SmartCliff
Career Mobility Solutions

Unit III
NoSQL Databases

Contents

Introduction to NoSQL

Relational vs Non relational database

- Non-relational and relational databases are designed to support different application requirements and they typically co-exist in most enterprises.
- The key decision points on when to use which include the following:

Relational Database	Non-Relational Database
Centralized applications	Decentralized applications
Moderate to high availability	Continuous availability
Moderate velocity data	High velocity data
Data ingestion from one location	Data ingestion from multiple locations
Structured data	Semi structured or unstructured data
Complex transactions	Simple transactions
Maintain moderate data volumes	Maintain high data volume; retain forever

What is NoSQL?

- NoSQL databases are **non-tabular databases** and store data differently than relational tables.
- NoSQL databases come in a variety of types based on their data model. The main types are **document, key-value, wide-column, and graph.**
- They provide **flexible schemas and scale easily** with large amounts of data and high user loads.
- The term NoSQL originally could be taken at its word -- that is, SQL was not used as the API to access data.
- However, the ubiquity and usefulness of SQL caused **many NoSQL databases to add support for SQL**.
- Today it is commonly accepted that NoSQL stands for "**Not Only SQL**"

Why should you use a NoSQL database?

- NoSQL databases are a great fit for many **modern applications** such as mobile, web, and gaming that require **flexible, scalable, high-performance, and highly functional databases** to provide great user experiences.

  Flexibility: NoSQL databases generally provide flexible schemas that enable faster and more iterative development. The flexible data model makes NoSQL databases ideal for **semi-structured and unstructured data**.

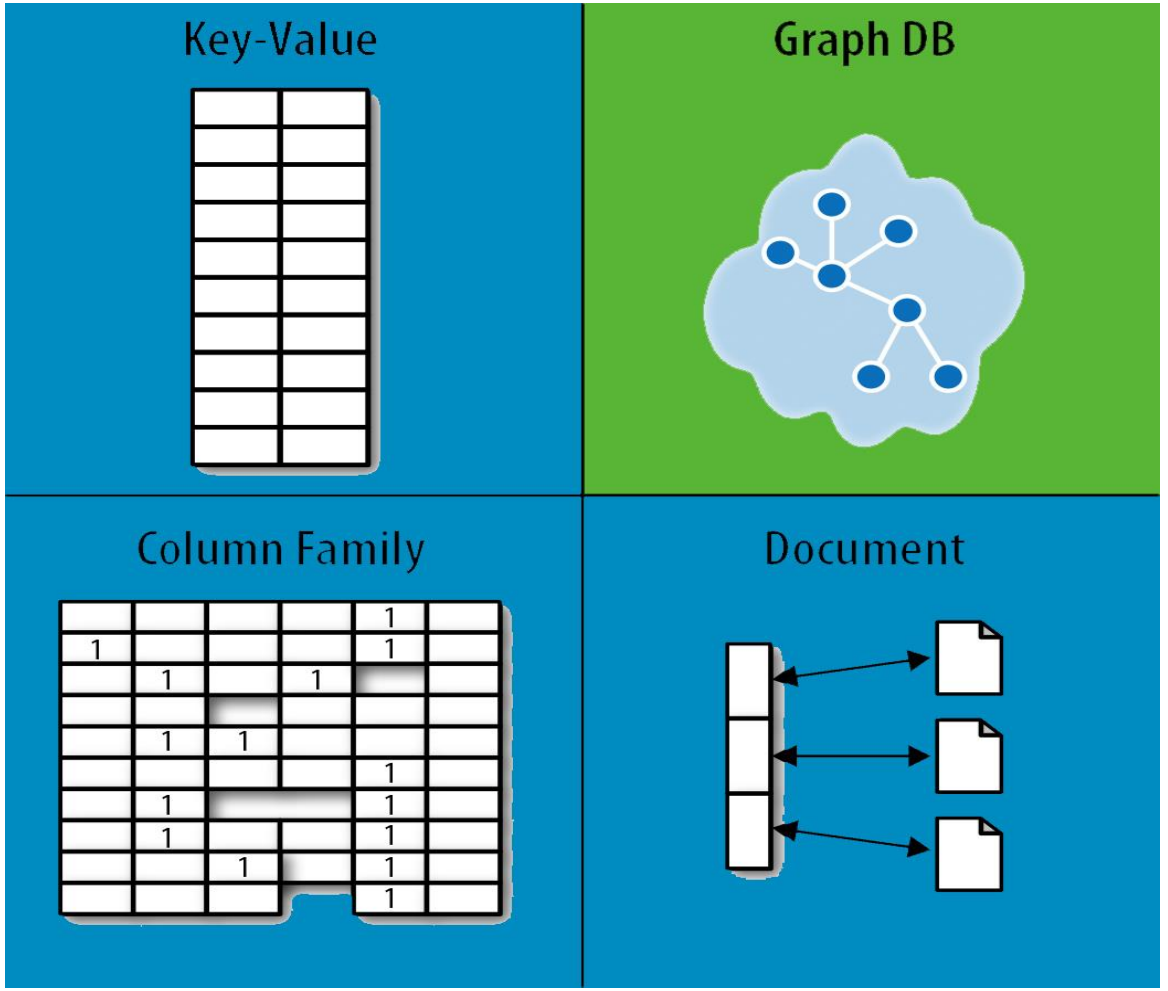
  Scalability: NoSQL databases are generally designed to scale out by using **distributed clusters** of hardware instead of scaling up by adding expensive and robust servers. Some cloud providers handle these operations behind-the-scenes as a fully managed service.

  High-performance: NoSQL database are **optimized** for specific data models and access patterns that enable higher performance than trying to accomplish similar functionality with relational databases.

  Highly functional: NoSQL databases provide highly functional **APIs** and data types that are purpose built for each of their **respective data models**.

Types of NoSQL Databases

- Key-value databases
- Wide-column stores
- Document databases
- Graph databases



Key-value databases(key-value store)

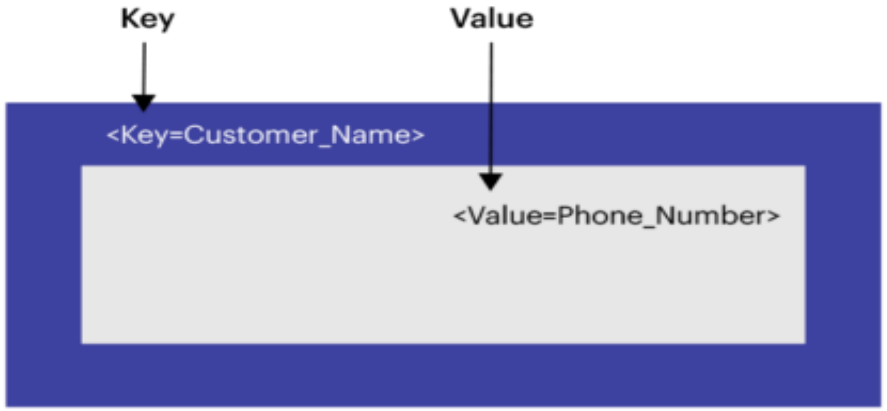
- Use the **associative array** (also called a **map or dictionary**) as their fundamental data model.
- Data is represented as a collection of **key–value pairs**.
- A value can typically only be retrieved by referencing its key.
- Key-value stores have no query language but they do provide a way to **add and remove key-value pairs**.
- Values cannot be queried or searched upon.
- **Only the key can be queried**.

Key-value databases(key-value store)

- **Popular Databases:**
  - Amazon DynamoDB, Oracle NoSQL Database, Infinity DB, Redis, Aerospike, Riak KV, Voldemort
- **Example:**

Phone directory

Key	Value
Paul	(091) 9786453778
Greg	(091) 9686154559
Marco	(091) 9868564334



Key-value databases(key-value store)

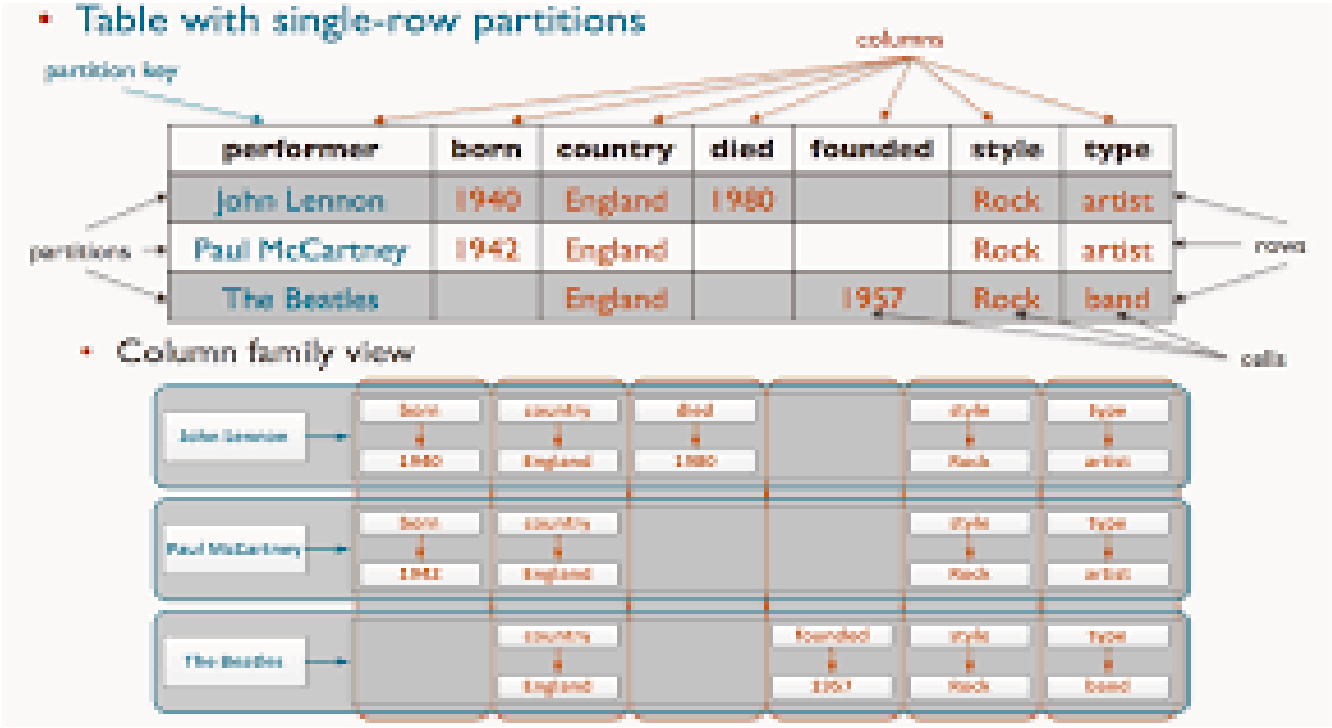
- **Use cases**
  - Key-value data stores are ideal **for storing user profiles, blog comments, product recommendations, and session information.**
    - **Twitter** uses Redis to deliver your Twitter timeline
    - **Pinterest** uses Redis to store lists of users, followers, unfollowers, boards.
    - **Coinbase** uses Redis to enforce rate limits and guarantee correctness of Bitcoin transactions
    - **Quora** uses Memcached to cache results from slower, persistent databases

Wide-column stores

- Store data in tables, rows, and **dynamic columns**.
- Wide-column stores provide a lot of flexibility over relational databases because each row is not required to have the same columns
- Wide-column stores are great for when you need to store large amounts of data and you can predict what your query patterns will be.
- Wide-column stores are commonly used for **storing Internet of Things data and user profile data**.

Wide-column stores

- **Popular Databases:**
  - Cassandra, HBase, Scylla, Google Bigtable, Cosmos DB
- **Example:**



Wide-column stores

- **Use cases**
  - Wide-column stores are commonly used for storing Internet of Things data and user profile data.
    - **Spotify** uses Cassandra to store user profile attributes and metadata about artists, songs, etc. for better personalization
    - **Facebook** use HBase for services like the Nearby Friends feature and search indexing
    - **Outbrain** uses Cassandra to serve over 190 billion personalized content recommendations each month

Document databases

- Document store databases use **JSON, XML, or BSON** documents to store data.
- Each document contains pairs of fields and values.
- The values can typically be a **variety of types** including things like strings, numbers, booleans, arrays, or objects, and their structures typically align with objects developers are working with in code.
- They can horizontally scale-out to accommodate large data volumes.
- **Horizontal scaling, or scaling out** - add more databases or divide your large database into smaller nodes, using a **data partitioning approach called sharding**, which can be managed faster and more easily across servers.

Document databases

- **Popular Databases:**
  - MarkLogic, MongoDB, Couchbase, IBM Cloudant, CouchDB
- **Example:**

Document databases

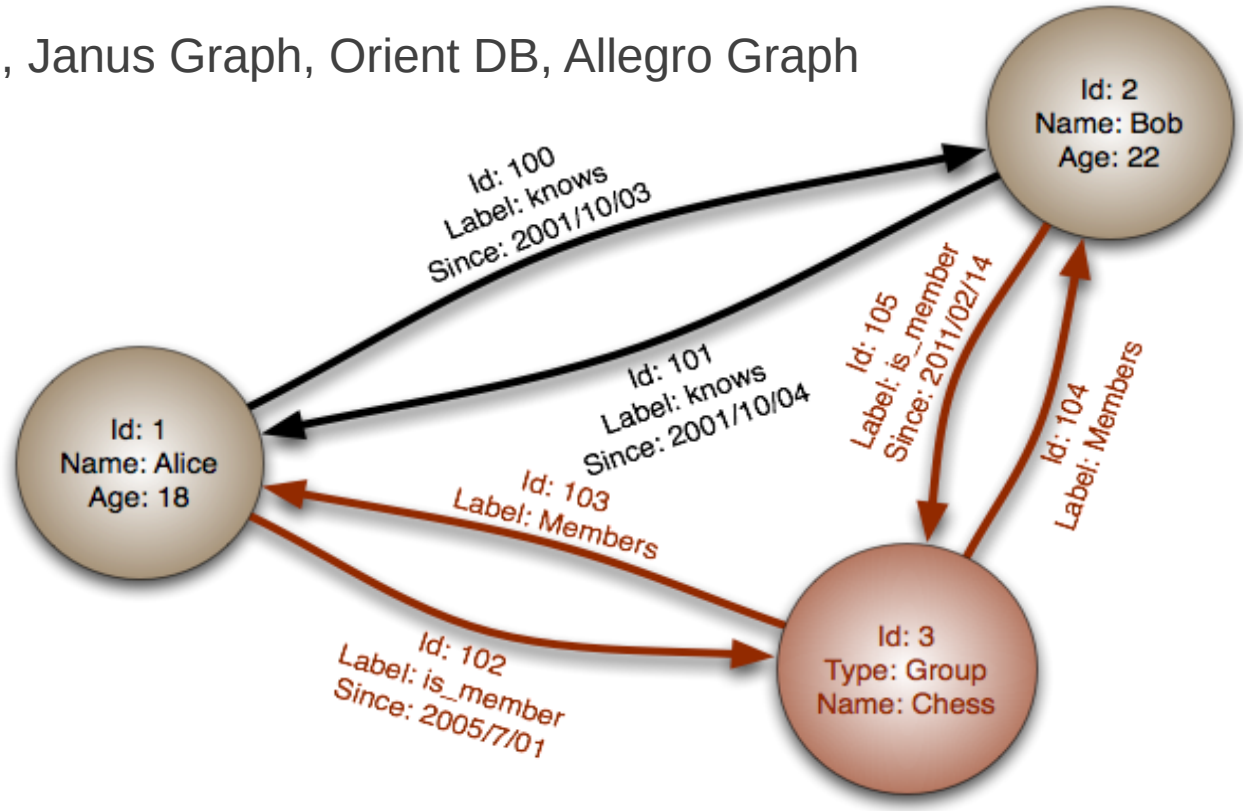
- **Use cases**
  - **SEGA** uses **MongoDB** for handling 11 million **in-game** accounts
  - **Cisco** moved its **VSRM** (video session and research manager) platform to **Couchbase** to achieve greater scalability
  - Built on **MongoDB,** The Weather Channel's **iOS and Android apps** deliver **weather alerts** to 40 million users in real-time

Graph databases

- Nodes and relationships are the bases of graph databases.
- A **node** represents an **entity**, like a user, category, or a piece of data.
- A **relationship** represents how two nodes are **associated**.
- Graph databases use nodes that contain lists of relationship records.
- These records represent relationships with other nodes, which eliminates the (time-consuming) search-match operation found in relational databases.

Graph databases

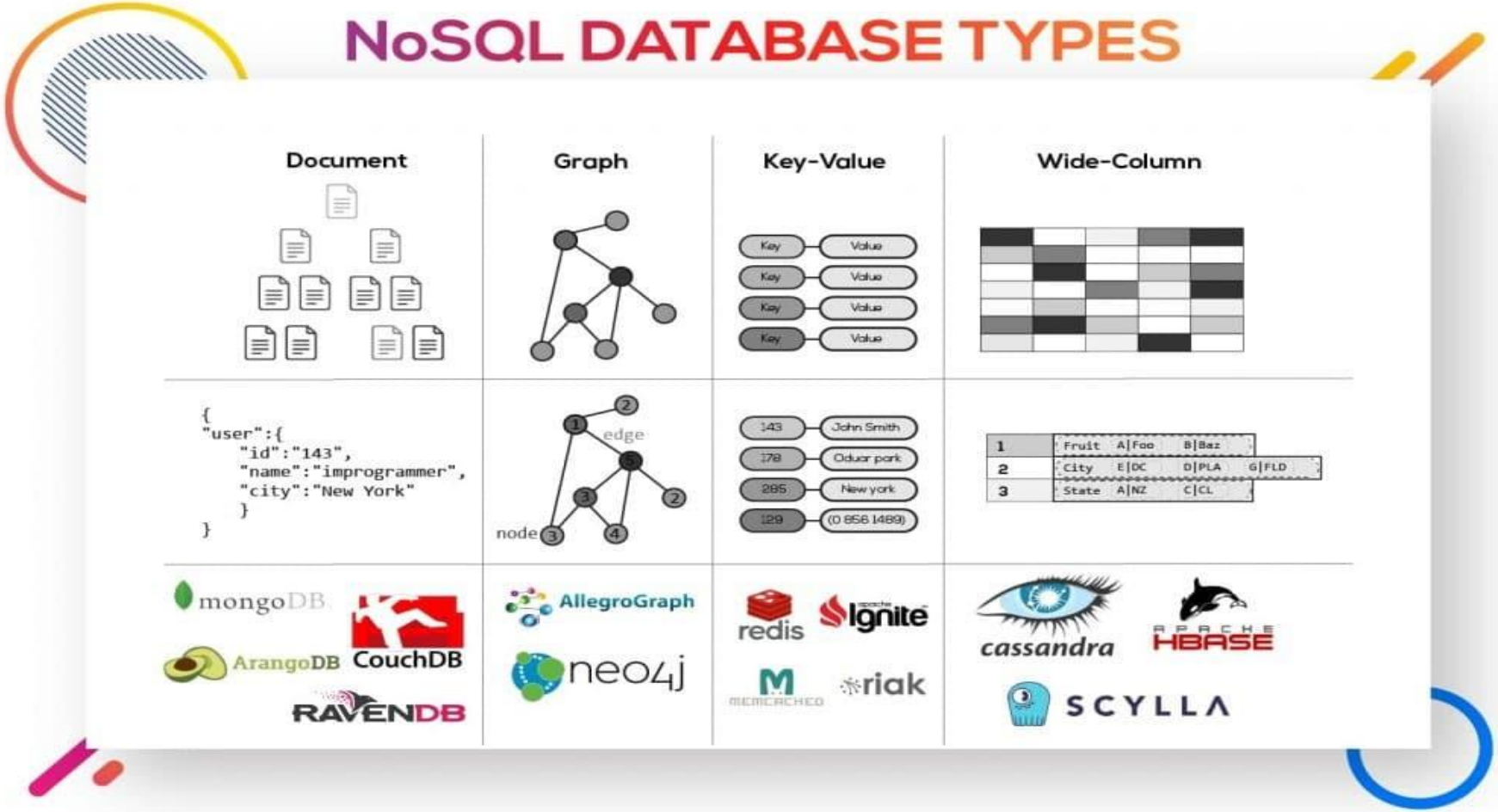
- **Popular Databases:**
  - Neo4j, Janus Graph, Orient DB, Allegro Graph
- **Example:**



Graph databases

- **Use cases**
  - Graph databases are great for establishing data relationships especially when dealing with large data sets.
  - **Walmart** uses **Neo4j** to provide customers personalized, relevant product recommendations and promotions
  - **Medium** uses **Neo4j** to build their social graph to enhance content personalization
  - **Cisco** uses **Neo4j** to mine customer support cases to anticipate bugs

Types of NoSQL Databases



CAP Theorem

CAP Theorem

- The CAP theorem is used to makes system designers aware of the trade-offs while designing networked shared-data systems.
- CAP theorem has influenced the design of many distributed data systems.
- It is very important to understand the CAP theorem as it makes the basics of **choosing any NoSQL database based on the requirements.**
- CAP theorem states that in networked shared-data systems or distributed systems, we can only achieve **at most two** out of three guarantees for a database: **Consistency, Availability and Partition Tolerance.**

Consistency in CAP Theorem

- Consistency means that all clients see the **same data at the same time**, no matter which node they connect to.
- For this to happen, whenever data is written to one node, it must be **instantly forwarded or replicated** to all the other nodes in the system before the write is deemed ‘successful.’
- It means that the nodes will have the same copies of a replicated data item visible for various transactions.
- A guarantee that every node in a distributed cluster returns the same, most recent, successful write.
- Consistency refers to every client having the **same view of the data**.
- Consistency in CAP refers to **sequential consistency**, a very strong form of consistency.

Availability in CAP Theorem

- Availability means that that any client making a request for data **gets a response**, even if one or more nodes are down.
- Another way to state this—all working nodes in the distributed system **return a valid response** for any request, without exception.
- It means that each read or write request for a data item will either be processed successfully or will receive a message that the operation cannot be completed.
- **Every non-failing node** returns a response for all read and write requests in a reasonable amount of time. The key word here is every.
- To be available, every node on (either side of a network partition) must be able to respond in a **reasonable amount of time**.

Partition tolerance in CAP Theorem

- A **partition is a communications break** within a distributed system—a lost or temporarily delayed connection between two nodes.
- **Partition tolerance** means that the **cluster must continue to work** despite any number of communication breakdowns between nodes in the system.
- It means that the system can continue operating if the network connecting the nodes has a fault that results in two or more partitions, where the nodes in each partition can only communicate among each other.
- That means, the **system continues to function and upholds its consistency** guarantees in spite of network partitions. Network partitions are a fact of life.
- Distributed systems guaranteeing partition tolerance can gracefully recover from partitions once the partition heals.

CAP theorem NoSQL database types

NoSQL databases are classified based on the two CAP characteristics they support:

- **CP database:** A CP database delivers **consistency and partition tolerance** at the expense of availability. When a partition occurs between any two nodes, the system has to shut down the non-consistent node (i.e., make it unavailable) until the partition is resolved.
- **AP database:** An AP database delivers **availability and partition tolerance** at the expense of consistency. When a partition occurs, all nodes remain available but those at the wrong end of a partition might return an older version of data than others. (When the partition is resolved, the AP databases typically resync the nodes to repair all inconsistencies in the system.)
- **CA database:** A CA database delivers **consistency and availability** across all nodes. It can’t do this if there is a partition between any two nodes in the system, however, and therefore can’t deliver fault tolerance.

Sharding

Sharding

- It is a **partitioning pattern** that places each partition in potentially separate servers - potentially all over the world.
- This scale out works well for accessing different parts of the data set with performance

Features

- Makes the Database smaller
- Makes the Database faster
- Makes the Database much more easily manageable
- Can be a complex operation sometimes
- Reduces the transaction cost of the Database

Sharding

Example

Consider database of a college in which all the student's record (present and past) in the whole college are maintained in a single database. So, it would contain very large number of data, say 100, 000 records.

- Now when we need to find a student from this Database, each time around 100, 000 transactions has to be done to find the student, which is very costly.
- Now consider the same college students records, **divided into smaller data shards based on years**.
- Now each data shard will have around 1000-5000 students records only.
- So not only the database became much **more manageable**, but also the **transaction cost** of each time also **reduces** by a huge factor, which is achieved by Sharding.

Introduction to MongoDB

Introduction to MongoDB

- Cross-platform ,**document-oriented** database.
- Leading modern, **general purpose database** platform.
- Designed for ease of **development and scaling**.
- Developed by MongoDB Inc. and licensed under the **Server Side Public License (SSPL).**
- Classified as a **NoSQL database**.

Introduction to MongoDB

- MongoDB has improved performance
  - Makes data storage **faster and easier** by using **dynamic database schemas** similar to **JSON** instead of traditional relational database systems of tables and SQL.
- Dynamic database schema used is called the **BSON** (Binary JavaScript Object Notation - binary-encoded serialization of JSON documents)

Introduction to MongoDB

- MongoDB was founded in **2007** by **Dwight Merriman, Eliot Horowitz and Kevin Ryan** – the team behind DoubleClick.
- Internet **advertising company DoubleClick** (now owned by Google), the team developed and used many custom data stores to work around the shortcomings of existing databases.
- The business served 400,000 ads per second, but often struggled with both scalability and agility.
- Frustrated, the team was inspired to create a database that tackled the challenges it faced at DoubleClick. This was when MongoDB was born.

MongoDB vs RDBMS

MongoDB	RDBMS
It is a **non-relational** and **document-oriented** database.	It is a **relational** database.
It is suitable for **hierarchical data storage**.	It is not suitable for hierarchical data storage.
It has a **dynamic schema**.	It has a **predefined schema**.
It centers around the **CAP theorem** (Consistency, Availability, and Partition tolerance).	It centers around **ACID properties** (Atomicity, Consistency, Isolation, and Durability).
In terms of performance, it is much **faster** than RDBMS.	In terms of performance, it is **slower** than MongoDB.

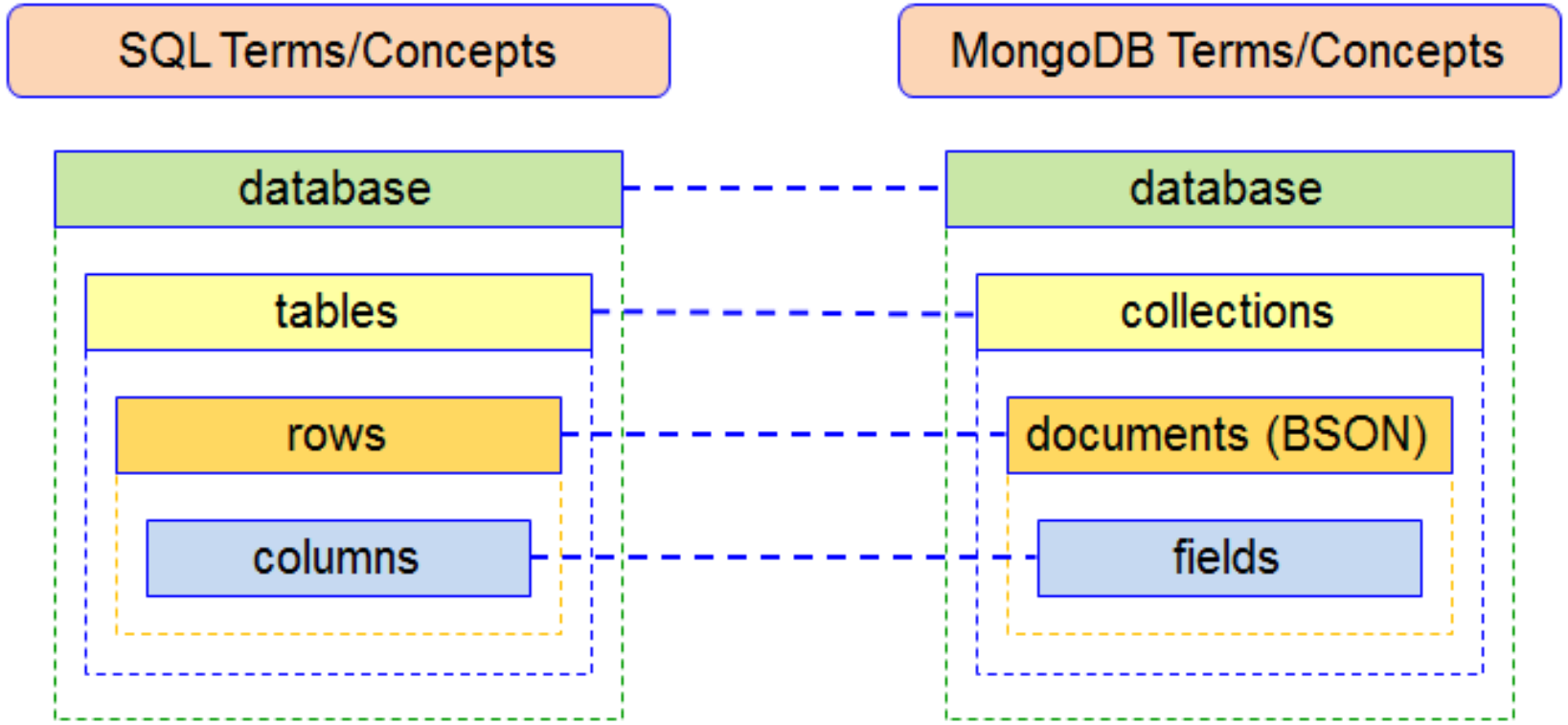
MongoDB Features

- Ad–hoc queries
- Document Data model
- Indexes
- Replication
- Speed and Durability
- Scaling

Database, Collections and Documents in MongoDB

- **Database** is a physical container for collections. Each database gets its own set of files on the file system. A single MongoDB server typically has multiple databases.
- **Collections** is a group of MongoDB documents. It is the **equivalent of an RDBMS table**.
- A collection exists within a single database.
- Collections do not enforce a schema. **Documents within a collection can have different fields**.
- **Document** is a set of **key-value** pairs. Documents have **dynamic schema**.

Database, Collections and Documents in MongoDB



Database, Collections and Documents in MongoDB

Relational

ID	first_name	last_name	cell	city	year_of_birth	location_x	location_y
1	'Mary'	'Jones'	'516-555-2048'	'Long Island'	1986	'-73.9876'	'40.7574'

ID	user_id	profession
10	1	'Developer'
11	1	'Engineer'

ID	user_id	name	version
20	1	'MyApp'	1.0.4
21	1	'DocFinder'	2.5.7

ID	user_id	make	year
30	1	'Bentley'	1973
31	1	'Rolls Royce'	1965

MongoDB

```
first_name: "Mary",
last_name: "Jones",
cell: "516-555-2048",
city: "Long Island",
year_of_birth: 1986,
location: {
        type: "Point",
        coordinates: [-73.9876, 40.7574]
},
profession: ["Developer", "Engineer"],
apps: [
 { name: "MyApp",
   version: 1.0.4 },
 { name: "DocFinder",
   version: 2.5.7 }
 ],
cars: [
   { make: "Bentley",
     year: 1973 },
   { make: "Rolls Royce",
     year: 1965 }
 ]
```

Document Database - MongoDB

- A record in MongoDB is a document, which is a data structure composed of keys- name and value pairs
- MongoDB documents are similar to JSON Objects.
- **The value of the field may include other documents, arrays and array of documents.**

JSON – JavaScript Object Notation

- **Lightweight** data-interchange format.
- It is easy for humans to read and write.
- It is easy for machines to parse and generate.
- It is based on a subset of the JavaScript Programming Language Standard ECMA-262 3rd Edition.
- **Commonly used for API s and configs**

JSON – JavaScript Object Notation

- In JSON file, the data inside are represented in a key: value pair, just like a traditional JavaScript object.
- JSON and objects aren't exactly the same.
- The core difference is that the key in JSON must be in double-quotes, and the values apart from the number and null types must be in double-quotes, too.

JSON – JavaScript Object Notation

JSON's basic data types

- **Number :** a signed decimal number that may contain a fractional part and may use exponential notation, but cannot include non-numbers such as NaN.
- **String** : a sequence of zero or more Unicode characters. Strings are delimited with double-quotation marks and support a backslash escaping syntax.
- **Boolean :** either of the values true or false
- **Array :** an ordered list of zero or more elements, each of which may be of any type. Arrays use square bracket notation with comma-separated elements.
- **Object :** a collection of field –value pairs where the field(also called keys) are strings. Each key is unique within an object.
- **Null :** an empty value, using the word null

JSON – JavaScript Object Notation

Example

```
{
   "firstName": "Rack" ,
   "lastName": "Jackson" ,
   "gender": "man" ,
   "age": 24 ,
   "address": {
                  "street Address": "126",
                  "city": "San Jone ",
                  "state": "CA",
                  "postalCode": "394221"
    } ,
   "phoneNumbers": [
                { "type": "home", "number": "7383627627" }
    ]
}
```

BSON

- Binary JavaScript Object Notation- BSON.
- It is a binary-encoded serialization of JSON documents.
- BSON has been **extended** to add some optional non-JSON-native data types, like **dates and binary data.**
- BSON has a published specification : https://bsonspec.org/spec.html
- Example : A document such as {"hello":"world"} will be stored as:

```
\x16\x00\x00\x00          // total document size
\x02                      // 0x02 = type String
hello\x00                 // field name
\x06\x00\x00\x00world\x00 // field value (size of value, value, null terminator)
\x00                      // 0x00 = type EOO ('end of object')
```

Datatypes in MongoDB

MongoDB supports many datatypes. Some of them are −

- String − This is the most commonly used datatype to store the data. String in MongoDB must be UTF-8 valid.

  Ex: { “FullName” : “Alice John” }
- Integer − This type is used to store a numerical value. Integer can be 32 bit or 64 bit depending upon your server.

  Ex: { “Age” : 20 }
- Boolean − This type is used to store a boolean (true/ false) value.

  Ex: { “isEmployed” : true }
- Double − This type is used to store floating point values.

  Ex: { “PromotionChances” : 78.55 }
- Min/ Max keys − This type is used to compare a value against the lowest and highest BSON elements.

Datatypes in MongoDB

- Arrays − This type is used to store arrays or list or multiple values into one key.

  Ex: { “Languages” : [“English”, “Spanish”, “French”] }

- Timestamp − This can be handy for recording when a document has been modified or added.
- Object − This datatype is used for embedded documents.

  Ex: { “workDetail” : { “title”: “Recruitment”, “sector” : “Mgmt”, “Manager”: “Bob Fred” } }

- Null − This type is used to store a Null value.

  Ex: { “EmailID” : null }

- Symbol − This datatype is used identically to a string; however, it's generally reserved for languages that use a specific symbol type.

Datatypes in MongoDB

- Date − This datatype is used to store the current date or time in UNIX time format. You can specify your own date time by creating object of Date and passing day, month, year into it.

  Ex: { “JoiningDate” : ISODate("2022-03-15T18:52:28.476Z") }

- Object ID − This datatype is used to store the document’s ID.

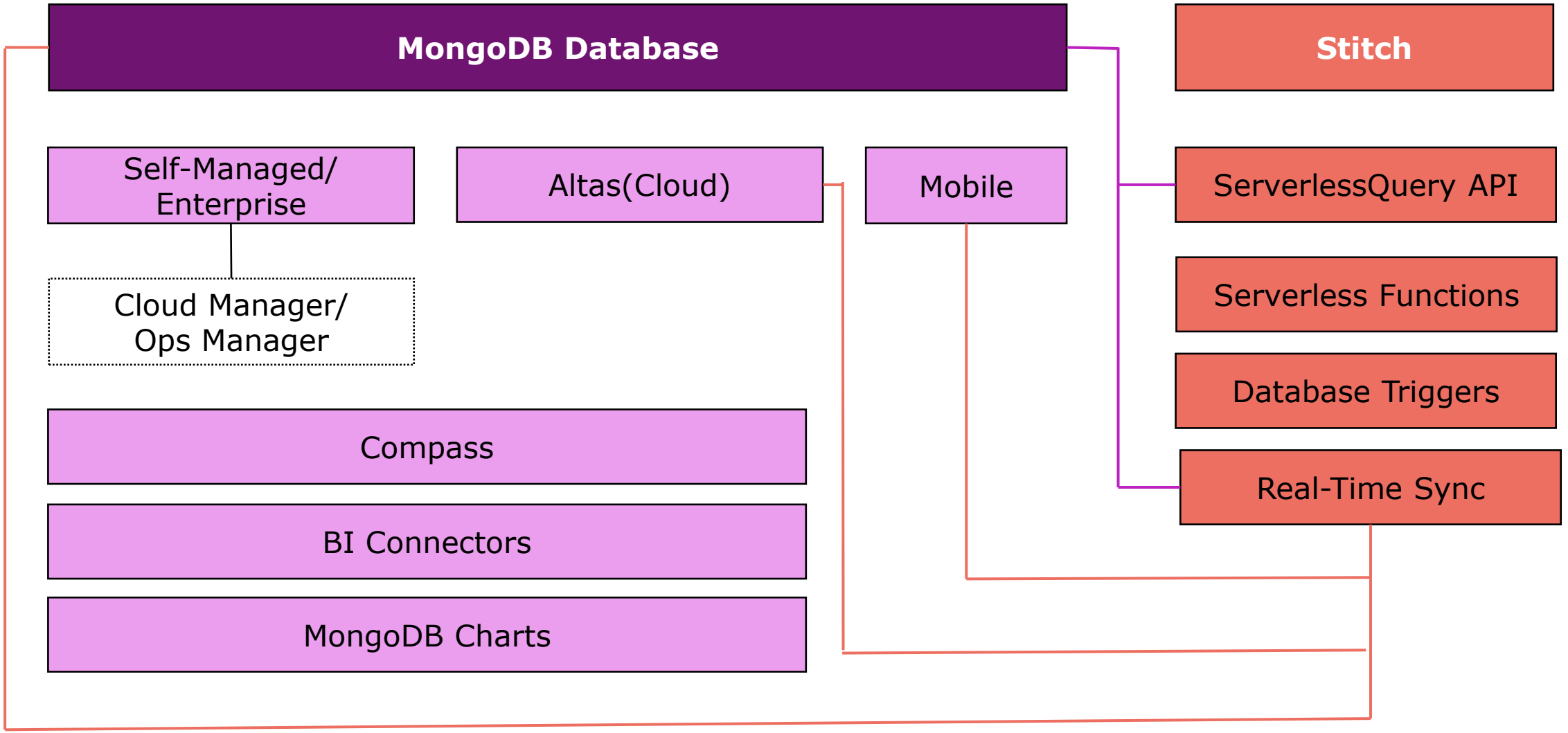
  Ex : { _id : ObjectId("5ad324589d34982a4c582bb6") }

- Binary data − This datatype is used to store binary data.
- Code − This datatype is used to store JavaScript code into the document.
- Regular expression − This datatype is used to store regular expression.

Sample Document

```
{
  _id: ObjectId(6fe25ca4619d)
  title: 'MongoDB',
  description: 'MongoDB is a NoSQL database',
  by: ‘SmartCliff',
  url: 'http://www.mongodb.com',
  tags: ['database','mongodb', 'NoSQL'],
  likes: 100,
  comments: [
    {
      user:'alice',
      message: 'My first comment',
      dateCreated: new Date(2021,4,15,4,10),
      like: 25
    },
    {
      user:'bob',
      message: 'My second comments',
      dateCreated: new Date(2022,3,28,5,10),
      like: 40
    }
  ]
}
```

MongoDB Ecosystem - Products



MongoDB Ecosystem - Products

The company behind the MongoDB database solution is also called MongoDB

- **MongoDB database** — Self-managed, Cloud, mobile and enterprise solution
- **Compass**— Allows you to connect to your database and look into it with a nice user interface.
- **BI Connectors or MongoDB charts** — Allows you to connect different analytics tools.
- **Stitch** — Serverless backend solution that is a tool set which you can use to efficiently query your database directly from inside your client-side apps.
- **Database triggers** — Service that allows you to listen to events in a database, such as a document being inserted and then execute a function in response to that and that function (i. e. send an email to a user)

MongoDB Installation - Community Edition

1) Download the .msi installer from the following link.

https://www.mongodb.com/try/download/community

a. In the **Version** dropdown, select the version of MongoDB to download.

b. In the **Platform** dropdown, select **Windows**.

c. In the **Package** dropdown, select **msi**.

d. Click **Download**.

2) Run the MongoDB installer.

− For example, from the Windows Explorer/File Explorer:

a. Go to the directory where you downloaded the MongoDB installer (.msi file). By default, this is your Downloads directory.

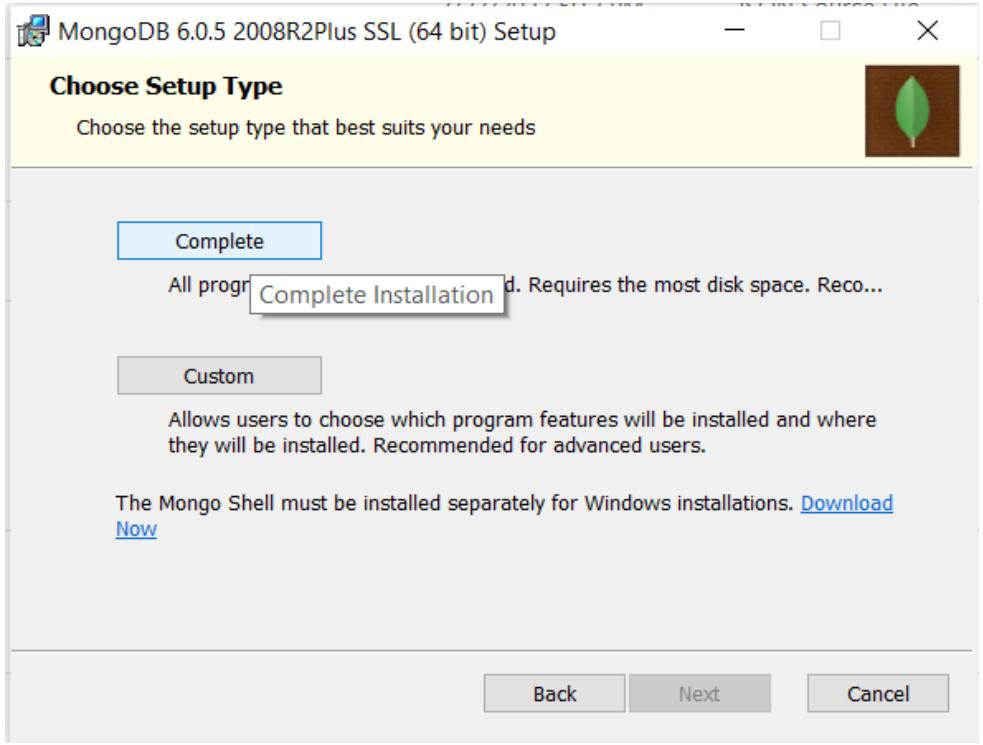
b. Double-click the .msi file.

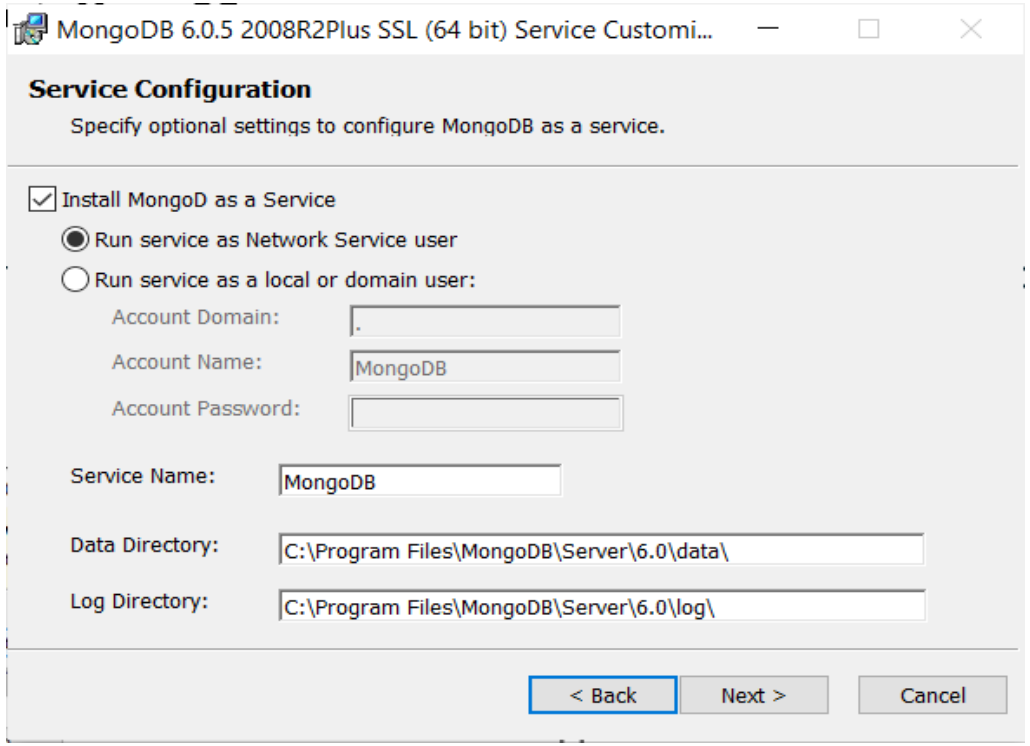
MongoDB Installation - Community Edition

3) Follow the MongoDB Community Edition installation wizard.

The wizard steps you through the installation of MongoDB and MongoDB Compass.

a)**Choose Setup Type** - Complete (recommended for most users) b) **Service Configuration** - Installs and configures MongoDB- -as a Windows service.





MongoDB Installation - Community Edition

4) Install MongoDB Compass

Optional. To have the wizard install MongoDB Compass, select **Install MongoDB Compass** (Default).

5) When ready, click **Install**

6) Install mongosh : The .msi installer does not include mongosh.

Follow the below instructions to download and install the shell separately.

a. Open the MongoDB Shell download page.

https://www.mongodb.com/try/download/shell

a. In the ***Platform*** dropdown, select ***Windows 64-bit (8.1+) (MSI)***

b. Click ***Download***.

c. Double-click the installer file.

d. Follow the prompts to install mongosh.

MongoDB Installation

Install Node.js

- First, make sure you have a supported version of Node.js installed.

Install the MongoDB Node.js Driver

- The MongoDB Node.js Driver allows to easily interact with MongoDB databases from within Node.js applications. You'll need the driver in order to connect to your database and execute the queries described in this Quick Start series.
- We can install MongoDB Node.js Driver with the following command.

```
npm install mongodb
```

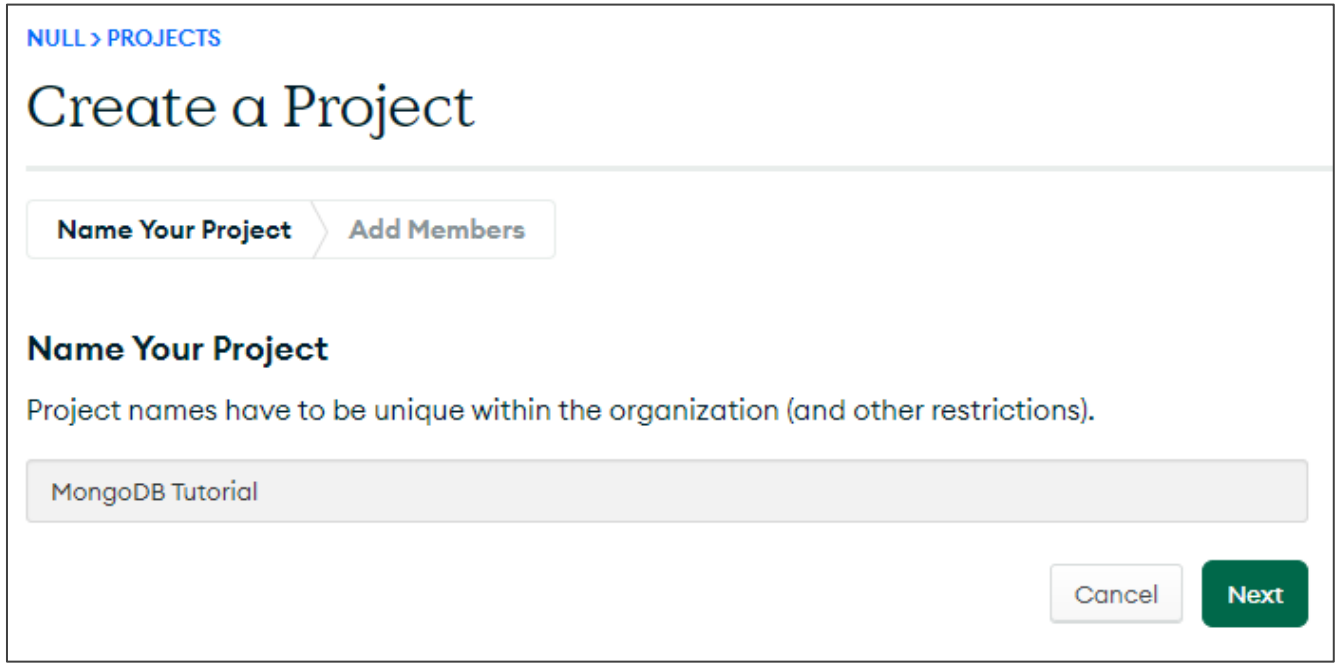
MongoDB Installation

Create a free MongoDB Atlas cluster

- MongoDB Atlas is a cloud service by MongoDB. It is built for developers who'd rather spend time building apps than managing databases. This service is available on AWS, Azure, and GCP.
- It is the worldwide cloud database service for modern applications that give best-in-class automation and proven practices guarantee availability, scalability, and compliance with the foremost demanding data security and privacy standards.
- We can use MongoDB's robust ecosystem of drivers, integrations, and tools to create faster and spend less time managing our database.

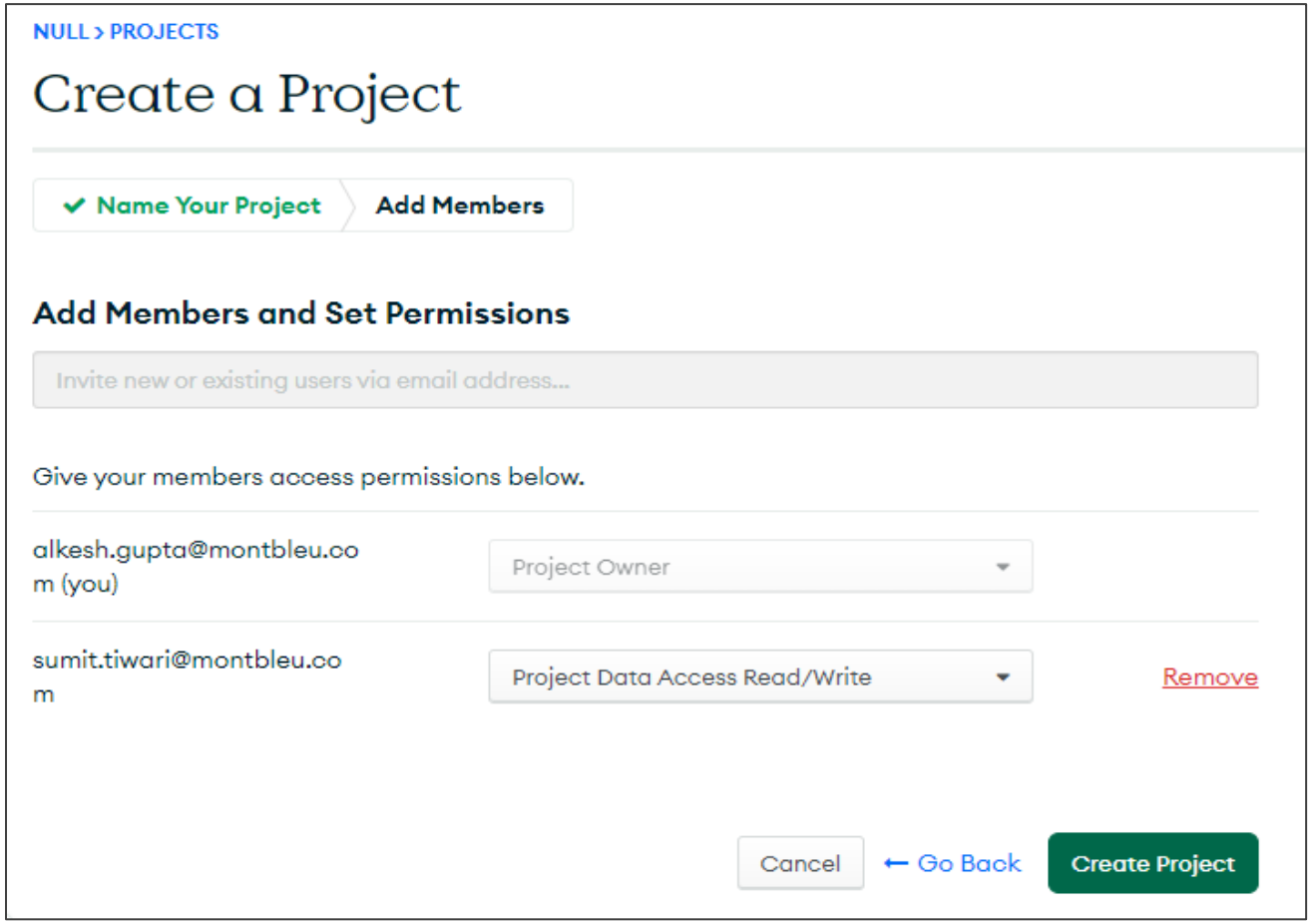
Connection in MongoDB

- First go to the following url https://www.mongodb.com/ then create a account, after login you will be redirected to the mongoDB atlas dashboard.
- Then create a project in as shown below.



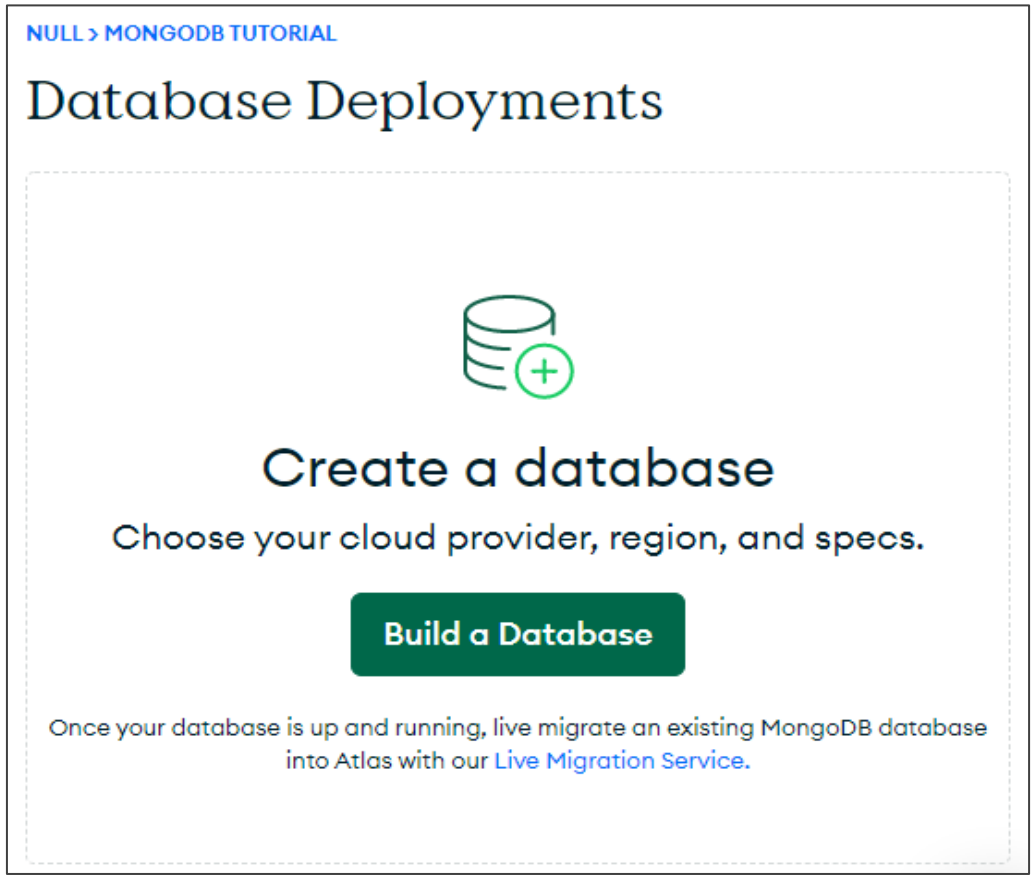
Connection in MongoDB

- Then we need to add members to our project and set their permissions as shown here.



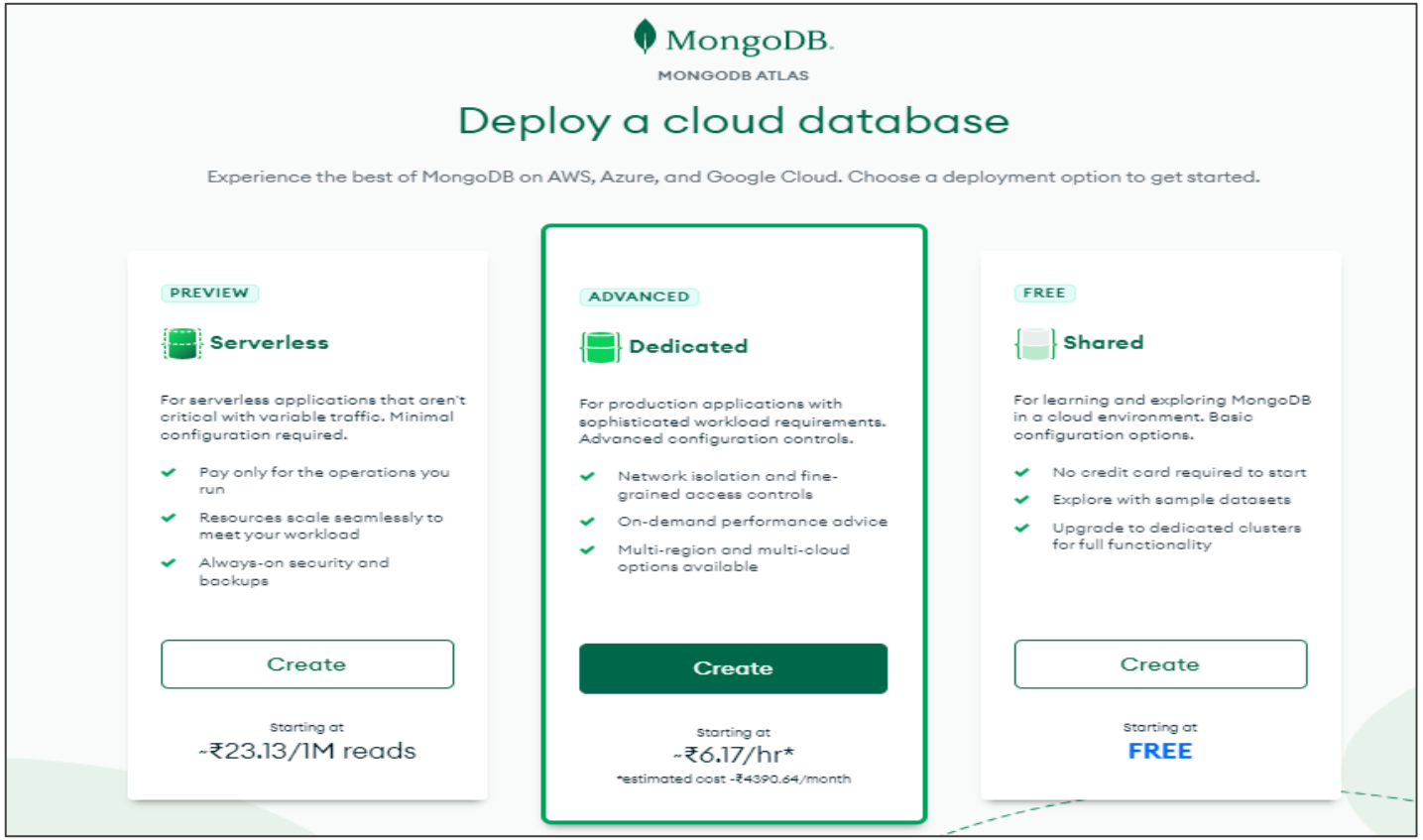
Connection in MongoDB

- Once the project is created, you will be redirected to this page where now we have to create our database.



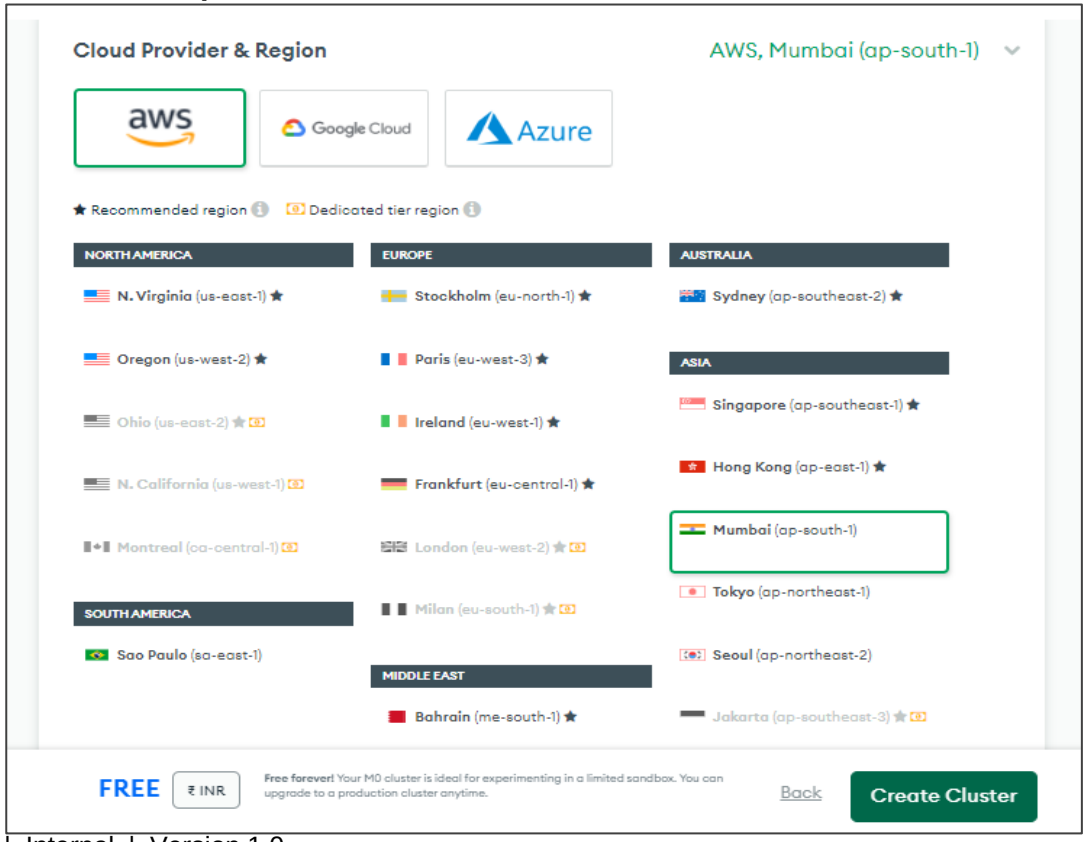
Connection in MongoDB

- In order to create a database, we have three options viz. “Serverless”, “Dedicated” and “Shared”.
- For learning purpose, we will choose “Shared”.



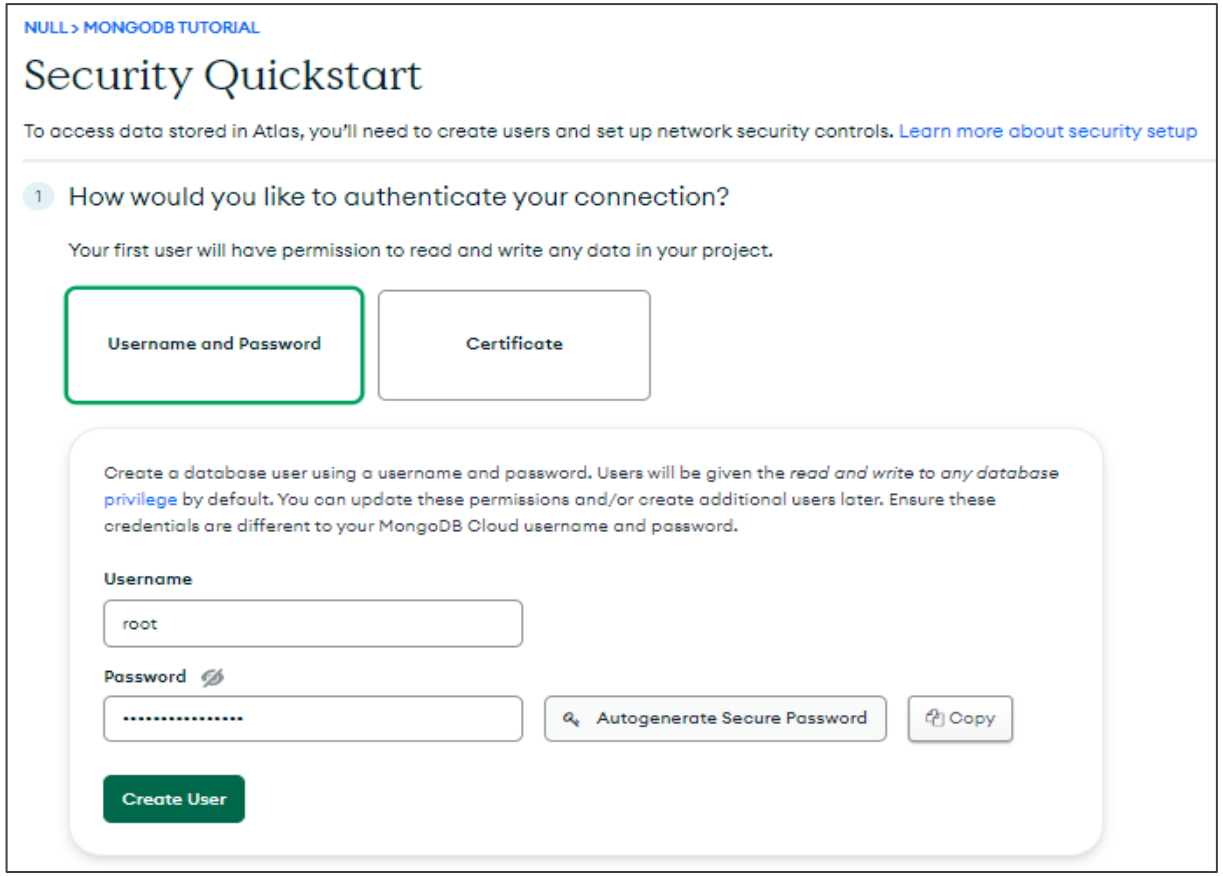
Connection in MongoDB

- Next, we have to choose a Cloud Provider and a Region to where our database will be assigned for storage.
- Choose any of options and proceed with “Create Cluster”.



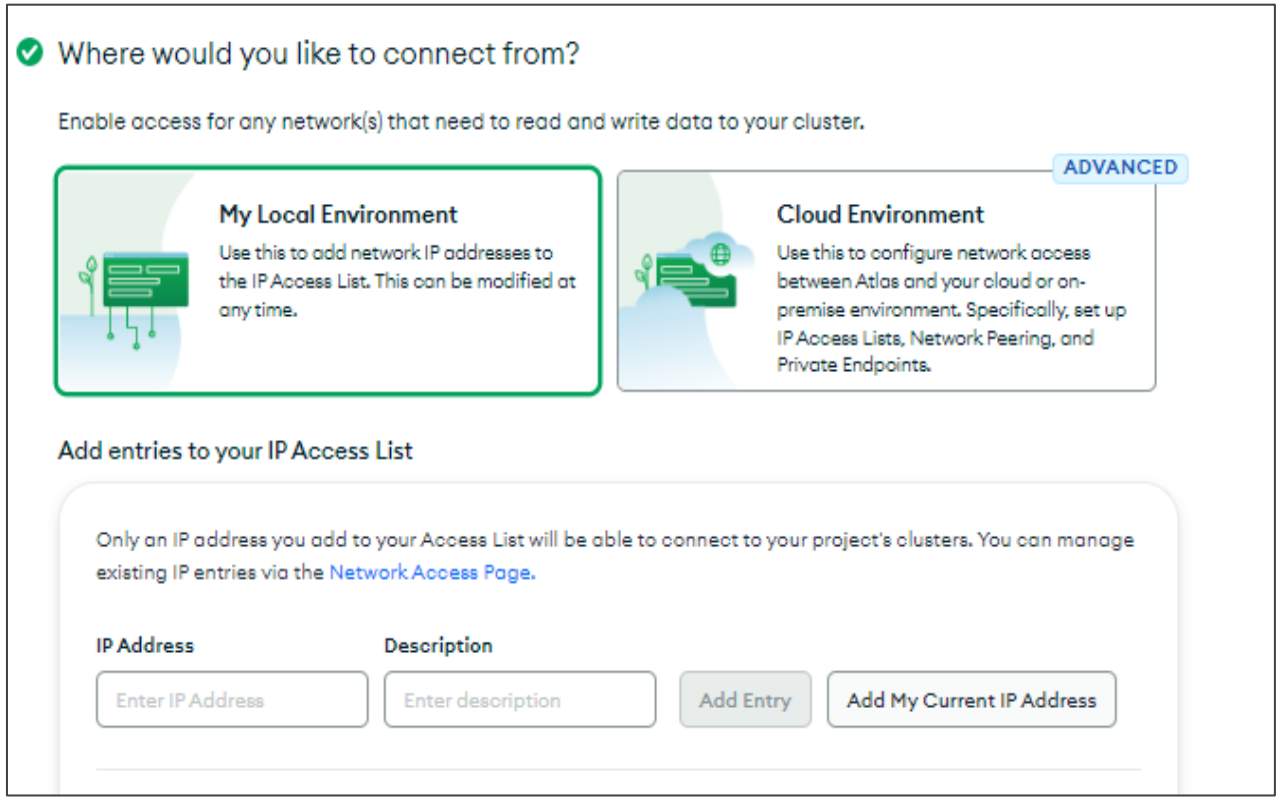
Connection in MongoDB

- Next we have to do basic setup in Security Quickstart as shown here.
- First, we have to create a username and password through which we will access our database through our application.



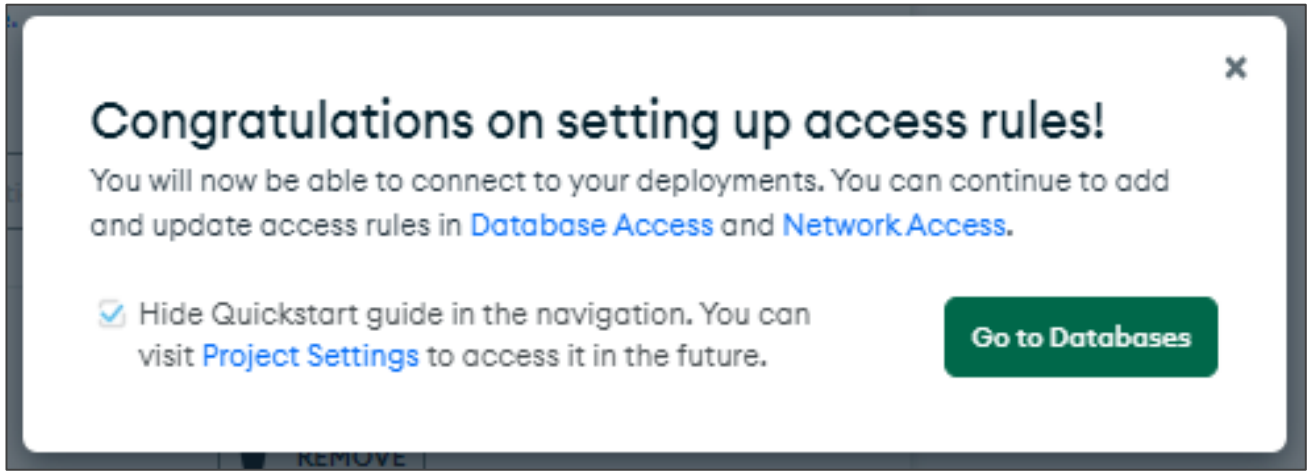
Connection in MongoDB

- Next in Security Quickstart, we have to add IP access list through which our application will be allowed to access our database.
- Here, we need to add “0.0.0.0” for access from any IP address.



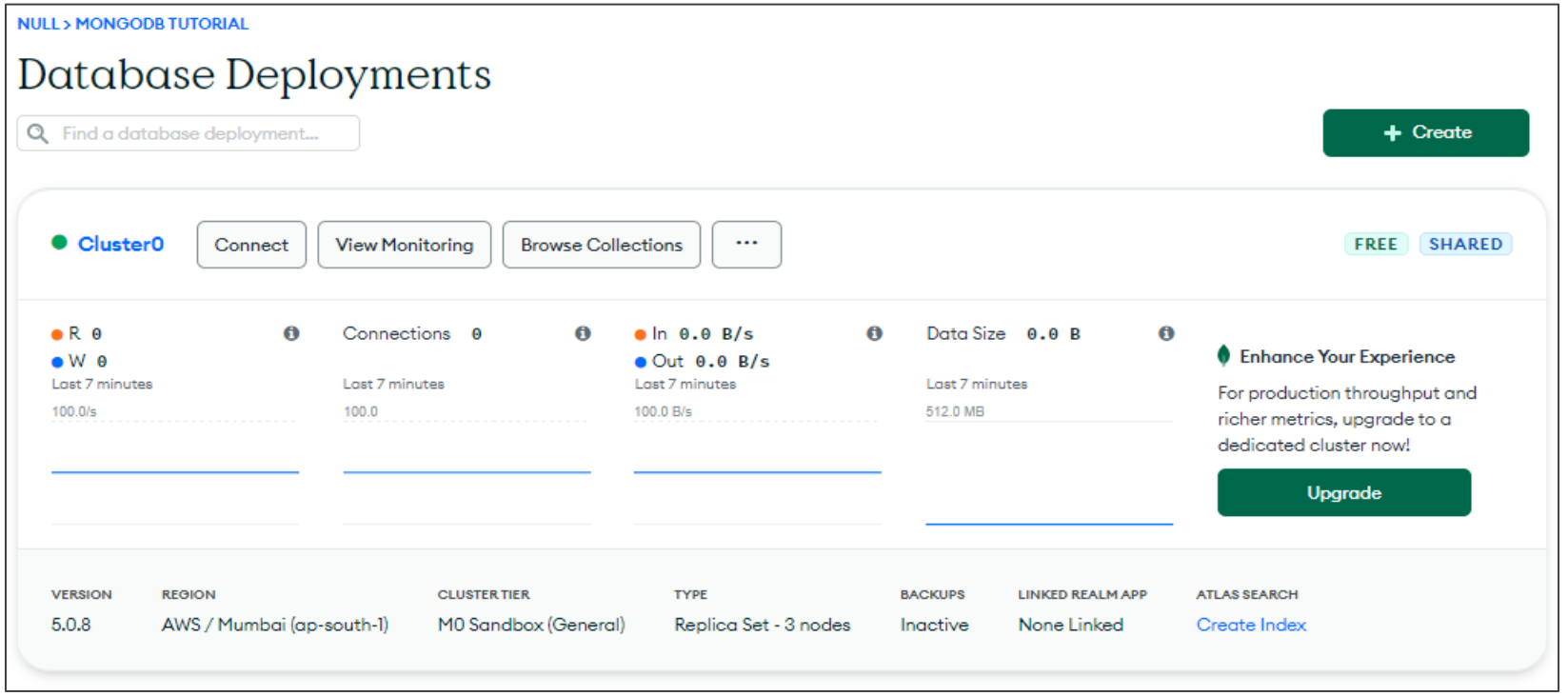
Connection in MongoDB

- Once the access rules setting is done properly, we will receive this popup so that we can now navigate to our database.



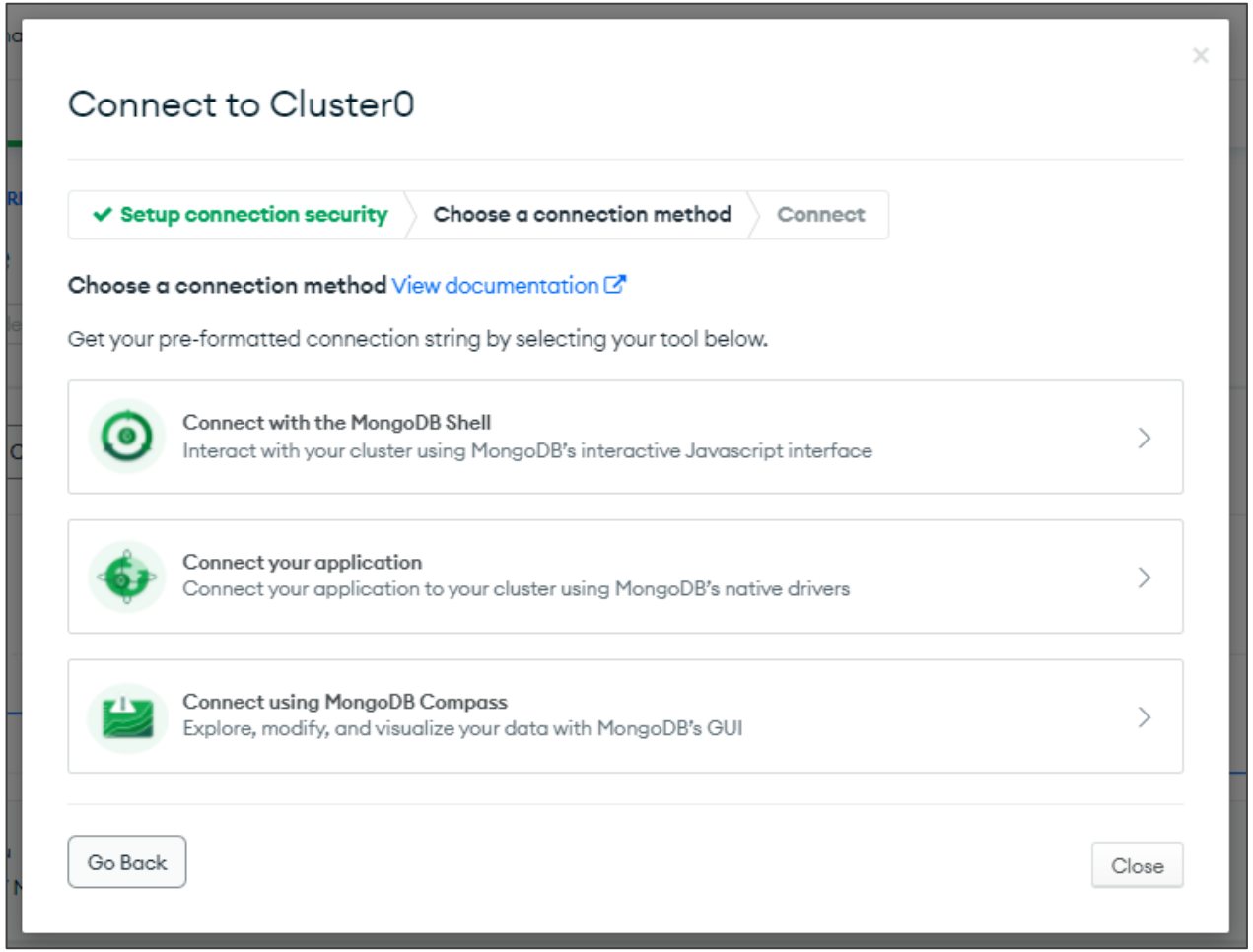
Connection in MongoDB

- Now, you are directed to your dashboard where we have three main options viz. Connect, View Monitoring, Browse Collection.
- We now go for Connect to setup connection with our application.



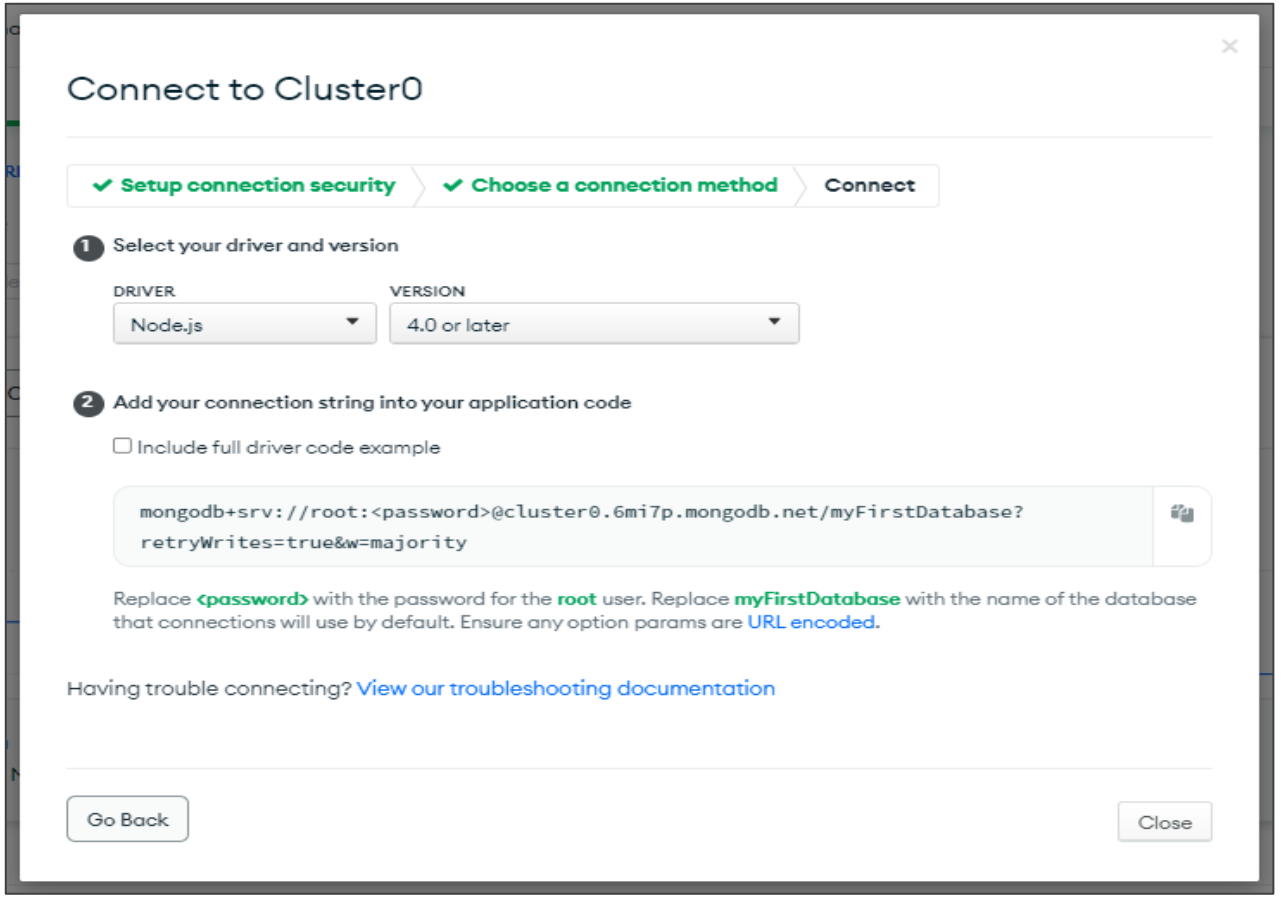
Connection in MongoDB

- Here, we have to select “Connect your application” option to connect our application with our database.



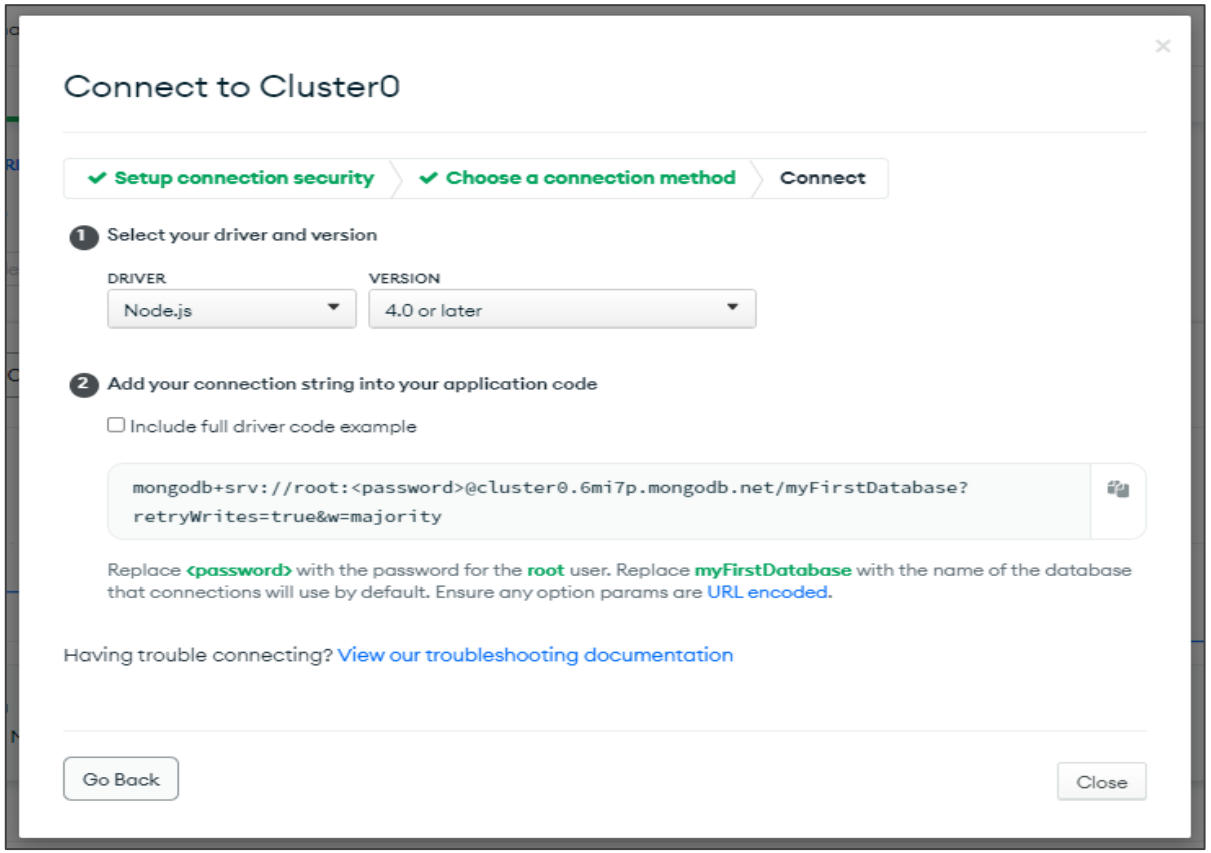
Connection in MongoDB

- Now we finally got our mongodb uri as shown in pointer 2. Copy this uri as it will be needed in our database connection setup and close this popup.



Connection in MongoDB

- Now we finally got our mongodb uri as shown in pointer 2. Copy this uri as it will be needed in our database connection setup and close this popup.



- It’s now time to use Mongoose to create the database connection.

SHARDING IN MONGODB

Sharding in MongoDB

- Sharding is a methodology to **distribute data across multiple machines**.
- Basically used for deployment with a large dataset and high throughput operations.
- The single database cannot handle a database with large datasets as it requires larger storage, and bulk query operations can use most of the CPU cycles, which slows down processing.
- For such scenarios, we need more **powerful systems**.
- One approach is to add more capacity to a single server, such as adding more memory and processing units or adding more RAM on the single server, this is also called **vertical scaling**.
- Another approach is to divide a large dataset across multiple systems and serve a data application to query data from multiple servers. This approach is called **horizontal scaling**.

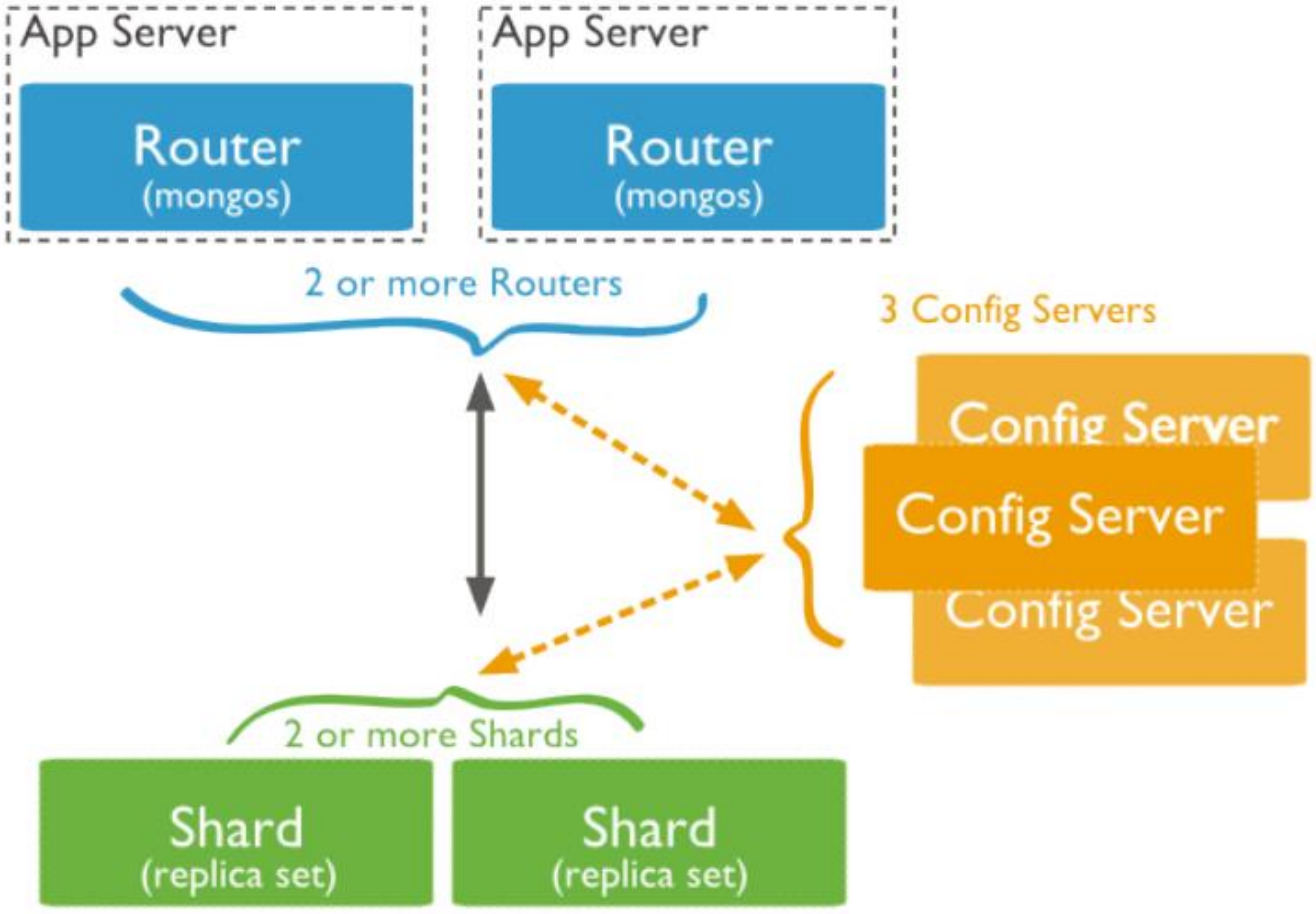
Sharding in MongoDB

- MongoDB handles horizontal scaling through sharding.

Sharded clusters - MongoDB's sharding consists of the following **components**:

- Shard: Each shard stores a subset of sharded data. Also, each shard can be deployed as a replica set.
- Mongos: Mongos provide an interface between a client application and sharded cluster to route the query.
- Config server: The configuration server stores the metadata and configuration settings for the cluster. The MongoDB data is sharded at the collection level and distributed across sharded clusters.
- Shard keys: To distribute documents in collections, MongoDB partitions the collection using the shard key. MongoDB shards data into chunks. These chunks are distributed across shards in sharded clusters

Sharding in MongoDB using sharded cluster



Sharding in MongoDB using sharded cluster

Query Routers

- Query routers are basically mongo instances, interface with client applications and direct operations to the appropriate shard.
- The query router processes and targets the operations to shards and then returns results to the clients.
- A sharded cluster can contain more than one query router to divide the client request load.
- A client sends requests to one query router.
- Generally, a sharded cluster have many query routers.

Sharding in MongoDB

Advantages of sharding

- When we use sharding, the load of the read/write operations gets distributed across sharded clusters.
- As sharding is used to distribute data across a shard cluster, we can increase the storage capacity by adding shards horizontally.
- MongoDB allows continuing the read/write operation even if one of the shards is unavailable.
- In the production environment, shards should deploy with a replication mechanism to maintain **high availability** and add **fault tolerance** in a system.

REPLICATION IN MONGODB

Replication

- Replication is the process of **synchronizing data across multiple servers**.
- Replication provides **redundancy and increases data availability** with multiple copies of data on different database servers.
- Replication protects a database from the loss of a single server.
- Replication also allows you to **recover** from hardware failure and service interruptions.
- With additional copies of the data, you can dedicate one to disaster recovery, reporting, or backup.

Why Replication?

- To keep your data safe
- High (24*7) availability of data
- Disaster recovery
- No downtime for maintenance (like backups, index rebuilds, compaction)
- Read scaling (extra copies to read from)
- Replica set is transparent to the application

Replication in MongoDB

- MongoDB achieves replication by the use of **replica set**.
- A replica set is a **group of mongod instances** that host the same data set.
- In a replica, one node is **primary node** that receives all write operations.
- All other instances, such as **secondaries**, apply operations from the primary so that they have the same data set.
- Replica set can have only one primary node.
- Replica set is a group of two or more nodes (generally minimum 3 nodes are required).
- In a replica set, one node is primary node and remaining nodes are secondary.
- All data replicates from primary to secondary node.

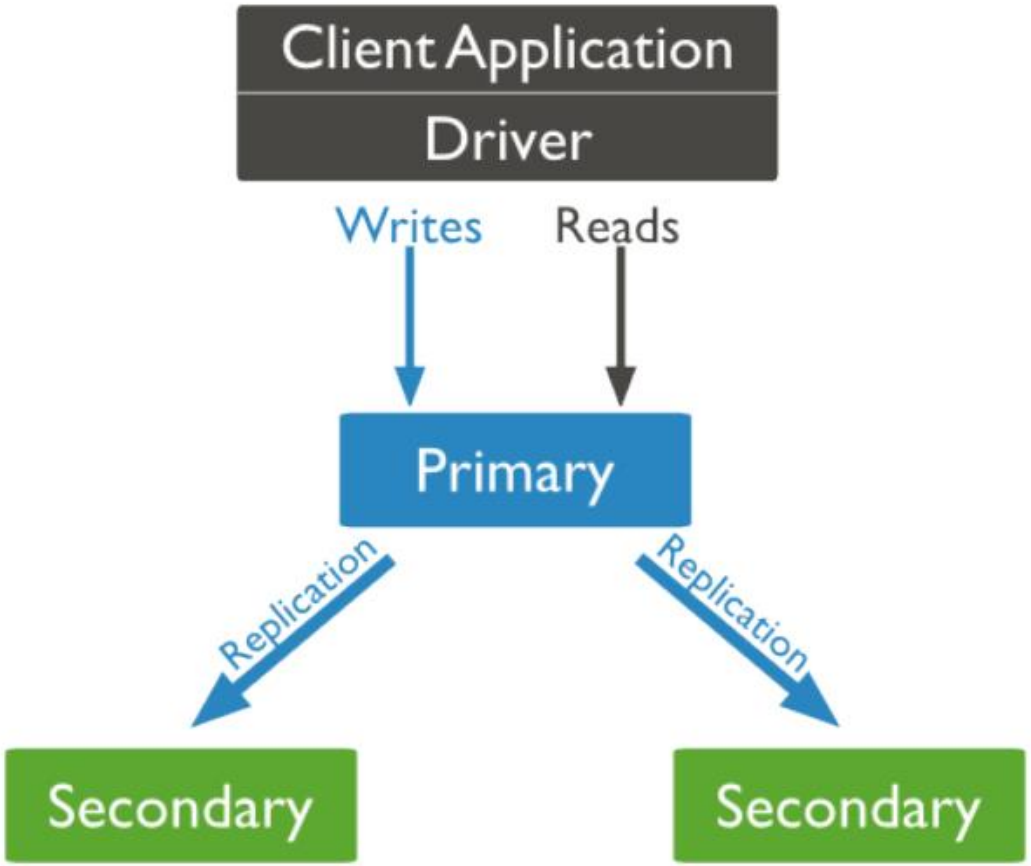
Replication in MongoDB

- At the time of automatic failover or maintenance, election establishes for primary and a new primary node is elected.
- After the recovery of failed node, it again join the replica set and works as a secondary node.

Replica Set Features

- A cluster of N nodes
- Any one node can be primary
- All write operations go to primary
- Automatic failover, Automatic recovery
- Consensus election of primary

Replication in MongoDB



Data Modeling in MongoDB

What is Data Modeling?

- **Data modeling** is the process of **building a structure** for a **database system**.
- **Data modeling** is the **first step** of **designing an application**, where **data can be grouped together** based on the **business** use cases and accordingly a **schema can be designed to store** the data in the **database.**
- The goal of data modeling is to ensure that **data is organized** in a way that is **efficient, easy to maintain, and meets the application criteria.**

What is Data Modeling?

- **Example**: let's say you have a **website where students can enroll in one or more courses**. A common way is for all the **student data to be stored together, and similarly the course data be stored together. Students and courses will then be linked based on their relationship, like "is", "has" or "contains".** There might be other ways, we can **embed the most important details of the course a particular student has enrolled in, with the student data itself.** The choice of modeling depends on many factors, like the workload and business use cases of your application.

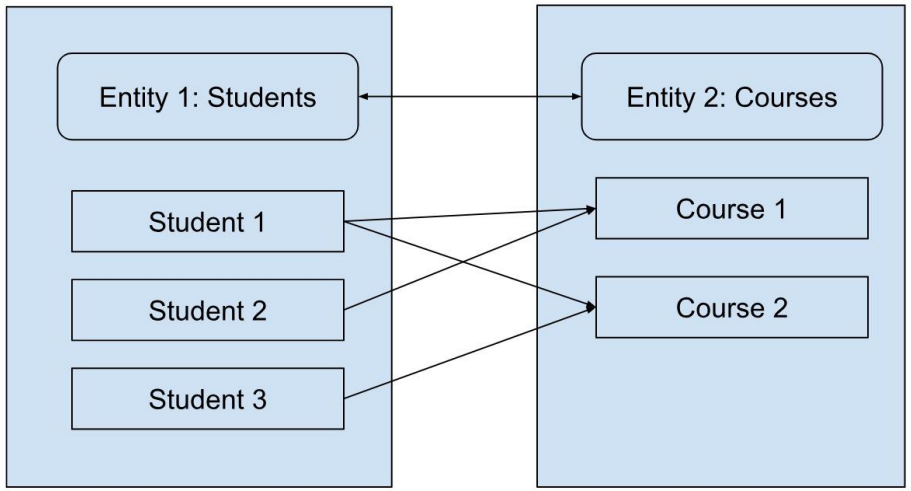


Fig1: Student enrolled more than one course

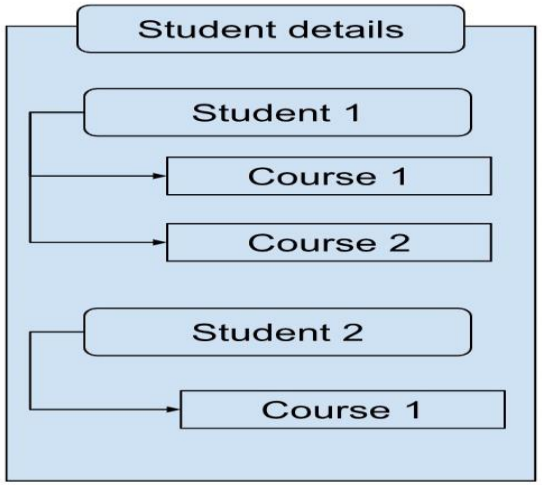
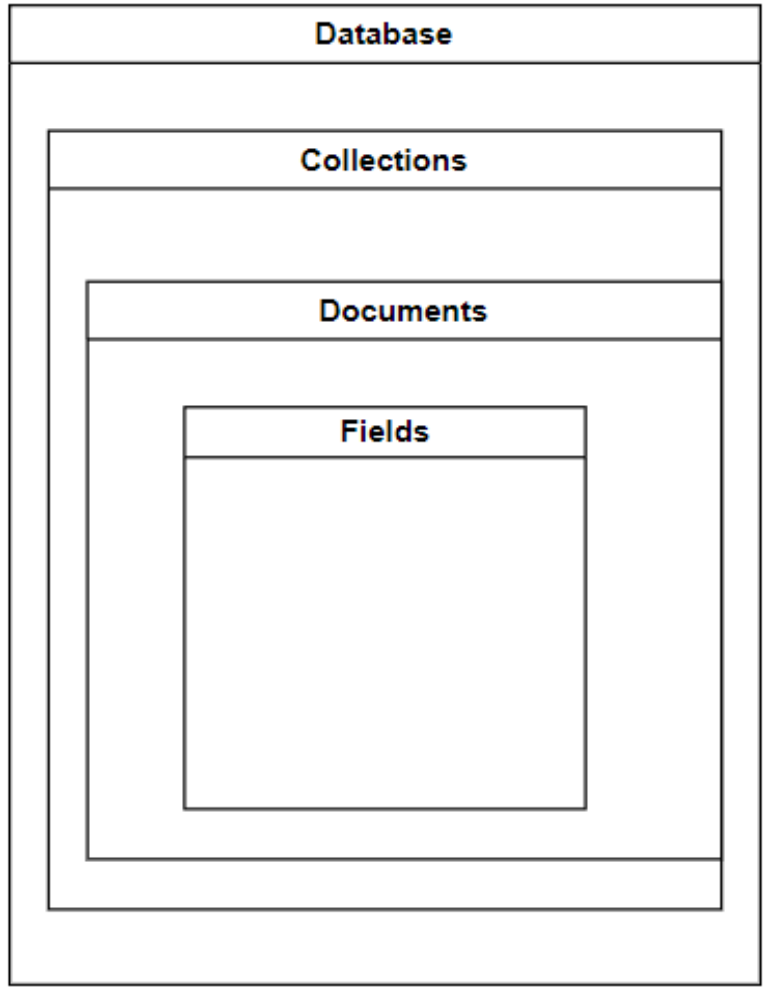


Fig 2: Embed the data in student itself

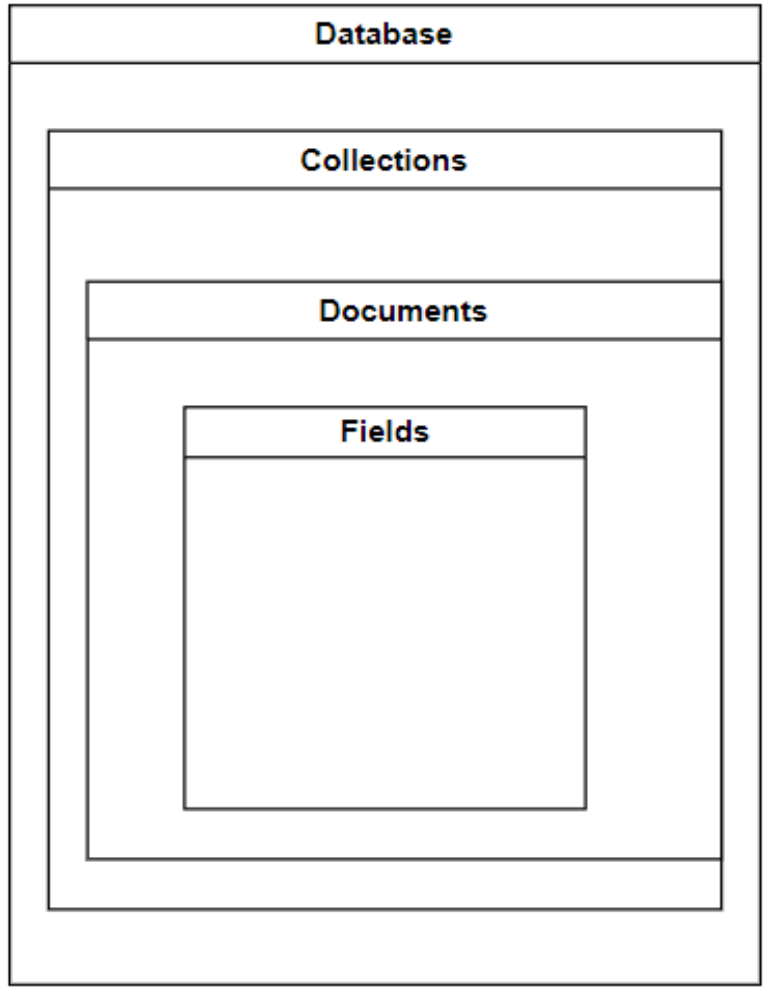
What is Data Modeling?

- Since **MongoDB** is a **NoSQL database**, it does things a bit differently from what you might be used to with **SQL databases** that is **Instead of tables**, **MongoDB uses** what are called **collections**, and **instead of rows**, it **stores documents**.
- That is MongoDB follows **Document based data modeling** where stored data in MongoDB is arranged in a **hierarchical structure** where the **databases are at the top**, then the **collections within databases**, and finally **documents at the collection level**.



What is Data Modeling?

- **Documents** represent an **object and its attributes**, allowing for **easy adaptation to changes in data structure and application requirements**.
- This approach also simplifies data access by eliminating the need for **complex joins and foreign keys in relational databases.**
- MongoDB's requires that a **single document** contain **related data to improves query performance, making retrieval faster and more efficient.**



Why Data Modeling is Important?

- **Consistency:** Ensures that data is consistent and accurately reflects business processes.
- **Efficiency:** Helps in designing efficient database structures that reduce redundancy and improve performance.
- **Communication:** Provides a clear framework for communication between stakeholders, developers, and database administrators.
- **Documentation:** Acts as documentation for the system's data, making it easier to understand and maintain.

Important Terminologies in Data Modeling

1. **Normalization and denormalization : Normalization** involves **breaking down data** into **several manageable** documents, while d**enormalization**, involves **packing related data into a single document**.
2. **Entity and attribute: Entities** is an **object** that need to be **modeled**. **Every entity** should be **unique.** An **entity should have a set of attributes** that describe the entity.
3. **Relationship:** A **relationship indicates** the **relationship between** the **various entities used in the application.**
4. **Indexing: Indexing** is essential to maintain a **good performance**. It is **preferred to index the fields (columns) that are most used in queries**, so that the **search can be faster.** However, indexing should be done carefully, as it comes with a cost.

Traditional Schema Vs MongoDB Schema

- In a **relational database**, the **data dictates** how you **develop your application**.This is because while **designing the schema** you would **normalize your data** and then as **you develop your application** and you may have some **concerns** in the **way you access the data**.
- As the **database structure** is **rigid** in a relational database, **you cannot change your schema** often, even if you do so, and **deformalize your data**, and you could get into **serious performance concerns**.
- In **MongoDB**, however, you can **follow an iterative approach** into **developing** and **improving the data** model as you develop your application.
- Since **MongoDB** offers a **flexible schema**, the **changes to the structure** of the data **do not need any downtime and do not affect the performance.**

Traditional Schema Vs MongoDB Schema

- For **example**, if you **created two entities** for **Student and Course**, and as you notice later, that **teachers or admins lookup for the student data and their corresponding course data together**, you can consider **embedding both into the same document**. To reduce the **data duplication**, you could **add only the required fields** from the **course collection into the Student collection.**

Data Modeling Process with Example

- **Data Modeling has following process:**

    1. Workload Identification.

    2.Realtionship Identification

    3. Decide on Embedding Vs Referencing

    4.Applying schema Patterns

Data Modeling Process with Example

1. Workload Identification:

- In this process we identify the **main use cases** and **entities** required in the application.
- For Example: In our student and course app defining the entities and related attributes as follows

S.No	Entity	Attribute
1	Student	Name, Age, Email, EnrolledCourses (array of Course IDs)
2	Course	Name, Type (e.g., dance, music, instrument), Tutors (array of Tutor IDs), Students (array of Student IDs)
3	Tutor	Name, YearsIntoTeaching, Hobbies, OtherSkills, ClassDetails (array of ClassSlot subdocuments), Reviews (array of Review subdocuments)
4	Review	Reviewer (Student ID), Reviewee (Tutor ID), Rating, Comments, Timestamp
5	ClassSlot	Day, TimeSlot, Available (boolean), Course (Course ID)

Data Modeling Process with Example

- Once we **identify the entities and attributes**, we need to **quantify them** -
  - How many enrolments are we expecting in a year?
  - How many reviews are we expecting in a year?
  - How many website visits and enquiries should we expect per day?
  - How many users click on the tutor feedback per day?
- Based on these and more similar questions, we can **estimate the reads and writes in the application**.
- For that, we need to **examine each entity and find the operations** to be **performed**, the information needed, and the type of operation (read or write).

Data Modeling Process with Example

- **Examples:**
  - To estimate reads and writes, we need to make some assumptions:
  - Enrollments: Targeting 1000 enrollments per year (~83 per month)
  - Reviews: Assuming 10% of students review their tutors (~100 reviews per year)
  - Website Visits and Enquiries: 1000 visits per day
  - Tutor Feedback Clicks: Assume 10% of visitors click on tutor feedback (~100 clicks per day)
- There might be many such **read and write operations**, and **identifying each of them will help create an efficient schema.**

Data Modeling Process with Example

2. Relationship Identification:

- Once the **workloads** are **identified** and **entities** are in place, we can **define the relationships** between the **entities**.
- **Identifying relationships** will help us **understand what patterns to apply while modeling the data**.
- There are **3 main types of relationships** as follows:
  - **One to One (1:1)** : A **one to one represents** one of **entity A** related to **one of entity B** and vice versa.

    Example: A teacher can take only one class slot at a time. So, the relationship between Tutor and ClassSlot is 1:1. This will help plan the availability of teachers during a particular time.

Data Modeling Process with Example

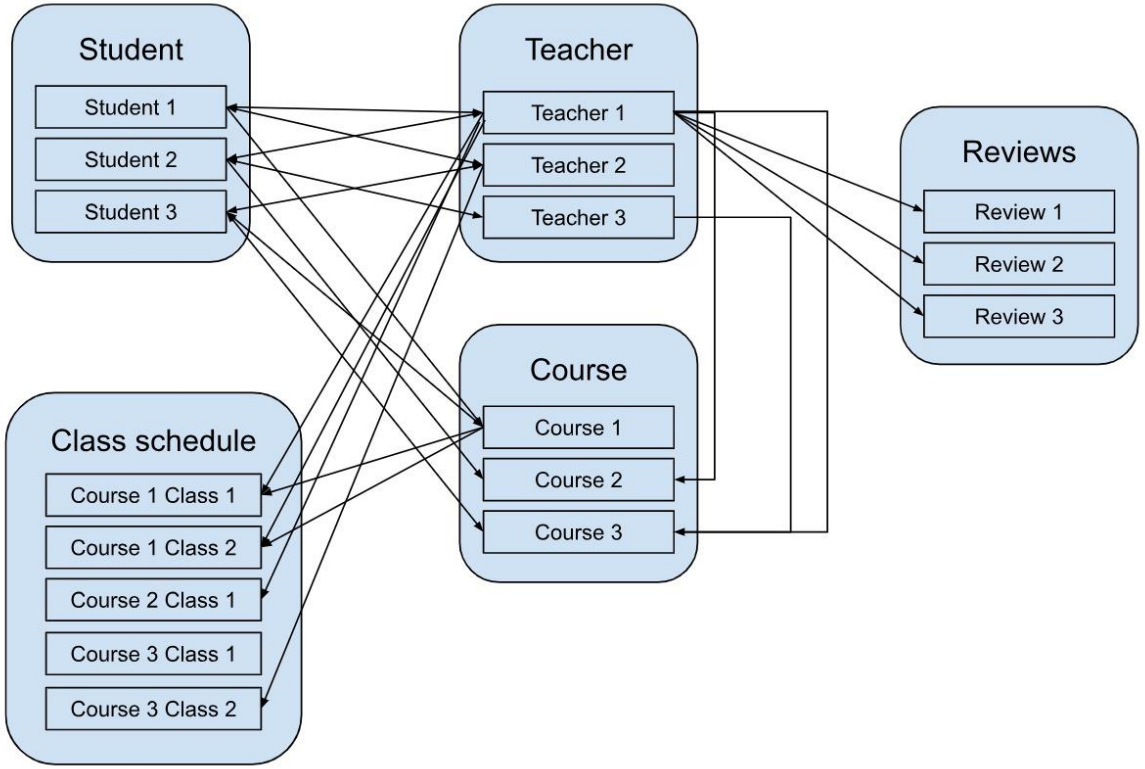
- **One to Many (1:m) :**If entity A is related to many of entity B and entity B is related to only one of entity A, the relationship is referred to as 1:many.

  Example: A teacher can have multiple reviews, however one review can be for only one tutor.

- **Many to Many (m:m):** In this case, many of entity A are related to many of entity B and vice versa.

  Example: Student can enroll in many classes and one class can have multiple students. Similarly, a teacher can teach multiple courses, and one course can have multiple teachers too.

Data Modeling Process with Example



- Based on the **nature of these relationships**, you can either **embed or reference documents**.

Data Modeling Process with Example

3. Deciding Embedded Vs Reference:

- **Embedding** is done by **creating document arrays or subdocuments**.
- In a **many to many relationship**, **embedding** can **create data duplication** and it depends on our project whether we can **tolerate some amount of data duplication** for a **significant performance boost.**
- **Referencing** is done using an **array of references for child elements** in the **parent document** or using the **array of references** of **parent elements in the child document**.

Data Modeling Process with Example

- To decide whether to **embed or reference** the entities, **MongoDB provides** a **set of guidelines**, based on which you can answer simple questions.
- These **guidelines** include **simplicity, relationship b/w the entities, archival, document size, individuality, update complexity, query atomicity** and so on.
- **Example:** let us consider the relationship between the teacher and review entities
  - **Simplicity** - Would your code be simple if the teacher and review entities are kept embedded? - yes.
  - **Document size** - would it become too huge and unmanageable if the entities are kept together - no? - however, if we are expecting reviews in 1000s then we may consider a yes.
  - Is the relationship between the teacher and review entities 'has a' or 'contains' - yes
  - Does the app query the information together - yes - a user would most likely like to see the reviews along with the teacher details

Data Modeling Process with Example

- In this way, we can **refer to each guideline** and **decide** whether to **embed or reference**. In general, you would **embed the entity** in the **same document level or create a sub document**.

Example: Data Model

```
// Embedded/Denormalized Model:
// Student Document
{
  "_id": "student_id",
  "name": "John Doe",
  "age": 20,
  "email": "john.doe@example.com",
  "enrolledCourses": [
```

Data Modeling Process with Example

```
    {
      "courseId": "course_id_1",
      "courseName": "Piano",
      "tutors": ["tutor_id_1", "tutor_id_2"]
    },
    ...
  ]
}
```

Data Modeling Process with Example

```
// Tutor Document
{
  "_id": "tutor_id",
  "name": "Jane Smith",
  "yearsIntoTeaching": 10,
  "hobbies": ["Reading", "Traveling"],
  "otherSkills": ["Violin", "Conducting"],
  "classDetails": [
```

Data Modeling Process with Example

```
{
      "day": "Monday",
      "timeSlot": "10:00 - 11:00",
      "course": "course_id_1"
    },
    ...
  ],
"reviews": [
    {
      "reviewer": "student_id",
      "rating": 5,
```

Data Modeling Process with Example

```
"comments": "Great tutor!",
      "timestamp": "2023-06-10T10:00:00Z"
    },
    ...
  ]
}
```

Data Modeling Process with Example

// Referenced / Normalized model

1. Student Document:

```
{
  "_id": "student_id",
  "name": "John Doe",
  "age": 20,
  "email": "john.doe@example.com",
  "enrolledCourses": ["course_id_1", "course_id_2"]
}
```

Data Modeling Process with Example

2. Course Document:

```
{
  "_id": "course_id",
  "name": "Piano",
  "type": "instrument",
  "tutors": ["tutor_id_1", "tutor_id_2"],
  "students": ["student_id_1", "student_id_2"]
}
```

Data Modeling Process with Example

3. Tutor Document

```
{
  "_id": "tutor_id",
  "name": "Jane Smith",
  "yearsIntoTeaching": 10,
  "hobbies": ["Reading", "Traveling"],
  "otherSkills": ["Violin", "Conducting"],
  "classDetails": [
    {
      "day": "Monday",
      "timeSlot": "10:00 - 11:00",
      "courseId": "course_id_1"
    },
    ...
  ]
}
```

Data Modeling Process with Example

4. Review Document

```
{
  "_id": "review_id",
  "reviewer": "student_id",
  "reviewee": "tutor_id",
  "rating": 5,
  "comments": "Great tutor!",
  "timestamp": "2023-06-10T10:00:00Z"
}
```

Data Modeling Process with Example

4. Applying patterns:

- The last step is to **collate all the information** we gathered in the **previous steps** and apply the **relevant schema patterns**. MongoDB provides many patterns, few of which are highlighted below:

1. Inheritance pattern: Polymorphism is a key feature of the document model of MongoDB. The inheritance pattern is based on this principle, where you can group similar documents together in one collection. This, as golden rules of MongoDB "Data that is accessed together should be stored together." You can use the MongoDB aggregation framework to apply inheritance pattern to existing collections.

Data Modeling Process with Example

2. Computed pattern : This pattern precomputes certain fields before the read operation. For example, calculating the avg review of a particular tutor, calculating monthly sales. We can do this using the aggregation framework. Whenever data changes, these operations can be performed and result is stored in a document for quick reference.

3. Approximation pattern : This pattern is used when data is expensive or difficult to calculate, and getting the exact result is not critical. The pattern is very useful for big data calculations and helps reduce resource usage by reducing numbers of writes and updates, when not needed.

Data Modeling Process with Example

4. Schema versioning pattern : This pattern allows you to update the schema of a database without any application downtime. This feature is a big distinguisher between relational databases and MongoDB, as relational databases allow only one schema per database or table, whereas in MongoDB you can have multiple schemas for the same database.

BASIC OPERATIONS

MongoDB - Basics

1) To view the **list of databases,** use the command

```
>show dbs / show databases
```

2) To check your **currently selected** database, use the command

```
>db
```

3) To **create the Database** ,use the command

```
>use flights
```

4) To **create the Collection** in the database ,use the command

```
flights >db.createCollection(“flightData”)
```

MongoDB - Basics

5) To view the **list of collections** in the database, use the command

```
flights>show collections
```

6) To **drop the collection** in the database, use the command

```
flights>db.flightData.drop()
```

7) To **drop the Database** ,use the command

```
flights>db.dropDatabase()
```

MongoDB – Insert

- **Insert Document**

Syntax :

```
db.collection.insertOne(
   <document>,
   {
      writeConcern: <document>
   }
)
```

Returns: A document containing:

- A boolean acknowledged as true if the operation ran with write concern or false if write concern was disabled.
- A field insertedId with the _id value of the inserted document.

Parameter	Type	Description
document	document	A document to insert into the collection.
writeConcern	document	Optional. A document expressing the write concern. Omit to use the default write concern. Do not explicitly set the write concern for the operation if run in a transaction

MongoDB - Insert

- **Insert Document**

Example :

```
db.inventory.insertOne(
  {
        item: "canvas",
        qty: 100,
        tags: ["cotton"],
        size: {
              h: 28,
              w: 35.5,
              uom: "cm"
            }
  }
)
```

Returns :

```
{
  acknowledged: true,
  insertedId: ObjectId("642bb2166953787703d27464")
}
```

MongoDB - Insert

- **Insert Multiple Documents**

Syntax :

```
db.collection.insertMany(
   [ <document 1> , <document 2>, ... ],
   {
      writeConcern: <document>,
      ordered: <boolean>
   }
)
```

Returns: A document containing:

- A boolean acknowledged as true if the operation ran with write concern or false if write concern was disabled.
- An insertedIds array, containing _id values for each successfully inserted document

Parameter	Type	Description
document	document	A document to insert into the collection.
writeConcern	document	Optional. A document expressing the write concern. Omit to use the default write concern. Do not explicitly set the write concern for the operation if run in a transaction
ordered	boolean	Optional. A boolean specifying whether the mongod instance should perform an ordered or unordered insert. Defaults to true.

MongoDB – Read

- **Query Documents - Select All Documents in a Collection**

```
retail> db.inventory.find({})
[
  {
    _id: ObjectId("642bb2166953787703d27464"),
    item: 'canvas',
    qty: 100,
    tags: [ 'cotton' ],
    size: { h: 28, w: 35.5, uom: 'cm' }
  },
  {
    _id: ObjectId("642bb96c19a82b81b61c2bae"),
    item: 'journal',
    qty: 25,
    tags: [ 'blank', 'red' ],
    size: { h: 14, w: 21, uom: 'cm' }
  },
```

MongoDB – Read

- **Query Documents - Select All Documents in a Collection - Specify Equality Condition**

Syntax ;

{ <field1>: <value1>, ... }

```
retail> db.inventory.find( { status: "D" } )
[
  {
    _id: ObjectId("642bba1b19a82b81b61c2bb3"),
    item: 'paper',
    qty: 100,
    size: { h: 8.5, w: 11, uom: 'in' },
    status: 'D'
  },
  {
    _id: ObjectId("642bba1b19a82b81b61c2bb4"),
    item: 'planner',
    qty: 75,
    size: { h: 22.85, w: 30, uom: 'cm' },
    status: 'D'
  }
]
```

MongoDB – Read

Query Documents - Select All Documents in a Collection - Specify Conditions Using Query Operators

Syntax ;

{ <field1>: { <operator1>: <value1> }, ... }

```
retail> db.inventory.find( { status: {$in :["A","D"] } } )
[
  {
    _id: ObjectId("642bba1b19a82b81b61c2bb1"),
    item: 'journal',
    qty: 25,
    size: { h: 14, w: 21, uom: 'cm' },
    status: 'A'
  },
  {
    _id: ObjectId("642bba1b19a82b81b61c2bb2"),
    item: 'notebook',
    qty: 50,
    size: { h: 8.5, w: 11, uom: 'in' },
    status: 'A'
  },
```

MongoDB – Read

Query Documents - Select All Documents in a Collection - Specify *AND* Conditions

Syntax ;

{ <field1>: { <operator1>: <value1> }, ... }

```
retail> db.inventory.find( { status: "A", qty: { $lt: 30 } } )
[
  {
    _id: ObjectId("642bba1b19a82b81b61c2bb1"),
    item: 'journal',
    qty: 25,
    size: { h: 14, w: 21, uom: 'cm' },
    status: 'A'
  }
]
```

MongoDB – Read

Query Documents - Select All Documents in a Collection - Specify *OR* Conditions

Syntax ;

{ <field1>: { <operator1>: <value1> }, ... }

```
]
retail> db.inventory.find( { $or: [ { status: "A" }, { qty: { $lt: 30 } } ] } )
[
  {
    _id: ObjectId("642bb96c19a82b81b61c2bae"),
    item: 'journal',
    qty: 25,
    tags: [ 'blank', 'red' ],
    size: { h: 14, w: 21, uom: 'cm' }
  },
  {
    _id: ObjectId("642bb96c19a82b81b61c2bb0"),
    item: 'mousepad',
    qty: 25,
    tags: [ 'gel', 'blue' ],
    size: { h: 19, w: 22.85, uom: 'cm' }
  }
```

MongoDB – Read

Query Documents - Select All Documents in a Collection - Match an Embedded/Nested Document

```
retail> db.inventory.find( { size: { h: 14, w: 21, uom: "cm" } } )
[
  {
    _id: ObjectId("642bb96c19a82b81b61c2bae"),
    item: 'journal',
    qty: 25,
    tags: [ 'blank', 'red' ],
    size: { h: 14, w: 21, uom: 'cm' }
  },
  {
    _id: ObjectId("642bba1b19a82b81b61c2bb1"),
    item: 'journal',
    qty: 25,
    size: { h: 14, w: 21, uom: 'cm' },
    status: 'A'
  }
]
```

MongoDB – Read

Query Documents - Select All Documents in a Collection - Specify Equality Match on a Nested Field

```
retail> db.inventory.find( { "size.uom": "in" } )
[
  {
    _id: ObjectId("642bba1b19a82b81b61c2bb2"),
    item: 'notebook',
    qty: 50,
    size: { h: 8.5, w: 11, uom: 'in' },
    status: 'A'
  },
  {
    _id: ObjectId("642bba1b19a82b81b61c2bb3"),
    item: 'paper',
    qty: 100,
    size: { h: 8.5, w: 11, uom: 'in' },
    status: 'D'
  }
]
```

MongoDB – Read

Query Documents - Select All Documents in a Collection - Match an Array

```
retail> db.inventory.find( { tags: ["red", "blank"] } )
[
  {
    _id: ObjectId("642bcedb9dc9c1f2b3172280"),
    item: 'notebook',
    qty: 50,
    tags: [ 'red', 'blank' ],
    dim_cm: [ 14, 21 ]
  }
]
retail> db.inventory.find( { tags: { $all: ["red", "blank"] } } )
[
  {
    _id: ObjectId("642bb96c19a82b81b61c2bae"),
    item: 'journal',
    qty: 25,
    tags: [ 'blank', 'red' ],
    size: { h: 14, w: 21, uom: 'cm' }
  },
  {
    _id: ObjectId("642bcedb9dc9c1f2b317227f"),
    item: 'journal',
    qty: 25,
    tags: [ 'blank', 'red' ],
    dim_cm: [ 14, 21 ]
  },
```

MongoDB – Read

Query Documents - Select All Documents in a Collection - Project Fields to Return from Query

```
}
retail> db.inventory.find( { status: "A" }, { item: 1, status: 1 } )
[
  {
    _id: ObjectId("642bba1b19a82b81b61c2bb1"),
    item: 'journal',
    status: 'A'
  },
  {
    _id: ObjectId("642bba1b19a82b81b61c2bb2"),
    item: 'notebook',
    status: 'A'
  },
```

```
retail> db.inventory.find( { status: "A" }, { item: 1, status: 1, _id: 0 } )
[
  { item: 'journal', status: 'A' },
  { item: 'notebook', status: 'A' },
  { item: 'postcard', status: 'A' },
  { item: 'journal', status: 'A' },
  { item: 'notebook', status: 'A' },
  { item: 'postcard', status: 'A' }
]
```

MongoDB – Update

- In mongoDB,**db.collection.update()** method, modifies an existing document or documents in a collection.
- The method can modify specific fields of an existing document or documents or replace an existing document entirely, depending on the **update parameter.**
- **Methods**

  db.collection.updateOne(<filter>, <update>, <options>)

  db.collection.updateMany(<filter>, <update>, <options>)

  db.collection.replaceOne(<filter>, <update>, <options>)
- To use the update operators, pass to the update methods an update document of the form:

```
{
  <update operator>: { <field1>: <value1>, ... },
  <update operator>: { <field2>: <value2>, ... },
  ...
}
```

MongoDB – UpdateOne Method

```
}
retail> db.inventory.updateOne(
...     { item: "paper" },
...     {
...       $set: { "size.uom": "cm", status: "P" },
...       $currentDate: { lastModified: true }
...     }
... );
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

MongoDB – UpdateMany

```
retail> db.inventory.updateMany(
...     { "qty": { $lt: 50 } },
...     {
...       $set: { "size.uom": "in", status: "P" },
...       $currentDate: { lastModified: true }
...     }
... );
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 12,
  modifiedCount: 12,
  upsertedCount: 0
}
```

MongoDB – ReplaceOne

```
retail> db.inventory.replaceOne(
...     { item: "paper" },
...     { item: "paper", instock: [ { warehouse: "A", qty: 60 }, { warehouse: "B", qty: 40 } ] }
... );
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

MongoDB – Delete

- Delete All Documents that match a condition

Example:

db.inventory.deleteMany({})

- Delete All Documents that Match a Condition

1) To specify equality conditions, use <field>:<value> expressions in the query filter document

```
{ <field1>: <value1>, ... }
```

2) A query filter document can use the query operators to specify conditions in the following form:

```
{ <field1>: { <operator1>: <value1> }, ... }
```

```
retail> db.inventory.deleteMany({ status : "A" })
{ acknowledged: true, deletedCount: 0 }
retail>
```

MongoDB – Indexing

- Indexes support the **efficient resolution of queries**.
- Without indexes, MongoDB must scan every document of a collection to select those documents that match the query statement.
- This scan is highly inefficient and require MongoDB to process a large volume of data.
- Indexes are **special data structures**, that store a small portion of the data set in an easy-to-traverse form.
- The index stores the value of a specific field or set of fields, ordered by the value of the field as specified in the index.

MongoDB – Indexing

Syntax:

db.COLLECTION_NAME.createIndex({KEY:1})

- Key is the name of the field on which you want to create index and 1 is for ascending order.
- To create index in descending order you need to use -1.

Example:

db.users.createIndex({ age: 1 });

- 1 means ascending order.
- use -1 for descending order.

Example:

db.users.find({ age: 28 }).explain("executionStats");

MongoDB – Indexing

Type	Description	Example
Single Field	Index on one field	{ age: 1 }
Compound Index	Index on multiple fields	{ age: 1, name: 1 }
Unique Index	No duplicate values	{ email: 1 }, { unique: true }
Text Index	For text search	{ bio: "text" }
Hashed Index	For sharded collections	{ user_id: "hashed" }
Wildcard Index	Indexes dynamic fields	{ "$**": 1 }

MongoDB – Indexing

- Single Field Index

```
db.products.createIndex({ price: 1 });
db.products.find({ price: { $gt: 700 } });
```

- Compound Index

```
db.products.createIndex({ name: 1, price: -1 });
db.products.find({ name: "Laptop" }).sort({ price: -1 });
```

MongoDB – Indexing

- Unique Index - Ensures all values in the indexed field are unique (no duplicates allowed).

```
db.users.createIndex({ email: 1 }, { unique: true });
db.users.insert({ name: "Alice", email: "alice@example.com" });
db.users.insert({ name: "Bob", email: "alice@example.com" }); // ❌ Error
```

- Text Index

```
db.articles.insertMany([
  { title: "Node.js Tutorial", content: "Learn Node.js step by step" },
  { title: "MongoDB Guide", content: "Indexing and performance tips" }
]);
db.articles.createIndex({ content: "text" });
db.articles.find({ $text: { $search: "MongoDB" } });
```

MongoDB – Indexing

- Hashed Index - Indexes using a hashed value of the field. Mainly used for sharding (distributed

```
db.users.createIndex({ user_id: "hashed" });
```

- Not useful for range queries ($gt, $lt), but good for evenly distributing data across shards.

- Wildcard Index - Used for indexing fields dynamically, especially when the schema is not known or varies between documents. This can index all fields, including nested ones.

```
db.logs.insert({ a: 1, b: 2, c: { x: 10, y: 20 } });
// Create a wildcard index on all fields
db.logs.createIndex({ "$**": 1 });
db.logs.find({ "c.x": 10 });
```

MongoDB – Indexing

- Deleting an index

```
db.users.dropIndex({ age: 1 });
```

- Drop all index

```
db.users.dropIndexes();
```

- To see all indexes on a collection

```
db.collection.getIndexes()
```


THANK YOU
SmartCliff
Career Mobility Solutions