

Data refers to any collection of information, facts, or statistics that can be stored, processed, and interpreted to derive meaningful insights. Data can take many forms, such as numbers, text, images, audio, video, and more. In the context of computing and technology, data is often represented and manipulated using binary code (0s and 1s), which forms the basis of all digital information.

A database, on the other hand, is a structured collection of data that is organized and stored in a way that allows for efficient storage, retrieval, and manipulation. Databases are designed to manage large volumes of data, making it easier to store, access, update, and analyze information. They are used in various applications such as business operations, scientific research, e-commerce, social media, and more.

Databases typically consist of tables, which are structured collections of data organized into rows and columns. Each row represents a record, and each column represents a field or attribute of that record. Databases provide mechanisms for querying (retrieving specific data), updating, and deleting data, as well as for ensuring data integrity, security, and scalability.



There are different types of databases, including:

- 1.Relational Databases:** These databases use a structured schema where data is organized into tables with predefined relationships between them. Examples include MySQL, PostgreSQL, Microsoft SQL Server, and Oracle Database.
- 2.NoSQL Databases:** These databases are designed to handle unstructured or semi-structured data and offer more flexible schemas. They include types like document stores (MongoDB), key-value stores (Redis), column-family stores (Cassandra), and graph databases (Neo4j).
- 3.In-Memory Databases:** These databases store data in system memory (RAM) instead of on disk, leading to faster data retrieval. Examples include Redis and Memcached.
- 4.Graph Databases:** These databases are optimized for managing data with complex relationships, making them suitable for applications like social networks and recommendation systems. Neo4j is a well-known example.
- 5.Time-Series Databases:** These databases are designed to handle and analyze time-stamped data, commonly used in applications like IoT sensor data and financial trading systems.
- 6.Spatial Databases:** These databases specialize in storing and querying spatial data, such as geographic information system (GIS) data used for mapping and location-based applications.



## SQL (Relational Databases):

### Data Model:

Relational databases store data in structured tables with predefined schemas.

Example: In a student database, you might have a "**Students**" table and a "**Courses**" table with defined **relationships**.

Schema:

Rigid **schema**: Changes to the schema require careful planning and may affect existing data.

Example: MySQL, PostgreSQL.

Data Relationships:

Well-defined relationships using primary and foreign keys.

Example: A "Customers" table linked to an "Orders" table using customer IDs.

Query Language:

SQL (**Structured Query Language**) is used for querying and managing data.

Example: `SELECT * FROM Employees WHERE Department = 'Sales'.`

Transactions:

ACID (**Atomicity, Consistency, Isolation, Durability**) compliance ensures data integrity.

Example: In banking, transferring money from one account to another is an atomic operation.



## **NoSQL Databases:**

### **1.Data Model:**

1. Various data models: document, key-value, column-family, graph, etc.
2. Example: MongoDB (document), Redis (key-value), Neo4j (graph).

### **2.Schema:**

1. Flexible schema: Allows dynamic changes to data structure.
2. Example: In a document database, each document can have different fields.

### **3.Data Relationships:**

1. Flexibility: Relationships can be defined but may be less rigid than in relational databases.
2. Example: In a document database, a blog post can embed comments as sub-documents.

### **4.Query Language:**

1. NoSQL databases often use their own query languages.
2. Example: MongoDB's query language for document databases.

### **5.Transactions:**

1. Eventual consistency: May not provide full ACID compliance due to distributed nature.
2. Example: In a distributed system, data changes might take some time to propagate.

### **6.Scalability:**

1. Horizontal scaling (sharding) is common, enabling databases to handle large amounts of data and traffic.
2. Example: Cassandra's ability to distribute data across nodes.



### **Why Choose SQL:**

- When data has a structured schema and well-defined relationships.
- For applications requiring complex querying, reporting, and analytics.
- When data integrity, consistency, and **ACID** compliance are crucial (e.g., financial applications).
- When dealing with a smaller dataset that fits well in a single server's resources.

### **Why Choose NoSQL:**

- For handling large volumes of unstructured or semi-structured data.
- When data schema is dynamic or evolving frequently.
- For applications with high write and read volumes and the need for fast data retrieval.
- When scalability across multiple servers or nodes is required.
- In scenarios where real-time data processing and low latency are critical.



## 1.Atomicity:

- Atomicity ensures that a transaction is treated as a single, indivisible unit of work. Either all the changes within the transaction are successfully completed, or none of them are applied.
- If a transaction fails at any point, all changes made so far are rolled back, and the database is left in its original state.
- Example: Imagine transferring money from one bank account to another. The transaction deducts the amount from one account and credits it to another. If the transfer fails midway, neither account should be affected.

## 2.Consistency:

- Consistency ensures that a transaction brings the database from one valid state to another. It enforces any defined constraints or rules and maintains data integrity.
- The database should remain consistent before and after the transaction, even if it involves multiple operations.
- Example: If a transaction involves transferring money between accounts, the sum of account balances before and after the transaction should be the same, ensuring that no money is created or lost during the process.

## 3.Isolation:

- Isolation ensures that multiple transactions can occur concurrently without interfering with each other. Transactions are isolated from one another until they are completed.
- Each transaction should appear to be executed in isolation, regardless of the actual order of execution.
- Example: If two users attempt to transfer money simultaneously, their transactions should not affect each other. The balance updates should occur independently and consistently.

## 4.Durability:

- Durability guarantees that once a transaction is successfully committed, its changes are permanent and will survive any system failures (e.g., power loss, crashes).
- Committed changes are stored in a durable medium, typically a disk, to ensure that they are not lost.
- Example: After a successful money transfer transaction, the changes to account balances are permanently recorded, even if the system crashes immediately afterward.



**1.Schema Flexibility:** NoSQL databases like MongoDB offer a flexible schema design, allowing you to store different types of data within the same collection without a predefined schema. This flexibility is especially beneficial during the development phase when requirements may change frequently.

**2.Object-Oriented Nature:** NoSQL databases, particularly document-oriented ones like MongoDB, map well to object-oriented programming paradigms used in JavaScript (used with both React and Node.js). Data stored as JSON-like documents in NoSQL databases can be directly consumed in JavaScript applications without extensive mapping or conversion.

**3.JSON-Like Format:** NoSQL databases store data in formats like JSON, which closely aligns with the way data is represented in JavaScript objects. This makes it easier to work with data within the stack's components (React, Node.js).

**4.Scalability:** NoSQL databases are designed for horizontal scalability, which is crucial for handling the potentially large user bases of modern web applications. MongoDB, for example, can distribute data across multiple servers or clusters to handle increased traffic.



**5. Agile Development:** In the fast-paced development environment of web applications, where requirements evolve rapidly, NoSQL databases' schema flexibility allows developers to adapt to changing needs more easily compared to traditional SQL databases.

**6. Document-Oriented Approach:** NoSQL databases like MongoDB are well-suited for scenarios where data is naturally hierarchical or nested, which is common in web applications. Documents can contain arrays and nested documents, mirroring the structure of modern web applications' UI components.

**7. Read/Write Performance:** Many NoSQL databases, including MongoDB, excel at high-speed read and write operations. This is important for real-time applications that require quick response times and dynamic updates.

**8. Geospatial Capabilities:** MongoDB provides strong support for geospatial data and queries, making it well-suited for location-based applications, maps, and geospatial analysis.

**9. Real-Time Data:** NoSQL databases are often favored for applications that require real-time updates and dynamic data synchronization, such as messaging apps, social networks, and collaborative tools.




Aspect	SQL (Relational Databases)	NoSQL Databases
Data Model	Tabular (Rows and Columns)	Various (Document, Key-Value, Column-Family, Graph, etc.)
Schema	Rigid, Fixed Schema	Flexible, Dynamic Schema
Data Relationships	Well-Defined, Foreign Keys	Flexible, Often Denormalized
Query Language	SQL (Structured Query Language)	Vendor-Specific Query Languages
Transactions	ACID (Atomicity, Consistency, Isolation, Durability) Compliance	Eventual Consistency (varies), Lack of Full ACID
Scaling	Vertical Scaling	Horizontal Scaling (Sharding)
Data Integrity	High, Enforced by Constraints	May Vary, Depends on Implementation
Flexibility	Limited for Unstructured Data	Suited for Unstructured and Semi-Structured Data
Performance	Complex Queries, Aggregations	Read/Write Performance Varies
Use Cases	Complex Queries, Structured Data	High Write/Read Volumes, Scalability
Examples	MySQL, PostgreSQL, SQL Server	MongoDB, Redis, Cassandra, Neo4j, Couchbase



## JSON Example:


json

 Copy code

```
{
  "name": "John Doe",
  "age": 30,
  "isStudent": false,
  "languages": ["English", "Spanish"],
  "address": {
    "city": "New York",
    "zipCode": "10001"
  }
}
```

## BSON Example:

less

 Copy code

```
\x34\x00\x00\x00      // Total BSON document size
\x02                   // Type: String
name\x00               // Field name: "name"
\x0B\x00\x00\x00John Doe\x00 // Value: "John Doe" (null-terminated)
\x10                   // Type: 32-bit Integer
age\x00                // Field name: "age"
\x1E\x00\x00\x00       // Value: 30
\x08                   // Type: Boolean
isStudent\x00          // Field name: "isStudent"
\x00                   // Value: false
\x04                   // Type: Array
languages\x00           // Field name: "languages"
\x19\x00\x00\x00       // Array total size
\x02                   // Type: String
\x07\x00\x00\x00English\x00 // Value: "English" (null-terminated)
\x02                   // Type: String
\x08\x00\x00\x00Spanish\x00 // Value: "Spanish" (null-terminated)
\x03                   // Type: Embedded Document
address\x00             // Field name: "address"
\x15\x00\x00\x00       // Document total size
\x02                   // Type: String
city\x00                // Field name: "city"
\x0A\x00\x00\x00New York\x00 // Value: "New York" (null-terminated)
\x02                   // Type: String
zipCode\x00             // Field name: "zipCode"
\x05\x00\x00\x0010001\x00 // Value: "10001" (null-terminated)
\x00                   // End of Document
```



## **File System:**

- A file system is a method for organizing and storing files and data on storage devices like hard drives or network storage.
- It provides a hierarchical structure of directories (folders) and files, much like a filing cabinet.
- It's suitable for basic storage and retrieval of files, but doesn't provide advanced querying, relationships, or structured data management.
- Each file is typically associated with a filename, extension, and location in the directory structure.
- File systems are often used for personal files, documents, images, and program files.
- They lack features like data relationships, querying capabilities, and transactions.
- Examples of file systems include FAT32, NTFS (used in Windows), HFS+ (used in macOS), and ext4 (used in Linux)

## **Database:**

- A database is a structured collection of data organized for efficient storage, retrieval, and manipulation.
- It offers features like data integrity, consistency, transactions, and powerful querying.
- Databases can be relational (SQL) or non-relational (NoSQL), each catering to different data storage and retrieval needs.



## **Key Differences:**

### **1.Data Structure:**

1. File systems store data in files and folders, without inherent support for complex relationships between data elements.
2. Databases store data in structured tables with well-defined relationships, allowing for efficient data retrieval and manipulation.

### **2.Data Integrity and Consistency:**

1. Databases enforce data integrity and consistency through constraints, normalization, and transaction management.
2. File systems do not inherently enforce data integrity and are more limited in terms of maintaining consistent relationships between data elements.

### **3.Querying and Reporting:**

1. Databases provide powerful querying capabilities using SQL, allowing for complex queries and data retrieval based on various criteria.
2. File systems offer basic file operations but lack sophisticated querying and reporting features.

### **4.Concurrency and Multi-User Support:**

1. Databases manage concurrent access by multiple users to ensure data consistency.
2. File systems might not handle concurrent access and data consistency as effectively in multi-user scenarios.



A **schema** is a fundamental concept in database management systems that defines the structure, organization, and relationships of the data stored in a database. It serves as a blueprint for how data is organized and represented within the database.

A schema provides a structured framework for organizing data within a database, ensuring data consistency, integrity, and efficient querying. It serves as a guideline for both database designers and users on how data is structured and manipulated within the database system.

The schema specifies the following aspects:

**1. Table Structure:** In a relational database, the schema defines the tables that will hold the data. Each table has a set of columns, and each column has a defined data type (such as text, number, date) that specifies what kind of data can be stored in that column.

**2. Column Constraints:** The schema also specifies constraints on columns, such as whether a column must contain a unique value (a primary key constraint), whether a column can contain null values, and whether certain values are allowed (data validation constraints).

**3. Relationships:** For relational databases, the schema outlines the relationships between tables. These relationships are established using keys—specifically, primary keys and foreign keys—which help maintain data integrity and enable data retrieval across related tables.



**4.Indexes:** Schema definition also includes decisions about which columns need indexing. Indexes enhance the efficiency of data retrieval operations by creating data structures that speed up the search process.

**5.Data Integrity Rules:** The schema can enforce rules to maintain the consistency and integrity of the data. This might involve specifying that certain data values are required, setting up referential integrity between related tables, and defining rules for data validation and transformation.

**6.Views:** A schema can include the definition of database views. Views are virtual tables that display data from one or more underlying tables in a customized way, often to simplify complex queries or present specific subsets of data to users.

**7.Security and Permissions:** Schema definition can also involve specifying access controls and permissions, determining who can read, write, modify, or delete data within the database.



# What Is Difference Between Collections and Tables

## Collections:

- Collections are a term commonly used in NoSQL databases, especially in document-oriented databases like MongoDB.
- In a NoSQL context, a collection is a grouping of similar or related documents. A document is a data structure that contains key-value pairs, similar to a JSON object.
- Each document within a collection can have a different structure, meaning that different documents can have different fields.
- Collections are schema-flexible, allowing you to add new fields to documents without affecting others in the collection.
- Collections are used to store data in a semi-structured or unstructured format.

## Tables:

- Tables are the fundamental structure in relational databases.
- In a relational database, a table is a two-dimensional structure made up of rows and columns.
- Each row represents a record, and each column represents an attribute or field of that record.
- The structure of a table is defined by its schema, which specifies the columns and their data types.
- All rows in a table follow the same structure defined by the schema. Any changes to the schema typically affect the entire table.
- Relational databases enforce data integrity through defined relationships between tables and use the concept of keys (primary keys and foreign keys) to ensure data consistency.




**Install MongoDB:** official MongoDB website (<https://www.mongodb.com/try/download/community>).

**Start MongoDB Server:** Once MongoDB is installed, you'll need to start the MongoDB server. Open your command prompt or terminal and navigate to the directory where MongoDB is installed. Run the following command to start the MongoDB server:



- To show all available databases:


sql

 Copy code

```
show dbs
```

- To switch to a specific database (or create it if it doesn't exist):


perl

 Copy code

```
use mydb
```

- To show all collections in the current database:

sql


 Copy code

```
show collections
```

### 1. Creating a Database:

To create a new database, use the `use` command followed by the desired database name. MongoDB will create the database if it doesn't already exist.

shell


 Copy code

```
use mydatabase
```

### 2. Creating a Collection:

Once you've selected a database, you can create a collection using the `db.createCollection()` command.

shell

 Copy code

```
db.createCollection("mycollection")
```

Replace `"mycollection"` with the name you want to give to your collection.



```
Atlas atlas-6jk5ln-shard-0 [primary] test> show dbs
doctor-app    24.00 KiB
doctorapp    128.00 KiB
ecommerce     88.00 KiB
test         108.00 KiB
admin        248.00 KiB
local        38.43 GiB
Atlas atlas-6jk5ln-shard-0 [primary] test>
```

```
Atlas atlas-6jk5ln-shard-0 [primary] test> use student
switched to db student
Atlas atlas-6jk5ln-shard-0 [primary] student> show dbs
doctor-app    24.00 KiB
doctorapp    128.00 KiB
ecommerce     88.00 KiB
test         108.00 KiB
admin        248.00 KiB
local        38.43 GiB
Atlas atlas-6jk5ln-shard-0 [primary] student>
```

```
Atlas atlas-6jk5ln-shard-0 [primary] test> show collection
MongoshInvalidInputError: [COMMON-10001] 'collection' is not a valid arg
ument for "show".
Atlas atlas-6jk5ln-shard-0 [primary] test> show collections
users
Atlas atlas-6jk5ln-shard-0 [primary] test>
```

```
Atlas atlas-6jk5ln-shard-0 [primary] student> show collections
data
Atlas atlas-6jk5ln-shard-0 [primary] student>
```

```
Atlas atlas-6jk5ln-shard-0 [primary] student> db.collections('data')
TypeError: db.collections is not a function
Atlas atlas-6jk5ln-shard-0 [primary] student> db.createCollections('data
')
TypeError: db.createCollections is not a function
Atlas atlas-6jk5ln-shard-0 [primary] student> db.createCollection('data'
)
{ ok: 1 }
Atlas atlas-6jk5ln-shard-0 [primary] student>
```


```
Atlas atlas-6jk5ln-shard-0 [primary] student> show dbs
doctor-app    24.00 KiB
doctorapp    128.00 KiB
ecommerce     88.00 KiB
student       8.00 KiB
test         108.00 KiB
admin        248.00 KiB
local        38.43 GiB
Atlas atlas-6jk5ln-shard-0 [primary] student>
```



### 3. Deleting a Collection:

To delete a collection, use the `db.collectionName.drop()` command.

shell

 Copy code


```
db.mycollection.drop()
```

Replace `"mycollection"` with the name of the collection you want to delete.

### 4. Deleting a Database:

To delete a database, first switch to a different database (e.g., the `"admin"` database) using the `use` command. Then, use the `db.dropDatabase()` command.

shell


 Copy code

```
use admin  
db.dropDatabase()
```

### 3. Inserting with Custom `_id`:

By default, MongoDB will automatically generate an `_id` field for each inserted document. However, you can specify your own `_id` value if needed.

shell

 Copy code


```
db.collectionName.insertOne({ _id: 123, key1: "value1", key2: "value2" })
```

Replace `collectionName` with the actual name of your collection and provide the `_id` and other fields for the document.

### 1. Inserting a Single Document:

To insert a single document into a collection, you can use the `insertOne()` method.

shell

 Copy code


```
db.collectionName.insertOne({ key1: "value1", key2: "value2" })
```

Replace `collectionName` with the actual name of your collection and provide the keys and values for the document.

### 2. Inserting Multiple Documents:

To insert multiple documents into a collection, you can use the `insertMany()` method.

shell

 Copy code

```
db.collectionName.insertMany([  
  { key1: "value1", key2: "value2" },  
  { key1: "value3", key2: "value4" }  
)
```

Replace `collectionName` with the actual name of your collection and provide an array of documents.



```
Atlas atlas-6jk5ln-shard-0 [primary] student> use student
already on db student
Atlas atlas-6jk5ln-shard-0 [primary] student>
```

```
Atlas atlas-6jk5ln-shard-0 [primary] student> db.data.drop()
true
Atlas atlas-6jk5ln-shard-0 [primary] student> show collections

Atlas atlas-6jk5ln-shard-0 [primary] student>
```

```
Atlas atlas-6jk5ln-shard-0 [primary] student> db.dropDatabase()
MongoServerError: user is not allowed to do action [dropDatabase] on [student.]
Atlas atlas-6jk5ln-shard-0 [primary] student>
```

```
students> db.data.insertOne({'name':'vinod', age:29})
{
  acknowledged: true,
  insertedId: ObjectId("64d5c719e857eb95ebc6c6a8")
}
students> db.data.insertMany([ {'name':'vinod', age:29}, {'name':'binamra', age:20},
{'name':'arjun', age:30}  ])
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId("64d5c75ce857eb95ebc6c6a9"),
    '1': ObjectId("64d5c75ce857eb95ebc6c6aa"),
    '2': ObjectId("64d5c75ce857eb95ebc6c6ab")
  }
}
```



```
students> db.data.insertMany([ {'name': 'vinod', age: 29}, { _id: ObjectId("64d5c75ce857eb95ebc6c6ab"), 'name': 'arjun', age: 30}, {'name': 'arjun', age: 30} ])  
Uncaught:
```

```
Write Errors: [  
  WriteError {  
    err: {  
      index: 1,  
      code: 11000,  
      errmsg: "E11000 duplicate key error collection: students.data  
key: { _id: ObjectId('64d5c75ce857eb95ebc6c6ab') }",  
      errInfo: undefined,  
      op: {  
        _id: ObjectId("64d5c75ce857eb95ebc6c6ab"),  
        name: 'arjun',  
        age: 30  
      }  
    }  
  }  
]  
]
```



```
students> db.data.insertMany([ {'name':'vinod', age:29}, { _id: ObjectId("64d5c75ce857eb95ebc6c6ab"), 'name':'arjun', age:30}, {'name':'arjun', age:30} ], {ordered:false})
Uncaught:
MongoBulkWriteError: E11000 duplicate key error collection: students.data index: _id
_ dup key: { _id: ObjectId('64d5c75ce857eb95ebc6c6ab') }
Result: BulkWriteResult {
  insertedCount: 2,
  matchedCount: 0,
  modifiedCount: 0,
  deletedCount: 0,
  upsertedCount: 0,
  upsertedIds: {},
```

## Ordered and Unordered Inserts

When executing bulk write operations, "ordered" and "unordered" determine the batch behavior.

### Ordered Inserts

Default behavior is ordered, where MongoDB stops on the first error.

```
db.<collection-name>.insertMany([ doc1, doc2, ... ]);
```

### Unordered Inserts

When executing bulk write operations with unordered flag, MongoDB continues processing after encountering an error.

```
db.<collection-name>.insertMany([ doc1, doc2, ... ], { ordered: false });
```





## Case Sensitivity in MongoDB

- Collection names are case-sensitive.
- Field names within documents are also case-sensitive.
- `db.Product.insertOne({ name: 'thapa', age: 30 });`
- `db.product.insertOne({ name: 'thapa', age: 30 });`



```
students> db.data.find({'name': 'vinod'})
[
  { _id: ObjectId("64d5c719e857eb95ebc6c6a8"), name: 'vinod', age: 29 },
  { _id: ObjectId("64d5c75ce857eb95ebc6c6a9"), name: 'vinod', age: 29 },
  {
    _id: ObjectId("64d5c806e857eb95ebc6c6ac"),
    name: 'vinod',
    age: 29,
    'course name': 'cs'
  },
  { _id: ObjectId("64d5c8cee857eb95ebc6c6ad"), name: 'vinod', age: 29 },
  { _id: ObjectId("64d5c9afe857eb95ebc6c6af"), name: 'vinod', age: 29 }
]
```



## Comparison Operators

\$eq

\$ne

\$gt

\$gte

\$lt

\$lte

\$in

\$nin

❖ `db.collectionName.find({'fieldname': {$operator: value } })`

❖ `db.products.find({ 'price': { $eq: 699 } });`

❖ `db.category.find({ price: { $in: [249, 129, 39] } });`



# Cursor Methods

count()

limit()

skip()

sort()

- db.products.find({ price: { \$gt: 250 } }).count();
- db.products.find({ price: { \$gt: 250 } }).limit(5);
- db.products.find({ price: { \$gt: 250 } }).limit(5).skip(2);
- db.products.find({ price: { \$gt: 1250 } }).limit(3).sort({ price: 1 });
  - (1) for ascending and (-1) for descending





```
shop> db.products.find({'price': {$gt: 1250 }}).limit(3).sort({'price':1})
```

```
[
  {
    _id: ObjectId("64c23601e32f4a51b19b9263"),
    name: 'Laptop Pro',
    company: '64c23350e32f4a51b19b9231',
    price: 1299,
    colors: [ '#333333', '#cccccc', '#00ff00' ],
    image: '/images/product-laptop.png',
    category: '64c2342de32f4a51b19b924e',
    isFeatured: true
  },
  {
    _id: ObjectId("64c236a2e32f4a51b19b9281"),
    name: 'Diamond Ring',
    company: '64c23350e32f4a51b19b923a',
    price: 1999,
    colors: [ '#000000', '#cc6600', '#663300' ],
    image: '/images/product-diamond-ring.png',
    category: '64c2342de32f4a51b19b9259',
    isFeatured: false
  },
  {
    _id: ObjectId("64c23707e32f4a51b19b9296"),
    name: 'Diamond Ring',
```



ESP - NED  
Game score



Search





## Logical Operators

\$and

\$or

\$not

\$nor


- { \$and: [ { condition1 }, { condition2 }, ... ] }

- { field: { \$not: { operator: value } } }

```
shop> db.products.find({$and: [{ 'price':{$gt: 100}}, { 'name': 'Diamond Ring' } ] })
[
  {
    _id: ObjectId("64c236a2e32f4a51b19b9281"),
    name: 'Diamond Ring',
    price: 150
  }
]
```



javascript


 Copy code

```
db.products.deleteMany({ brand: "Apple" })
```

#### 4. Delete Documents Based on a Range:

Delete documents that match a range condition (e.g., delete products with a price less than \$100).

javascript


 Copy code

```
db.collection.deleteMany({ price: { $lt: 100 } })
```

#### 6. Delete Products with No Stock:

Delete products that are out of stock (assuming there's a "stock" field).

javascript

 Copy code

```
db.products.deleteMany({ stock: 0 })
```



```
shop> db.products.find({'price':{'$eq:100'}}).count()
```

```
7
```

```
shop> db.products.find({'price':{'$not: {$eq:100}} }).count()
```

```
10354
```

```
shop> db.products.find({'price':{'$ne:100}} ).count()
```

```
10354
```

```
shop> |
```



# Complex Expressions

- The `$expr` operator allows using aggregation expressions within a query.
- Useful when you need to compare fields from the same document in a more complex manner.

## • Syntax

• `{ $expr: { operator: [field, value] } }`

## • Example

• `db.products.find({ $expr: { $gt: ['$price', 1340] } });`

```
shop> db.sales.find({ $expr: { $gt: [{ $multiply: ['$quantity', '$price'] }, '$targetPrice' ] } }  
)  
[  
  { _id: 5, quantity: 5, price: 55, targetPrice: 150 },  
  { _id: 3, quantity: 6, price: 35, targetPrice: 100 },  
  { _id: 1, quantity: 10, price: 15, targetPrice: 120 },  
  { _id: 4, quantity: 5, price: 55, targetPrice: 150 },  
  { _id: 2, quantity: 5, price: 25, targetPrice: 100 }  
]
```



## Elements Operator

\$exists

\$type

\$size

• { field: { \$exists: <boolean> } }

```
shop> db.products.find({price: {$exists: true}}).count()  
10355
```

```
shop> db.products.find({price: {$exists: false}, price: {$gt: 1250} })  
[  
  {  
    _id: ObjectId("64c236a2e32f4a51b19b9281"),  
    name: 'Diamond Ring',
```



```
shop> db.products.find({price: {$type: 'number'}}).count()  
10355  
shop> db.products.find({price: {$type: 'string'}}).count()  
0  
shop>
```

## Projection

- `db.collection.find({}, { field1: 1, field2: 1 })`
- To include specific fields, use projection with a value of 1 for the fields you want.
- To exclude fields, use projection with a value of 0 for the fields you want to exclude.
- You cannot include and exclude fields simultaneously in the same query projection.






#### 1. Find All Documents:

To retrieve all documents from the collection, you can use the `find()` method without any arguments:

shell


 Copy code

```
db.products.find()
```

#### 2. Limit the Number of Results:

If you have a large collection and want to limit the number of results returned, you can use the `limit()` method:

shell


 Copy code

```
db.products.find().limit(5)
```

#### 3. Filter Documents with a Condition:

You can use the `find()` method with a query object to filter documents based on specific criteria. For example, to find all products with a brand of "Apple":

shell


 Copy code

```
db.products.find({ brand: "Apple" })
```

#### 4. Projection - Select Specific Fields:

You can use the second argument of the `find()` method to specify which fields to include in the returned documents. For example, to retrieve only the "title" and "price" fields of products with a brand of "Apple":

shell


 Copy code

```
db.products.find({ brand: "Apple" }, { title: 1, price: 1, _id: 0 })
```

#### 5. Sorting Documents:

You can sort the results using the `sort()` method. For example, to retrieve products sorted by price in ascending order:

shell

 Copy code

```
db.products.find().sort({ price: 1 })
```



1. **\*\*Product Information:\*\***

- What are the details of the product with ID 5?
- List all products from the "laptops" category.
- Retrieve the title, price, and brand of products with a discount percentage greater than 15%.

2. **\*\*Brand and Category Queries:\*\***

- How many products belong to the "Samsung" brand?
- Find all products that are categorized as "smartphones" and have a rating greater than 4.5.
- Which brands have products in the "fragrances" category?

3. **\*\*Price and Discount Queries:\*\***

- What is the average price of all products?
- Show the products with a price between \$30 and \$50.
- List the top 5 products with the highest discounts.

4. **\*\*Stock and Availability:\*\***

- How many products have a stock quantity of less than 50?
- Find the products that are out of stock (stock = 0).
- Which category has the highest total stock quantity?

5. **\*\*Rating and Reviews:\*\***

- Retrieve products that have a rating above 4.8 and at least 100 units in stock.
- List the products with a rating below 4.0 and provide their titles and descriptions.
- Calculate the average rating for products in the "skincare" category.



### 1. Details of the Product with ID 5:

javascript

Copy code

```
db.products.find({ "id": 5 })
```

This query will retrieve the details of the product with ID 5.

### 2. List of Products from the "laptops" Category:

javascript

Copy code

```
db.products.find({ "category": "laptops" })
```

### 3. Retrieve Title, Price, and Brand of Products with Discount > 15%:

javascript

Copy code

Copy code

```
db.products.find({ "discountPercentage": { $gt: 15 } }, { "title": 1, "price": 1, "brand": 1, "_id": 0 })
```

```
products> db.items.find({'brand': 'Samsung'}).count()
```

```
2
```


```
products> db.items.find({ "category": "smartphones", "rating": { $gt: 4.5 } })
```

```
[
  {
    _id: ObjectId("64f149b8529a0e0394baf53c"),
    id: 1,
    title: 'iPhone 9',
    description: 'An apple mobile which is nothing like apple',
    price: 549,
    discountPercentage: 12.96,
    rating: 4.69,
    stock: 94,
    brand: 'Apple',
    category: 'smartphones',
    thumbnail: 'https://i.dummyjson.com/data/products/1/thumbnail.jpg',
```



### 3. Brands that have products in the "fragrances" category:


javascript

 Copy code

```
db.products.distinct("brand", { "category": "fragrances" })
```

### 1. Average price of all products:

javascript


 Copy code

```
db.products.aggregate([
  { $group: { _id: null, avgPrice: { $avg: "$price" } } }
])
```

This aggregation query will calculate the average price of all products.

### 2. Products with a price between \$30 and \$50:

javascript


 Copy code

```
db.products.find({ "price": { $gte: 30, $lte: 50 } })
```

This query will retrieve products with a price between \$30 and \$50.

### 3. Top 5 products with the highest discounts:

javascript

 Copy code


```
db.products.find().sort({ "discountPercentage": -1 }).limit(5)
```

This query will retrieve the top 5 products with the highest discount percentages, sorted in descending order.



### 1. Number of products with a stock quantity of less than 50:

javascript


 Copy code

```
db.products.count({ "stock": { $lt: 50 } })
```

This query will count the number of products that have a stock quantity of less than 50.

### 2. Products that are out of stock (stock = 0):

javascript


 Copy code

```
db.products.find({ "stock": 0 })
```

This query will retrieve products that are out of stock (stock quantity = 0).

### 3. Category with the highest total stock quantity:

javascript

 Copy code

```
db.products.aggregate([
  { $group: { _id: "$category", totalStock: { $sum: "$stock" } } },
  { $sort: { totalStock: -1 } },
  { $limit: 1 }
])
```

```
products> db.items.aggregate([ { $group: { _id: "$category", totalStock: { $sum: "$stock" } } } ]])
[
  { _id: 'smartphones', totalStock: 319 },
  { _id: 'laptops', totalStock: 386 },
  { _id: 'home-decoration', totalStock: 263 },
  { _id: 'groceries', totalStock: 465 },
  { _id: 'skincare', totalStock: 470 },
  { _id: 'fragrances', totalStock: 397 }
]
```





Certainly! Here are the solutions to your new queries:



1. **Products with a rating above 4.8 and at least 100 units in stock:**

javascript

Copy code

```
db.products.find({ "rating": { $gt: 4.8 }, "stock": { $gte: 100 } })
```

This query will retrieve products with a rating above 4.8 and at least 100 units in stock.

2. **Products with a rating below 4.0 and their titles and descriptions:**

javascript

Copy code

```
db.products.find({ "rating": { $lt: 4.0 } }, { "title": 1, "description": 1
```

This query will retrieve products with a rating below 4.0 and display their titles and descriptions.

3. **Average rating for products in the "skincare" category:**


javascript

Copy code

```
db.products.aggregate([
  { $match: { "category": "skincare" } },
  { $group: { _id: null, avgRating: { $avg: "$rating" } } }
])
```



javascript

 Copy code

```
db.products.aggregate([
  { $match: { "category": { $in: ["skincare", "laptops"] } } },
  { $group: { _id: null, avgRating: { $avg: "$rating" } } }
])
```

```
products> db.items.aggregate([ { $group: { _id: "$category", avgRating: { $avg: "$rating" } } }, { $project: { _id: 0, category: "$_id", avgRating: 1 } } ] )
[
  { avgRating: 4.322, category: 'smartphones' },
  { avgRating: 4.444, category: 'laptops' },
  { avgRating: 4.5340000000000001, category: 'home-decoration' },
  { avgRating: 4.734, category: 'groceries' },
  { avgRating: 4.478, category: 'skincare' },
  { avgRating: 4.351999999999999, category: 'fragrances' }
]
```



. **\*\*Image URLs and Thumbnails:\*\***

- Show the URLs of images associated with the product with ID 10.
- Retrieve the thumbnails of products with a brand of "Apple".

7. **\*\*Mixed Queries:\*\***

- List products with a discount percentage between 10% and 20% and a rating greater than 4.2.
- Find laptops that are priced above \$1000 and have a rating above 4.5.

8. **\*\*Category Analysis:\*\***

- How many products are there in each category? Provide a count for each category.

9. **\*\*Brands and Discounts:\*\***

- Calculate the average discount percentage for each brand.
- Which brand has the highest average discount percentage?


10. **\*\*Combining Conditions:\*\***

- Retrieve products from the "groceries" category with a price less than \$15 and a stock quantity greater than 50.



1. Show the URLs of images associated with the product with ID 10:


javascript

 Copy code

```
db.products.findOne({ "id": 10 }, { "images": 1 })
```

1. Retrieve the thumbnails of products with a brand of "Apple":


javascript

 Copy code

```
db.products.find({ "brand": "Apple" }, { "thumbnail": 1 })
```

1. How many products are there in each category? Provide a count for each category:


javascript

 Copy code

```
db.products.aggregate([
  {
    $group: {
      _id: "$category",
      count: { $sum: 1 }
    }
  }
])
```

1. List products with a discount percentage between 10% and 20% and a rating greater than 4.2:


javascript

 Copy code

```
db.products.find({
  "discountPercentage": { $gte: 10, $lte: 20 },
  "rating": { $gt: 4.2 }
})
```

1. Find laptops that are priced above \$1000 and have a rating above 4.5:

javascript


 Copy code

```
db.products.find({
  "category": "laptops",
  "price": { $gt: 1000 },
  "rating": { $gt: 4.5 }
})
```



1. Calculate the average discount percentage for each brand:


javascript

 Copy code

```
db.products.aggregate([
  {
    $group: {
      _id: "$brand",
      avgDiscountPercentage: { $avg: "$discountPercentage" }
    }
  }
])
```


To find out which brand has the highest average discount percentage, you can sort the result in descending order and limit it to the first document:

javascript

 Copy code

```
db.products.aggregate([
  {
    $group: {
      _id: "$brand",
      avgDiscountPercentage: { $avg: "$discountPercentage" }
    }
  },
  {
    $sort: { avgDiscountPercentage: -1 }
  },
  {
    $limit: 1
  }
])
```

javascript

 Copy code

```
db.products.find({
  "category": "groceries",
  "price": { $lt: 15 },
  "stock": { $gt: 50 }
})
```




11. **\*\*Best Selling Products:\*\***
  - List the top 5 best selling products based on their stock availability.
12. **\*\*Variety in Brands:\*\***
  - How many different brands are present in the dataset?
  - Which brands have products in more than one category?
13. **\*\*Image Analysis:\*\***
  - Find products with the maximum number of images. Provide their titles and the number of images.
14. **\*\*Discount Distribution:\*\***
  - Create a histogram of discount percentages to visualize the distribution of discounts.
15. **\*\*Popular Categories:\*\***
  - Identify the category with the highest average rating.
  - List the categories in descending order of the average rating.
16. **\*\*Brand Comparison:\*\***
  - Compare the average ratings of products from "Apple" and "Samsung" brands.
  - Compare the average prices of products from "HP Pavilion" and "Dermive" brands.
17. **\*\*Stock vs. Discount:\*\***
  - Analyze the relationship between stock availability and discount percentages. Do products with lower stock tend to have higher discounts?



### Best Selling Products:


javascript

 Copy code

```
db.products.find().sort({ "stock": -1 }).limit(5)
```

### Variety in Brands - Count of Different Brands:


javascript

 Copy code

```
db.products.distinct("brand").length
```

### Variety in Brands - Brands with Products in More Than One Category:


javascript

 Copy code

```
db.products.aggregate([
  {
    $group: {
      _id: "$brand",
      categories: { $addToSet: "$category" }
    }
  },
  {
    $match: {
      categories: { $size: { $gt: 1 } }
    }
  }
])
```

### Image Analysis - Products with Maximum Number of Images:

javascript


 Copy code

```
db.products.aggregate([
  {
    $project: {
      title: 1,
      imageCount: { $size: "$images" }
    }
  },
  {
    $sort: {
      imageCount: -1
    }
  },
  {
    $limit: 1
  }
])
```



### Popular Categories - Category with Highest Average Rating:


javascript

 Copy code

```
db.products.aggregate([
  {
    $group: {
      _id: "$category",
      avgRating: { $avg: "$rating" }
    }
  },
  {
    $sort: {
      avgRating: -1
    }
  },
  {
    $limit: 1
  }
])
```

### Popular Categories - Categories in Descending Order of Average Rating:

javascript


 Copy code

```
db.products.aggregate([
  {
    $group: {
      _id: "$category",
      avgRating: { $avg: "$rating" }
    }
  },
  {
    $sort: {
      avgRating: -1
    }
  }
])
```



### Brand Comparison - Average Ratings of "Apple" and "Samsung" Products:


javascript

 Copy code

```
db.products.aggregate([
  {
    $match: {
      brand: { $in: ["Apple", "Samsung"] }
    }
  },
  {
    $group: {
      _id: "$brand",
      avgRating: { $avg: "$rating" }
    }
  }
])
```

### Brand Comparison - Average Prices of "HP Pavilion" and "Dermive" Products:


javascript

 Copy code

```
db.products.aggregate([
  {
    $match: {
      brand: { $in: ["HP Pavilion", "Dermive"] }
    }
  },
  {
    $group: {
      _id: "$brand",
      avgPrice: { $avg: "$price" }
    }
  }
])
```



javascript

 Copy code

```
db.products.aggregate([
  {
    $project: {
      stock: 1,
      discountPercentage: {
        $multiply: [{ $divide: [{ $subtract: ["$originalPrice", "$price"] }, "$originalPrice"] }, 100]
      }
    }
  },
  {
    $group: {
      _id: "$stock",
      avgDiscount: { $avg: "$discountPercentage" }
    }
  },
  {
    $sort: {
      _id: 1
    }
  }
])
```

18. **Category Insights:**

- Show the products in the "home-decoration" category with ratings above 4.5.
- Calculate the average price of products in the "skincare" category.

19. **Product Availability:**

- List products that are in stock and have a discount percentage greater than 10%.

20. **Comparison by Rating:**

- Compare the average ratings of products in the "smartphones" and "laptops" categories.

21. **Discount Range Analysis:**

- Calculate the percentage of products with discount percentages in the range of 5% to 15%.

22. **Brand and Price:**

- Find the brand with the highest average price.
- Find the brand with the lowest average price.

23. **High-Value Products:**

- Show the products with a price above \$1000 and a discount percentage greater than 10%.



25. **\*\*Product and Category Count:\*\***

- Count the number of products in each category and display the results.

26. **\*\*Limited Availability:\*\***

- List products with a stock quantity of 20 or less.

27. **\*\*Discounted Fragrances:\*\***

- Display the products in the "fragrances" category that have a discount percentage above 15%.

28. **\*\*Category Comparison:\*\***

- Compare the average prices of products in the "groceries" and "home-decoration" categories.


29. **\*\*Best Value Products:\*\***

- Identify the products with the highest rating-to-price ratio.

30. **\*\*Discounted Electronics:\*\***

- List products in the "smartphones" and "laptops" categories that have a discount percentage greater than 10%.

```
shop> db.comments.find({'comments.user': {$all: ['Alice', 'Vinod']} })
[
  {
    _id: 7,
    title: 'Introduction to MongoDB',
    content: 'MongoDB is a popular NoSQL database...',
    author: 'Vinod Thapa',
    comments: [
      { user: 'Alice', text: 'Awesome article!' },
      { user: 'Vinod', text: 'Thanks for sharing.' }
    ],
    metadata: { views: 1000, likes: 70 }
  }
]
```




```
shop> db.comments.find({'comments': {$elemMatch: {'user': 'Vinod', 'text': 'Thanks for sharing.'}}})
[
  {
    _id: 7,
    title: 'Introduction to MongoDB',
    content: 'MongoDB is a popular NoSQL database...',
    author: 'Vinod Thapa',
    comments: [
      { user: 'Alice', text: 'Awesome article!' },
      { user: 'Vinod', text: 'Thanks for sharing.' }
    ],
    metadata: { views: 1000, likes: 70 }
  }
]
```






csharp

 Copy code

- Find the brand with the lowest average price.

javascript


 Copy code

```
db.products.aggregate([
  {
    $group: {
      _id: "$brand",
      averagePrice: { $avg: "$price" }
    }
  },
  {
    $sort: { averagePrice: 1 }
  },
  {
    $limit: 1
  }
])
```

### 1. Discount Range Analysis:

- Calculate the percentage of products with discount percentages in the range of 5% to 15%.

javascript


 Copy code

```
db.products.count({
  "discountPercentage": { $gte: 5, $lte: 15 }
})
```

## 1. Comparison by Rating:


- Compare the average ratings of products in the "smartphones" and "laptops" categories.

javascript

 Copy code


```
db.products.aggregate([
  {
    $group: {
      _id: "$category",
      averageRating: { $avg: "$rating" }
    }
  },
  {
    $match: {
      _id: { $in: ["smartphones", "laptops"] }
    }
  }
])
```

arduino

 Copy code

- Calculate the average price of products in the "skincare" category.

javascript

 Copy code


```
db.products.aggregate([
  { $match: { "category": "skincare" } },
  {
    $group: {
      _id: null,
      averagePrice: { $avg: "$price" }
    }
  }
])
```



## 1. Product Availability:


- List products that are in stock and have a discount percentage greater than 10%.

javascript

 Copy code

```
db.products.find({  
  "stock": { $gt: 0 },  
  "discountPercentage": { $gt: 10 }  
})
```

javascript

 Copy code

```
db.products.aggregate([  
  {  
    $group: {  
      _id: "$category",  
      count: { $sum: 1 }  
    }  
  }  
])
```

Calculate the number of products in each price range (e.g., \$0-500, \$500-1000, \$1000+) and display the results.

javascript

Copy code

```
db.products.aggregate([
  {
    $project: {
      priceRange: {
        $switch: {
          branches: [
            {
              case: { $lte: ["$price", 500] },
              then: "$0-500"
            },
            {
              case: { $lte: ["$price", 1000] },
              then: "$500-1000"
            }
          ],
          default: "$1000+"
        }
      }
    },
    {
      $group: {
        _id: "$priceRange",
        count: { $sum: 1 }
      }
    },
    {
      $sort: { _id: 1 }
    }
  ]
})
```

Compare the average prices of products in the "groceries" and "home-decoration" categories.

javascript

Copy code

```
db.products.aggregate([
  {
    $match: {
      $or: [
        { category: "groceries" },
        { category: "home-decoration" }
      ]
    }
  },
  {
    $group: {
      _id: "$category",
      averagePrice: { $avg: "$price" }
    }
  }
])
```



▼	📁 connection	●
	JS db.js	U
▼	📁 model	●
	JS userModel.js	U
	JS userschem...	U
>	📁 node_modu...	●
	📄 .env	U
	📄 package-loc...	U
	📄 package.json	U

```


1  import mongoose from "mongoose";
2  import chalk from "chalk";
3  // Use import
4  import dotenv from "dotenv";
5  dotenv.config();
6
7  const connectDB = async () => {
8    try {
9      const conn = await mongoose.connect(process.env.MONGO_URL);
10     console.log(
11       chalk.bgMagenta.white(`Connected to MongoDB ${conn.connection.host}`)
12     );
13   } catch (error) {
14     console.error(chalk.bgRed.white(`Error in connection ${error}`));
15   }
16 };
17 export default connectDB;
18

```


```
import mongoose from "mongoose";
import Product from "../userschema.js";
import connectDB from "../connection/db.js";
```

```
const createDocument = async () => {
  try {
```

```
    // Connect to the database
    await connectDB();
```

```
>  const newItem = new Product({ ...
  });

  const result = await newItem.save();
  console.log("Document saved successfully:", result);
} catch (error) {
  console.error("Error occurred:", error);
} finally {
  // Close the database connection
  mongoose.connection.close();
}
};
createDocument();
```

```
9
10  const newItem = new Product({
11   id: 31,
12   title: "iPhone 99",
13   description: "An apple mobile which is in space",
14   price: 54990,
15   discountPercentage: 88.96,
16   rating: 9.69,
17   stock: 9,
18   brand: "Apple",
19   category: "smartphones",
20   thumbnail: "https://i.dummyjson.com/data/products/1/thumbnail.jpg",
21   images: [
22     "https://i.dummyjson.com/data/products/1/1.jpg",
23     "https://i.dummyjson.com/data/products/1/2.jpg",
24     "https://i.dummyjson.com/data/products/1/3.jpg",
25     "https://i.dummyjson.com/data/products/1/4.jpg",
26     "https://i.dummyjson.com/data/products/1/thumbnail.jpg",
27   ],
28   });
29
```





```

1  import mongoose from "mongoose";
2
3  const productSchema = new mongoose.Schema({
4    id: Number,
5    title: String,
6    description: String,
7    price: Number,
8    discountPercentage: Number,
9    rating: Number,
10   stock: Number,
11   brand: String,
12   category: String,
13   thumbnail: String,
14   images: [String],
15 });
16
17 const Product = mongoose.model("Product", productSchema);
18
19 export default Product;
20

```

javascript

Copy code

```

const getProducts = async () => {
  try {
    // Connect to the database
    await connectDB();

    // Use the find() method to retrieve all documents from the Products collection
    // Sort the products by the "rating" field in descending order (-1 for descending)
    const Products = await Product.find().sort({ rating: -1 });

    // Log the retrieved Products
    console.log("Retrieved Products (sorted by rating in descending order):");
    console.log(Products);
  } catch (error) {
    console.error("Error occurred:", error);
  } finally {
    // Close the database connection
    mongoose.connection.close();
  }
};

// Call the getProducts function to retrieve and log the sorted Products
getProducts();

```

















