MongoDB:

**MongoDB** is a type of database that helps you store and manage data. Instead of using tables like traditional databases (like MySQL), MongoDB uses **collections** to store **documents**. Think of it like a digital filing cabinet: each file (document) can be different, and you don't need to organize them in strict rows and columns.

1. Data Storage:

* Relational Databases (RDBMS): Data is stored in tables with rows and columns. Each row is a record, and each column represents a field or attribute.
* MongoDB: Data is stored in documents (similar to JSON objects) inside collections (similar to tables). Each document can have a unique structure, meaning you don’t need to predefine a schema (fields can vary between documents).

2. Schema:

* Relational Databases: The schema is fixed, meaning you must define the structure of the data (tables and columns) upfront. Data types are strictly enforced.
* MongoDB: It is schema-less, meaning there is no need to define the structure of the document in advance. You can store documents with different structures in the same collection.

3. Primary Keys:

* Relational Databases: Each table has a primary key, typically used to uniquely identify records (e.g., an ID column).
* MongoDB: Each document has a unique identifier called \_id (automatically generated by MongoDB if not provided), but you can also create your own.

4. Relationships:

* Relational Databases: Use foreign keys to establish relationships between tables (e.g., one-to-many, many-to-many).
* MongoDB: Instead of foreign keys, embedding and referencing are used for relationships. You can embed documents within other documents or reference them by storing their \_id. For example:
  + Embedding: Store a list of comments inside a blog post document (one-to-many).
  + Referencing: Store a reference to another document (e.g., an order referencing a customer).

5. Querying:

* Relational Databases: Use SQL (Structured Query Language) to query the data. You perform operations like SELECT, INSERT, UPDATE, and DELETE.
* MongoDB: Uses its own query language, which is more intuitive for working with documents. Queries are typically written in a format that resembles JSON, and you use methods like find(), insert(), update(), and delete().

**Simple Example:**

Imagine you're making a **contacts app** and you need to store information about each person.

* In **MySQL** (SQL database), you’d have a table with columns like **name**, **phone number**, and **email**.
* In **MongoDB**, you'd have a **document** for each person, and each document can have a **name**, **phone number**, **email**, and **address**. If some people don't have an address, that's okay—you just leave that field out in their document.

**NoSQL** stands for "Not Only SQL" and refers to databases that don't use the traditional table-based structure of SQL databases. Instead, they use different ways to store and manage data, making them more flexible for certain types of applications.

Here are the key points:

1. **Flexible Structure**: NoSQL databases can store data in various formats like documents, key-value pairs, graphs, or wide-columns, rather than rows and columns like in SQL.
2. **Scalability**: They are designed to scale easily across many servers to handle large amounts of data.
3. **No Fixed Schema**: You don’t need to define a strict structure for your data, making it easier to change as needs evolve.
4. **Faster for Certain Tasks**: They can be faster and more efficient for certain applications like social media, big data, or real-time analytics.

In short, NoSQL databases offer a more flexible, scalable, and faster alternative to traditional SQL databases.

**JSON** stands for **JavaScript Object Notation**. It's a simple way to store and exchange data in a format that's easy for both humans and computers to read. It looks like a collection of key-value pairs, where the key is a name (a string) and the value can be various types of data (like a number, string, array, etc.).

Here's a simple example:

{

"name": "John",

"age": 30,

"city": "New York",

"skills": ["JavaScript", "Python", "SQL"]}

In this example:

* "name", "age", "city", and "skills" are keys.
* "John", 30, "New York", and ["JavaScript", "Python", "SQL"] are their respective values.

JSON is widely used for transferring data between a server and a web application because it is lightweight and easy to parse.

**BSON** stands for **Binary JSON**. It’s a format similar to JSON, but it’s more efficient for storage and speed because it’s in binary (a more compact, machine-readable format).

While JSON is human-readable and often used for data exchange, BSON allows MongoDB (and other databases) to store and process data faster. It supports additional data types like **dates**, **binary data**, and even **embedded documents** (which JSON doesn't handle as efficiently).

In short, BSON is just a faster, more efficient version of JSON, but it’s not meant to be human-readable—it's optimized for computers.

**1. Basic Data Types:**

**String**

* **Description**: Used to store text.
* **Example**: "Hello World"
* **Usage**: Most commonly used data type for text data.

{name: "John Doe"}

**Integer**

* **Description**: Used to store whole numbers (32-bit or 64-bit, depending on the platform).
* **Example**: 42
* **Usage**: For numeric fields without decimals.

{age: 30}

**Double**

* **Description**: Used to store floating-point numbers (decimals).
* **Example**: 3.14
* **Usage**: For numbers that require decimal points (e.g., currency or measurements).

{price: 10.99}

**Boolean**

* **Description**: Used to store true or false values.
* **Example**: true or false
* **Usage**: For flags, toggles, or binary states.

{isActive: true}

**Null**

* **Description**: Used to store null values, which signify the absence of any value.
* **Example**: null
* **Usage**: For fields that have no value or are undefined.

{middleName: null}

**Date**

* **Description**: Used to store dates (with millisecond precision).
* **Example**: ISODate("2025-02-25T15:00:00Z")
* **Usage**: To store dates and times.

{createdAt: ISODate("2025-02-25T15:00:00Z")}

**ObjectId**

* **Description**: A unique identifier for each document in MongoDB (used as the default primary key).
* **Example**: ObjectId("507f191e810c19729de860ea")
* **Usage**: Automatically generated when a document is created if no \_id is provided.

{\_id: ObjectId("507f191e810c19729de860ea")}

**2. Complex Data Types:**

**Array**

* **Description**: Used to store a list of values. Arrays in MongoDB can store different data types in a single list.
* **Example**: [1, "apple", true, {name: "John"}]
* **Usage**: Useful for fields that require a list of items (e.g., tags, comments, or an inventory list).

{tags: ["mongodb", "database", "nosql"]}

**Embedded Document (Sub-document)**

* **Description**: A document within another document, which allows you to store nested data structures.
* **Example**: {address: {street: "123 Main St", city: "New York"}}
* **Usage**: When you want to store related data together, such as an address inside a user document.

{address: {street: "123 Main St", city: "New York", zip: "10001"}}

**Binary Data (BinData)**

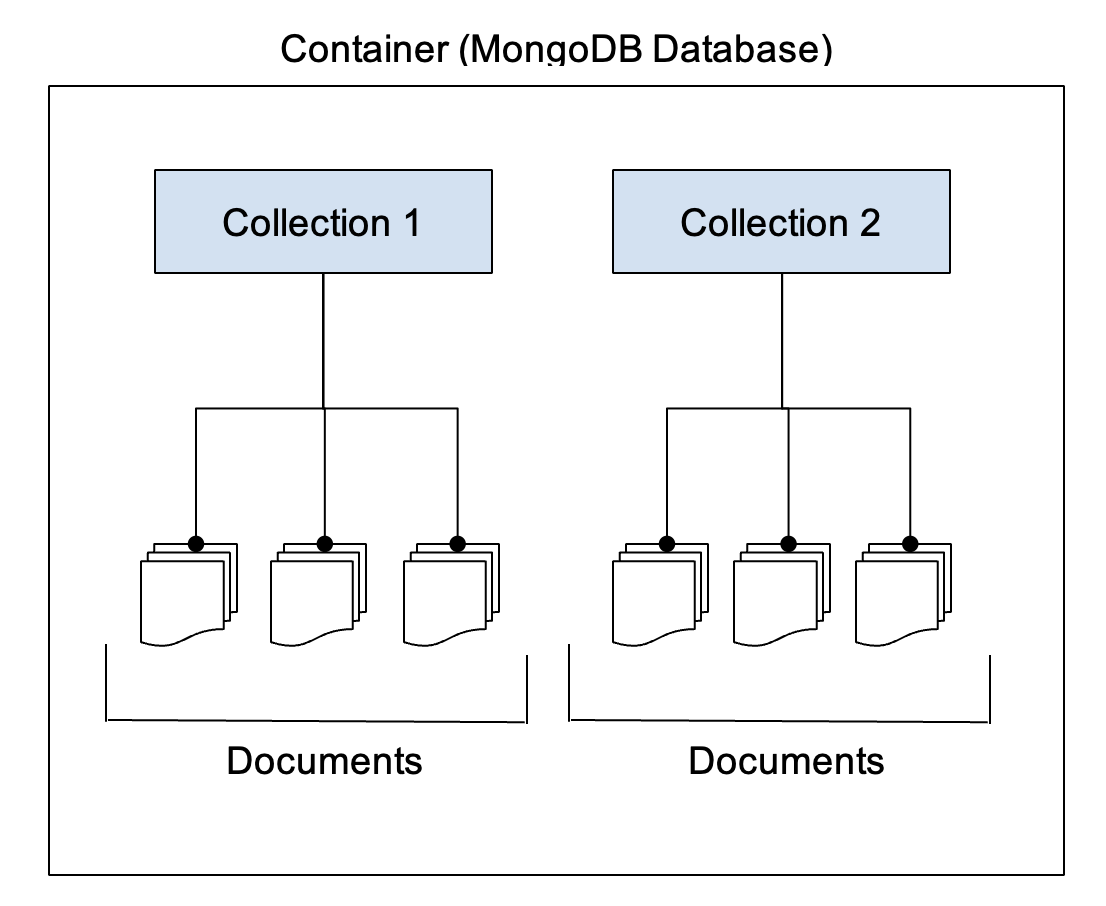
* **Description**: Used to store binary data such as images, files, or any non-text data.
* **Example**: BinData(0, "somebinarydatahere")
* **Usage**: Storing files like images or encrypted data.

{image: BinData(0, "iVBORw0KGgoAAAANS...")}

**Summary of MongoDB Data Types:**

* **Basic Types**: String, Integer, Double, Boolean, Null, Date, ObjectId
* **Complex Types**: Array, Embedded Document (Sub-document), Binary Data

**MongoDB Structure:**

****

**Database**

* A **MongoDB database** is a container for collections.
* Each database in MongoDB can contain multiple collections, and each collection can store documents.
* You can create multiple databases in a single MongoDB instance.
* MongoDB uses **"db"** to refer to a specific database.

**Collection**

* A **collection** is a group of MongoDB documents (equivalent to a table in a relational database).
* Collections are schema-less, meaning documents within the same collection don’t need to have the same structure or fields.
* Collections are created automatically when you insert the first document.

**Document**

* A **document** is a record in a MongoDB collection (similar to a row in a SQL database).
* Documents are written in **BSON (Binary JSON)** format, which is a binary representation of JSON (JavaScript Object Notation).
* A document can have multiple fields (key-value pairs), and each field can store different types of data (strings, integers, arrays, nested documents, etc.).
* Documents have a special field, \_id, which acts as the primary key and is unique for each document in a collection.

To create a database and collections in MongoDB on Windows and connect it to **VS Code**, you’ll need to follow these steps. These instructions will guide you through installing MongoDB, connecting it to VS Code, and writing MongoDB queries.

**Step 1: Install MongoDB on Windows**

1. **Download MongoDB**:
   * Go to the official MongoDB download page.
   * Select **Windows** as your OS.
   * Choose the **.msi** installer (recommended for simplicity).
   * Download the installer.
2. **Install MongoDB**:
   * Run the downloaded .msi installer.
   * Follow the installation wizard and select "Complete" installation.
   * During installation, ensure to check the box that says **"Install MongoDB as a Service"**. This will automatically start MongoDB whenever your machine is booted up.
3. **Verify MongoDB Installation**:
   * Once the installation is complete, open the **Command Prompt** and type:

mongod

This should start the MongoDB server. If it starts successfully, it means MongoDB is installed and running.

* You can stop the server by pressing Ctrl+C.

**Add MongoDB to Path** (if not automatically added):

* If MongoDB isn’t recognized in your Command Prompt, you need to add it to your PATH manually.
* Find the MongoDB bin directory, typically located here:

C:\Program Files\MongoDB\Server\<version>\bin

Add this directory to your PATH environment variable:

* Open **Control Panel** > **System and Security** > **System** > **Advanced system settings** > **Environment Variables**.
* Under **System Variables**, find and select **Path**, then click **Edit**.
* Add the path to the MongoDB bin folder.

**Step 2: Install MongoDB Extension for VS Code**

1. **Install Visual Studio Code** (if you don't have it):
   * Download and install [**VS Code**](https://code.visualstudio.com/).
2. **Install MongoDB for VS Code Extension**:
   * Open **VS Code**.
   * Go to the **Extensions** view by clicking on the **Extensions icon** in the Activity Bar on the side of the window or press Ctrl+Shift+X.
   * In the search box, type **MongoDB**.
   * Install the **MongoDB for VS Code** extension from **MongoDB, Inc.**.

**Step 3: Connect MongoDB with VS Code**

1. **Start MongoDB**:
   * Open **Command Prompt in** Adminstrator mode and run:

mongod

This will start the MongoDB server on the default port 27017.

**Connect to MongoDB in VS Code**:

* In **VS Code**, click on the **MongoDB** icon on the left sidebar (from the MongoDB extension you installed).
* Click on **Connect** and then select **Add Connection**.
* You will need to enter the connection string. For a local instance, use:

mongodb://localhost:27017

After adding the connection, it will appear in the MongoDB Explorer view on the left side of the VS Code.

**Create a New Database**:

* MongoDB doesn’t require you to explicitly create a database first. It will be created when you insert your first document into it.
* To switch to a new database (which will be created once you insert data), use the following command:

use myDatabase

This switches to a new database called myDatabase. If myDatabase does not exist, MongoDB will create it as soon as you insert data into it.

**Create a Collection and Insert Data**:

* MongoDB will create a collection automatically when you insert the first document into it. For example, let’s create a users collection and insert a document:

db.createCollection("users")

This command will create the users collection

Insert():

To insert data into a MongoDB collection, you can use the insertOne() or insertMany() methods. Here's how you can do it:

**General Syntax:**

* **insertOne()**: Inserts a single document.

db.collection.insertOne({ field1: value1, field2: value2 });

* **insertMany()**: Inserts multiple documents as an array.

db.collection.insertMany([{ field1: value1 }, { field2: value2 }]);

**Notes:**

* The \_id field is automatically added by MongoDB if it’s not explicitly provided.
* You can add more fields or nested objects to the documents as needed.

SORTING:

Sorting in MongoDB is a straightforward process that allows you to order the results of a query based on one or more fields. MongoDB provides a powerful sort() function to help you sort documents in ascending or descending order.

**Basic Syntax for Sorting:**

db.collection.find().sort({ field: 1 })

* **1** indicates **ascending order** (from smallest to largest, A to Z).
* **-1** indicates **descending order** (from largest to smallest, Z to A).

Limiting:

In MongoDB, **limiting** allows you to restrict the number of documents returned by a query. This is especially useful when working with large datasets, as it helps improve performance and ensures that only a subset of data is returned.

Basic Syntax for Limiting Results:

db.collection.find().limit(number)

**number** specifies the maximum number of documents to return.

Find():

The find() method in MongoDB is used to query documents in a collection. It’s a very versatile and powerful method that allows you to retrieve documents based on specific conditions, filter the results, and apply various operations such as sorting, limiting, and projecting.

Basic Syntax of find():

db.collection.find(query, projection)

**query**: A filter object that defines the criteria for selecting the documents. If you want to retrieve all documents, you can use an empty object {}.

**projection**: Specifies which fields to include or exclude from the documents. This is optional. If not specified, all fields are returned.

Update():

In MongoDB, the **update()** method allows you to modify existing documents in a collection. There are several ways to update documents, depending on your requirements. MongoDB provides two primary methods for updating documents:

1. **updateOne()**: Updates a single document that matches the filter.
2. **updateMany()**: Updates multiple documents that match the filter.
3. **replaceOne()**: Replaces a single document with a new document.

**Basic Syntax:**

**updateOne() Syntax:**

**db.collection.updateOne(**

**<filter>,**

**<update>,**

**<options>)**

**filter**: Specifies the condition to match the document(s) you want to update.

**update**: Defines the modification to apply to the matched document.

**options**: Optional parameters (e.g., upsert, which creates a new document if no match is found).

updateMany() Syntax:

db.collection.updateMany(

<filter>,

<update>,

<options>

)

replaceOne() Syntax:

db.collection.replaceOne(

<filter>,

<replacement>,

<options>

)

Delete():

In MongoDB, the **delete()** operation is used to remove documents from a collection. However, it's important to note that **delete()** is an older method and has been replaced by more specific methods in MongoDB, such as:

1. **deleteOne()**: Removes a single document that matches the filter.
2. **deleteMany()**: Removes multiple documents that match the filter.

Syntax for deleteOne():

db.collection.deleteOne(

<filter>,

<options>

)

* **<filter>**: The condition that specifies which document to delete (similar to the query in find()).
* **<options>**: Optional settings (e.g., collation or writeConcern).

This method deletes the first document that matches the provided filter.

Syntax for deleteMany():

db.collection.deleteMany(

<filter>,

<options>

)

* **<filter>**: The condition that specifies which documents to delete.
* **<options>**: Optional settings (e.g., collation or writeConcern).

This method deletes all documents that match the provided filter.

Comparison Operators:

MongoDB provides a variety of comparison operators to help you filter data based on specific conditions. These operators can be used within queries to compare field values against specified conditions.

| **Operator** | **Description** |
| --- | --- |
| $eq | Equal to |
| $ne | Not equal to |
| $gt | Greater than |
| $gte | Greater than or equal to |
| $lt | Less than |
| $lte | Less than or equal to |
| $in | In an array of values |
| $nin | Not in an array of values |
| $exists | Field exists (true or false) |
| $type | Matches a specific data type |
| $all | Field is an array that contains all specified values |
| $size | Array has a specific number of elements |
|  |  |

AGGREGATION FUNCTION:

Aggregation in MongoDB is a way to **transform** and **analyze** your data. Unlike basic queries that just return documents (like rows in a table), the **aggregate** function helps you perform more complex operations, such as:

* **Summing up values** (e.g., total sales)
* **Grouping data** (e.g., total sales by each customer)
* **Filtering** data (e.g., getting only completed orders)
* **Sorting** data (e.g., sorting customers by total spend)

**Why Use Aggregation?**

Imagine you're working with a collection of **orders** in a store. You might want to:

* Get the total money spent by each customer.
* Find the most popular product.
* Count how many orders each customer has placed.

These tasks go beyond just retrieving a few documents—they involve **calculations** and **grouping**. That’s where **aggregation** comes in!

**How Does the Aggregate Function Work?**

The aggregate function works by passing data through a **pipeline**. A **pipeline** is just a series of steps, and each step transforms or processes the data in some way.

Pipeline:

In MongoDB aggregation, a **pipeline** is a sequence of **stages** through which the data passes, and each stage performs a specific operation on the data. Think of it like a conveyor belt where each stage processes the data in some way (filters, groups, sorts, etc.), and the result from one stage becomes the input for the next.

**Key Points about the Pipeline:**

1. **Series of Stages**: The pipeline consists of multiple stages, and each stage processes the data in a particular way. Each stage transforms the data and passes it to the next stage.
2. **Flow of Data**: The data "flows" through the stages in the order they are defined. Each stage takes the input (documents), does something with them, and passes the modified documents to the next stage.
3. **Transformations**: Each stage can transform the data by:
   * Filtering out unwanted documents ($match).
   * Grouping documents together based on a specific field ($group).
   * Sorting documents ($sort).
   * Adding new fields or reshaping data ($project).

**1. $group:**

* **Purpose**: Groups documents by a specified identifier and performs aggregation operations like summing values, finding averages, counting items, etc.
* **\_id**: The \_id field is always used to define the grouping criteria. You cannot group without specifying it, so when you group by a field (e.g., category or customer), it becomes the value of the \_id for each group.
* **Why use \_id**: MongoDB uses \_id as the default field for grouping. When you group by a specific field, that field will be stored under \_id. You can also rename or add other fields as part of the group.

Example of grouping by category:

{

$group: {

\_id: "$category", // Group by the 'category' field

totalSales: { $sum: "$sales" } // Sum the 'sales' field

}

}

**2. $match:**

* **Purpose**: Filters the documents that pass through the aggregation pipeline. It's similar to the WHERE clause in SQL. It's often used at the start of the pipeline to filter out documents early on, improving performance.
* **Efficiency**: Using $match early in the pipeline can speed up the query because fewer documents pass through subsequent stages.

Example of matching documents where status is "active":

{

$match: { status: "active" }

}

**3. $project:**

* **Purpose**: Reshapes the documents by including or excluding fields, adding new fields, or modifying the values of existing fields. It's similar to SELECT in SQL, but with more flexibility.

**1 and 0 in $project**:

* **1** means **include** the field in the output.
* **0** means **exclude** the field from the output.

**Why use 1 and 0**:

* By default, MongoDB includes all fields in the output unless you explicitly exclude them. To control what gets included or excluded, you can specify 1 for the fields you want, and 0 for the fields you want to omit.
* You cannot mix 1 and 0 in the same $project stage (except for \_id). If you include one field, you cannot exclude it in the same stage.

**Examples**:

* **Include specific fields**:

{

$project: {

name: 1, // Include the 'name' field

price: 1, // Include the 'price' field

\_id: 0 // Exclude the default '\_id' field

}

}

This includes name and price and excludes \_id from the result.

* **Add computed fields**:

{

$project: {

name: 1, // Include 'name'

price: 1, // Include 'price'

discountPrice: { $multiply: ["$price", 0.9] } // Add a new field 'discountPrice' as 90% of 'price'

}

}

In this example, discountPrice is calculated as 90% of the original price.

**Important Notes**:

* If you set \_id: 0, MongoDB will exclude the \_id field from the results. But, if you don’t explicitly exclude it (\_id: 0), it will be included by default in the output.

**Example of Excluding Specific Fields**:

{

$project: {

name: 1, // Include 'name'

\_id: 0 // Exclude '\_id'

}

}

**4. $sort:**

* **Purpose**: Orders the documents based on specified fields in ascending (1) or descending (-1) order. Similar to SQL's ORDER BY.

Example of sorting by price in descending order:

{

$sort: { price: -1 }

}

**5. $limit:**

* **Purpose**: Limits the number of documents that pass through the pipeline. This is useful when you need to reduce the result set size, similar to SQL's LIMIT.

Example to return only 5 documents:

{

$limit: 5

}

**6. $skip:**

* **Purpose**: Skips a specified number of documents, commonly used for pagination. After skipping documents, the remaining documents pass through to the next stage.

Example to skip the first 10 documents:

{

$skip: 10

}

**7. $sample:**

* **Purpose**: Randomly selects a specified number of documents. This is useful for sampling a subset of your data for analysis or testing.

Example to select 3 random documents:

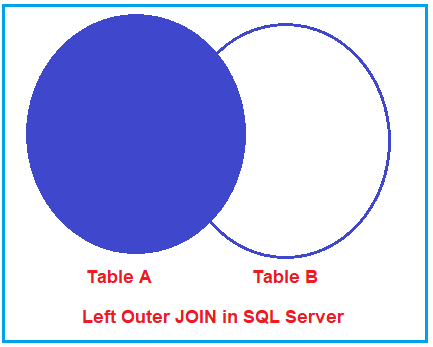
{

$sample: { size: 3 }

}

**8. $lookup:**

* **Purpose**: Performs a left outer join to another collection. This is equivalent to SQL's JOIN. You can combine documents from different collections based on a common field.



**from:**

* This specifies the collection that you want to join with the current collection.
* It's the "source" collection from which you want to pull matching documents.

**localField:**

* This is the field in the current (or "local") collection that will be used for the join.
* It is the key or field that will be matched against a field in the "from" collection.

**foreignField:**

* This refers to the field in the "from" collection (the collection you're joining with) that you want to match against the localField in the current collection.
* It's the field you're trying to find a match for in the "from" collection.

**as:**

* This is the name of the new field that will be added to each document in the local collection as a result of the join.
* The field will hold an array of matching documents from the "from" collection.

In MongoDB, the collection you're working with in an aggregation pipeline is always **the current collection** — that's the collection you're starting the aggregation from (in your case, orders).

So, even though orders is your "current collection," you need to tell MongoDB that you are joining it with **another collection** (customer).

* **from: "customer"** tells MongoDB: "Join the orders collection with the customer collection."
* **localField: "customerId"** tells MongoDB: "In orders, use the customerId field to find the related documents."
* **foreignField: "\_id"** tells MongoDB: "In the customer collection, match the customerId field in orders with the \_id field in customer."

This way, MongoDB can find the correct customer information for each order and include it in the customer\_info array.

Example of performing a lookup:

{

$lookup: {

from: "orders",

localField: "customerId",

foreignField: "customerId",

as: "customerOrders"

}

}

This join will take documents from the Shop collection and join them with the corresponding documents from the orders collection based on the customerId. The resulting array will be placed in the customerOrders field.

**$ before a field (e.g., "$customer")**:

* When you use $ before a field name (like "$customer"), you're referring to the value of the customer field in each document.
* Without the $ symbol, MongoDB would treat it as a string literal, not as a field in the document. So, the $ indicates that you're referring to the value of the field customer inside each document

This will add a new field customerOrders containing the matching documents from the orders collection for each document in the current collection.

**Summary of Operators:**

* **$group**: Groups documents by a specified key and performs aggregation operations on them.
* **$match**: Filters documents based on a condition.
* **$project**: Reshapes the documents by including, excluding, or adding new fields.
* **$sort**: Sorts documents based on specified fields.
* **$limit**: Limits the number of documents returned.
* **$skip**: Skips a specified number of documents.
* **$sample**: Selects a random sample of documents.
* **$lookup**: Joins documents from another collection.

VALIDATION:

**validation** allows you to enforce rules on the data being inserted or updated in a collection. You can define these rules using **schema validation** with **JSON Schema** or custom validation rules. MongoDB allows you to enforce constraints such as data types, required fields, or ranges for numeric values to ensure data integrity.

**Scenario:**

We want to store information about users in the collection, where each user must have:

* A **name** (string).
* An **age** (integer) that should be between 18 and 100.
* An **email** (string) that must follow a basic email format.

We will create the collection with validation rules and then test it by inserting data that either passes or fails the validation.

**Step 1: Create Collection with Validation**

First, let's create a collection myCollection with validation rules.

1. Open the **MongoDB shell** in your terminal by typing mongo (ensure your MongoDB server is running).
2. Use the following command to create a collection with validation rules:

db.createCollection("myCollection", {

validator: {

$jsonSchema: {

bsonType: "object", // Document must be an object

required: ["name", "age", "email"], // Fields that are required

properties: {

name: {

bsonType: "string", // 'name' must be a string

description: "Name must be a string"

},

age: {

bsonType: "int", // 'age' must be an integer

minimum: 18, // Age must be at least 18

maximum: 100, // Age must be at most 100

description: "Age must be an integer between 18 and 100"

},

email: {

bsonType: "string", // 'email' must be a string

pattern: "^.+@.+$", // 'email' must match the pattern of a basic email address

description: "Email must be a valid email address"

}

}

}

},

validationAction: "error" // Reject documents that don't meet the validation criteria

});

**Explanation:**

* **$jsonSchema**: We define a **JSON Schema** to validate the structure of documents.
* **bsonType**: Specifies the data type for each field (string, integer).
* **required**: Specifies that the name, age, and email fields must be present.
* **minimum/maximum**: Limits the age field to values between 18 and 100.
* **pattern**: Ensures the email is in a basic valid format (using regular expressions).
* **validationAction**: If set to "error", MongoDB will reject any document that does not conform to the validation rules.

**Step 2: Insert Valid Data**

Now that we have set up validation, let’s try inserting a valid document into the myCollection collection

db.myCollection.insertOne({

name: "Alice",

age: 25,

email: "alice@example.com"

});

* **This will succeed** because:
  + name is a string.
  + age is an integer between 18 and 100.
  + email matches the pattern of a basic email address.

**Step 3: Insert Invalid Data**

Next, let’s try inserting data that does **not** meet the validation criteria.

**Example 1: Age is too low (below 18):**

db.myCollection.insertOne({

name: "Bob",

age: 15, // Invalid age

email: "bob@example.com"

});

* **This will fail** because the age is less than 18, and the validation rule requires age to be between 18 and 100.

**Example 2: Missing email:**

db.myCollection.insertOne({

name: "Charlie",

age: 30, // Valid age

// Missing email

});

* **This will fail** because the email field is **required**, but it is missing.

**Example 3: Invalid email format:**

db.myCollection.insertOne({

name: "David",

age: 40,

email: "david.com" // Invalid email (missing @ symbol)

});

* **This will fail** because the email doesn't match the specified pattern (^.+@.+$), which is a basic email format validation.

**Step 4: Check the Valid Documents**

Now, you can retrieve the valid documents from the collection:

db.myCollection.find()

You should only see the valid document for **Alice**, since the other inserts failed.