NOSQL DATABASES (23CSD472A)

R.Eswar Reddy,
Assistant Professor,
Sreenivasa Institute of Technology and Management Studies

July 20, 2026

Contents

UNIT–1: Introduction to NOSQL

1 Introduction to NOSQL

A NoSQL database is a type of database that stores data using non-relational data models instead of traditional tables. Designed to efficiently manage large volumes of structured, semi-structured and unstructured data. Work with cloud platforms and distributed systems using tools such as Docker, Kubernetes and Apache Cassandra. Optimizes performance by reducing complex JOIN operations. Used by high-traffic applications such as social media, e-commerce, gaming and real-time analytics to handle massive volumes of data and concurrent users.

Brief History of NoSQL Databases

- **1998:** Carlo Strozzi uses the term *NoSQL* for his lightweight, open-source relational database.
- **2000:** Graph database **Neo4j** is launched.
- **2004:** Google introduces **BigTable**, a distributed storage system for managing structured data.
- **2005: CouchDB** is launched as a document-oriented database.
- **2007:** Amazon releases the research paper on **Dynamo**, a highly available distributed key-value store.
- **2008:** Facebook open-sources the **Cassandra** project.
- **2009:** The term *NoSQL* is reintroduced and gains popularity within the database community.

Features

- **Horizontal Scalability:** Distributes data across multiple nodes, enabling the database to handle increasing workloads by adding more servers.
- **Distributed Data Storage:** Stores and manages data across a cluster of machines, improving availability and supporting large-scale deployments.
- **Multiple Data Models:** Supports document, key-value, column-family, and graph models, allowing developers to choose the most suitable model for different use cases.
- **High Availability:** Uses replication and automatic failover mechanisms to ensure continuous access to data even when individual nodes become unavailable.
- **Fault Tolerance:** Detects node failures and continues serving requests using replicated data and distributed storage.

Popular NoSQL Databases

1. MongoDB
2. Apache Cassandra
3. Redis
4. Apache HBase
5. Neo4j
6. CouchDB
7. Couchbase
8. Amazon DynamoDB
9. Azure Cosmos DB
10. Google Cloud Bigtable

Applications of NoSQL Databases

- **Content Management Systems (CMS):** Efficiently stores and retrieves articles, multimedia files, and metadata with varying structures.
- **Internet of Things (IoT):** Manages high-frequency sensor and device data generated by connected systems.
- **Caching and Session Management:** Provides low-latency storage for user sessions, authentication tokens, and frequently accessed data.
- **Big Data Processing:** Stores and processes massive datasets generated from distributed applications and analytics platforms.

Challenges of NoSQL Databases

- **Lack of Standardization:** Different NoSQL databases use different query languages and interfaces, making migration and interoperability challenging.
- **Limited Complex Queries:** Joins and complex relational operations are generally less efficient compared to traditional relational databases.
- **Management Complexity:** Distributed architectures require careful configuration, monitoring, and maintenance, increasing administrative complexity.

SQL vs NoSQL Databases: Comparison

Feature	SQL Databases	NoSQL Databases
Data Structure	Structured data with fixed schema	Unstructured or semi-structured with flexible schema
Scalability	Vertical scalability (Scale-Up)	Horizontal scalability (Scale-Out)
Complex Queries	Excellent support for JOINs and complex queries	Limited support for complex queries
Transactions	ACID compliance	BASE model, limited ACID support
Consistency	Strong consistency	Eventual consistency
Performance	Efficient for structured and smaller datasets	Efficient for large-scale distributed data
Flexibility	Schema modifications required	Schema changes without downtime
Examples	MySQL, PostgreSQL, Oracle, SQL Server	MongoDB, Cassandra, CouchDB, DynamoDB

Feature	SQL Databases	NoSQL Databases
Data Structure	Structured data with fixed schema tables	Unstructured or semi-structured data with flexible schema
Scalability	Vertical scalability (scale-up)	Horizontal scalability (scale-out)
Complex Queries	Excellent support for complex queries and JOIN operations	Limited support for complex queries and JOINs
Transactions	ACID-compliant transactions	Typically follows the BASE model; may not fully support ACID
Consistency	High data consistency	Eventual consistency; updates may not be immediate
Speed and Performance	Efficient for structured and smaller datasets	Optimized for large-scale distributed environments
Flexibility	Schema changes require database modifications	Schemas can be changed dynamically with minimal disruption
Examples	MySQL, PostgreSQL, Oracle, SQL Server	MongoDB, Cassandra, CouchDB, DynamoDB

Table 1: Comparison of SQL and NoSQL Databases

Comparison of SQL and NoSQL Databases

Advantages

1. **Structured Data Management:** SQL databases efficiently manage structured data using predefined schemas and integrity constraints.

2. **ACID Compliance:** Ensures Atomicity, Consistency, Isolation, and Durability,

making transactions reliable.

3. **Mature Ecosystem:** Provides extensive tools, frameworks, documentation, and community support.
4. **Standardized Language:** SQL is standardized by ANSI and ISO, enabling portability across database systems.
5. **Strong Data Integrity:** Supports constraints, triggers, and stored procedures to maintain data consistency.

Disadvantages

1. **Scalability Challenges:** Vertical scaling can be expensive and has practical limitations.
2. **Schema Rigidity:** Schema modifications often require migrations or downtime.
3. **Performance Bottlenecks:** Complex queries and large datasets can affect performance.
4. **High Overhead:** Requires additional memory and storage for indexing and structured storage.
5. **Limited Support for Semi-Structured Data:** Handling JSON, XML, and other flexible formats may require additional processing.

Advantages

1. **Flexible Schema:** Supports dynamic, semi-structured, and unstructured data without predefined schemas.
2. **Scalability:** Designed for horizontal scaling across multiple servers.
3. **High Performance:** Provides fast read and write operations, especially in distributed environments.
4. **Support for Semi-Structured Data:** Natively handles JSON, XML, and other flexible data formats.
5. **Horizontal Scalability:** Easily accommodates growing workloads and large datasets.

Disadvantages

1. **Consistency Trade-offs:** May sacrifice strong consistency to achieve high availability and partition tolerance.
2. **Limited ACID Support:** Full transactional guarantees are not available in all NoSQL databases.
3. **Complexity in Data Modeling:** Requires careful schema and query design for optimal performance.

4. **Tooling and Ecosystem Maturity:** Some NoSQL platforms have fewer tools and community resources compared to SQL systems.

5. **Learning Curve:** Developers familiar with relational databases may need time to adapt to NoSQL concepts and query models.

1.1 Key Concepts: Documents, Collections, Databases

- **Document**: A record in MongoDB, stored as a JSON-like (BSON) object.
- **Collection**: A group of documents, analogous to a table in relational databases but schema-less.
- **Database**: A container for collections. A MongoDB instance can host multiple databases.

The Document Database Model is a type of NoSQL database model that stores data in the form of documents instead of rows and columns. Documents are usually stored in formats such as JSON, BSON or XML, making them flexible and easy to manage.

Data is stored as documents rather than rows and tables. Supports flexible and dynamic schemas. Documents can contain nested data and complex structures. Designed for high scalability and fast data retrieval.

Popular Document Databases

- MongoDB
- CouchDB
- Amazon DocumentDB
- Azure Cosmos DB

Advantages of Document Database Model

1. **Flexible Schema:** Documents can have different fields and structures without affecting other documents.
2. **Easy Data Representation:** Complex and nested data can be stored naturally within a single document.
3. **High Scalability:** Data can be distributed across multiple servers to support large-scale applications.
4. **Fast Data Access:** Documents can be retrieved quickly without requiring complex joins.
5. **Developer Friendly:** JSON-like document formats are easy to understand and work with in modern applications.

Limitations of Document Database Model

1. **Data Redundancy:** Similar information may be repeated across multiple documents, increasing storage usage.
2. **Complex Relationships:** Managing relationships between documents can be more difficult than in relational databases.
3. **Limited Join Support:** Most document databases provide limited or less efficient join operations.

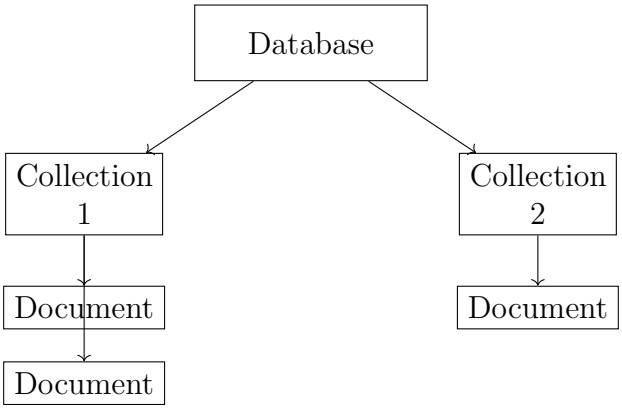


Figure 1: Hierarchy of Database, Collections, and Documents

Document Structure

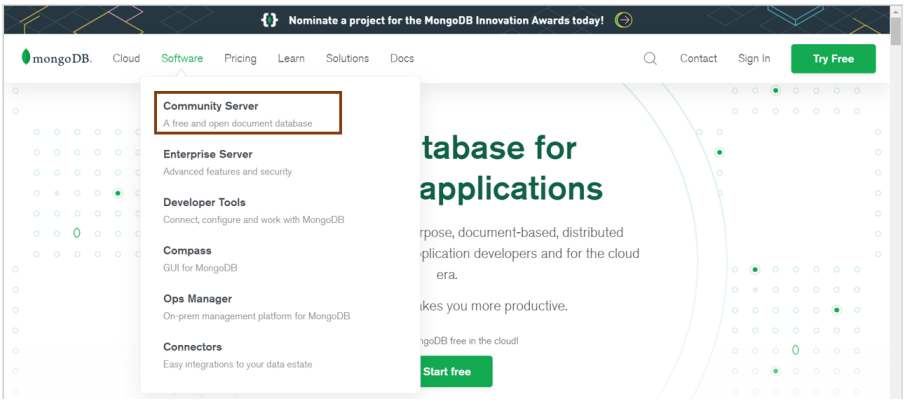
```
{ "_id" : ObjectId(...),
"name": "John Doe",
"age": 30,
"city": "New York",
"hobbies": ["reading", "gaming"] }
```

Figure 2: Example Document Structure in BSON

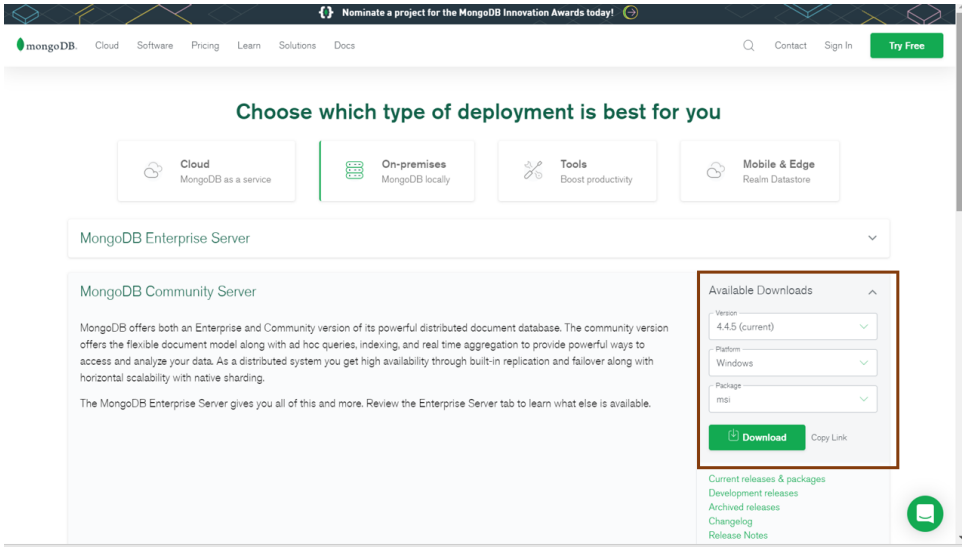
Getting and Starting MongoDB

To start using MongoDB:

1. **Download**: Visit `https://www.mongodb.com/try/download/community`.

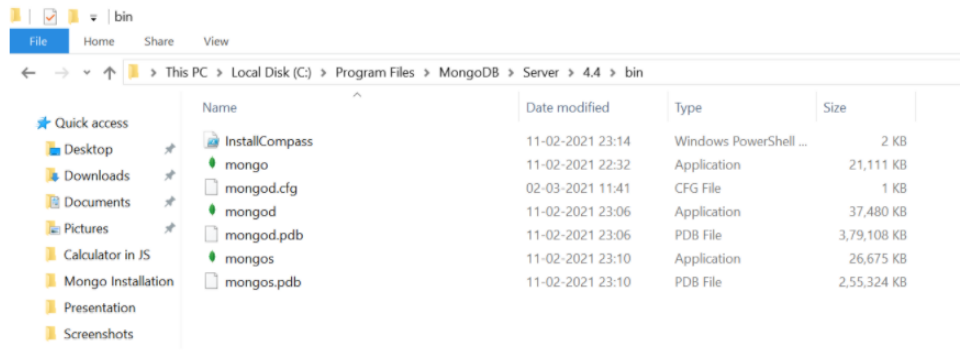


figureMongoDB Setup Wizard - Screenshot

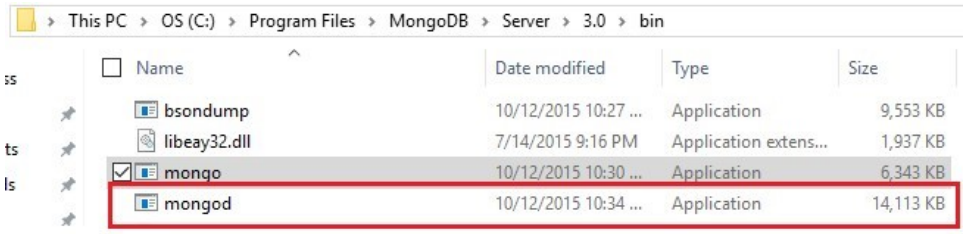


figureMongoDB System Install steps Make sure that the specifications to the right of the screen are correct. At the time of writing, the latest version is 4.4.5. Ensure that the platform is Windows, and the package is MSI. Go ahead and click on download.

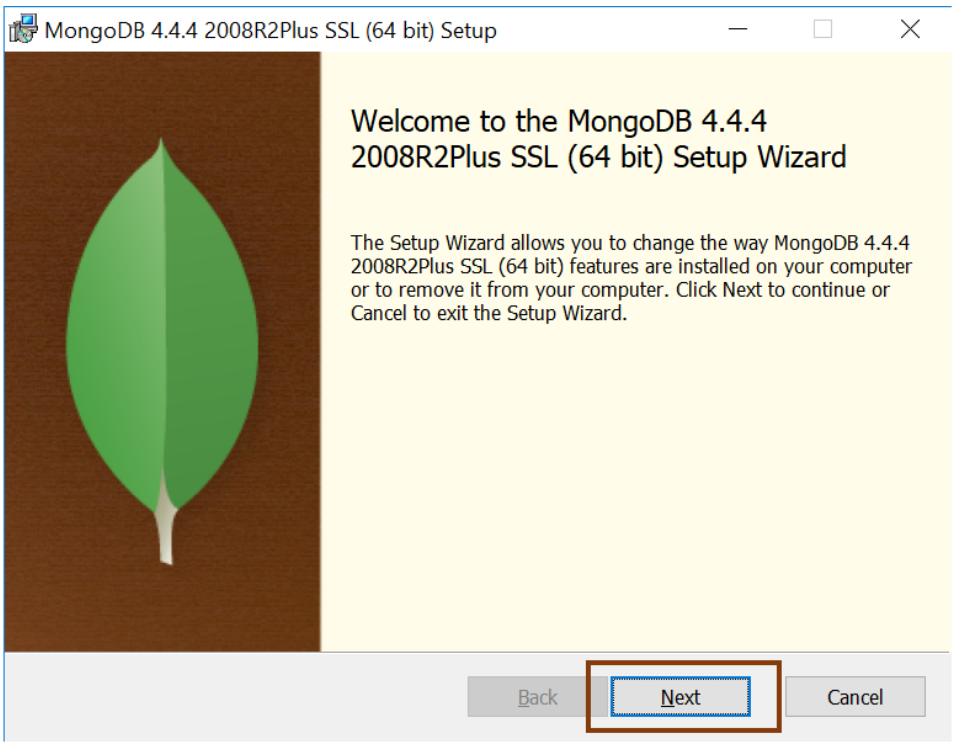
2. **Installation**: Follow platform-specific instructions.



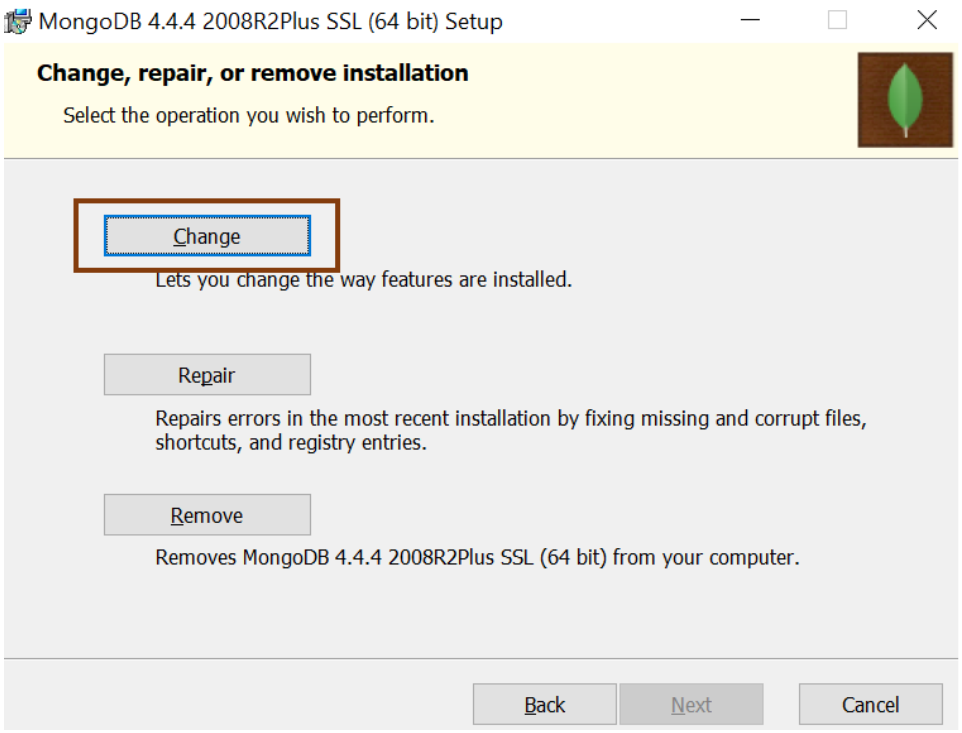
figureMongoDB System Install steps



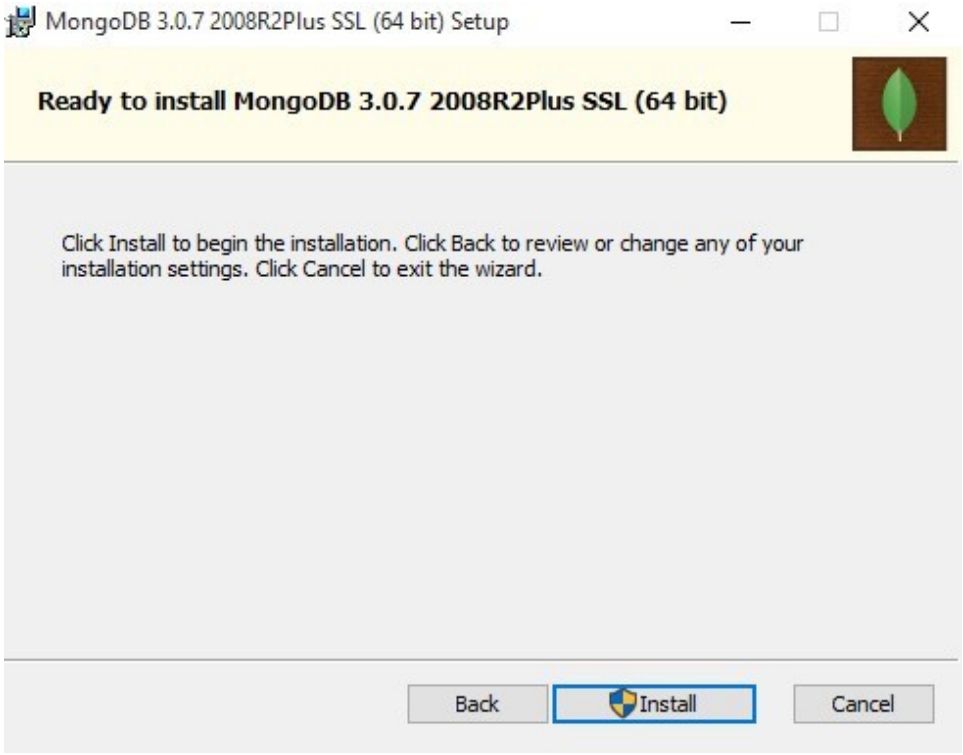
figureMongoDB System



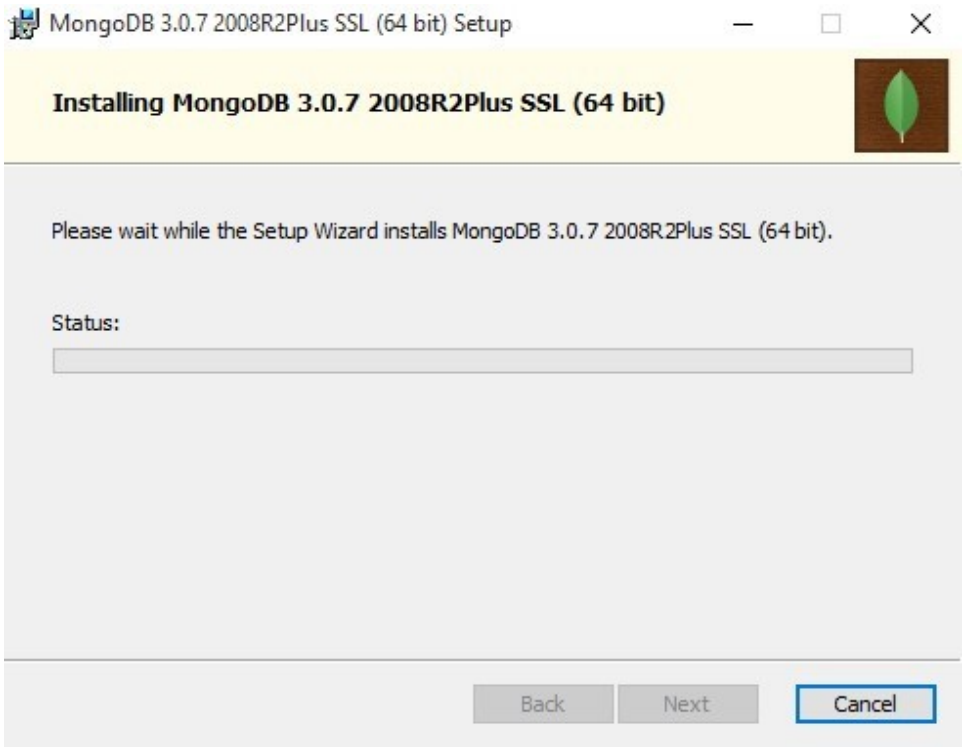
figureMongoDB System Architecture
MongoDB Installation You can find the downloaded file in the downloads directory. You can follow the steps mentioned there and install the software.



figureMongoDB System Architecture

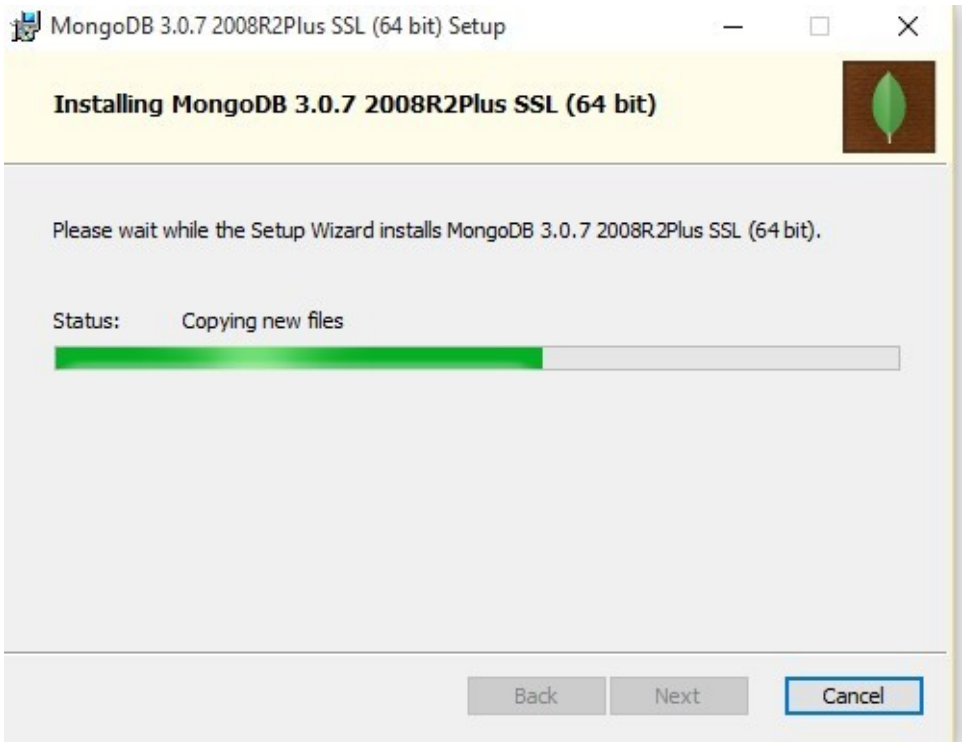


figureMongoDB System Architecture



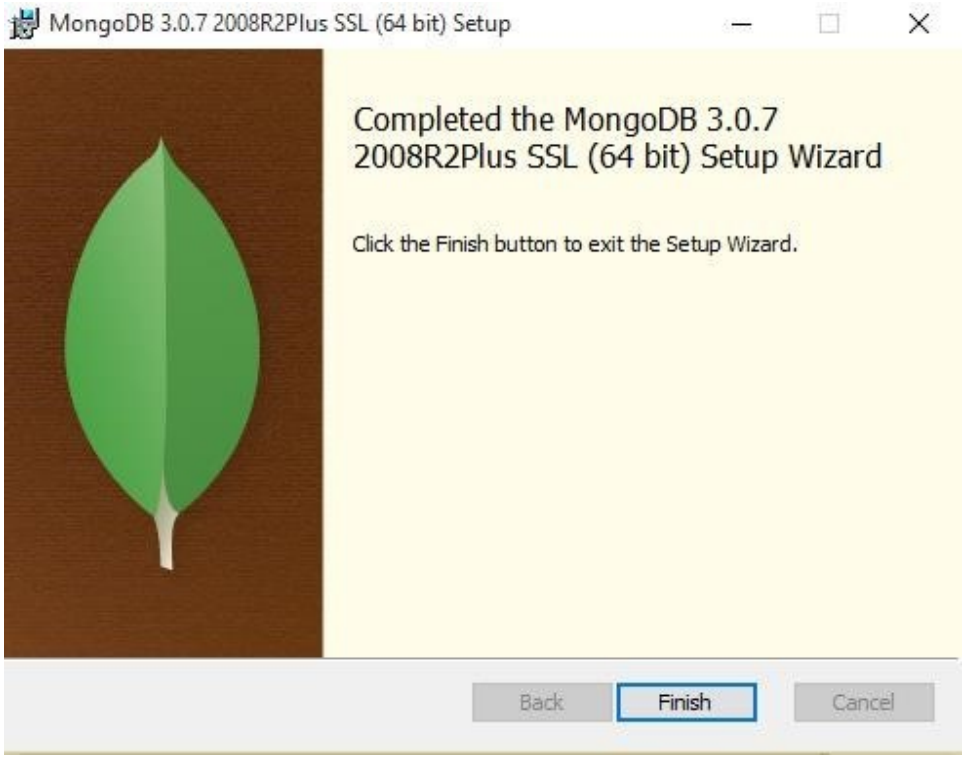
figureMongoDB System

On completing the installation successfully, you will find the software package in your C drive. C : .4.



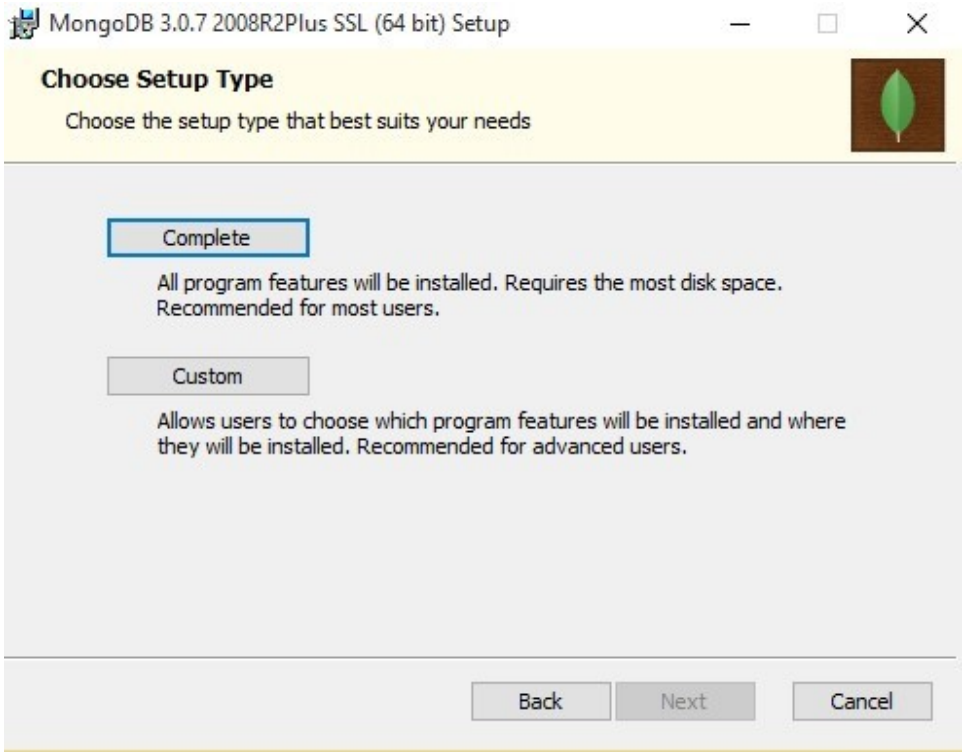
figureMongoDB System Architecture

8. Create folders in C drive: data¿db
9. Add below path in the environment variable
10. Navigate to $C :> Program Files > MongoDB > Server > 3.0 > bin$



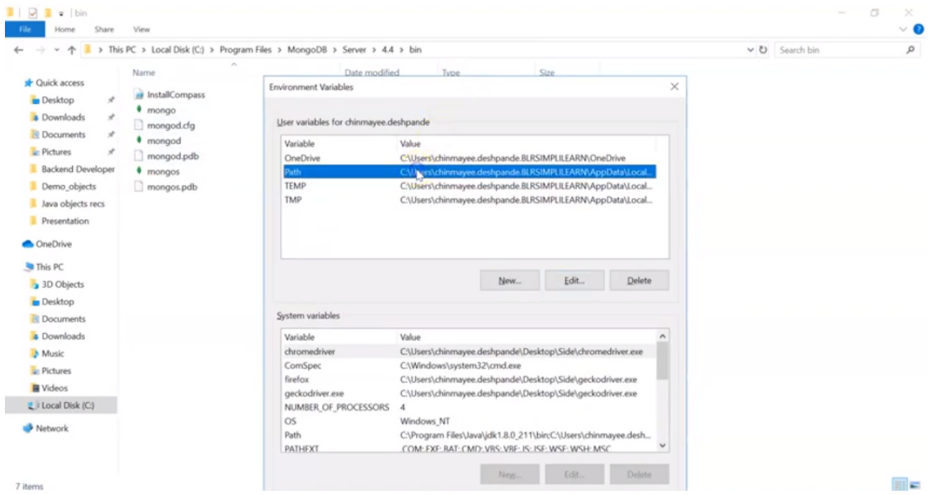
figureMongoDB System Architecture

8. Create folders in C drive: data¿db
9. Add below path in the environment variable
10. Navigate to $C :> Program Files > MongoDB > Server > 3.0 > bin$

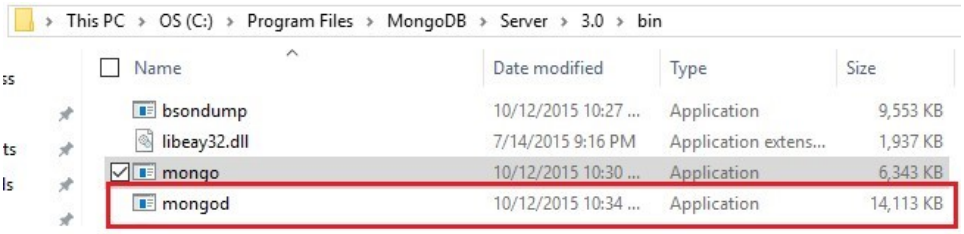


figureMongoDB System

On completing the installation successfully, you will find the software package in your C drive. C : .4.



figureMongoDB System Architecture

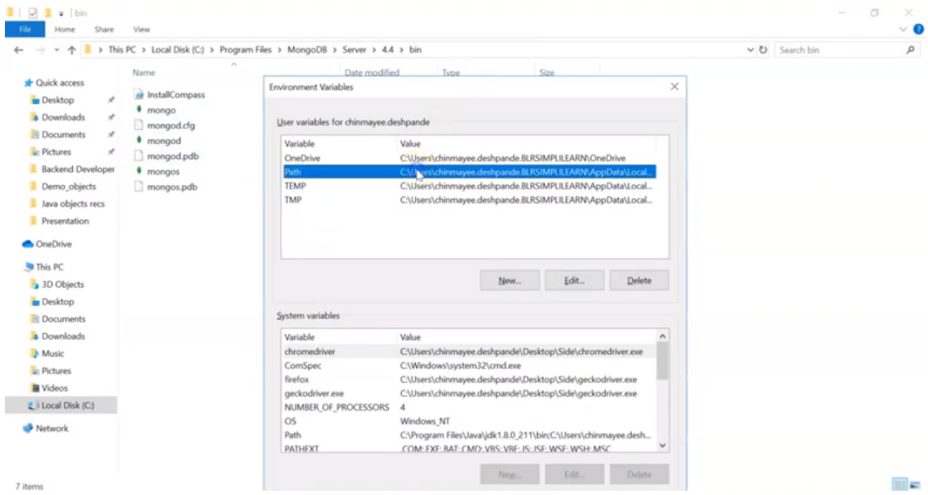


figureMongoDB System
On completing the installation successfully, you will find the software package in your C drive. $C : Files.4$.

8. Create folders in C drive: data¿db
9. Add below path in the environment variable
10. Navigate to $C :> ProgramFiles > MongoDB > Server > 3.0 > bin$

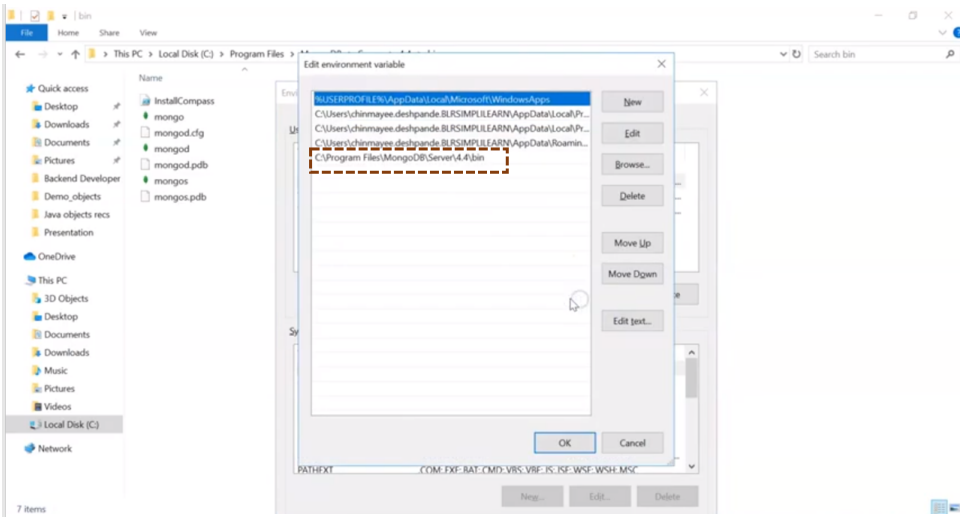
On completing the installation successfully, you will find the software package in your C drive. $C : Files.4$.

3. **Connect**: Use the MongoDB shell (`mongosh`) or a driver. You can see that there are mongo and mongod executable files. The mongod file is the daemon process that does the background jobs like accessing, retrieving, and updating the database.

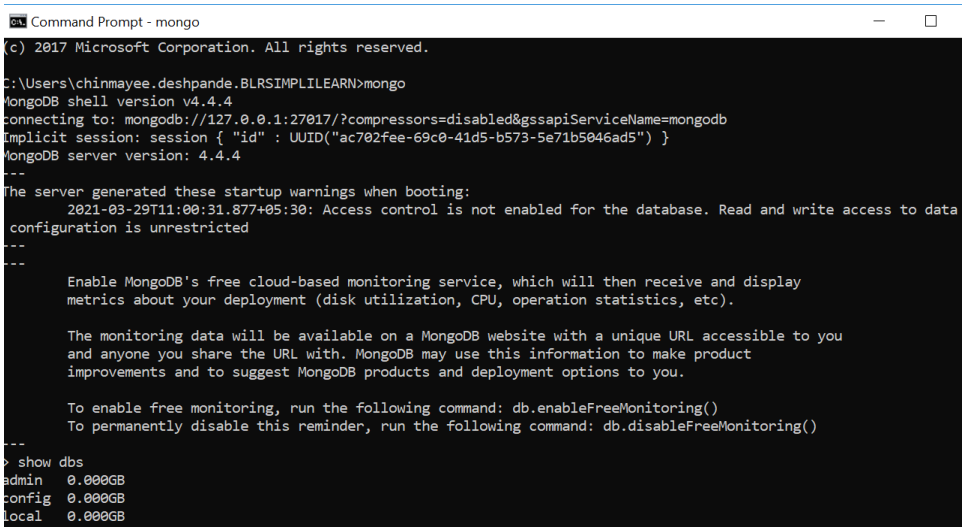


figureMongoDB System Architecture

Create an Environment Variable It's best practice to create an environment variable for the executable file so that you don't have to change the directory structure every time you want to execute the file.

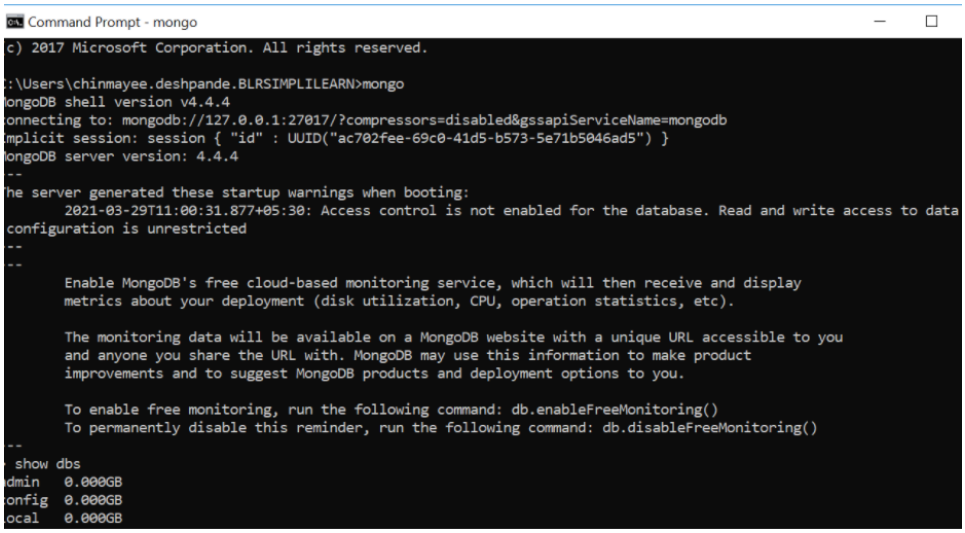


figureMongoDB System Architecture

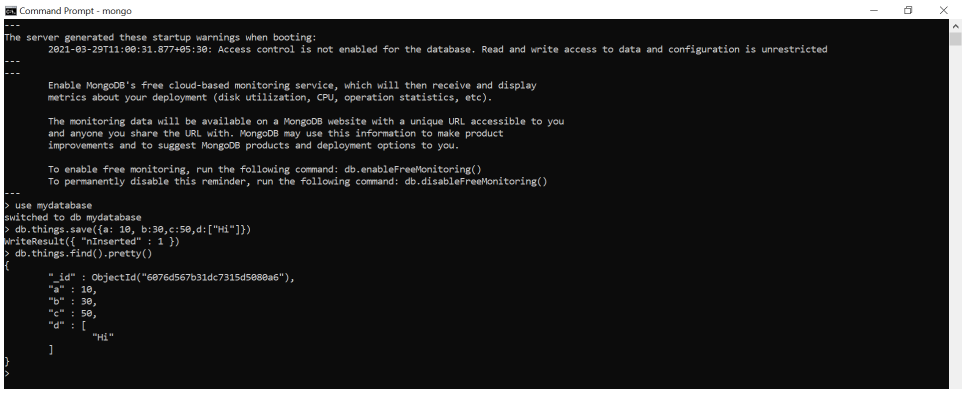


figureMongoDB System Architecture

Execute the Mongo App
After creating an environment path, you can open the command prompt and just type in mongo and press enter.



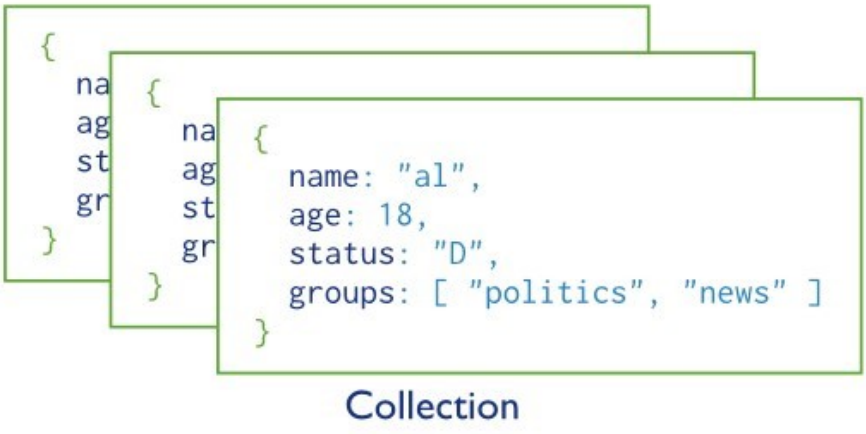
figureMongoDB System Architecture
Verify the Setup
To verify if it did the setup correctly, type in the command show DBS.



figureMongoDB System Architecture
With that, you have successfully installed and set up MongoDB on your Windows system.
This demo sample has also created a database called mydatabase, with some data added to it. It has also displayed the same using the find() method.

figureMongoDB System Architecture
With that, you have successfully installed and set up MongoDB on your Windows system.

This demo sample has also created a database called mydatabase, with some data added to it. It has also displayed the same using the find() method.



figureMongoDB System Architecture
With that, you have successfully installed and set up MongoDB on your Windows system.
This demo sample has also created a database called mydatabase, with some data added to it. It has also displayed the same using the find() method.

```
> mongod
> mongo
> show dbs
admin      40.00 KiB
config     72.00 KiB
local      40.00 KiB
```

1.2 Types of Nosql Databases

1.3 Key-Value Databases

Key-value databases, also known as **key-value stores**, are a type of NoSQL database in which data is stored as a collection of *key-value pairs*. Each key is unique and is used to retrieve its associated value efficiently. The value can be a simple data type such as a string or number, or a complex object such as JSON documents, arrays, or binary data.

A key-value pair follows the general structure:

```
key :  value
```

For example:

```
"name" : "John Drake"
```

History of Key-Value Databases

Key-value storage is not a new concept. One of the earliest examples is the Microsoft Windows Registry, which stores configuration information in a key-value format. Over time, modern distributed databases adopted this model to provide scalable and high-performance data storage.

Types of Key-Value Databases

Depending on the use case, different types of key-value databases are available:

(a) **In-Memory Key-Value Databases**
- Store data primarily in memory.
- Provide extremely fast read and write operations.
- Example: Redis.

(b) **Persistent Key-Value Databases**
- Store data on disk for durability.
- Suitable for long-term storage.
- Example: MongoDB.

(c) **Multi-Model Key-Value Databases**
- Support multiple data models such as document, graph, and key-value.
- Example: MongoDB.

How Key-Value Databases Work

A key-value database associates a unique key with a value. The key is used to perform operations on the stored data.

The three basic operations are:

(a) `put(key, value)` – Insert or update a value.

(b) `get(key)` – Retrieve a value.

(c) `delete(key)` – Remove a value.

Schema of Key-Value Databases

Unlike relational databases, key-value databases do not require a fixed schema. This flexibility allows them to efficiently handle structured, semi-structured, and unstructured data.

Grouping Similar Items

Related records can be grouped using a common key prefix.

Example:

```
customer:1:name   -> John Drake
customer:1:email  -> john.drake@gmail.com
customer:1:dob    -> 24/11/1982
customer:1:mobile -> 7843241098
```

This naming convention helps organize and retrieve related data efficiently.

Features of Key-Value Databases

(a) **Flexible Data Models**
- No predefined schema.
- Easy adaptation to changing requirements.

(b) **No Query Language Required**
- CRUD operations are performed directly using keys.
- Fast and simple data access.

(c) **Support for Complex Data Types**
- Strings, numbers, arrays, JSON documents, media files, and nested structures.

(d) **Indexing Support**
- Hash indexing.
- Tree-based indexing.
- Secondary indexing.
- Wildcard indexing.

When to Use Key-Value Databases

Key-value databases are suitable for:

- Real-time applications.
- User session management.
- Online gaming platforms.
- Financial systems.
- Caching frequently accessed data.
- Configuration management.

MongoDB as a Key-Value Store

Although MongoDB is primarily a document database, it can effectively function as a key-value store due to its flexible document structure and indexing capabilities.

Example:

```
{
    session_id : "ueyrt-jshdt-6783-utyrts",
    create_time : 1122384666000
}
```

MongoDB supports:

- Secondary Indexes
- Wildcard Indexes
- Nested Key-Value Structures
- Flexible Schema Design

Key-Value Database vs Cache

Key-Value Database	**Cache**
Persistent storage on disk	Usually memory-based
Data survives restarts	Data may be lost after restart
Suitable for long-term storage	Suitable for temporary storage
Provides durability	Optimized for speed

Table 2: Key-Value Database vs Cache

Advantages of Key-Value Databases

(a) Simple and efficient data model.
(b) Extremely fast read and write operations.
(c) High scalability.
(d) Flexible schema design.
(e) Easy implementation for caching and session management.
(f) Supports distributed architectures.
(g) Efficient indexing and retrieval mechanisms.

Limitations of Key-Value Databases

(a) Limited support for complex queries.

(b) Relationships between data items are difficult to model.

(c) Lack of standard query language.

(d) Data duplication may occur.

(e) Less suitable for applications requiring complex joins.

1.4 Wide-column stores

The **Wide-column stores** is a type of NoSQL database model that stores data in columns instead of rows. It is designed to handle large amounts of data efficiently and provides high scalability.

- Designed to support scalable data storage across multiple nodes.
- Works with big data platforms such as Apache Cassandra and Apache HBase.
- Supports horizontal scaling while maintaining high availability.

Components of Wide-column stores

Row Key

The row key is a unique identifier for each record and is used to locate and retrieve data efficiently.

Example:

```
EMP101
```

Column Family

A column family is a collection of related columns stored together.

Example:

```
Personal_Info
```

Columns

Columns store individual attributes such as Name, Age, Department, or Salary.

Example:

```
Name : John
Age  : 30
```

Cell

A cell stores the value of a column for a specific row.

Example:

```
Salary : 70000
```

Timestamp

Many columnar databases associate timestamps with cells to maintain multiple versions of data.

Example of Columnar Data Model

Row Key	Personal_Info:Name	Personal_Info:Age	Academic_Info:Course
101	Ethan	21	Data Science
102	Harper	20	Computer Science

Table 3: Example of Columnar Data Model

- Row Keys: 101, 102
- Column Families: Personal_Info, Academic_Info
- Columns: Name, Age, Course
- Values: Ethan, 21, Data Science, etc.

This approach is commonly used by Apache Cassandra and Apache HBase.

Working of Columnar Data Model

Step 1: Store Data in Columns

Row Key	Name	Course
101	Ethan	Data Science
102	Harper	Computer Science
103	Mason	Mathematics

Each record is identified using a unique row key.

Step 2: Organize Data by Column Families

```
Row Key: 101

Personal_Info
    Name = Ethan

Academic_Info
    Course = Data Science
```

Related columns are grouped into column families.

Step 3: Insert Data

Row Key	Name	Course
104	Avery	Statistics

The values are stored under the corresponding row key and column families.

Step 4: Retrieve Required Columns

Applications can retrieve only the required columns.

```
SELECT Name
FROM Students
WHERE RowKey = 101;
```

Only the requested column is returned.

Step 5: Scale for Large Datasets

Columnar databases distribute data across multiple servers to provide high scalability and availability.

Example: A university storing millions of student records can distribute data across multiple servers for faster storage and retrieval.

Popular Columnar Databases

- Apache Cassandra
- Apache HBase
- Google Bigtable
- ScyllaDB

Advantages of Columnar Data Model

(a) **Well-Structured Storage:** Data is stored in a highly organized column-wise format, improving storage efficiency.

(b) **High Scalability:** Data can be distributed across multiple servers to handle large-scale workloads.

(c) **Fast Data Loading:** Large volumes of data can be loaded and processed efficiently.

Limitations of Columnar Data Model

(a) **Complex Data Modeling:** Designing an efficient schema requires careful planning based on query patterns.

(b) **Limited Query Support:** Complex joins and aggregations are less efficient than in relational databases.

(c) **Data Redundancy:** Data is often duplicated to improve query performance, increasing storage requirements.

1.5 Graph Databases

A **Graph Database** is a type of NoSQL database that stores data as nodes, edges, and properties. It is designed to represent and manage complex relationships between data efficiently.

- Stores highly connected data using graph structures.
- Optimized for relationship-based queries.
- Widely used in social networks, recommendation systems, and fraud detection.
- Supports flexible schema and dynamic relationships.

Components of Graph Databases

1.5.1 Node

A node represents an entity or object in the graph.

Examples:

- Person
- Student
- Product
- Organization

Edge (Relationship)

An edge represents a connection or relationship between two nodes.

Examples:

- FRIEND_OF
- PURCHASED
- ENROLLED_IN
- WORKS_FOR

Properties

Properties are attributes associated with nodes or edges.

Example:

```
Name : John
Age  : 25
City : New York
```

Example of Graph Database

John — FRIEND_OF — Alice
John — WORKS_FOR — Company X
Alice — ENROLLED_IN — Data Science

Working of Graph Databases

Step 1: Create Nodes

```
(Person: John)
(Person: Alice)
(Company: Company X)
```

Step 2: Create Relationships

```
John --> FRIEND_OF --> Alice
John --> WORKS_FOR --> Company X
```

Step 3: Store Properties

```
John:
    Age = 25
    City = New York
```

Step 4: Query Relationships

Example Query:

```
Find all friends of John
```

The graph database traverses relationships directly without expensive joins.

Popular Graph Databases

- Neo4j
- Amazon Neptune
- ArangoDB
- OrientDB
- TigerGraph

Applications of Graph Databases

- Social Networking Platforms
- Recommendation Systems
- Fraud Detection
- Network and IT Operations
- Knowledge Graphs
- Supply Chain Management

Advantages of Graph Databases

(a) **Efficient Relationship Management**
   - Relationships are stored directly between nodes.

(b) **High Query Performance**
   - Fast traversal of connected data.

(c) **Flexible Schema**
   - Easy to add new nodes and relationships.

(d) **Real-Time Insights**
   - Suitable for recommendation engines and fraud detection.

(e) **Scalability**
   - Can manage large interconnected datasets efficiently.

Limitations of Graph Databases

(a) **Complex Data Modeling**
- Designing graph structures requires careful planning.

(b) **Limited Standardization**
- Different graph databases use different query languages.

(c) **Resource Intensive**
- Large graphs may require significant memory and processing power.

(d) **Not Ideal for Simple Transactions**
- Relational databases may perform better for simple tabular data.

1.6 Examining Two Simple Examples

1.6.1 Example 1: Social Network

Consider a social networking application where users are connected through friendships.

- Nodes: Users
- Relationship: FRIEND_OF

Graph Representation:

```
John ---- FRIEND_OF ---- Alice
Alice --- FRIEND_OF ---- Bob
```

Query Example:

Find all friends of Alice.

Result:

- John
- Bob

1.6.2 Example 2: Online Shopping System

Consider an e-commerce application where customers purchase products.

- Nodes: Customer, Product
- Relationship: PURCHASED

Graph Representation:

```
John ---- PURCHASED ---- Laptop
John ---- PURCHASED ---- Mobile
Emma ---- PURCHASED ---- Laptop
```

Query Example:

Find all products purchased by John.

Result:

- Laptop
- Mobile

Benefits of Graph Databases in These Examples

- Fast traversal of relationships.
- Efficient storage of connected data.
- Simplifies recommendation and relationship queries.

Table 4: Car Make and Model Dataset

S.No.	Name	Location	Year	Kilometers_Driven	Fuel_Type	Transmission	Owner_Type	Mileage	Engine	Power	Seats	New_Price	Price
1	Hyundai i20	Chennai	2018	45000	Petrol	Manual	First	18.6	1197	82	5	8.50	5.20
2	Maruti Swift	Bangalore	2019	32000	Petrol	Manual	First	21.2	1197	83	5	7.20	5.80
3	Honda City	Hyderabad	2017	55000	Diesel	Manual	Second	25.6	1498	99	5	13.50	7.40
4	Toyota Innova	Mumbai	2016	78000	Diesel	Manual	First	14.3	2494	102	7	18.20	10.50
5	Ford EcoSport	Delhi	2018	42000	Diesel	Manual	First	22.7	1498	100	5	11.80	7.20
6	Hyundai Creta	Pune	2020	25000	Petrol	Automatic	First	17.0	1497	113	5	16.50	12.80
7	Tata Nexon	Kolkata	2021	18000	Petrol	Manual	First	17.4	1199	118	5	10.20	8.90
8	Mahindra XUV500	Chennai	2017	65000	Diesel	Automatic	Second	15.1	2179	140	7	19.50	9.80
9	Volkswagen Polo	Bangalore	2019	28000	Petrol	Manual	First	18.8	1198	75	5	8.90	6.50
10	Renault Kwid	Hyderabad	2020	15000	Petrol	Manual	First	24.0	999	68	5	5.10	4.20

Working With Language Bindings

Language bindings allow developers to interact with databases using different programming languages. A database system provides drivers or APIs that act as a bridge between the application programming language and the database. Using language bindings, developers can perform database operations such as connecting to a database, inserting data, updating records, deleting data, and executing queries without directly using the database command-line interface.

Need for Language Bindings

Modern applications are developed using multiple programming languages such as Python, Java, JavaScript, C++, and C#. Database language bindings provide the following advantages:

- Easy integration between applications and databases.
- Platform-independent database access.
- Support for application development using preferred programming languages.
- Reduced complexity in database communication.
- Improved productivity through predefined database APIs.

Language Bindings in NoSQL Databases

NoSQL databases provide language-specific drivers that enable applications to communicate with the database. Different NoSQL databases support multiple language bindings.

Advantages of Language Bindings

- Provides simple programming interfaces.
- Supports rapid application development.
- Enables interaction with distributed databases.
- Provides error handling and security features.
- Reduces dependency on database-specific commands.

UNIT–2: INTERACTING WITH NoSQL

NoSQL databases provide flexible, scalable, and high-performance data storage solutions for modern applications. Unlike traditional relational databases, NoSQL systems support unstructured, semi-structured, and large-scale distributed data. Interacting with NoSQL databases involves connecting applications, performing database operations, and managing stored information using different programming languages and APIs.

If NoSQL Then What?

NoSQL (Not Only SQL) databases are designed to overcome the limitations of traditional relational database management systems (RDBMS). They provide flexible schemas, horizontal scalability, and efficient processing of large volumes of data. NoSQL databases are preferred when applications require:

- Handling large volumes of rapidly changing data.
- Flexible data models without fixed schemas.
- High availability and fault tolerance.
- Distributed data storage across multiple servers.
- Fast read and write operations.

Types of NoSQL Data Stores

The major categories of NoSQL databases are:

(a) **Key–Value Stores:** Store data as key and value pairs. Example: Redis, Amazon DynamoDB.

(b) **Document Stores:** Store data as JSON-like documents. Example: MongoDB, CouchDB.

(c) **Column-Family Stores:** Store data in column-oriented structures. Example: Apache Cassandra, HBase.

(d) **Graph Databases:** Store data using nodes, edges, and relationships. Example: Neo4j.

Language Bindings for NoSQL Data Stores

Language bindings allow programming languages to communicate with NoSQL databases through drivers and APIs. They provide methods for connecting, querying, and manipulating database records.

Table 5: Language Bindings for NoSQL Databases

Database	Language	Binding/Driver
MongoDB	Python	PyMongo
MongoDB	Java	MongoDB Java Driver
MongoDB	JavaScript	Node.js Driver
Cassandra	Java	DataStax Driver
Redis	Python	Redis-py
Neo4j	Python	Neo4j Driver

Examples of NoSQL Language Bindings

Example: Connecting MongoDB Using Python

```
from pymongo import MongoClient

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

db = client["University"]

collection = db["Student"]

print("Connected Successfully")
```

2 Performing CRUD Operations

CRUD represents the four basic operations performed on database records.

- **Create:** Adding new records into the database.
- **Read:** Retrieving existing records from the database.
- **Update:** Modifying existing records.
- **Delete:** Removing records from the database.

Creating Records

Creating records involves inserting new documents or data items into a NoSQL database.

2.1 MongoDB Insert Operation

Example:

```
db.Student.insertOne(
{
"name":"Rahul",
"age":21,
"course":"CSE"
}
)
```

Multiple records can be inserted using:

```
db.Student.insertMany([
{
"name":"Anu",
"age":20
},
{
"name":"Ravi",
"age":22
}
])
```

String

Used to store textual data.

Command:

```
> db.types.insertOne({ name: "MongoDB", type: "String" })
```

Output:

```
{ "acknowledged" : true, "insertedId" : ObjectId("...") }
```

Integer

Stores numerical values without decimals.

Command:

```
> db.types.insertOne({ quantity: 100, type: "Integer" })
```

Output:

```
{ "acknowledged" : true, "insertedId" : ObjectId("...") }
```

Double (Floating Point)

Stores decimal values.

Command:

```
> db.types.insertOne({ price: 99.99, type: "Double" })
```

Output:

```
{ "acknowledged" : true, "insertedId" : ObjectId("...") }
```

Boolean

Stores either `true` or `false`.

Command:

```
> db.types.insertOne({ isActive: true, type: "Boolean" })
```

Array

Stores multiple values in a single key.

Command:

```
> db.types.insertOne({ colors: ["Red", "Green", "Blue"],
   type: "Array" })
```

Object

Stores embedded documents (nested fields).

Command:

```
> db.types.insertOne({
    type: "Object",
    specs: { engine: "V8", fuel: "Petrol" }
})
```

Date

Stores date/time values.

Command:

```
> db.types.insertOne({
    created_at: new Date(),
    type: "Date"
})
```

Null

Represents a null or missing value.

Command:

```
> db.types.insertOne({ value: null, type: "Null" })
```

ObjectId

Every document has a unique `_id` field of type ObjectId.

Command:

```
> db.types.findOne()
```

Output (sample):

```
{
  "_id" : ObjectId("64f0b4bfe0abf12345678901"),
  ...
}
```

2.2 Accessing Data

Accessing data refers to retrieving stored records from a NoSQL database.

MongoDB Find Operation

Retrieve all documents:

```
db.Student.find()
```

Retrieve a specific document:

```
db.Student.find(
{"name":"Rahul"}
)
```

2.3 Updating Data

Updating modifies existing records in the database.

MongoDB Update Operation

Example:

```
db.Student.updateOne(
{"name":"Rahul"},
{
$set:
{
"age":22
}
}
)
```

2.4 Deleting Data

Deleting removes unwanted records from a NoSQL database.

2.5 MongoDB Delete Operation

Delete a single record:

```
db.Student.deleteOne(
{"name":"Rahul"}
)
```

Delete multiple records:

```
db.Student.deleteMany(
{"course":"CSE"}
)
```

Data Types in MongoDB

MongoDB uses BSON, supporting:

- **String**: UTF-8 strings.
- **Integer**: 32-bit or 64-bit integers.
- **Double**: Floating-point numbers.
- **Boolean**: `true`/`false`.
- **ObjectId**: Unique document identifier.
- **Array**: Lists, e.g., `["item1", "item2"]`.
- **Object**: Nested documents.
- **Date**: ISODate objects.

- **Null**: Null or missing values.

Example:

```
{
    "_id": ObjectId("507f1f77bcf86cd799439011"),
    "name": "Alice",
    "age": 25,
    "active": true,
    "scores": [90, 85, 88],
    "address": { "city": "NY" },
    "createdAt": ISODate("2025-06-27T00:00:00Z")
}
```

3 MongoDB CRUD Operations Using Car4U Database

CRUD operations are the basic operations performed on MongoDB collections. CRUD represents Create, Read, Update, and Delete operations. In this example, a **Car4U** database is created to store car details. The operations are performed on the **car** collection.

Creating Database and Collection

Create (insert) operations are used to add new documents to a collection, creating the collection automatically if it does not already exist. Adds new documents to a collection. Automatically creates the collection if missing. Generates a unique *id* if not provided. Supports inserting single or multiple documents. MongoDB automatically creates a database and collection when data is inserted.The following command creates a Car4U database and a car collection.

```
use Car4U

db.createCollection("car")
```

Output:

```
switched to db Car4U

{ "ok" : 1 }
```

Create Operations

Create operations are used to insert new documents into a MongoDB collection. MongoDB provides two methods:

- insertOne() - Inserts a single document.
- insertMany() - Inserts multiple documents.

Inserting a Single Car Record

```
db.car.insertOne(
{
 "Name":"Hyundai Creta",
 "Location":"Hyderabad",
 "Year":2022,
 "Kilometers_Driven":15000,
 "Fuel_Type":"Petrol",
 "Transmission":"Automatic",
 "Owner_Type":"First",
 "Mileage":"16.8 kmpl",
 "Engine":"1497 CC",
 "Power":"113 bhp",
 "Seats":5,
 "Price":15.50
}
)
```

Output:

```
{
 acknowledged: true,
 insertedId: ObjectId('65a3f21a89')
}
```

Inserting Multiple Car Records

```
db.car.insertMany([
{
"Name":"Toyota Innova",
"Year":2021,
"Fuel_Type":"Diesel",
"Price":18.75
},

{
"Name":"Maruti Swift",
"Year":2023,
"Fuel_Type":"Petrol",
"Price":8.25
}
])
```

Output:

```
{
```

```
 acknowledged: true,
 insertedIds:
 {
 0:ObjectId('65a3f21a90'),
 1:ObjectId('65a3f21a91')
 }
}
```

3.1 Read Operations

Read operations retrieve documents from MongoDB collections. MongoDB provides:

- find() - Retrieves multiple documents.
- findOne() - Retrieves a single document.

3.1.1 Display All Cars

```
db.car.find().pretty()
```

Output:

```
{
 Name:"Hyundai Creta",
 Location:"Hyderabad",
 Year:2022,
 Fuel_Type:"Petrol",
 Price:15.50
}

{
 Name:"Toyota Innova",
 Year:2021,
 Fuel_Type:"Diesel",
 Price:18.75
}

{
 Name:"Maruti Swift",
 Year:2023,
 Fuel_Type:"Petrol",
 Price:8.25
}
```

Finding a Specific Car

```
db.car.findOne(
{
"Name":"Hyundai Creta"
}
)
```

Output:

```
{
 Name:"Hyundai Creta",
 Year:2022,
 Fuel_Type:"Petrol",
 Price:15.50
}
```

3.2 Update Operations

Update operations modify existing documents in MongoDB.

MongoDB update methods:

- updateOne()
- updateMany()
- replaceOne()

Updating a Single Car Record

Updating the price of Hyundai Creta:

```
db.car.updateOne(
{
"Name":"Hyundai Creta"
},
{
$set:
{
"Price":16.25
}
}
)
```

Output:

```
{
 acknowledged:true,
 matchedCount:1,
 modifiedCount:1
}
```

Updating Multiple Records

Updating fuel type of all petrol cars:

```
db.car.updateMany(
{
"Fuel_Type":"Petrol"
},
{
$set:
{
"Transmission":"Automatic"
}
}
)
```

Output:

```
{
 acknowledged:true,
 matchedCount:2,
 modifiedCount:2
}
```

Replacing a Complete Document

```
db.car.replaceOne(
{
"Name":"Maruti Swift"
},
{
"Name":"Maruti Baleno",
"Year":2024,
"Fuel_Type":"Petrol",
"Price":9.50
}
)
```

Output:

```
{
 acknowledged:true,
 matchedCount:1,
 modifiedCount:1
}
```

Delete Operations

Delete operations remove documents from MongoDB collections.

MongoDB provides:

- deleteOne() - Deletes one document.
- deleteMany() - Deletes multiple documents.

Deleting a Single Car Record

```
db.car.deleteOne(
{
"Name":"Toyota Innova"
}
)
```

Output:

```
{
 acknowledged:true,
 deletedCount:1
}
```

Deleting Multiple Car Records

Delete all cars with price less than 10 lakhs:

```
db.car.deleteMany(
{
"Price":
{
$lt:10
}
}
)
```

Output:

```
{
 acknowledged:true,
 deletedCount:1
}
```