

2. 2009: Reintroduction of the Term by Johan Oskarsson
   - Johan Oskarsson, a developer at Last.fm, reintroduced the term "NoSQL" in early 2009 when he organized a meetup to discuss "open-source distributed, non-relational databases." This event aimed to address the growing interest in new types of databases that did not fit the traditional relational model, including open-source clones of Google's Bigtable and Amazon's DynamoDB. The term became a label for the broad movement of non-relational, distributed data stores that prioritized scalability and flexibility.
   -

KEY FEATURES OF NOSQL DATABASES:

1. Scalability: NoSQL databases are designed to scale out horizontally by distributing data across multiple servers, making them ideal for handling large amounts of data and high user loads.

2. Flexibility in Data Models: They support a wide range of data models, allowing for the storage of structured, semi-structured, and unstructured data. This flexibility makes NoSQL databases suitable for diverse use cases, such as content management, real-time analytics, and Internet of Things (IoT) applications.

3. High Performance: NoSQL databases are optimized for read and write performance, which is critical for applications that need to process large amounts of data in real time.

4. Schema-less Architecture: Unlike SQL databases that require a predefined schema, NoSQL databases allow you to add fields as needed without restructuring the entire database, providing more agility in application development.

5. Distributed Data Storage: Data is distributed across multiple nodes, enhancing data availability and fault tolerance. This is particularly useful in cloud and distributed environments.

TYPES OF NOSQL DATABASES:

1. **Key-Value Stores**: Data is stored as a collection of key-value pairs, where each key is unique. Examples include Redis, DynamoDB, and Riak.

2. **Document Stores:** Data is stored in document formats like JSON, BSON, or XML. Examples include MongoDB, CouchDB, and Couchbase.

3. **Column-Family Stores**: Data is stored in columns rather than rows, allowing for efficient retrieval of large datasets. Examples include Apache Cassandra and HBase.

4. **Graph Databases:** These databases store data as nodes, edges, and properties, making them ideal for applications with complex relationships. Examples include Neo4j, Amazon Neptune, and ArangoDB.

USE CASES FOR NOSQL DATABASES:

1. Real-time Big Data Applications: NoSQL databases can handle massive amounts of data at high speed, making them suitable for big data analytics, IoT, and real-time data processing.

2. Content Management Systems: Their flexible data models allow for easy storage of diverse content types, such as articles, images, and videos.

3.E-commerce and Retail: They can manage product catalogs, customer data, and transaction histories efficiently.

4. Fraud Detection and Prevention: NoSQL databases can quickly analyze patterns in large datasets, making them ideal for fraud detection systems where real-time analysis is crucial.

5. Social Networks and Recommendations: Graph databases, in particular, are great for handling social network data and generating recommendations based on user interactions.

ADVANTAGES OF NOSQL DATABASES:

1. Scalability: Easily scales out horizontally by adding more servers to handle increased load.
2. Flexible Schema: Allows for dynamic or schema-less data models, accommodating varying data structures.
3. Performance: Optimized for high-speed read and write operations, often resulting in lower latency.
4. High Availability: Includes built-in replication and distribution features to enhance fault tolerance and uptime.
5. Handling Unstructured Data: Effectively manages unstructured or semi-structured data formats, like JSON.

DISADVANTAGES OF NOSQL DATABASES:

1. Lack of Standardization
   - NoSQL databases do not have a standard query language like SQL, leading to inconsistency across different NoSQL systems and making it difficult to switch between them.

2. Data Consistency Issues
   - Many NoSQL databases offer eventual consistency rather than strong consistency, which can compromise data integrity, especially in applications requiring real-time accuracy.

3. Limited ACID Compliance
   - NoSQL databases often do not fully support ACID transactions, making them less suitable for applications requiring strict transactional guarantees.

4. Complexity in Joins and Relationships

- NoSQL databases are not optimized for complex joins, requiring developers to handle relationships between data manually, which can be cumbersome and lead to data duplication.

5. High Learning Curve
   - The diverse data models and query languages in NoSQL databases present a steep learning curve for developers familiar with relational databases, potentially increasing development time.

CAP THEOREM(Consistency Availability and Partition Tolerance):-

The CAP theorem, also known as the CAP principle, can be used to explain some of the competing requirements in a Distributed System with replication. It was created to make system designers aware of the trade-offs while designing networked shared-data systems. In this article, we will discuss the CAP theorem, and the tradeoffs offered by it using various real-life examples. After reading this article, we can ensure you that you can differentiate between each property of the CAP theorem and prioritize them based on your use case.

The three letters in CAP refer to three desirable properties of Distributed Systems with replicated data: Consistency (among replicated copies), Availability (of the system for read and write operations) and Partition Tolerance (of the nodes in the network being partitioned by a network fault).

The CAP Theorem, also known as Brewer's Theorem, is a principle in distributed systems that describes the trade-offs between three key properties:

1. Consistency: Every read receives the most recent write or an error. All nodes see the same data at the same time, ensuring that there are no discrepancies in the data across the system.

2. Availability: Every request receives a response, whether it's a successful read or write, even if some nodes are down. The system is always operational and responds to queries.

3. Partition Tolerance: The system continues to operate even if network partitions or communication breakdowns occur between nodes. It can handle the situation where nodes cannot communicate with each other.

According to the CAP Theorem, a distributed system can achieve at most two of these three properties simultaneously but not all three. Here's how the trade-offs typically work:

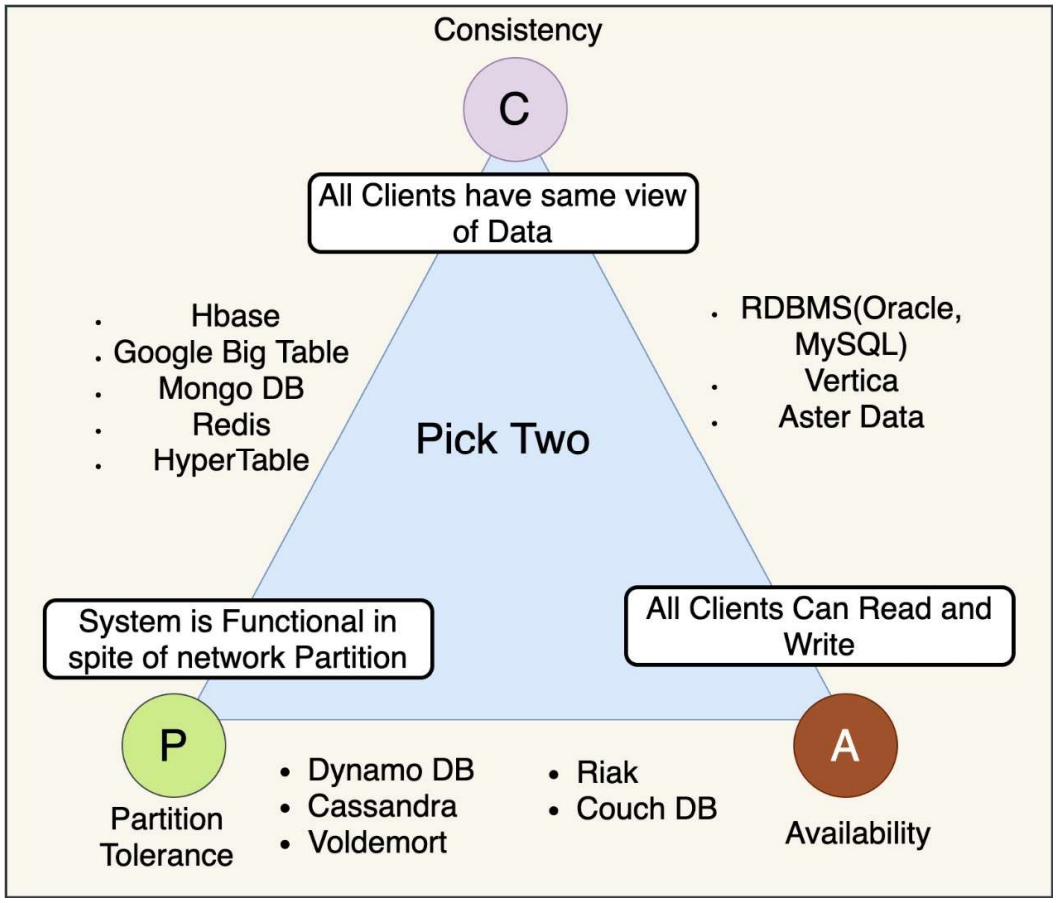
- Consistency and Availability (CA): The system provides strong consistency and is available, but it might not handle network partitions well. An example is a traditional SQL database in a single

data center.

- Consistency and Partition Tolerance (CP): The system maintains consistency and handles network partitions, but it may sacrifice some availability during partitions. An example is HBase or MongoDB in a configuration that prioritizes consistency.

- Availability and Partition Tolerance (AP): The system remains available and can handle network partitions but may not provide strong consistency. An example is Couchbase or DynamoDB, which might offer eventual consistency rather than immediate consistency.

Understanding these trade-offs helps in designing distributed systems and selecting the appropriate technology based on the needs and priorities of the application.



The CAP theorem states that it is not possible to guarantee all three of the desirable properties – Consistency, availability, and partition tolerance- at the same time in a distributed system with data replication, and it can only support any two properties at a time.

EXPERIMENT-2

(CS3EL17): NOSQL DATABASE	
AIM: NoSQL vs. RDBMS, Data Types and NoSQL Classification	Page 6 to 8

OBJECTIVE:-

- Understand the basic difference between NoSQL and RDBMS.
- Learn the data types of NoSQL databases.
- Understand the Classification of NoSQL.

NoSQL (Not Only SQL) vs. RDBMS (Relational Database Management Systems):-

1. NoSQL (Not Only SQL) Databases:

- Definition: NoSQL databases are designed to handle large volumes of unstructured or semi-structured data, offering high flexibility, scalability, and performance. They do not use a fixed schema or rely on SQL for querying.

- Key Features:
  - Schema Flexibility: NoSQL databases do not have a rigid schema. You can store different types of data in the same collection or table without predefined formats.
  - Horizontal Scalability: NoSQL databases are designed to scale out by distributing data across multiple servers.
- High Performance: NoSQL databases are optimized for fast read/write operations.
  - Use Cases: Real-time web applications, big data analytics, content management systems, and IoT applications.
- Examples: MongoDB, Cassandra, Redis, DynamoDB, CouchDB.

2. RDBMS (Relational Database Management Systems):

- Definition: RDBMSs are based on the relational model and use SQL (Structured Query Language) for querying and managing data. They are ideal for structured data with well-defined relationships.

- Key Features:
  - Schema-based: RDBMSs require a fixed schema where tables are defined with specific columns and data types.

- ACID Compliance: They ensure strong consistency, atomicity, isolation, and durability.
- Vertical Scalability: RDBMSs typically scale vertically by adding more resources (CPU, RAM) to a single server.
- Use Cases: Transactional systems, financial applications, CRM systems, and any application requiring complex querying and data integrity.

- Examples: MySQL, PostgreSQL, Oracle, Microsoft SQL Server.

NoSQL vs. RDBMS

	NoSQL	RDBMS
Types of Databases	Non-Relational.	Relational.
Schema	Dynamic Schema.	Predefined Schema.
database category	4 types of data.	Tabular-based data.
Scalability	Horizontally scalable.	Vertically scalable.
Language	UQL.	SQL.
Data sets	Preferred for large datasets.	Not preferred for large datasets.
Base property	Follows CAP Theorem.	Follows ACID Property.
Example	MongoDB, CouchDB, Redis.	Oracle, SQL, My SQL.

Data Types in NoSQL:

NoSQL databases generally support various types of data, such as:

- **String/Text:** Represents textual data.
- **Number/Integer/Float:** Represents numerical data.
- **Boolean:** Represents true or false values.
- **Date/Time:** Represents dates and times.
- **Array/List:** Represents a collection of ordered elements.
- **Document/Object:** Represents complex data types like JSON or BSON objects.
- **Binary Data:** Represents data stored in binary format, such as images or files.

- **Geospatial Data:** Represents geographical coordinates for location-based queries.

NoSQL Classification:

NoSQL databases are classified based on their data models:

1. Document Stores:

- Description: Store data in the form of documents, usually JSON, BSON, or XML.
- Features: Flexible schema, hierarchical data representation, support for complex data types, easy indexing.
- Examples: MongoDB, CouchDB.
- Use Cases: Content management systems, e-commerce applications, real-time analytics.

2. Key-Value Stores:

- Description: Store data as key-value pairs.
- Features: Simple data model, fast read/write operations, highly scalable, minimal schema.
- Examples: Redis, DynamoDB, Riak.
- Use Cases: Caching, session management, real-time data processing.

3. Column-Family Stores (Wide-Column Stores):

- Description: Store data in columns rather than rows. Each row can have a different number of columns.
- Features: High write and read throughput, optimized for large-scale data, support for distributed storage.
- Examples: Apache Cassandra, HBase.
- Use Cases: Time-series data, IoT data storage, big data applications.

4. Graph Databases:

- Description: Store data in nodes and edges, representing entities and their relationships.
- Features: Optimized for complex relationship queries, efficient for traversing and analyzing graphs.
- Examples: Neo4j, Amazon Neptune, ArangoDB.
- Use Cases: Social networks, recommendation engines, fraud detection.

EXPERIMENT-3

(CS3EL17): NOSQL DATABASE	
AIM: Case Study Key-Value Databases	Page 9 to 10

Key value databases :-

Key value databases, also known as key value stores, are database types where data is stored in a "key-value" format and optimized for reading and writing that data. The data is fetched by a unique key or a number of unique keys to retrieve the associated value with each key. The values can be simple data types like strings and numbers or complex objects.

MongoDB covers a wide range of database examples and use cases, supporting key-value pair data concepts. With its flexible schema and rich query language with secondary indexes, MongoDB is a compelling store for "key-value" data. Learn more in this article and try it with MongoDB Atlas, MongoDB's Database-as-a-Service platform.

What is a key-value database?

Over the years, database systems have evolved from legacy relational databases storing data in rows and columns to NoSQL distributed databases allowing a solution per use case. Key-value pair stores are not a new concept and were already with us for the last few decades. One of the known stores is the old Windows Registry allowing the system/applications to store data in a "key-value" structure, where a key can be represented as a unique identifier or a unique path to the value.

Data is written (inserted, updated, and deleted) and queried based on the key to store/retrieve its value.

How do key-value databases work?

A key-value database, AKA key-value store, associates a value (which can be anything from a number or simple string to a complex object) with a key, which is used to keep track of the object. In its simplest form, a key-value store is like a dictionary/array/map object as it exists in most programming paradigms, but which is stored in a persistent way and managed by a Database Management System (DBMS).

Key-value databases use compact, efficient index structures to be able to quickly and reliably locate a value by its key, making them ideal for systems that need to be able to find and retrieve data in constant time. Redis, for instance, is a key-value database that is optimized for tracking relatively simple data structures (primitive types, lists, heaps, and maps) in a persistent database. By only supporting a limited number of value types, Redis is able to expose an extremely simple interface to querying and manipulating them, and when configured optimally is capable of high throughput.

What are the features of a key-value database?

A key-value database is defined by the fact that it allows programs or users of programs to retrieve data by keys, which are essentially names, or identifiers, that point to some stored value. Because key-value databases are defined so simply, but can be extended and optimized in numerous ways, there is no global list of features, but there are a few common ones:

- Retrieving a value (if there is one) stored and associated with a given key
- Deleting the value (if there is one) stored and associated with a given key
- Setting, updating, and replacing the value (if there is one) associated with a given key

Modern applications will probably require more than the above, but this is the bare minimum for a key-value store.

When to use a key-value database

There are several use-cases where choosing a key value store approach is an optimal solution:

- Real time random data access, e.g., user session attributes in an online application such as gaming or finance.
- Caching mechanism for frequently accessed data or configuration based on keys.
- Application is designed on simple key-based queries.

Example:- Here is the example of bank database where unique key is used to store values.

BANK DATABASE	
Key	Value
1	ID:1 Joining Date: 15-July-1985 Designation: Cashier
2	ID:2 Joining Date: 19-March-1982 Designation: Manager
3	ID:3 Joining Date: 4-April-1988 Designation: Front Desk Officer

EXPERIMENT-4

(CS3EL17): NOSQL DATABASE	
AIM: MongoDB Installation and configuration in Windows.	Page 11 to 17

How to Install MongoDB on Windows?

Installing **MongoDB** on **Windows** involves setting up a 64-bit version of the database management system, which supports platforms like **Windows Server 2022, Windows Server 2019** and **Windows 11**. The process includes downloading the MongoDB server, configuring the environment and running the MongoDB service.

In this article, we will learn **How to Install MongoDB on Windows** in detail.

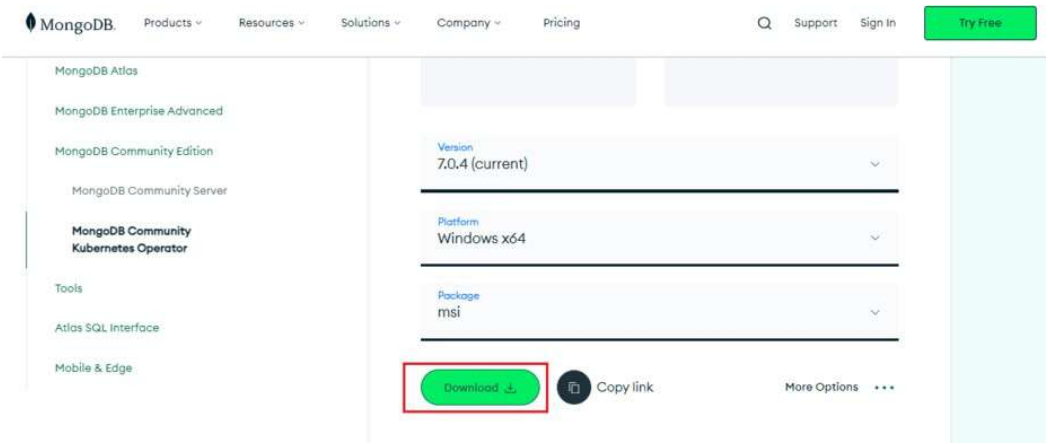
Requirements to Install MongoDB on Windows

- **MongoDB 4.4** and later only **support 64-bit versions** of Windows.
- MongoDB 7.0 Community Edition supports the following **64-bit versions** of Windows on x86_64 architecture:
  - **Windows Server 2022**
  - **Windows Server 2019**
  - **Windows 11**
- Ensure that the user is running **mongod** and **mongos** has the necessary permissions from the following groups:
  - Performance Monitor Users
  - Performance Log Users

Steps to Install MongoDB on Windows using MSI

To install MongoDB on Windows, first, download the MongoDB server and then install the MongoDB shell. The Steps below explain the installation process in detail and provide the required resources for the smooth **download and install MongoDB**.

Step 1: Go to the MongoDB Download Center to download the MongoDB Community Server.

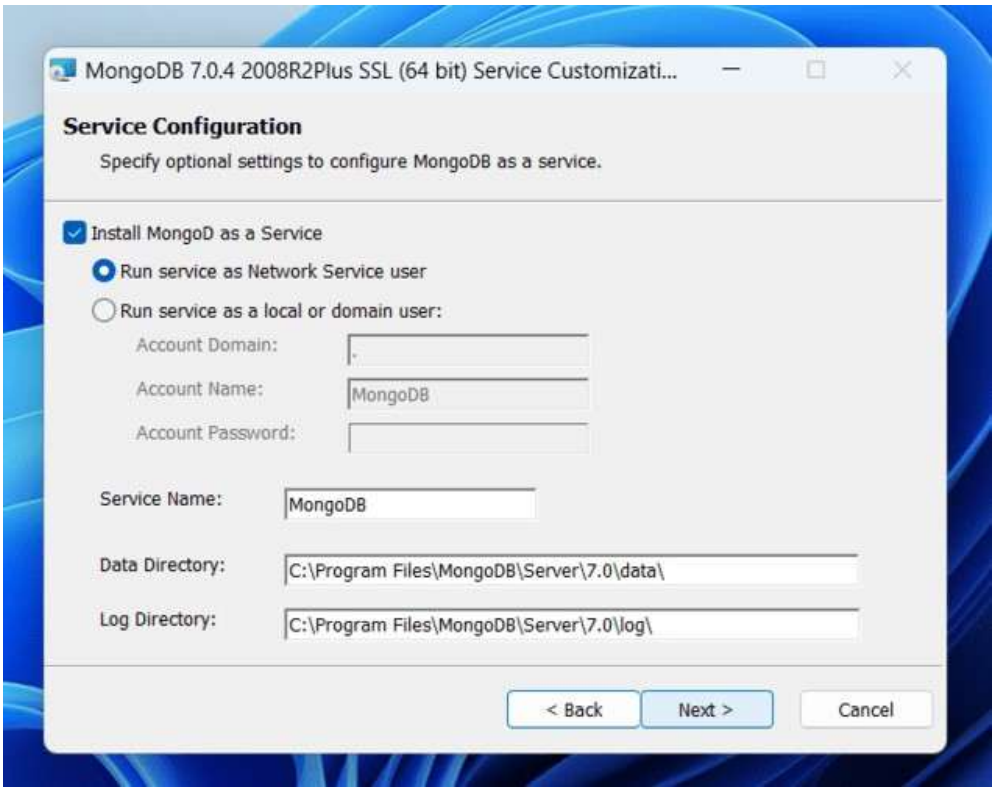


Here, You can select any version, Windows, and package according to your requirement. For Windows, we need to choose:

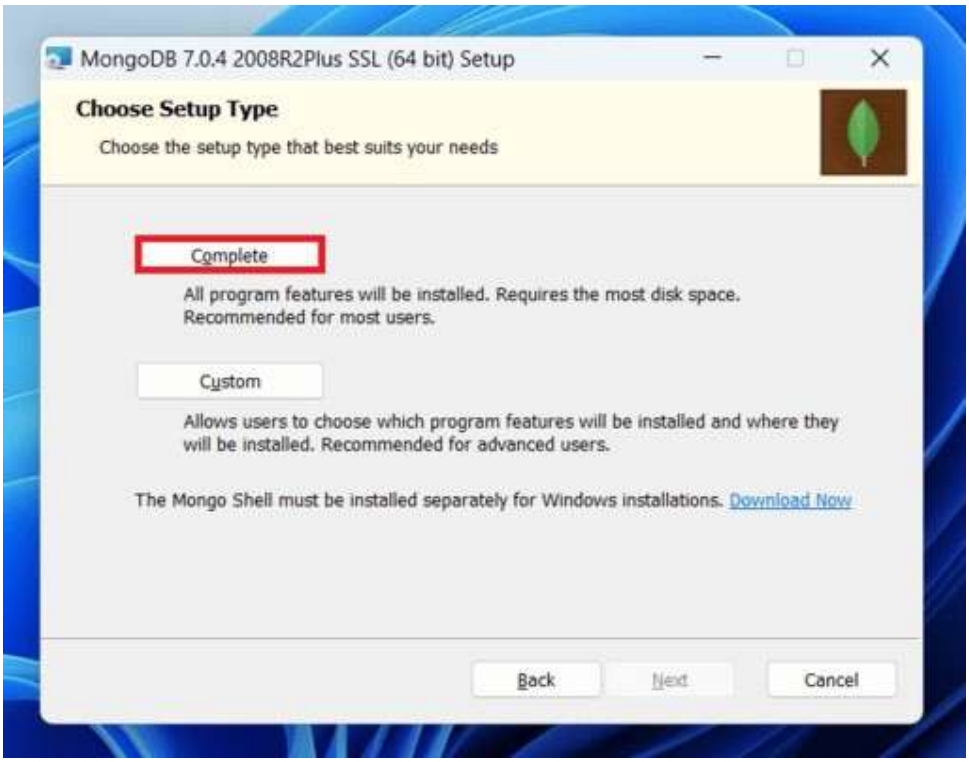
- **Version: 7.0.4**
- **OS: Windows x64**
- **Package: msi**

Step 2: When the download is complete open the msi file and click the *next button* in the startup screen:

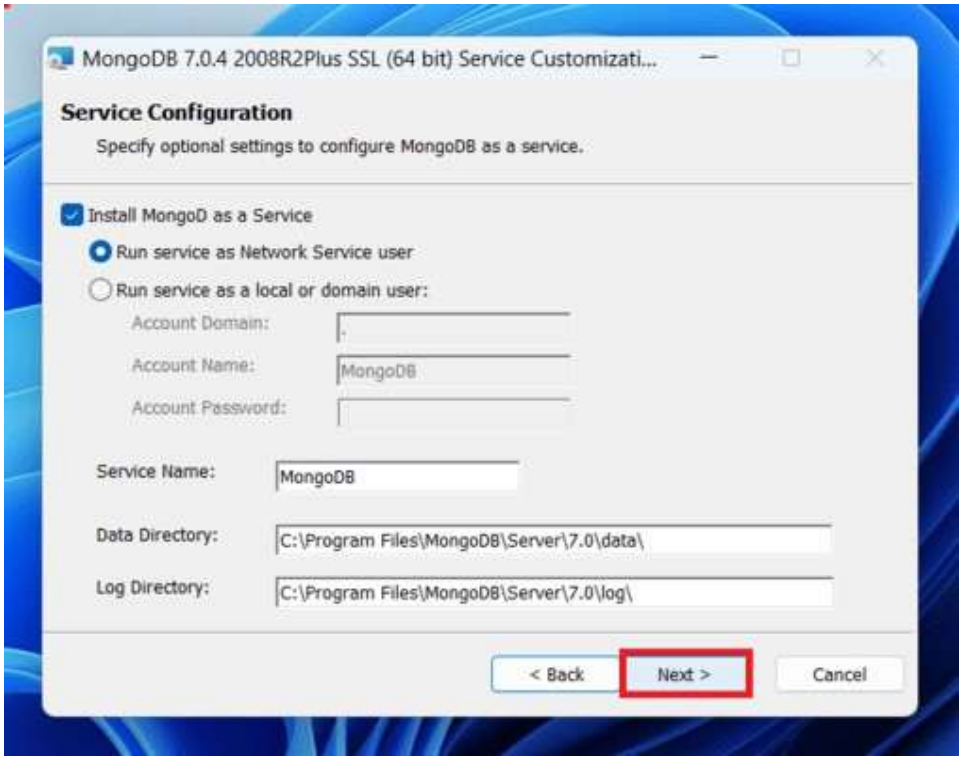
Step 3: Now accept the **End-User License Agreement** and click the next button:



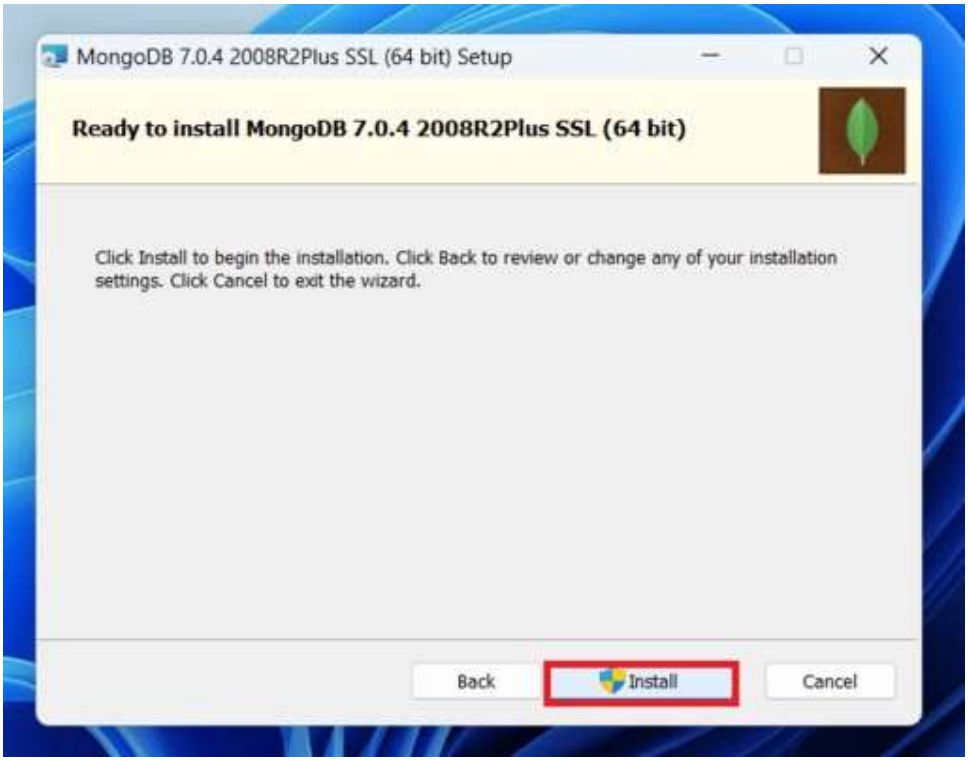
Step 4: Now select the **complete option** to install all the program features. Here, if you can want to install only selected program features and want to select the location of the installation, then use the **Custom option**:



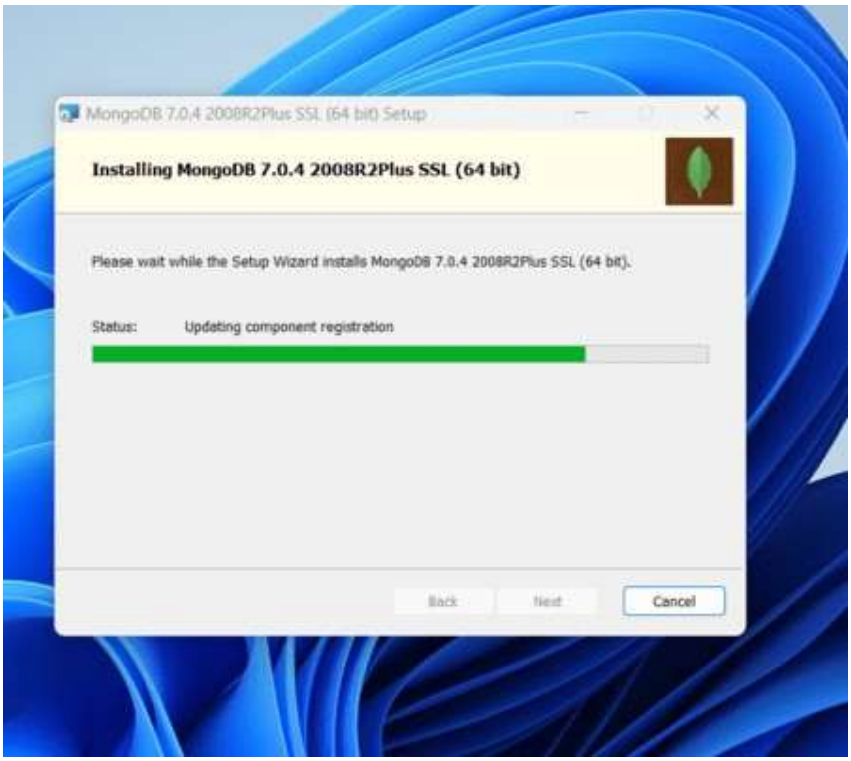
Step 5: Select "**Run service as Network Service user**" and copy the path of the data directory. **Click Next**:



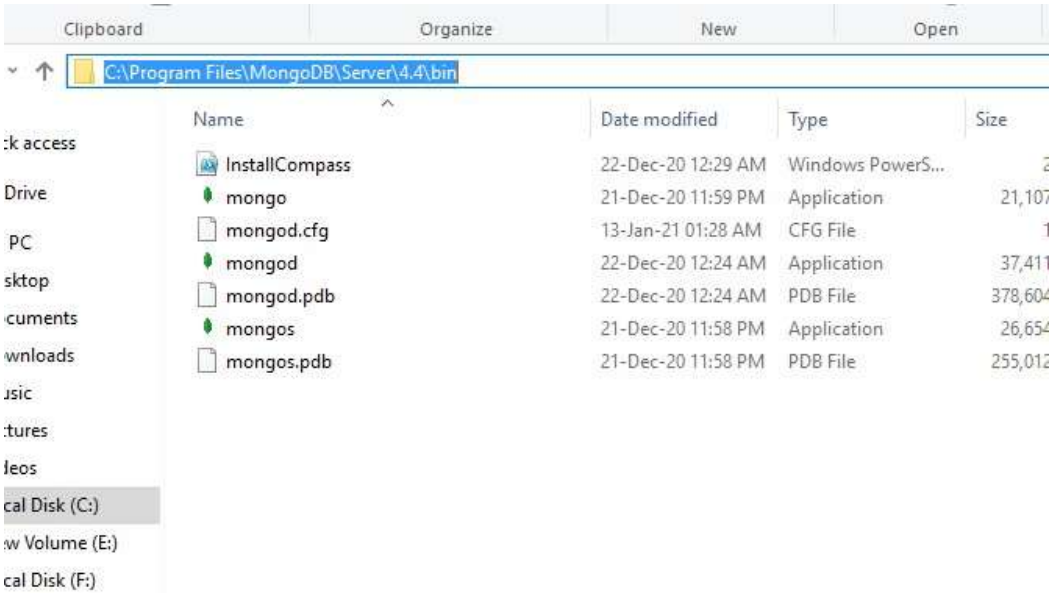
Step 6: Click the **Install button** to start the MongoDB installation process:



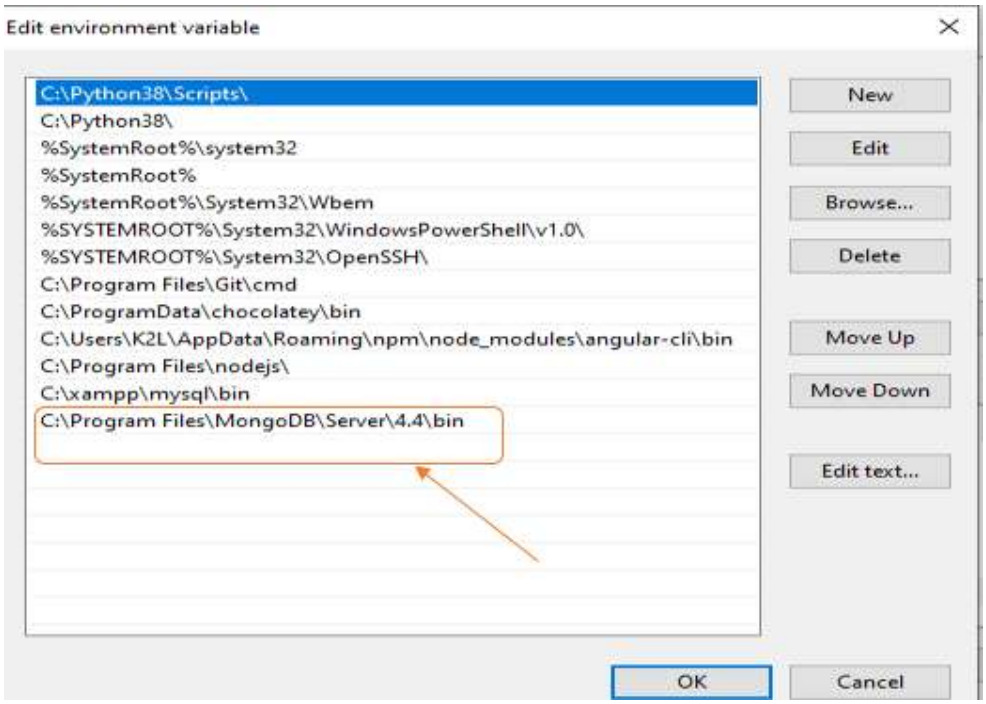
Step 7: After clicking on the install button installation of MongoDB begins:



Step 8: Now click the **Finish button** to complete the MongoDB installation process: **Step 9:** Now we go to the location where MongoDB installed in step 5 in your system and copy the **bin** path:



Step 10: Now, to create an environment variable open system **properties >> Environment Variable >> System variable >> path >> Edit Environment variable** and paste the copied link to your environment



system and **click Ok**:

Step 11: After setting the environment variable, we will run the MongoDB server, i.e. **mongod**. So, open the **command prompt** and run the following command:

mongod

When you run this command you will get an error i.e. **C:/data/db/ not found**.

Step 12: Now, Open **C** drive and create a folder named "**data**" inside this folder create another folder named "db". After creating these folders. Again open the command prompt and run the following command:

mongod

Now, this time the MongoDB server(i.e., mongod) will run successfully.

```
C:\Users\NIkhil Chhipa>mongod
{"t":{"$date":"2021-01-31T00:56:54.081+05:30"},"s":"I",  "c":"CONTROL",  "id":23285,   "ctx"
ify --sslDisabledProtocols 'none'"}
{"t":{"$date":"2021-01-31T00:56:54.087+05:30"},"s":"W",  "c":"ASIO",     "id":22601,   "ctx"
}
{"t":{"$date":"2021-01-31T00:56:54.088+05:30"},"s":"I",  "c":"NETWORK",  "id":4648602, "ctx"
{"t":{"$date":"2021-01-31T00:56:54.090+05:30"},"s":"I",  "c":"STORAGE",  "id":4615611, "ctx"
bPath":"C:/data/db/","architecture":"64-bit","host":"DESKTOP-L9MUQ7N"}}
{"t":{"$date":"2021-01-31T00:56:54.090+05:30"},"s":"I",  "c":"CONTROL",  "id":23398,   "ctx"
rgetMinOS":"Windows 7/Windows Server 2008 R2"}}
{"t":{"$date":"2021-01-31T00:56:54.090+05:30"},"s":"I",  "c":"CONTROL",  "id":23403,   "ctx"
gitVersion":"913d6b62acfbb344dde1b116f4161360acd8fd13","modules":[],"allocator":"tcmalloc","
}}}}
{"t":{"$date":"2021-01-31T00:56:54.090+05:30"},"s":"I",  "c":"CONTROL",  "id":51765,   "ctx"
ndows 10","version":"10.0 (build 14393)"}}}
{"t":{"$date":"2021-01-31T00:56:54.090+05:30"},"s":"I",  "c":"CONTROL",  "id":21951,   "ctx"
{"t":{"$date":"2021-01-31T00:56:54.157+05:30"},"s":"I",  "c":"STORAGE",  "id":22270,   "ctx"
:{"dbpath":"C:/data/db/","storageEngine":"wiredTiger"}}
{"t":{"$date":"2021-01-31T00:56:54.158+05:30"},"s":"I",  "c":"STORAGE",  "id":22315,   "ctx"
ize=1491M,session_max=33000,eviction=(threads_min=4,threads_max=4),config_base=false,statist
le_manager=(close_idle_time=100000,close_scan_interval=10,close_handle_minimum=250),statisti
ess],"}}
{"t":{"$date":"2021-01-31T00:56:54.395+05:30"},"s":"I",  "c":"STORAGE",  "id":22430,   "ctx"
95788][3708:140713908197088], txn-recover: [WT_VERB_RECOVERY_PROGRESS] Recovering log 20 thr
{"t":{"$date":"2021-01-31T00:56:54.631+05:30"},"s":"I",  "c":"STORAGE",  "id":22430,   "ctx"
```

Run mongo Shell

Step 13: Now we are going to connect our server (mongod) with the mongo shell. So, keep that mongod window and open a new command prompt window and write **mongo.** Now, our mongo shell will successfully connect to the mongod.

Important Point: Please do not close the mongod window if you close this window your server will stop working and it will not able to connect with the mongo shell.

```
C:\Users\NIkhil Chhipa>mongo
MongoDB shell version v4.4.3
connecting to: mongodb://127.0.0.1:27017/?compressors=disabled&gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("96cca5da-dc9f-4a40-aabb-732ee37600c0") }
MongoDB server version: 4.4.3
---
The server generated these startup warnings when booting:
        2021-01-28T20:56:52.570+05:30: Access control is not enabled for the database. Read and write access
configuration is unrestricted
---
---
        Enable MongoDB's free cloud-based monitoring service, which will then receive and display
        metrics about your deployment (disk utilization, CPU, operation statistics, etc).

        The monitoring data will be available on a MongoDB website with a unique URL accessible to you
        and anyone you share the URL with. MongoDB may use this information to make product
        improvements and to suggest MongoDB products and deployment options to you.

        To enable free monitoring, run the following command: db.enableFreeMonitoring()
        To permanently disable this reminder, run the following command: db.disableFreeMonitoring()
---
>
```

Now, you are ready to write queries in the mongo Shell.

Run MongoDB

Now you can make a new database, collections, and documents in your shell. Below is an example of how to make a new database:

The use **Database_name** command makes a new database in the system if it does not exist, if the database exists it uses that database:

use gfg

Now your database is ready of name gfg.

The **db.Collection_name** command makes a new collection in the gfg database and the insertOne() method inserts the document in the **student** collection:

db.student.insertOne({Akshay:500})

```
> use gfg
switched to db gfg
> db.student.insertOne({Akshay:500})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("60083bf8b7388ed4d54157c9")
}
> db.student.find().pretty()
{ "_id" : ObjectId("60083bf8b7388ed4d54157c9"), "Akshay" : 500 }
>
```

Hence MongoDB is successfully installed and its working absolutely perfect!

EXPERIMENT-5

(CS3EL17): NOSQL DATABASE	
AIM: Querying all the documents in JSON Format.	Page 18 to 21

Instead of fetching all the documents from collection, we can fetch selected documents based on a criterion.

1. Equality Criteria
2. Greater Than Criteria:
3. Less than Criteria:
4. Not Equals Criteria:
5. Greater than equals Criteria:
6. Less than equals Criteria:

1. Equality Criteria

Used to fetch documents where a field's value matches a specific value.

- Operator: { field: value }

Query:

Find all documents where age is equal to 25.

javascript

```
db.collection.find({ "age": 25 });
```

Output: json

```
[
{ "_id": 1, "name": "Alice", "age": 25, "status": "active", "price": 200, "quantity": 10 }
]
```

2. Greater Than Criteria

Used to retrieve documents where the field's value is greater than a given value.

- Operator: { field: { $gt: value } }

Query:

Find all documents where age is greater than 30.

javascript

```
db.collection.find({ "age": { "$gt": 30 } });
```

Output: json

```
[
{ "_id": 4, "name": "David", "age": 35, "status": "active", "price": 300, "quantity": 80
}
]
```

3. **Less Than Criteria**

Used to find documents where the field's value is less than a given value.

- Operator: { field: { $lt: value } }

Query:

Find all documents where price is less than 150.

javascript

```
db.collection.find({ "price": { "$lt": 150 } });
```

Output: json

```
[
{ "_id": 3, "name": "Charlie", "age": 20, "status": "active", "price": 120, "quantity": 5
}
]
```

4. **Not Equals Criteria**

Used to fetch documents where the field's value does not match the specified value.

- Operator: { field: { $ne: value } }

Query:

Find all documents where status is not "inactive".

javascript

```
db.collection.find({ "status": { "$ne": "inactive" } });
```

Output: json

```
[
{ "_id": 1, "name": "Alice", "age": 25, "status": "active", "price": 200, "quantity": 10 },
{ "_id": 3, "name": "Charlie", "age": 20, "status": "active", "price": 120, "quantity": 5},
{ "_id": 4, "name": "David", "age": 35, "status": "active", "price": 300, "quantity": 80}
]
```

5. **Greater Than or Equals Criteria**

Used to retrieve documents where the field's value is greater than or equal to the specified

value.

- Operator: { field: { $gte: value } }

Query:

Find all documents where price is greater than or equal to 150.

javascript

```
db.collection.find({ "price": { "$gte": 150 } });
```

Output: json

```
[
{ "_id": 1, "name": "Alice", "age": 25, "status": "active", "price": 200, "quantity": 10 },
{ "_id": 2, "name": "Bob", "age": 30, "status": "inactive", "price": 150, "quantity": 50},
{ "_id": 4, "name": "David", "age": 35, "status": "active", "price": 300, "quantity": 80}
]
```

6. Less Than or Equals Criteria

Used to fetch documents where the field's value is less than or equal to the specified value.

- Operator: { field: { $lte: value } }

Query:

Find all documents where quantity is less than or equal to 50.

javascript

```
db.collection.find({ "quantity": { "$lte": 50 } });
```

Output: json

```
[
{ "_id": 1, "name": "Alice", "age": 25, "status": "active", "price": 200, "quantity": 10 },
{ "_id": 2, "name": "Bob", "age": 30, "status": "inactive", "price": 150, "quantity": 50},
{ "_id": 3, "name": "Charlie", "age": 20, "status": "active", "price": 120, "quantity": 5}
]
```

7. Multiple Criteria Query

Combining multiple criteria using logical operators.

- Syntax: Combine queries using { field: { $operator: value } } format for multiple fields.

Query:

Find documents where age is greater than 25 and status is not "inactive".

javascript

```
db.collection.find({   "age":
{ "$gt": 25 },
"status": { "$ne": "inactive" }
});
```

Output: json

```
[
{ "_id": 4, "name": "David", "age": 35, "status": "active", "price": 300, "quantity": 80}
]
```

EXPERIMENT-6

(CS3EL17): NOSQL DATABASE	
AIM: How to create and drop a database in mongoDB also Creating Collection in MongoDB on the fly.	Page 22 to 23

```
C:\Users\Yaman>mongosh
Current Mongosh Log ID: 66ed04fec49c54ffa1c73bf7
Connecting to:          mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+2.3.1
Using MongoDB:          7.0.14
Using Mongosh:          2.3.1

For mongosh info see: https://www.mongodb.com/docs/mongodb-shell/


   2024-09-13T11:13:51.955+05:30: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted
```

```
test> show databases
admin   40.00 KiB
config  72.00 KiB
local   40.00 KiB
```

1. Creating a Database in MongoDB:

MongoDB does not require explicitly creating a database; it creates one on the fly when you insert the first document. However, you can switch to a new database and start using it as follows:

```
test> use yaman
switched to db yaman
```

```
yaman> db.myCollection.insertOne({ name: "Yaman", age: 22 })
{
  acknowledged: true,
  insertedId: ObjectId('66ed0666c49c54ffa1c73bf8')
}
yaman>
```

2. Creating a Collection on the Fly:

MongoDB collections are created automatically when you first insert data into them. There's no need for explicit collection creation unless you want to specify options like capped collections or collection validators.

```
yaman> show databases
admin   40.00 KiB
config  96.00 KiB
local   40.00 KiB
yaman   40.00 KiB
yaman>
```

3. Dropping a Database:

To drop an existing database:

```
yaman> db.dropDatabase()
{ ok: 1, dropped: 'yaman' }
yaman> show databases
admin   40.00 KiB
config  96.00 KiB
local   40.00 KiB
yaman>
```

4. Show Collections:

```
yaman> show collections
myCollection
```

EXPERIMENT-7

(CS3EL17): NOSQL DATABASE	
AIM: Mongodb insert, update, and delete in a document.	Page 24 to 25

1. Inserting a Document

You can insert a document into a MongoDB collection using the insertOne() or insertMany() method.

Insert a Single Document

```
yaman> use myDatabase  # Switch to the database
switched to db myDatabase
myDatabase>

myDatabase> db.myCollection.insertOne({ name: "Yaman", age: 28, occupation: "Developer" })
{
  acknowledged: true,
  insertedId: ObjectId('66ed0a5fc49c54ffa1c73bfa')
}
myDatabase>
```

Insert Multiple Documents

```
myDatabase> db.myCollection.insertMany([
...   { name: "Alice", age: 30, occupation: "Engineer" },
...   { name: "Bob", age: 25, occupation: "Designer" }
... ])
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId('66ed0b0ac49c54ffa1c73bfb'),
    '1': ObjectId('66ed0b0ac49c54ffa1c73bfc')
  }
}
myDatabase>
```

2. Updating a Document

You can update documents using the updateOne(), updateMany(), or replaceOne() methods. The updateOne() and updateMany() methods allow you to update specific fields, while replaceOne() replaces the entire document.

Update a Single Document

```
myDatabase> db.myCollection.updateOne( { name: "Yaman" }, {$set: {age: 21}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
myDatabase>
```

Update Multiple Documents

```
myDatabase> db.myCollection.updateMany( { occupation: "Developer" }, {$set: {occupation: "Senior Developer"}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
myDatabase>
```

3. Deleting a Document

You can delete documents using the deleteOne() or deleteMany() methods.

Delete a Single Document

```
myDatabase> db.myCollection.deleteOne({ name: "Yaman" })
{ acknowledged: true, deletedCount: 1 }
myDatabase>
```

Delete Multiple Documents

```
myDatabase> db.myCollection.deleteMany({ occupation: "Senior Developer" })
{ acknowledged: true, deletedCount: 0 }
myDatabase>
```

EXPERIMENT-8

(CS3EL17): NOSQL DATABASE	
AIM: Apply Limit( ), Skip( ), sort( ) method in MongoDB.	Page 26 to 28

Firstly, insert some data in Collection

```
myDatabase> db.myCollection.insertMany([
...   { name: "Person1", age: 21, occupation: "Engineer" },
...   { name: "Person2", age: 22, occupation: "Teacher" },
...   { name: "Person3", age: 23, occupation: "Doctor" },
...   { name: "Person4", age: 24, occupation: "Designer" },
...   { name: "Person5", age: 25, occupation: "Engineer" },
...   { name: "Person6", age: 26, occupation: "Teacher" },
...   { name: "Person7", age: 27, occupation: "Doctor" },
...   { name: "Person8", age: 28, occupation: "Engineer" },
...   { name: "Person9", age: 29, occupation: "Designer" },
...   { name: "Person10", age: 30, occupation: "Engineer" },
...   { name: "Person11", age: 31, occupation: "Teacher" },
...   { name: "Person12", age: 32, occupation: "Doctor" },
...   { name: "Person13", age: 33, occupation: "Designer" },
...   { name: "Person14", age: 34, occupation: "Engineer" },
...   { name: "Person15", age: 35, occupation: "Teacher" },
...   { name: "Person16", age: 36, occupation: "Doctor" },
...   { name: "Person17", age: 37, occupation: "Engineer" },
...   { name: "Person18", age: 38, occupation: "Designer" },
...   { name: "Person19", age: 39, occupation: "Engineer" },
...   { name: "Person20", age: 40, occupation: "Teacher" }
... ])
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId('66ed1073c49c54ffa1c73bfd'),
    '1': ObjectId('66ed1073c49c54ffa1c73bfe'),
    '2': ObjectId('66ed1073c49c54ffa1c73bff'),
    '3': ObjectId('66ed1073c49c54ffa1c73c00'),
    '4': ObjectId('66ed1073c49c54ffa1c73c01'),
    '5': ObjectId('66ed1073c49c54ffa1c73c02'),
    '6': ObjectId('66ed1073c49c54ffa1c73c03'),
    '7': ObjectId('66ed1073c49c54ffa1c73c04'),
    '8': ObjectId('66ed1073c49c54ffa1c73c05'),
    '9': ObjectId('66ed1073c49c54ffa1c73c06'),
    '10': ObjectId('66ed1073c49c54ffa1c73c07'),
    '11': ObjectId('66ed1073c49c54ffa1c73c08'),
    '12': ObjectId('66ed1073c49c54ffa1c73c09'),
    '13': ObjectId('66ed1073c49c54ffa1c73c0a'),
    '14': ObjectId('66ed1073c49c54ffa1c73c0b'),
    '15': ObjectId('66ed1073c49c54ffa1c73c0c'),
    '16': ObjectId('66ed1073c49c54ffa1c73c0d'),
    '17': ObjectId('66ed1073c49c54ffa1c73c0e'),
    '18': ObjectId('66ed1073c49c54ffa1c73c0f'),
    '19': ObjectId('66ed1073c49c54ffa1c73c10')
  }
}
myDatabase>
```

limit():

This method limits the number of documents returned.

```
myDatabase> db.myCollection.find().limit(2)
[
  {
    _id: ObjectId('66ed0b0ac49c54ffa1c73bfb'),
    name: 'Alice',
    age: 30,
    occupation: 'Engineer'
  },
  {
    _id: ObjectId('66ed0b0ac49c54ffa1c73bfc'),
    name: 'Bob',
    age: 25,
    occupation: 'Designer'
  }
]
myDatabase>
```

1. skip():

This method skips a specified number of documents from the result set.

```
myDatabase> db.myCollection.find().skip(20)
[
  {
    _id: ObjectId('66ed1073c49c54ffa1c73c0f'),
    name: 'Person19',
    age: 39,
    occupation: 'Engineer'
  },
  {
    _id: ObjectId('66ed1073c49c54ffa1c73c10'),
    name: 'Person20',
    age: 40,
    occupation: 'Teacher'
  }
]
myDatabase>
```

2. sort():

This method sorts the documents based on a specified field. Sorting can be done in ascending (1) or descending (-1) order.

For Ascending Order:

```
myDatabase> db.myCollection.find().sort({ age: 1 })
[
  {
    _id: ObjectId('66ed1073c49c54ffa1c73bfd'),
    name: 'Person1',
    age: 21,
    occupation: 'Engineer'
  },
  {
    _id: ObjectId('66ed1073c49c54ffa1c73bfe'),
    name: 'Person2',
    age: 22,
    occupation: 'Teacher'
  },
  {
    _id: ObjectId('66ed1073c49c54ffa1c73bff'),
    name: 'Person3',
    age: 23,
    occupation: 'Doctor'
  },
  {
    _id: ObjectId('66ed1073c49c54ffa1c73c00'),
    name: 'Person4',
    age: 24,
    occupation: 'Designer'
  },
  {
    _id: ObjectId('66ed0b0ac49c54ffa1c73bfc'),
    name: 'Bob',
    age: 25,
    occupation: 'Designer'
  },
  {
    _id: ObjectId('66ed1073c49c54ffa1c73c01'),
    name: 'Person5',
    age: 25,
    occupation: 'Engineer'
  },
  {
    _id: ObjectId('66ed1073c49c54ffa1c73c02'),
    name: 'Person6',
    age: 26,
    occupation: 'Teacher'
  },
  {
    _id: ObjectId('66ed1073c49c54ffa1c73c03'),
    name: 'Person7',
    age: 27,
    occupation: 'Doctor'
  },
```

For Descending Order:

```
myDatabase> db.myCollection.find().sort({ age: -1 })
[
  {
    _id: ObjectId('66ed1073c49c54ffa1c73c10'),
    name: 'Person20',
    age: 40,
    occupation: 'Teacher'
  },
  {
    _id: ObjectId('66ed1073c49c54ffa1c73c0f'),
    name: 'Person19',
    age: 39,
    occupation: 'Engineer'
  },
  {
```

EXPERIMENT-9

(CS3EL17): NOSQL DATABASE	
AIM: Case Study of Graph Databases, Column Family Databases.	Page 29 to 32

Case Study: Graph Databases

Graph Databases are designed to represent and store data in a graph structure, with nodes, edges, and properties. These databases excel at handling complex relationships between data points.

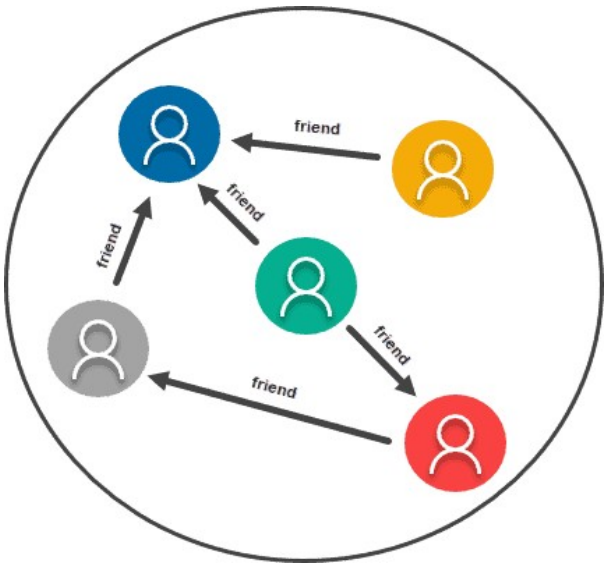
Case Study: Social Network Analysis Using Graph Databases

Scenario: A social media platform wants to analyze relationships between users, identify influencers, and provide personalized recommendations based on the users' connections and activities.

Challenges:

- Traditional relational databases struggle to efficiently model relationships such as friends, followers, likes, and comments.
- Join operations in relational databases become slow as the network grows in size.
- Complex queries to find shortest paths, mutual connections, or influencers are difficult to implement.

Solution: Implementing a Graph Database (e.g., Neo4j).



How Graph Databases Help:

- Efficient Relationship Modeling: Nodes represent users, and edges represent relationships (e.g., follows, likes, comments). Properties on nodes and edges can store user attributes (e.g., name, age) or relationship metadata (e.g., time of follow).
- Fast Traversals: Instead of costly joins, traversals (e.g., finding friends of friends) are performed along edges. Traversals in a graph database are efficient even with a large number of nodes and edges.
- Complex Queries: Graph algorithms like shortest path, centrality, or community detection can be easily applied for tasks like finding key influencers or identifying groups.

Example:

- Nodes: Represent users (e.g., User A, User B, User C).
- Edges: Represent relationships (e.g., User A follows User B, User B follows User C).
- Queries:
- Find all users who follow User A.
- Find the shortest path between User A and User C.
- Identify the most connected users (influencers).

Advantages:

- Performance: Queries that involve complex relationships (e.g., finding mutual friends or influencers) are executed quickly without requiring multiple joins.
- Flexibility: Graph databases are schema-less and can evolve with the application without major refactoring.
- Scalability: While handling billions of nodes and relationships, graph databases perform well under highly connected data scenarios.

Real-World Example:

- Twitter: Twitter uses graph databases to analyze its social graph, identifying influential users, recommending followers, and detecting communities.
- Facebook: Facebook's social graph is built to efficiently handle billions of connections and interactions, supporting friend recommendations, event invitations, and mutual friends suggestions.

Case Study: Column Family Databases

Column Family Databases are a type of NoSQL database designed to handle large-scale, distributed data efficiently. They store data in columns instead of rows, enabling fast data retrieval for wide, sparse datasets.

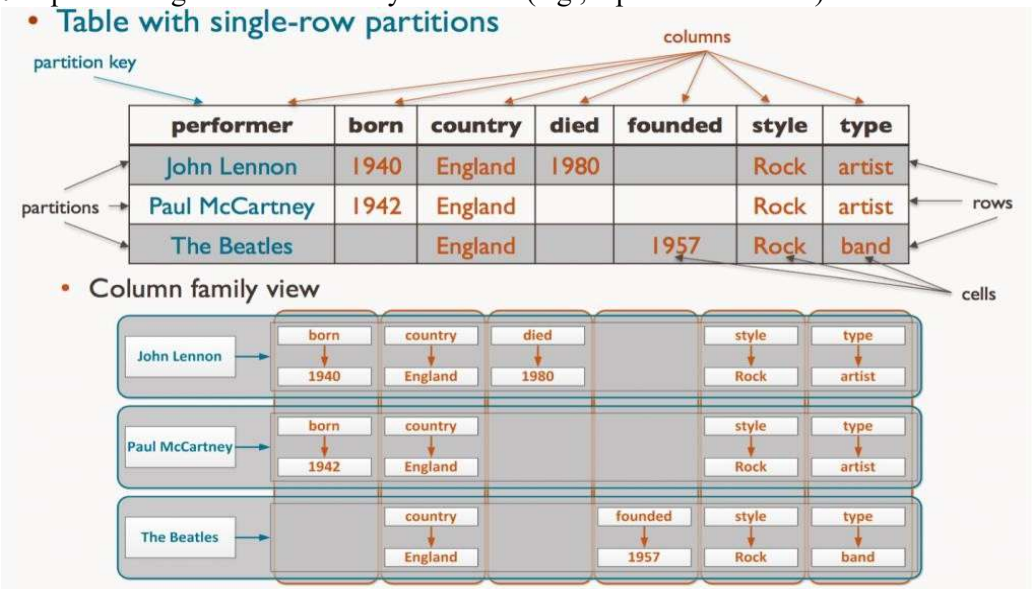
Case Study: Time-Series Data Analysis Using Column Family Databases

Scenario: A telecom company needs to store and analyze billions of records of network activity logs over time, each containing metadata such as timestamp, IP address, device type, and data usage.

Challenges:

- The data is time-series and growing rapidly, with millions of new records generated daily.
- Queries often focus on a specific time range or set of columns (e.g., retrieving all logs for a specific device type in the last 24 hours).
- Traditional relational databases are inefficient for handling wide datasets where the schema may change frequently.

Solution: Implementing a Column Family Database (e.g., Apache Cassandra).



How Column Family Databases Help:

- Flexible Schema: Unlike relational databases, column family databases store data in flexible column families (groupings of rows and columns). Each row in a column family can have a different set of columns.
- Efficient Time-Series Data Storage: Data can be organized into column families where each row key represents a unique identifier (e.g., a timestamp or device ID), and columns store data related to the key (e.g., IP, data usage).
- Distributed Architecture: Column family databases are designed to run on distributed clusters, enabling horizontal scaling as the data grows.

Example:

- Column Family: Stores network logs. Each row represents a log entry, and columns store various

properties like timestamp, device ID, and data usage.

- Row key: timestamp
- Columns: device_id, ip_address, data_usage, device_type

Queries:

- Retrieve all logs for a specific device within the last 24 hours.
- Retrieve only the data_usage and device_type columns for the past week's logs.

Advantages:

- High Throughput: Column family databases can efficiently write and read massive amounts of data across distributed clusters.
- Optimized for Time-Series: The row and column structure is perfect for time-series data, enabling fast retrieval of specific time ranges or subsets of columns.
- Scalability: Built to handle large datasets across multiple nodes with automatic sharding and replication.

Real-World Example:

- Netflix: Netflix uses Apache Cassandra to store and manage its time-series data for user activity tracking and log analysis. The columnar structure allows them to efficiently retrieve specific fields from massive datasets.
- Uber: Uber relies on Cassandra to store geospatial data, logs, and metrics from its ride-hailing service. The ability to scale horizontally helps them manage their rapidly growing data volume across multiple regions.

EXPERIMENT-10

(CS3EL17): NOSQL DATABASE	
AIM: Exploring Search Engines	Page 33 to 37

Exploring Search Engines: A Deep Dive

Search engines are essential tools for navigating the vast amount of information on the internet. They retrieve relevant information based on user queries, ranking the results based on various algorithms. Let's explore the key concepts, components, types, and technologies behind search engines, along with examples of popular ones.

1. How Search Engines Work

Search engines follow three main steps to provide relevant search results:

A. Crawling:

- Definition: Crawling is the process where search engines use bots (also called spiders or crawlers) to discover web pages by following links across the internet.
- How it works: Crawlers start by visiting known URLs and then follow links on those pages to discover other URLs. This process continues recursively, enabling the crawler to map the web.
- Example: Google's crawler, known as Googlebot, crawls websites to discover and update content in its search index.

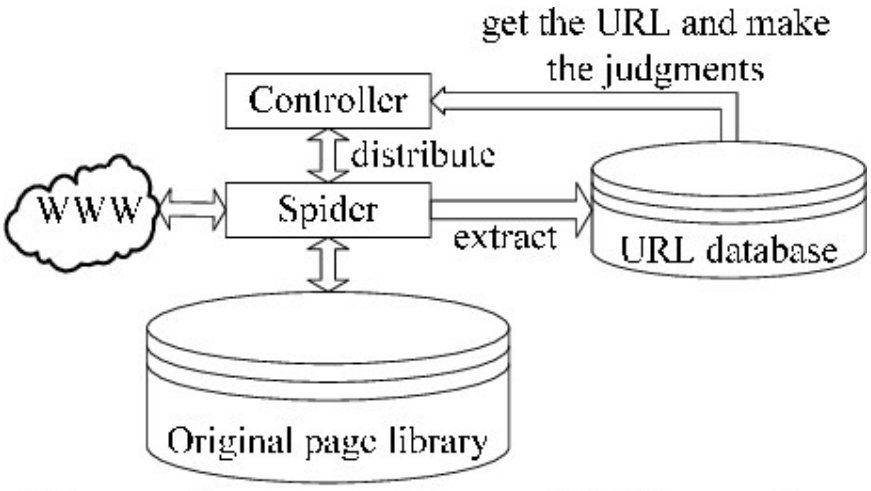
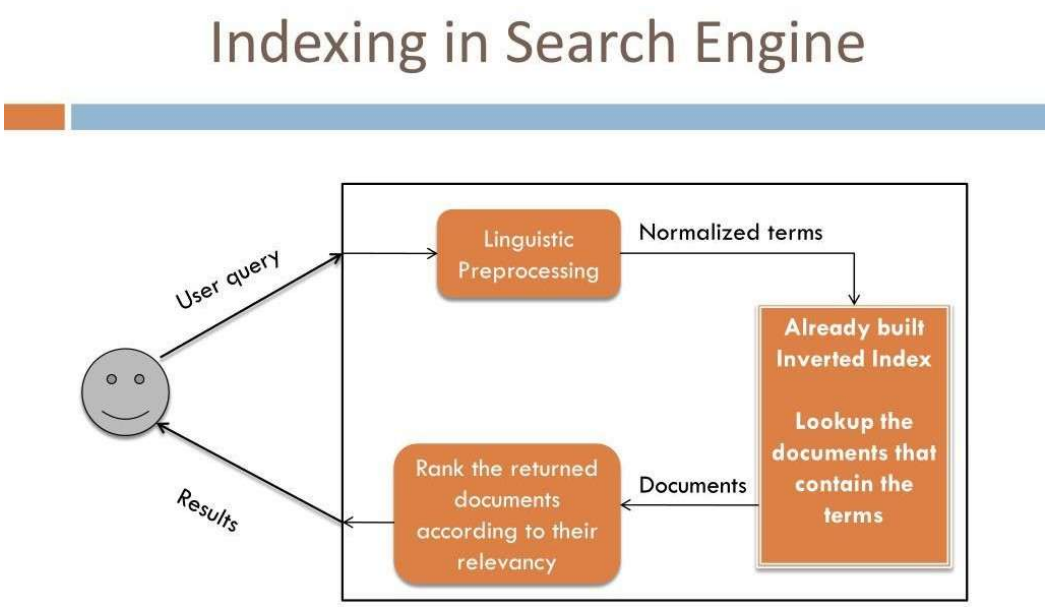


Figure 1. compositions of Web crawler

B. Indexing:

- Definition: Indexing is the process of storing and organizing the content discovered during crawling.
- How it works: Once a page is crawled, the search engine processes the text, metadata, and other content to build a structured index. The index serves as a database where key terms and their locations are stored, enabling quick retrieval when a user searches for information.
- Example: A page about "machine learning" will have keywords like “AI,” “algorithms,” and “training data” indexed, linking the page to related search queries.



C. Ranking & Retrieval:

- Definition: When a user submits a search query, the search engine retrieves relevant results from the index and ranks them according to relevance.
- How it works: Search engines use ranking algorithms (such as Google's PageRank) that analyze factors like keyword relevance, content quality, backlinks, user engagement, and freshness to order the results. The goal is to present the most useful results at the top.
- Example: If you search for "best pizza in New York," the search engine will rank local pizza places, reviews, and guides based on factors like location, popularity, and user ratings.

2. Key Components of a Search Engine

A. Web Crawlers (Spiders)

- Purpose: Crawl the web to discover new and updated content.
- Example: Googlebot, Bingbot.

B. Index

- Purpose: Store and organize crawled data for fast retrieval during searches.
  - Example: Google's index contains hundreds of billions of webpages organized for quick lookup.

C. Ranking Algorithms

- Purpose: Rank search results based on relevance, quality, and other factors.
  - Example: Google's PageRank algorithm considers the number and quality of backlinks to rank a page.

D. Query Processor

- Purpose: Interpret the user's query, search the index, and return relevant results.
  - Example: Google's BERT (Bidirectional Encoder Representations from Transformers) helps process natural language queries.

E. User Interface (UI)

- Purpose: Display search results in a way that is easy to read and interact with.
  - Example: Google Search's UI displays results, including snippets, images, and maps, in a user-friendly layout.

3. Types of Search Engines

A. Web Search Engines:

The most common type, designed to search the internet for relevant information based on a query.

Examples:

- Google: The dominant web search engine with an advanced ranking algorithm.
- Bing: Microsoft's search engine, offering similar features to Google.
- Yahoo: One of the earliest search engines, now powered by Bing.

B. Vertical Search Engines:

These are specialized search engines that focus on a specific type of content or niche industry.

Examples:

- Amazon: Focuses on product searches and e-commerce.
- Kayak: A travel search engine for flights, hotels, and rental cars.
- YouTube: A video search engine, part of Google, focused on video content.

C. Enterprise Search Engines:

Search engines designed for organizations to index and retrieve internal documents and information.

Examples:

- Elasticsearch: An open-source search engine used for enterprise-level indexing and searching.
- Microsoft SharePoint: A platform with integrated search capabilities for organizations.

D. Metasearch Engines:

These engines aggregate results from other search engines instead of building their own index.

Examples:

- DuckDuckGo: Focuses on privacy and aggregates results from multiple sources.
- Dogpile: Aggregates results from Google, Yahoo, and other search engines.

E. Semantic Search Engines:

Search engines that understand the intent behind a query rather than just matching keywords.

Examples:

- Google: Incorporates semantic search with its Knowledge Graph to understand entities and relationships.
- Wolfram Alpha: A computational search engine that focuses on providing factual answers, not just links.

4. Popular Search Engines

A. Google:

- Market Share: ~92% of global searches.
- Key Features: Google is known for its highly sophisticated search algorithm, leveraging AI, semantic search, and advanced ranking techniques to provide the most relevant results.
- Unique Offerings: Google's Knowledge Graph, featured snippets, and rich search results (images, videos, shopping).

B. Bing:

- Market Share: ~2.5% of global searches.
- Key Features: Bing focuses on delivering visually appealing search results and incorporates AI to improve its search capabilities.
- Unique Offerings: Image search, video previews, integration with Microsoft products (Office, Windows).

C. DuckDuckGo:

- Market Share: ~0.5% of global searches.
- Key Features: DuckDuckGo focuses on user privacy and does not track personal data or search history.
- Unique Offerings: Aggregated search results from multiple sources, privacy-focused features, zero personal profiling.

5. Search Engine Technologies

A. Natural Language Processing (NLP)

- How it works: NLP allows search engines to understand user queries better by interpreting the meaning behind words and context.
- Example: Google's BERT model helps the engine understand the context of prepositions in long queries, improving the search quality for natural language.

B. Machine Learning & AI

- How it works: Machine learning models continuously improve search results by analyzing user behavior, click patterns, and feedback.
- Example: Google's RankBrain, an AI-based system that helps refine search results by learning how users interact with different results.

C. Knowledge Graphs

- How it works: A knowledge graph connects facts and entities, allowing search engines to provide direct answers to queries instead of just links.
- Example: Google's Knowledge Graph displays information boxes with quick facts when you search for well-known entities (celebrities, companies, historical events).

D. Personalized Search

- How it works: Search engines personalize results based on a user's past searches, location, and behavior.
- Example: Google personalizes search results based on user data (such as browsing history, location, or interests) to show the most relevant results.