Table of Contents

1. Introduction to NoSQL

NoSQL stands for "Not only SQL". This means that NoSQL databases strive to break away from the traditional rigid structure of RDBMS and still accommodate the SQL-like query concepts.
These databases provide developers the ability to have a flexible data model that suits their application needs better. With the increase in apps that rely on real-time information and seek to customize the user experience, databases need to be able to grow and change on demand.

There are 4 types of NoSQL databases that can help manage the varying demands of modern apps.

- Graph databases : These databases store information in nodes. There can virtually be any number of relationships between any number of nodes. Graph databases are the best option to handle heavily interrelated information - social networks, for example.
- Column store databases : Data is stored in cells grouped in columns of data rather than as rows of data. Columns are logically grouped into column families. This makes for much faster read-write operations as compared to accessing rows in RDBMS. These databases are best suited for systems that have frequent information retrieval and storage - like blogging platforms.
- Key-value stores : Every instance of data is stored as a 'key' along with its value. The responsibility is on the application, rather than the database, to understand what was stored. This simple approach makes it faster to implement and use such a database to store information like shopping cart data.
- Document stores : Information is stored as a complex data structure known as a document. Documents can contain many different key-value pairs. This can be structured or semi-structured data. E-Commerce applications rely on such databases to maintain their product catalogs.

1.1 Characteristics of NoSQL

Now that you have seen why NoSQL databases are becoming a huge developer favorite, let's take a look at some important features that set these databases a class apart.

- Linear scalability

NoSQL databases don't follow the traditional master-slave architecture, instead it adopts a masterless, peer-to-peer architecture with all nodes being the same. This promotes faster, cheaper and easier scalability to handle the complexity and massive volume of data from modern apps. When new processors or servers are added, a consistent increase in performance and higher read/write speeds are observable.

- Easier querying and no schema restrictions

The lack of joins makes writing queries much simpler. The difference in query language is minimal. The flexible schema makes it easier to add and retrieve unstructured or semi-structured data. The development process can continue without having a schema design set in stone since these databases can easily adapt to new schema designs.

- Distributed systems

A characteristic feature of NoSQL databases is that they are distributed systems. This means the same data is stored across multiple servers or nodes.

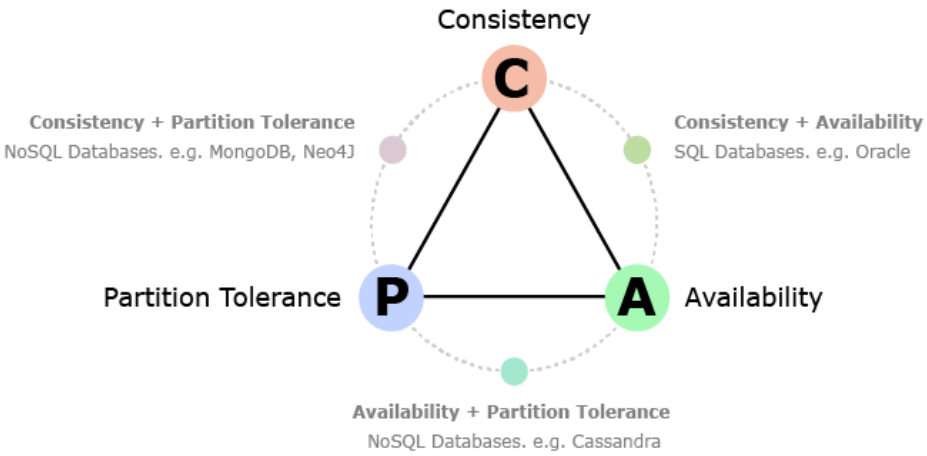
Distributed systems make information retrieval much faster. These systems are also more reliable if a server goes down, a user request can still be handled by remaining servers. This greatly minimizes downtime of an application.

1.2 Eric Brewer's CAP theorem

This theorem states that a distributed system cannot satisfy all three characteristics - Consistency, Availability, and Partition tolerance.

- Consistency - An end user must be able to see the latest data at all times.
- Availability - Every database request must be responded by the server.
- Partition tolerance - The database must continue to function even if there is a network partition. A network partition implies that systems in a network cannot communicate with each other.

NoSQL databases are partition tolerant. In case of a network partition, there is a trade-off between consistency and availability – some NoSQL systems give importance to consistency while others give importance to availability.



2. Introduction to MongoDB

- MongoDB is an open source document oriented database written in C++.
- Data in MongoDB is stored in key-value pairs in JSON format called as documents, which are roughly equivalent to rows in RDBMS.
- Multiple documents are stored in what is called a collection in MongoDB. A collection is roughly equivalent to a table in RDBMS.
- MongoDB is considered as schema-less as it can store documents having varying sets of fields with different types of each field.
- MongoDB stores these JSON documents in a binary-encoded format called BSON (Binary JSON) for faster storage and retrieval of data.

2.1 Features of MongoDB

- It supports flexible schema at the document level, that can vary from one document to another. It does not have a fixed structure defined at the collection level
- It has rich query language that supports CRUD operations, aggregation, text search and so on
- As documents can be embedded within each other, it does not need complex joins.
- It also provides support for indexes for faster querying

2.2 Advantages of MongoDB

MongoDB offers solutions to several challenges presented by the traditional RDBMS approach.

Given below are a few more advantages of using MongoDB over traditional RDBMS:

- Table-less flexibility: MongoDB is a non-relational database, hence, very different from SQL databases. This means the databases are easier to manage and provide a higher level of flexibility to accommodate newer data models.
- No need to develop a detailed database model: The non-relational nature of MongoDB allows database architects to quickly create a database without needing to develop a detailed (fine-grained) database model. This saves a lot of development time.
- Dynamic Schema: It gives you the flexibility to evolve your data schema without affecting the existing data
- Speed: All the related data are in individual documents, this eliminates the need for join operations and improves the speed and performance of queries.
- Scalability: It is horizontally scalable, i.e. you can easily add more servers to handle the load as data keeps increasing. RDBMS requires more hardware with increased processing power to handle increase in data.

- Manageable: The database doesn't require a database administrator. Since it is fairly user-friendly, it can be used by both developers and administrators without any special training to use and maintain.
- Sharding: MongoDB enables better performance for complex operations by distributing large data sets across multiple machines.

2.3 MongoDB v/s RDBMS

MongoDB	RDBMS
Non-relational document oriented database	Relational table oriented database
Supports JSON-like query language	Supports SQL query language
Collection based	Table based
Document based	Row based
Field based, uses key-value pairs	Column based
Schema is dynamic and flexible	Schema is predefined
Emphasis on CAP theorem	Emphasis on ACID properties
No need for JOINS	Supports JOINS across multiple tables
Better performance due to simpler queries	Complex queries cause slow performance

2.4 Data Types

MongoDB Server stores data using the BSON format which supports some additional data types that are not available using the JSON format. Compared to the legacy mongo shell, MongoDB Shell (mongosh) has type handling which is better aligned with the default types used by the MongoDB drivers.

This document highlights changes in type usage between mongosh and the legacy mongo shell. See the Extended JSON reference for additional information on supported types.

2.4.1 Date

mongosh provides various methods to return the date, either as a string or as a Date object:

- Date() method which returns the current date as a string.
- new Date() constructor which returns a Date object using the ISODate() wrapper.
- ISODate() constructor which returns a Date object using the ISODate() wrapper.

2.4.2 ObjectId

mongosh provides the ObjectId() wrapper class around the ObjectId data type. To generate a new ObjectId, use the following operation in mongosh:

- new ObjectId

2.4.3 Int32

If a number can be converted to a 32-bit integer, mongosh will store it as Int32. If not, mongosh defaults to storing the number as a Double. Numerical values that are stored as Int32 in mongosh would have been stored by default as Double in the mongo shell.

The Int32() constructor can be used to explicitly specify 32-bit integers.

```
db.types.insertOne(
{
"_id": 1,
"value": Int32("1"),
"expectedType": "Int32"
}
)
```

WARNING

Default Int32 and Double types may be stored inconsistently if you connect to the same collection using both mongosh and the legacy mongo shell.

2.4.4 Long

The Long() constructor can be used to explicitly specify a 64-bit integer.

```
db.types.insertOne(
{
"_id": 3,
"value": Long("1"),
"expectedType": "Long"
}
)
```

NOTE

In the legacy mongo shell NumberLong() accepted either a string or integer value. In mongosh, NumberLong() only accepts string values. Long() provides methods to manage conversions to and from 64-bit values.

2.4.5 Decimal128

Decimal128() values are 128-bit decimal-based floating-point numbers that emulate decimal rounding with exact precision.

This functionality is intended for applications that handle monetary data, such as financial, tax, and scientific computations.

The Decimal128 BSON type uses the IEEE 754 decimal128 floating-point numbering format which supports 34 decimal digits (i.e. significant digits) and an exponent range of −6143 to +6144.

```
db.types.insertOne(
{
"_id": 5,
"value": Decimal128("1"),
"expectedType": "Decimal128"
}
)
```

NOTE

To use the Decimal128 data type with a MongoDB driver, be sure to use a driver version that supports it.

Equality and Sort Order

Values of the Decimal128 type are compared and sorted with other numeric types based on their actual numeric value. Numeric values of the binary-based Double type generally have approximate representations of decimal-based values and may not be exactly equal to their decimal representations.

2.4.6 Timestamp

MongoDB uses a BSON Timestamp internally in the oplog. The Timestamp type works similarly to the Java Timestamp type. Use the Date type for operations involving dates.

A Timestamp signature has two optional parameters.

```
Timestamp( { "t": <integer>, "i": <integer> } )
```

Parameter, Type, Default, Definition

t, integer, Current time since UNIX epoch., Optional. A time in seconds.

i, integer, 1, Optional. Used for ordering when there are multiple operations within a given second. i has no effect if used without t.

For usage examples, see Timestamp a New Document, Create a Custom Timestamp.

2.4.7 Type Checking

Use the $type query operator or examine the object constructor to determine types.

The Javascript typeof operator returns generic values such as number or object rather than the more specific Int32 or ObjectId.

Type	Number	Alias
Double	1	"double"
String	2	"string"
Object	3	"object"
Array	4	"array"
Binary Data	5	"binData"
ObjectId	7	"objectId"
Boolean	8	"bool"
Date	9	"date"
Null	10	"null"
Regular Expression	11	"regex"
32-bit integer	16	"int"

2.5 Building blocks of MongoDB

The basic building blocks of MongoDB are:\

- Database

A MongoDB database contains all the data generated by the system as collections.

- Collection

Collection is a single entity that contains all the data related to a particular aspect.

Note: You cannot create multiple collections with the same name. Trying to do so throws an error as shown below:

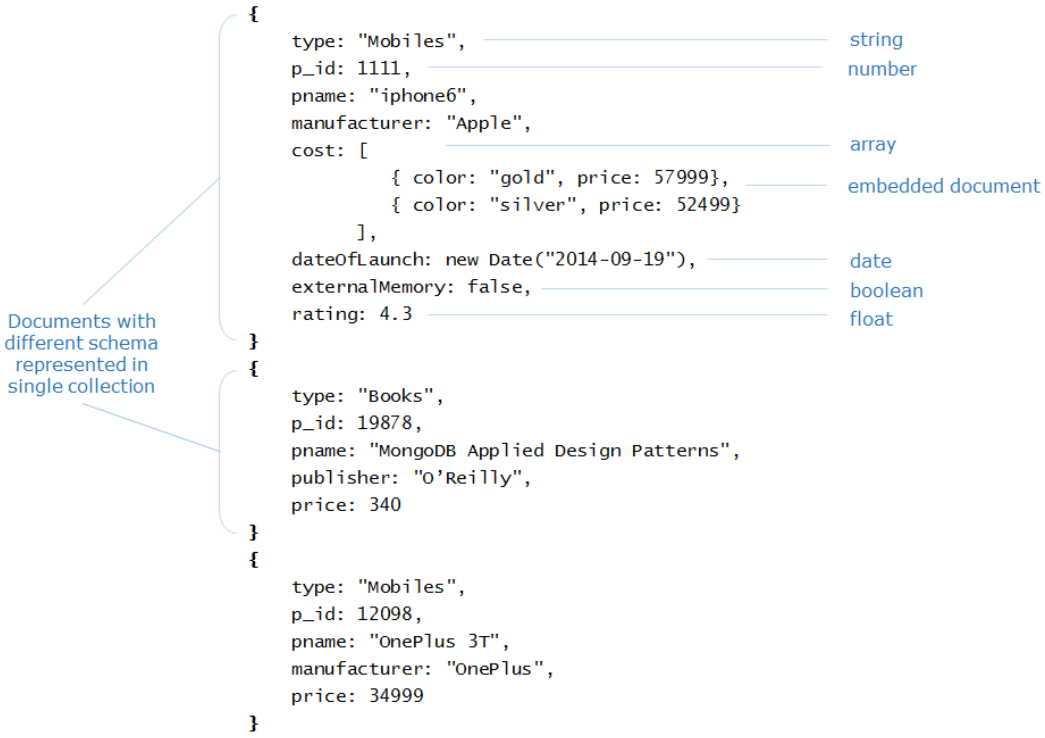
```
"<"ok":0, 'errmsg':"a collection 'databaseName.collectionName' already exists">;"
```

- Document

Data is stored as documents which are represented as a set of fields with associated values of corresponding datatypes.

2.6 Datatypes in MongoDB

The datatypes supported in MongoDB are string, number, array, boolean, date, float and so on.



3. CRUD Operations

3.1 Create operation

3.1.1 Creating a database

```
use databasename
```

3.1.2 Creating a collection implicitly

```
db.createCollection(name,options)
```

3.1.3 Dropping a collection and database

In case of any mistakes while creating the database or collection, you can drop it.

To drop a collection, replace collection_name in the below code with the actual name of the collection to be dropped. The drop() method returns true if the collection specified is successfully dropped.

```
db.collection_name.drop()
```

To drop the database currently in use, execute the below command.

```
db.dropDatabase()
```

3.1.4 Inserting one document at a time

The basic syntax of the insertOne() method is as shown below

Syntax:

```
db.collection_name.insertOne({document})
```

If a collection does not exist, the insertOne operation will create one.

3.1.5 Inserting multiple documents at a time

The db.collection_name.insertMany() method can also be used to insert multiple documents at a time.

To insert multiple documents into a collection, we need to slightly modify the syntax as shown

Observe the minor changes in the syntax here. We have to enclose the comma separated documents inside square brackets or [ ] (see line 4 and 7)

Syntax:

```
db.collection_name.insertMany(
[
{document1},
{document2}
]
)
```

If a collection does not exist, the insertMany operation will create one.

3.2 Read operation

3.2.1 find & findOne

We use the find() method to retrieve data from a collection.
This method returns all the documents in a collection which satisfies the specified query criteria on the collection.

```
db.collection.find(
{ <<query criteria>> },
{ <<projection criteria>> }
)
```

Returns one document that satisfies the specified query criteria on the collection

```
db.collection.findOne(
{ <<query criteria>> },
{ <<projection criteria>> }
)
```

3.2.1.1 Query

The first parameter, or the first document, is for the query. This document will specify the selection criteria, i.e, it specifies some kind of filter or restriction on the value of the key.

Observe the document given below

```
{
  prodid:7000013,
prodname:"Big Data: Principles and Best Practices",
publisher:"Dreamtech",
price:700
}
```

If you want to retrieve only documents that have 'Dreamtech' as the publisher, your query would be written as shown below

```
db.product_catalog.find( { publisher: "Dreamtech" } )
```

3.2.1.2 Projection

The second parameter, or the second document is for the projection. This parameter specifies which fields are to be returned in the documents that match the query selection criteria.

The projection parameter document is usually written as

```
{ field1: <value>, field2: <value> ... }
```

The <value> is always written as 0 or 1.

- 0 means omit this field
- 1 means include this field.

To continue our above example, if we want to display only the prodname and price of books with 'Dreamtech' as the publisher, the query will be modified as shown

```
db.product_catalog.find( { publisher: "Dreamtech" }, { prodname: 1, price: 1 } )
```

You can see that the output of the above query still contains an _id field, even though the column is not mentioned in the projection.

Each document stored in a collection has a unique _id value that acts as a primary key. If an inserted document omits the _id field, then MongoDB automatically generates a unique id and sets it to the _id field.

The long value that you see for the _id field is an auto-generated value.

If you want to avoid displaying the _id field, you can modify your query as shown

```
db.product_catalog.find({ }, {_id: 0}).pretty()
```

3.2.2 Relational Operators

3.2.2.1 $eq

Specifies equality condition. The $eq operator matches documents where the value of a field equals the specified value.

```
{ <field>: { $eq: <value> } }
```

Example:

```
{ _id: 1, item: { name: "ab", code: "123" }, qty: 15, tags: [ "A", "B", "C" ] }
{ _id: 2, item: { name: "cd", code: "123" }, qty: 20, tags: [ "B" ] }
{ _id: 3, item: { name: "ij", code: "456" }, qty: 25, tags: [ "A", "B" ] }
{ _id: 4, item: { name: "xy", code: "456" }, qty: 30, tags: [ "B", "A" ] }
{ _id: 5, item: { name: "mn", code: "000" }, qty: 20, tags: [ [ "A", "B" ], "C" ] }
```

```
db.inventory.find( { qty: { $eq: 20 } } )
```

```
{ _id: 2, item: { name: "cd", code: "123" }, qty: 20, tags: [ "B" ] }
{ _id: 5, item: { name: "mn", code: "000" }, qty: 20, tags: [ [ "A", "B" ], "C" ] }
```

3.2.2.2 $gt

$gt selects those documents where the value of the field is greater than (i.e. >) the specified value.

```
{ field: { $gt: value } }
```

Example:

```
[
{
"item": "nuts", "quantity": 30,
"carrier": { "name": "Shipit", "fee": 3 }
},
{
"item": "bolts", "quantity": 50,
```

```
"carrier": { "name": "Shipit", "fee": 4 }
},
{
"item": "washers", "quantity": 10,
"carrier": { "name": "Shipit", "fee": 1 }
}
]
```

```
db.inventory.find( { quantity: { $gt: 20 } } )
```

```
{
_id: ObjectId("61ba25cbfe687fce2f042414"),
item: 'nuts',
quantity: 30,
carrier: { name: 'Shipit', fee: 3 }
},
{
_id: ObjectId("61ba25cbfe687fce2f042415"),
item: 'bolts',
quantity: 50,
carrier: { name: 'Shipit', fee: 4 }
}
```

3.2.2.3 $gte

$gte selects the documents where the value of the field is greater than or equal to (i.e. >=) a specified value (e.g. value.)

```
{ field: { $gte: value } }
```

Example:

```
[
{
"item": "nuts", "quantity": 20,
"carrier": { "name": "Shipit", "fee": 3 }
},
{
"item": "bolts", "quantity": 50,
"carrier": { "name": "Shipit", "fee": 4 }
},
{
"item": "washers", "quantity": 10,
```

```
"carrier": { "name": "Shipit", "fee": 1 }
}
]
```

```
db.inventory.find( { quantity: { $gte: 20 } } )
```

```
{
_id: ObjectId("61bb51211b83c864e3bbe037"),
item: 'nuts',
quantity: 20,
carrier: { name: 'Shipit', fee: 3 }
},
{
_id: ObjectId("61bb51211b83c864e3bbe038"),
item: 'bolts',
quantity: 50,
carrier: { name: 'Shipit', fee: 4 }
}
```

3.2.2.4 $lt

$lt selects the documents where the value of the field is less than (i.e. <) the specified value.

```
{ field: { $lt: value } }
```

Example:

```
[
{
"item": "nuts", "quantity": 30,
"carrier": { "name": "Shipit", "fee": 3 }
},
{
"item": "bolts", "quantity": 50,
"carrier": { "name": "Shipit", "fee": 4 }
},
{
"item": "washers", "quantity": 10,
"carrier": { "name": "Shipit", "fee": 1 }
}
]
```

```
db.inventory.find( { quantity: { $lt: 20 } } )
```

```
{
  _id: ObjectId("61ba634dfe687fce2f04241f"),
  item: 'washers',
  quantity: 10,
  carrier: { name: 'Shipit', fee: 1 }
}
```

3.2.2.5 $lte

$lte selects the documents where the value of the field is less than or equal to (i.e. <=) the specified value.

```
{ field: { $lte: value } }
```

```
[
{
"item": "nuts", "quantity": 30,
"carrier": { "name": "Shipit", "fee": 3 }
},
{
"item": "bolts", "quantity": 50,
"carrier": { "name": "Shipit", "fee": 4 }
},
{
"item": "washers", "quantity": 10,
"carrier": { "name": "Shipit", "fee": 1 }
}
]
```

```
db.inventory.find( { quantity: { $lte: 10 } } )
```

```
{
_id: ObjectId("61ba453ffe687fce2f04241c"),
item: 'washers',
quantity: 10,
carrier: { name: 'Shipit', fee: 1 }
}
```

3.2.2.6 $ne

$ne selects the documents where the value of the field is not equal to the specified value. This includes documents that do not contain the field.

```
{ field: { $ne: value } }
```

```
[
{
"item": "nuts", "quantity": 30,
"carrier": { "name": "Shipit", "fee": 3 }
},
{
"item": "bolts", "quantity": 50,
"carrier": { "name": "Shipit", "fee": 4 }
},
{
"item": "washers", "quantity": 10,
"carrier": { "name": "Shipit", "fee": 1 }
}
]
```

```
db.inventory.find( { quantity: { $ne: 20 } } )
```

```
{
  _id: ObjectId("61ba667dfe687fce2f042420"),
  item: 'nuts',
  quantity: 30,
  carrier: { name: 'Shipit', fee: 3 }
},
{
  _id: ObjectId("61ba667dfe687fce2f042421"),
  item: 'bolts',
  quantity: 50,
  carrier: { name: 'Shipit', fee: 4 }
},
{
  _id: ObjectId("61ba667dfe687fce2f042422"),
  item: 'washers',
  quantity: 10,
  carrier: { name: 'Shipit', fee: 1 }
```

```
}
```

3.2.2.7 $in

The $in operator selects the documents where the value of a field equals any value in the specified array. To specify an $in expression, use the following prototype:

```
{ field: { $in: [<value1>, <value2>, ... <valueN> ] } }
```

```
[
{ "item": "Pens", "quantity": 350, "tags": [ "school", "office" ] },
{ "item": "Erasers", "quantity": 15, "tags": [ "school", "home" ] },
{ "item": "Maps", "tags": [ "office", "storage" ] },
{ "item": "Books", "quantity": 5, "tags": [ "school", "storage", "home" ] }
]
```

```
db.inventory.find( { quantity: { $in: [ 5, 15 ] } }, { _id: 0 } )
```

```
{ item: 'Erasers', quantity: 15, tags: [ 'school', 'home' ] },
{ item: 'Books', quantity: 5, tags: [ 'school', 'storage', 'home' ] }
```

3.2.2.8 $nin

$nin selects the documents where:

```
{ field: { $nin: [ <value1>, <value2> ... <valueN> ] } }
```

- the field value is not in the specified array or
- the field does not exist.

```
[{ "item": "Pens", "quantity": 350, "tags": [ "school", "office" ] },
{ "item": "Erasers", "quantity": 15, "tags": [ "school", "home" ] },
```

```
{ "item": "Maps", "tags": [ "office", "storage" ] },
{ "item": "Books", "quantity": 5, "tags": [ "school", "storage", "home" ] }
]
```

```
db.inventory.find( { quantity: { $nin: [ 5, 15 ] } }, { _id: 0 } )
```

```
{ item: 'Pens', quantity: 350, tags: [ 'school', 'office' ] },
{ item: 'Maps', tags: [ 'office', 'storage' ] }
```

3.2.3 Logical Operators

3.2.3.1 $and

$and performs a logical AND operation on an array of one or more expressions (<expression1>, <expression2>, and so on) and selects the documents that satisfy all the expressions.

```
{ $and: [ { <expression1> }, { <expression2> } , ... , { <expressionN> } ] }
```

NOTE:

MongoDB provides an implicit AND operation when specifying a comma separated list of expressions.

```
db.inventory.find( { $and: [ { price: { $ne: 1.99 } }, { price: { $exists: true } } ] } )
```

3.2.3.2 $or

The $or operator performs a logical OR operation on an array of one or more <expressions> and selects the documents that satisfy at least one of the <expressions>. The $or has the following syntax:

```
{ $or: [ { <expression1> }, { <expression2> }, ... , { <expressionN> } ] }
```

```
db.inventory.find( { $or: [ { quantity: { $lt: 20 } }, { price: 10 } ] } )
```

3.2.3.3 $nor

$nor performs a logical NOR operation on an array of one or more query expression and selects the documents that fail all the query expressions in the array. The $nor has the following syntax:

```
{ $nor: [ { <expression1> }, { <expression2> }, ... { <expressionN> } ] }
```

```
db.inventory.find( { $nor: [ { price: 1.99 }, { sale: true } ] } )
```

3.2.3.4 $not

$not performs a logical NOT operation on the specified <operator-expression> and selects the documents that do not match the <operator-expression>. This includes documents that do not contain the field.

```
{ field: { $not: { <operator-expression> } } }
```

```
db.inventory.find( { price: { $not: { $gt: 1.99 } } } )
```

3.2.4 Element Operators

3.2.4.1 $exists

When <boolean> is true, $exists matches the documents that contain the field, including documents where the field value is null. If <boolean> is false, the query returns only the documents that do not contain the field.

```
{ field: { $exists: <boolean> } }
```

```
db.inventory.find( { qty: { $exists: true, $nin: [ 5, 15 ] } } )
```

3.2.4.2 $type

$type selects documents where the value of the field is an instance of the specified BSON type(s). Querying by data type is useful when dealing with highly unstructured data where data types are not predictable.

A $type expression for a single BSON type has the following syntax:

```
{ field: { $type: <BSON type> } }
{ field: { $type: [ <BSON type1> , <BSON type2>, ... ] } }
```

3.2.4.3 $regex

Provides regular expression capabilities for pattern matching strings in queries. MongoDB uses Perl compatible regular expressions (i.e. "PCRE") version 8.42 with UTF-8 support.

To use $regex, use one of the following syntaxes:

```
{ <field>: { $regex: /pattern/, $options: '<options>' } }
{ <field>: { $regex: 'pattern', $options: '<options>' } }
{ <field>: { $regex: /pattern/<options> } }
```

The $options clause takes the following as parameters:

- i – Performs a case insensitive match
- m – Performs pattern match that consists of anchors i.e. ^ for the beginning, $ for the end
- x – Ignores all white space characters in the pattern
- s – Allows dot character (.) to match all characters

Search for the products whose prodname starts with iphone:

```
db.product_catalog.find(
{ prodname:{ $regex:/^iphone/i } }
)
```

3.2.4.4 $expr

Allows the use of aggregation expressions within the query language.

$expr has the following syntax:

```
{ $expr: { <expression> } }
```

```
{ "_id" : 1, "category" : "food", "budget": 400, "spent": 450 }
{ "_id" : 2, "category" : "drinks", "budget": 100, "spent": 150 }
{ "_id" : 3, "category" : "clothes", "budget": 100, "spent": 50 }
{ "_id" : 4, "category" : "misc", "budget": 500, "spent": 300 }
{ "_id" : 5, "category" : "travel", "budget": 200, "spent": 650 }
```

The following operation uses $expr to find documents where the spent amount exceeds the budget:

```
db.monthlyBudget.find( { $expr: { $gt: [ "$spent" , "$budget" ] } }
```

```
{ "_id" : 1, "category" : "food", "budget" : 400, "spent" : 450 }
{ "_id" : 2, "category" : "drinks", "budget" : 100, "spent" : 150 }
{ "_id" : 5, "category" : "travel", "budget" : 200, "spent" : 650 }
```

```
let discountedPrice = {
$cond: {
if: { $gte: ["$qty", 100] },
then: { $multiply: ["$price", NumberDecimal("0.50")] },
else: { $multiply: ["$price", NumberDecimal("0.75")] }
}
};
```

```
// Query the supplies collection using the aggregation expression
db.supplies.find( { $expr: { $lt:[ discountedPrice, NumberDecimal("5") ] } });
```

Document	Discounted Price	< NumberDecimal("5")
{"_id": 1, "item": "binder", "qty": 100, "price": NumberDecimal("12") }	NumberDecimal("6.00")	false
{"_id": 2, "item": "noteboook", "qty": 200, "price": NumberDecimal("8") }	NumberDecimal("4.00")	true
{"_id": 3, "item": "pencil", "qty": 50, "price": NumberDecimal("6") }	NumberDecimal("4.50")	true
{"_id": 4, "item": "eraser", "qty": 150, "price": NumberDecimal("3") }	NumberDecimal("1.50")	true
{"_id": 5, "item": "legal pad", "qty": 42, "price": NumberDecimal("10") }	NumberDecimal("7.50")	

3.2.5 Array Operators

3.2.5.1 $all

The $all operator selects the documents where the value of a field is an array that contains all the specified elements. To specify an $all expression, use the following prototype:

```
{ <field>: { $all: [ <value1> , <value2> ... ] } }
```

```
{
_id: ObjectId("5234cc89687ea597eabee675"),
code: "xyz",
tags: [ "school", "book", "bag", "headphone", "appliance" ],
qty: [
{ size: "S", num: 10, color: "blue" },
{ size: "M", num: 45, color: "blue" },
{ size: "L", num: 100, color: "green" }
]
}

{
_id: ObjectId("5234cc8a687ea597eabee676"),
code: "abc",
tags: [ "appliance", "school", "book" ],
qty: [
```

```
{ size: "6", num: 100, color: "green" },
{ size: "6", num: 50, color: "blue" },
{ size: "8", num: 100, color: "brown" }
]
}

{
_id: ObjectId("5234ccb7687ea597eabee677"),
code: "efg",
tags: [ "school", "book" ],
qty: [
{ size: "S", num: 10, color: "blue" },
{ size: "M", num: 100, color: "blue" },
{ size: "L", num: 100, color: "green" }
]
}

{
_id: ObjectId("52350353b2eff1353b349de9"),
code: "ijk",
tags: [ "electronics", "school" ],
qty: [
{ size: "M", num: 100, color: "green" }
]
}
```

```
db.inventory.find( { tags: { $all: [ "appliance", "school", "book" ] } } )
```

```
{
_id: ObjectId("5234cc89687ea597eabee675"),
code: "xyz",
tags: [ "school", "book", "bag", "headphone", "appliance" ],
qty: [
{ size: "S", num: 10, color: "blue" },
{ size: "M", num: 45, color: "blue" },
{ size: "L", num: 100, color: "green" }
]
}

{
_id: ObjectId("5234cc8a687ea597eabee676"),
code: "abc",
tags: [ "appliance", "school", "book" ],
qty: [
```

```
{ size: "6", num: 100, color: "green" },
{ size: "6", num: 50, color: "blue" },
{ size: "8", num: 100, color: "brown" }
]
}
```

3.2.5.2 $elemMatch

The $elemMatch operator matches documents that contain an array field with at least one element that matches all the specified query criteria.

```
{ <field>: { $elemMatch: { <query1>, <query2>, ... } } }
```

```
[
{ "_id": 1, "results": [ { "product": "abc", "score": 10 },
{ "product": "xyz", "score": 5 } ] },
{ "_id": 2, "results": [ { "product": "abc", "score": 8 },
{ "product": "xyz", "score": 7 } ] },
{ "_id": 3, "results": [ { "product": "abc", "score": 7 },
{ "product": "xyz", "score": 8 } ] },
{ "_id": 4, "results": [ { "product": "abc", "score": 7 },
{ "product": "def", "score": 8 } ] }
]
```

```
db.survey.find(
{ results: { $elemMatch: { product: "xyz", score: { $gte: 8 } } } }
)
```

```
{ "_id" : 3, "results" : [ { "product" : "abc", "score" : 7 },
{ "product" : "xyz", "score" : 8 } ] }
```

3.2.5.3 $size

The $size operator matches any array with the number of elements specified by the argument. For example:

```
db.collection.find( { field: { $size: 2 } } );
```

returns all documents in collection where field is an array with 2 elements.

For instance, the above expression will return { field: [red, green] } and { field: [apple, lime] } but not { field: fruit } or { field: [orange, lemon, grapefruit] }.

3.2.6 Projection operators

3.2.6.1 $(projection)

The positional $ operator limits the contents of an <array> to return the first element that matches the query condition on the array.

Use $ in the projection document of the find() method or the findOne() method when you only need one particular array element in selected documents.

```
{ "_id" : 7, semester: 3, "grades" : [ { grade: 80, mean: 75, std: 8 },
{ grade: 85, mean: 90, std: 5 },
{ grade: 90, mean: 85, std: 3 } ] }

{ "_id" : 8, semester: 3, "grades" : [ { grade: 92, mean: 88, std: 8 },
{ grade: 78, mean: 90, std: 5 },
{ grade: 88, mean: 85, std: 3 } ] }
```

```
db.students.find(
{ "grades.mean": { $gt: 70 } },
{ "grades.$": 1 }
)
```

```
{ "_id" : 7, "grades" : [  {  "grade" : 80,  "mean" : 75,  "std" : 8 } ] }
{ "_id" : 8, "grades" : [  {  "grade" : 92,  "mean" : 88,  "std" : 8 } ] }
```

3.2.6.2 $elemMatch(projection)

The $elemMatch operator limits the contents of an <array> field from the query results to contain only the first element matching the $elemMatch condition.

```
{
_id: 1,
zipcode: "63109",
students: [
{ name: "john", school: 102, age: 10 },
{ name: "jess", school: 102, age: 11 },
{ name: "jeff", school: 108, age: 15 }
]
}
{
_id: 2,
zipcode: "63110",
students: [
{ name: "ajax", school: 100, age: 7 },
{ name: "achilles", school: 100, age: 8 },
]
}
{
_id: 3,
zipcode: "63109",
students: [
{ name: "ajax", school: 100, age: 7 },
{ name: "achilles", school: 100, age: 8 },
]
}
{
_id: 4,
zipcode: "63109",
students: [
{ name: "barney", school: 102, age: 7 },
{ name: "ruth", school: 102, age: 16 },
]
}
```

```
db.schools.find( { zipcode: "63109" },
                 { students: { $elemMatch: { school: 102, age: { $gt: 10} } } } )
```

```
{ "_id" : 1, "students" : [ { "name" : "jess", "school" : 102, "age" : 11 } ] }
{ "_id" : 3 }
```

```
{ "_id" : 4, "students" : [ { "name" : "ruth", "school" : 102, "age" : 16 } ] }
```

3.2.6.3 $slice

The $slice projection operator specifies the number of elements in an array to return in the query result. The $slice has one of the following syntax forms:

```
db.collection.find(
<query>,
{ <arrayField>: { $slice: <number> } }
);
or
db.collection.find(
<query>,
{ <arrayField>: { $slice: [ <number>, <number> ] } }
);
```

3.3 Update operation

3.3.1 Updating multiple documents

Updates all documents that match the specified filter for a collection.

```
db.collection.updateMany(filter, update, options)
```

```
db.collection.updateMany(
<filter>,
<update>,
{
upsert: <boolean>,
writeConcern: <document>,
collation: <document>,
arrayFilters: [ <filterdocument1>, ... ],
hint: <document|string> // Available starting in MongoDB 4.2.1
}
)
```

3.3.2 Updating single document

Updates a single document within the collection based on the filter.

```
db.collection.updateOne(filter, update, options)
```

```
db.collection.updateOne(
<filter>,
<update>,
{
upsert: <boolean>,
writeConcern: <document>,
collation: <document>,
arrayFilters: [ <filterdocument1>, ... ],
hint: <document|string> // Available starting in MongoDB 4.2.1
})
```

3.3.3 Replace Single Document

Replaces a single document within the collection based on the filter.

```
db.collection.replaceOne(filter, replacement, options)
```

```
db.collection.replaceOne(
<filter>,
<replacement>,
{
upsert: <boolean>,
writeConcern: <document>,
collation: <document>,
hint: <document|string> // Available starting in 4.2.1
}
)
```

3.3.4 Upsert

In MongoDB, if you try to update a document that does not exist, a new document is inserted with just the fields specified.

Check the query given below as an example.

```
db.product_catalog.updateMany(
    { "price" : { $gt : 80000 }, "manufacturer" : "apple" },
    { $set: { "prodname" : "iphone 7 plus" } },
    { upsert: true }
)
```

Observe line 4, we have set the attribute upsert to true. This allows a new document to be inserted if there are no documents to be updated that match the criteria specified.

In the above query, there are no products matching the criteria of price and manufacturer. Thus, a new product is inserted with fields manufacturer and prodname.

3.3.5 Field Update Operators

3.3.5.1 $set

The $set operator replaces the value of a field with the specified value.

The $set operator expression has the following form:

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

To specify a <field> in an embedded document or in an array, use dot notation.

```
db.products.insertOne(
{
_id: 100,
quantity: 250,
instock: true,
reorder: false,
details: { model: "14QQ", make: "Clothes Corp" },
tags: [ "apparel", "clothing" ],
ratings: [ { by: "Customer007", rating: 4 } ]
}
)
```

```
db.products.updateOne(
{ _id: 100 },
{ $set:
{
quantity: 500,
details: { model: "2600", make: "Fashionaires" },
tags: [ "coats", "outerwear", "clothing" ]
}
}
)
```

```
{
_id: 100,
quantity: 500,
instock: true,
reorder: false,
details: { model: '2600', make: 'Fashionaires' },
tags: [ 'coats', 'outerwear', 'clothing' ],
ratings: [ { by: 'Customer007', rating: 4 } ]
}
```

```
db.products.updateOne(
{ _id: 100 },
{ $set:
{
"tags.1": "rain gear",
"ratings.0.rating": 2
}
}
)
```

```
{
_id: 100,
quantity: 500,
instock: true,
reorder: false,
details: { model: '2600', make: 'Kustom Kidz' },
tags: [ 'coats', 'rain gear', 'clothing' ],
ratings: [ { by: 'Customer007', rating: 2 } ]
}
```

3.3.5.2 $unset

The $unset operator deletes a particular field. Consider the following syntax:

```
{ $unset: { <field1>: "", ... } }
```

To specify a <field> in an embedded document or in an array, use dot notation.

```
db.products.insertMany( [
{ "item": "chisel", "sku": "C001", "quantity": 4, "instock": true },
{ "item": "hammer", "sku": "unknown", "quantity": 3, "instock": true },
{ "item": "nails", "sku": "unknown", "quantity": 100, "instock": true }
] )
```

```
db.products.updateOne(
{ sku: "unknown" },
{ $unset: { quantity: "", instock: "" } }
)
```

```
{
item: 'chisel',
sku: 'C001',
quantity: 4,
instock: true
},
{
item: 'hammer',
sku: 'unknown'
},
{
item: 'nails',
sku: 'unknown',
quantity: 100,
instock: true
}
```

3.3.5.3 $setOnInsert

If an update operation with upsert: true results in an insert of a document, then $setOnInsert assigns the specified values to the fields in the document. If the update operation does not result in an insert, $setOnInsert does nothing.

```
db.collection.updateOne(

<query>,

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

```
{ upsert: true }

)
```

```
db.products.updateOne(
{ _id: 1 },
{
$set: { item: "apple" },
$setOnInsert: { defaultQty: 100 }
},
{ upsert: true }
)
```

```
{ "_id" : 1, "item" : "apple", "defaultQty" : 100 }
```

When the upsert parameter is true db.collection.updateOne()

- creates a new document
- applies the $set operation
- applies the $setOnInsert operation

3.3.5.4 $rename

The $rename operator updates the name of a field and has the following form:

```
{$rename: { <field1>: <newName1>, <field2>: <newName2>, ... } }
```

The new field name must differ from the existing field name. To specify a <field> in an embedded document, use dot notation.

```
db.students.updateMany( {}, { $rename: { "nmae": "name" } } )

db.students.updateOne( { _id: 1 }, { $rename: { "name.first": "name.fname" } } )
```

3.3.5.5 $inc

The $inc operator increments a field by a specified value and has the following form:

```
{ $inc: { <field1>: <amount1>, <field2>: <amount2>, ... } }
```

To specify a <field> in an embedded document or in an array, use dot notation.

The $inc operator accepts positive and negative values.

```
db.products.insertOne(
{
_id: 1,
sku: "abc123",
quantity: 10,
metrics: { orders: 2, ratings: 3.5 }
}
)
```

```
db.products.updateOne(
{ sku: "abc123" },
{ $inc: { quantity: -2, "metrics.orders": 1 } }
)
```

```
{
_id: 1,
sku: 'abc123',
quantity: 8,
metrics: { orders: 3, ratings: 3.5 }
}
```

3.3.5.6 $mul

Multiply the value of a field by a number. To specify a $mul expression, use the following prototype:

```
{ $mul: { <field1>: <number1>, ... } }
```

The field to update must contain a numeric value.

To specify a <field> in an embedded document or in an array, use dot notation.

```
db.products.insertOne(
   { "_id" : 1, "item" : "Hats", "price" : Decimal128("10.99"), "quantity" : 25 }
)
```

```
db.products.updateOne(
   { _id: 1 },
   { $mul:
      {
        price: Decimal128( "1.25" ),
```

```
        quantity: 2
      }
   }
)
```

```
{ _id: 1, item: 'Hats', price: Decimal128("13.7375"), quantity: 50 }
```

3.3.5.7 $min

The $min updates the value of the field to a specified value if the specified value is less than the current value of the field.

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

To specify a <field> in an embedded document or in an array, use dot notation.

The lowScore for the document currently has the value 200. The following operation uses $min to compare 200 to the specified value 150 and updates the value of lowScore to 150 since 150 is less than 200:

```
db.scores.updateOne( { _id: 1 }, { $min: { lowScore: 150 } } )
```

The scores collection now contains the following modified document:

```
{ _id: 1, highScore: 800, lowScore: 150 }
```

3.3.5.8 $max

The $max operator updates the value of the field to a specified value if the specified value is greater than the current value of the field.

The $max operator expression has the form:

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

To specify a <field> in an embedded document or in an array, use dot notation.

The highScore for the document currently has the value 800. The following operation:

- Compares the highscore, 800, to the specified value, 950
- Updates highScore to 950 since 950 is greater than 800

```
db.scores.updateOne( { _id: 1 }, { $max: { highScore: 950 } } )
```

The scores collection now contains the following modified document:

```
{ _id: 1, highScore: 950, lowScore: 200 }
```

3.3.6 Array Update Operators

3.3.6.1 $push

The $push operator appends a specified value to an array.

The $push operator has the form:

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

To specify a <field> in an embedded document or in an array, use dot notation.

```
db.students.insertOne( { _id: 1, scores: [ 44, 78, 38, 80 ] } )
```

```
db.students.updateOne(
   { _id: 1 },
   { $push: { scores: 89 } }
)
```

```
{ _id: 1, scores: [ 44, 78, 38, 80, 89 ] }
```

3.3.6.2 $addToSet

The $addToSet operator adds a value to an array unless the value is already present, in which case $addToSet does nothing to that array.

The $addToSet operator has the form:

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

To specify a <field> in an embedded document or in an array, use dot notation.

$addToSet only ensures that there are no duplicate items added to the set and does not affect existing duplicate elements.

$addToSet does not guarantee a particular ordering of elements in the modified set.

3.3.6.3 $each

The $each modifier is available for use with the $addToSet operator and the $push operator.

Use with the $addToSet operator to add multiple values to an array <field> if the values do not exist in the <field>.

Use with the $addToSet operator to add multiple values to an array <field> if the values do not exist in the <field>.

```
{ $addToSet: { <field>: { $each: [ <value1>, <value2> ... ] } } }
```

Use with the $push operator to append multiple values to an array <field>.

```
{ $push: { <field>: { $each: [ <value1>, <value2> ... ] } } }
```

The following example appends each element of [90, 92, 85] to the scores array for the document where the name field equals joe:

```
db.students.updateOne(
{ name: "joe" },
{ $push: { scores: { $each: [ 90, 92, 85 ] } } }
)
```

3.3.6.4 $sort

The $sort modifier orders the elements of an array during a $push operation.

To use the $sort modifier, it must appear with the $each modifier. You can pass an empty array [] to the $each modifier such that only the $sort modifier has an effect.

```
{
$push: {
<field>: {
$each: [ <value1>, <value2>, ... ],
$sort: <sort specification>
}
}
}
```

For <sort specification>:

To sort array elements that are not documents, or if the array elements are documents, to sort by the whole documents, specify 1 for ascending or -1 for descending.

If the array elements are documents, to sort by a field in the documents, specify a sort document with the field and the direction, i.e. { field: 1 } or { field: -1 }. Do not reference the containing array field in the sort specification (e.g. { "arrayField.field": 1 } is incorrect).

```
db.students.insertOne(
{
"_id": 1,
"quizzes": [
{ "id" : 1, "score" : 6 },
{ "id" : 2, "score" : 9 }
]
}
)
```

```
db.students.updateOne(
   { _id: 1 },
   {
     $push: {
       quizzes: {
         $each: [ { id: 3, score: 8 }, { id: 4, score: 7 }, { id: 5, score: 6 } ],
         $sort: { score: 1 }
       }
     }
   }
)
```

```
{
  "_id" : 1,
  "quizzes" : [
     { "id" : 1, "score" : 6 },
     { "id" : 5, "score" : 6 },
     { "id" : 4, "score" : 7 },
     { "id" : 3, "score" : 8 },
     { "id" : 2, "score" : 9 }
  ]
}
```

3.3.6.5 $slice

The $slice modifier limits the number of array elements during a $push operation. To project, or return, a specified number of array elements from a read operation, see the $slice projection operator instead.

To use the $slice modifier, it must appear with the $each modifier. You can pass an empty array [] to the $each modifier such that only the $slice modifier has an effect.

```
{
$push: {
<field>: {
$each: [ <value1>, <value2>, ... ],
$slice: <num>
}
}
}
```

The <num> can be:

Value, Description

- Zero, To update the array <field> to an empty array.
- Negative, To update the array <field> to contain only the last <num> elements.
- Positive, To update the array <field> contain only the first <num> elements.

```
{ "_id" : 3, "scores" : [ 89, 70, 100, 20 ] }
```

```
db.students.updateOne(
{ _id: 3 },
{
$push: {
scores: {
$each: [ ],
$slice: -3
}
}
}
)
```

```
{ "_id" : 3, "scores" : [ 70, 100, 20 ] }
```

3.3.6.6 $position

The $position modifier specifies the location in the array at which the $push operator inserts elements. Without the $position modifier, the $push operator inserts elements to the end of the array. See $push modifiers for more information.

To use the $position modifier, it must appear with the $each modifier.

```
{
$push: {
<field>: {
$each: [ <value1>, <value2>, ... ],
$position: <num>
}
}}
```

<num> indicates the position in the array, based on a zero-based index:

A non-negative number corresponds to the position in the array, starting from the beginning of the array. If the value of <num> is greater or equal to the length of the array, the $position modifier has no effect and $push adds elements to the end of the array.

A negative number corresponds to the position in the array, counting from (but not including) the last element of the array. For example, -1 indicates the position just before the last element in the array. If you specify multiple elements in the $each array, the last added element is in the specified position from the end. If the absolute value of <num> is greater than or equal to the length of the array, the $push adds elements to the beginning of the array.

```
db.students.insertOne(
  { "_id" : 3, "scores" : [  50,  60,  20,  30,  70,  100 ] }
)
```

```
db.students.updateOne(
  { _id: 3 },
  {
    $push: {
      scores: {
        $each: [ 90, 80 ],
        $position: -2
      }
    }
  }
)
```

```
{ "_id" : 3, "scores" : [ 50, 60, 20, 30, 90, 80, 70, 100 ] }
```

3.3.6.7 $pop

The $pop operator removes the first or last element of an array. Pass $pop a value of -1 to remove the first element of an array and 1 to remove the last element in an array.

The $pop operator has the form:

```
{ $pop: { <field>: <-1 | 1>, ... } }
```

To specify a <field> in an embedded document or in an array, use dot notation.

```
db.students.insertOne( { _id: 10, scores: [ 9, 10 ] } )

db.students.updateOne( { _id: 10 }, { $pop: { scores: 1 } } )
```

```
{ _id: 10, scores: [ 9 ] }
```

3.3.6.8 $pull

The $pull operator removes from an existing array all instances of a value or values that match a specified condition.

The $pull operator has the form:

```
{ $pull: { <field1>: <value|condition>, <field2>: <value|condition>, ... } }
```

```
db.stores.insertMany( [
  {
    _id: 1,
    fruits: [ "apples", "pears", "oranges", "grapes", "bananas" ],
    vegetables: [ "carrots", "celery", "squash", "carrots" ]
  },
  {
    _id: 2,
    fruits: [ "plums", "kiwis", "oranges", "bananas", "apples" ],
    vegetables: [ "broccoli", "zucchini", "carrots", "onions" ]
  }
] )
```

```
db.stores.updateMany(
   { },
   { $pull: { fruits: { $in: [ "apples", "oranges" ] }, vegetables: "carrots" } }
)
```

```
{
  _id: 1,
  fruits: [ 'pears', 'grapes', 'bananas' ],
  vegetables: [ 'celery', 'squash' ]
},
{
  _id: 2,
  fruits: [ 'plums', 'kiwis', 'bananas' ],
  vegetables: [ 'broccoli', 'zucchini', 'onions' ]
}
```

3.3.6.9 $pullAll

The $pullAll operator removes all instances of the specified values from an existing array. Unlike the $pull operator that removes elements by specifying a query, $pullAll removes elements that match the listed values.

The $pullAll operator has the form:

```
{ $pullAll: { <field1>: [ <value1>, <value2> ... ], ... } }
```

To specify a <field> in an embedded document or in an array, use dot notation.

```
db.survey.insertOne( { _id: 1, scores: [ 0, 2, 5, 5, 1, 0 ] } )
db.survey.updateOne( { _id: 1 }, { $pullAll: { scores: [ 0, 5 ] } } )
```

```
{ "_id" : 1, "scores" : [ 2, 1 ] }
```

3.3.6.10 $

The positional $ operator identifies an element in an array to update without explicitly specifying the position of the element in the array.

The positional $ operator has the form:

```
{ "<array>.$" : value }
```

```
db.collection.updateOne(
  { <array>: value ... },
  { <update operator>: { "<array>.$" : value } }
)
```

- upsert

Do not use the positional operator $ with upsert operations because inserts will use the $ as a field name in the inserted document.

- Nested Arrays

The positional $ operator cannot be used for queries which traverse more than one array, such as queries that traverse arrays nested within other arrays, because the replacement for the $ placeholder is a single value

- Unsets

When used with the $unset operator, the positional $ operator does not remove the matching element from the array but rather sets it to null.

- Negations

If the query matches the array using a negation operator, such as $ne, $not, or $nin, then you cannot use the positional operator to update values from this array.

However, if the negated portion of the query is inside of an $elemMatch expression, then you can use the positional operator to update this field.

- Multiple Array Matches

The positional $ update operator behaves ambiguously when filtering on multiple array fields.

When the server executes an update method, it first runs a query to determine which documents you want to update. If the update filters documents on multiple array fields, the subsequent call to the positional $ update operator doesn't always update the required position in the array.

```
{
  _id: 4,
  grades: [
     { grade: 80, mean: 75, std: 8 },
     { grade: 85, mean: 90, std: 5 },
     { grade: 85, mean: 85, std: 8 }
  ]
}
```

```
db.students.updateOne(
   { _id: 4, "grades.grade": 85 },
   { $set: { "grades.$.std" : 6 } }
)
```

```
{
   "_id" : 4,
   "grades" : [
      { "grade" : 80, "mean" : 75, "std" : 8 },
      { "grade" : 85, "mean" : 90, "std" : 6 },
      { "grade" : 85, "mean" : 85, "std" : 8 }
   ]
}
```

3.3.6.11 $[]

The all positional operator $[] indicates that the update operator should modify all elements in the specified array field.
The $[] operator has the following form:

```
{ <update operator>: { "<array>.$[]" : value } }
```

```
db.students2.insertMany( [
   {
      "_id" : 1,
      "grades" : [
         { "grade" : 80, "mean" : 75, "std" : 8 },
         { "grade" : 85, "mean" : 90, "std" : 6 },
         { "grade" : 85, "mean" : 85, "std" : 8 }
      ]
   },
   {
      "_id" : 2,
      "grades" : [
```

```
        { "grade" : 90, "mean" : 75, "std" : 8 },
        { "grade" : 87, "mean" : 90, "std" : 5 },
        { "grade" : 85, "mean" : 85, "std" : 6 }
      ]
   }
] )
```

```
db.students2.updateMany(
   { },
   { $inc: { "grades.$[].std" : -2 } },
)
```

```
{
   "_id" : 1,
   "grades" : [
      { "grade" : 80, "mean" : 75, "std" : 6 },
      { "grade" : 85, "mean" : 90, "std" : 4 },
      { "grade" : 85, "mean" : 85, "std" : 6 }
   ]
}
{
   "_id" : 2,
   "grades" : [
      { "grade" : 90, "mean" : 75, "std" : 6 },
      { "grade" : 87, "mean" : 90, "std" : 3 },
      { "grade" : 85, "mean" : 85, "std" : 4 }
   ]
}
```

3.3.6.12 $[identifier]

The filtered positional operator $[<identifier>] identifies the array elements that match the arrayFilters conditions for an update operation,
e.g. db.collection.updateMany() and db.collection.findAndModify()

Used in conjunction with the arrayFilters option, the $[<identifier>] operator has the following form:

```
{ <update operator>: { "<array>.$[<identifier>]" : value } },
{ arrayFilters: [ { <identifier>: <condition> } ] }
```

```
db.students2.insertMany( [
  {
    "_id" : 1,
    "grades" : [
      { "grade" : 80, "mean" : 75, "std" : 6 },
      { "grade" : 85, "mean" : 90, "std" : 4 },
      { "grade" : 85, "mean" : 85, "std" : 6 }
    ]
  },
  {
    "_id" : 2,
    "grades" : [
      { "grade" : 90, "mean" : 75, "std" : 6 },
      { "grade" : 87, "mean" : 90, "std" : 3 },
      { "grade" : 85, "mean" : 85, "std" : 4 }
    ]
  }
] )
db.students2.updateMany(
  { },
  { $set: { "grades.$[elem].mean" : 100 } },
  { arrayFilters: [ { "elem.grade": { $gte: 85 } } ] }
)
```

```
{
  "_id" : 1,
  "grades" : [
    { "grade" : 80, "mean" : 75, "std" : 6 },
    { "grade" : 85, "mean" : 100, "std" : 4 },
    { "grade" : 85, "mean" : 100, "std" : 6 }
  ]
}
{
  "_id" : 2,
  "grades" : [
    { "grade" : 90, "mean" : 100, "std" : 6 },
    { "grade" : 87, "mean" : 100, "std" : 3 },
    { "grade" : 85, "mean" : 100, "std" : 4 }
  ]
}
```

3.4 Delete Operation

3.4.1 deleteOne

The deleteOne() function is used to delete a single document. By default, this function will delete only the first matching document. The syntax for this method is as shown

```
db.collection_name.deleteOne(
{ <<remove criteria>> }
)
```

3.4.2 deleteMany

db.collection_name.deleteMany() is used for deleting multiple documents that match the given criteria. The syntax remains the same as shown above, only the function name changes.

Observe that operators can be used to specify criteria, just like you saw previously in read operations.

This query will delete all the products that have price less than 1000.

3.4.3 Deleting all documents from a collection

To delete all the documents from the collection, the syntax is as shown

```
db.collection_name.deleteMany( {} )
```

3.5 bulkWrite

Performs multiple write operations with controls for order of execution.

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

Valid operations are:

- insertOne
- updateOne
- updateMany
- deleteOne
- deleteMany
- replaceOne

3.5.1 insertOne

```
db.collection.bulkWrite( [
   { insertOne : { "document" : <document> } }
] )
```

3.5.2 updateOne/updateMany

```
db.collection.bulkWrite( [
   { updateOne/updateMany :
      {
         "filter": <document>,
         "update": <document or pipeline>,          // Changed in 4.2
         "upsert": <boolean>,
         "collation": <document>,                   // Available starting in 3.4
         "arrayFilters": [ <filterdocument1>, ... ],  // Available starting in 3.6
         "hint": <document|string>                  // Available starting in 4.2.1
      }
   }
] )
```

3.5.3 deleteOne/deleteMany

```
db.collection.bulkWrite([
   { deleteOne/deleteMany : {
      "filter" : <document>,
      "collation" : <document>                  // Available starting in 3.4
   } }
] )
```

3.5.4 replaceOne

```
db.collection.bulkWrite([
   { replaceOne :
      {
         "filter" : <document>,
         "replacement" : <document>,
         "upsert" : <boolean>,
         "collation": <document>,                  // Available starting in 3.4
         "hint": <document|string>                 // Available starting in 4.2.1
      }
   }
] )
```

```
try {
   db.pizzas.bulkWrite( [
      { insertOne: { document: { _id: 3, type: "beef", size: "medium", price: 6 } } },
      { insertOne: { document: { _id: 4, type: "sausage", size: "large", price: 10 } } },
      { updateOne: {
         filter: { type: "cheese" },
         update: { $set: { price: 8 } }
      } },
      { deleteOne: { filter: { type: "pepperoni"} } },
      { replaceOne: {
         filter: { type: "vegan" },
         replacement: { type: "tofu", size: "small", price: 4 }
      } }
   ] )
} catch( error ) {
   print( error )
}
```

4. Indexing

Indexes support the efficient execution of queries in MongoDB. Without indexes, MongoDB must perform a *collection scan*, i.e. scan every document in a collection, to select those documents that match the query statement. If an appropriate index exists for a query, MongoDB can use the index to limit the number of documents it must inspect.

Indexes are special data structures [1] that store a small portion of the collection's 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. The ordering of the index entries supports efficient equality matches and range-based query operations. In addition, MongoDB can return sorted results by using the ordering in the index.

4.1 Types of Indexes

4.1.1 Single field index

We usually create an index on a single field for faster retrieval.
You can create indexes on fields within embedded documents, just as you can index top-level fields in documents.

```
db.collection.createIndex(
    { field:1/-1 }
)
```

4.1.2 Compound index

MongoDB supports *compound indexes*, where a single index structure holds references to multiple fields within a collection's documents.

Syntax:

```
db.collection.createIndex( { <field1>: <type>, <field2>: <type2>, ... } )
```

The index can specify any field as ascending or descending as per requirement.
Multikey Indexes for both fields in compound index is not allowed

4.1.3 Multikey Indexes

To index a field that holds an array value, MongoDB creates an index key for each element in the array. These multikey indexes support efficient queries against array fields. Multikey indexes can be constructed over arrays that hold both scalar values (e.g. strings, numbers) and nested documents.

```
db.collection.createIndex( { <field>: < 1 or -1 > } )
```

4.1.4 Text index

In the application, customers would like to perform text based searches, for example: search for either the main category such as 'electronics' or more specifically sub category such as 'smartphones'. Text indexes can be used for this purpose.

First, a text index is created and then $text operator is used to perform text search on the indexed field.

```
db.product_catalog.createIndex(
{ categories:"text" }
)
```

Now, the above index can be used to retrieve the details of 'smartphones':

```
db.product_catalog.find(
{ $text:{ $search:"smartphones" } }
)
```

4.1.5 TTL index

TTL collections make it possible to store data in MongoDB and have the mongod automatically remove data after a specified number of seconds or at a specific clock time.
Data expiration is useful for some classes of information, including machine generated event data, logs, and session information that only need to persist for a limited period of time.
A special TTL index property supports the implementation of TTL collections. The TTL feature relies on a background thread in mongod that reads the date-typed values in the index and removes expired documents from the collection.

```
db.log_events.createIndex( { "createdAt": 1 }, { expireAfterSeconds: 10 } )
```

4.1.6 Unique Index

A unique index ensures that the indexed fields do not store duplicate values; i.e. enforces uniqueness for the indexed fields. By default, MongoDB creates a unique index on the _id field during the creation of a collection.

```
db.collection.createIndex( <key and index type specification>, { unique: true } )
```

4.1.7 Partial index

Partial indexes only index the documents in a collection that meet a specified filter expression. By indexing a subset of the documents in a collection, partial indexes have lower storage requirements and reduced performance costs for index creation and maintenance.

- **Create a Partial Index**

To create a partial index, use the db.collection.createIndex() method with the partialFilterExpression option. The partialFilterExpression option accepts a document that specifies the filter condition using:

equality expressions (i.e. field: value or using the $eq operator),

- $exists: true expression,
- $gt, $gte, $lt, $lte expressions,
- $type expressions,
- $and operator,
- $or operator,
- $in operator

For example, the following operation creates a compound index that indexes only the documents with a rating field greater than 5.

```
db.restaurants.createIndex(
{ cuisine: 1, name: 1 },
{ partialFilterExpression: { rating: { $gt: 5 } } }
)
```

4.2 Index Methods

4.2.1 Get all indexes in a collection

To find all the indexes created in a collection, we can use the getIndexes() method.

The syntax is as shown below

```
db.collection_name.getIndexes()
```

4.2.2 Delete an index

To delete an index, we can use the dropIndex() method.

The syntax is as shown below if we know the name of the index

```
db.collection_name.dropIndex( "colors_1" )
```

4.2.3 Delete all indexes

To delete all user-defined indexes in a collection, we can use dropIndexes() method.

The syntax is as shown below

```
db.collection_name.dropIndexes()
```

4.3 Index Strategies

In order to select the most optimum index for your database design, there are certain factors that need to be taken into account, some of these are: -

- Type of queries that are expected most,
- The ratio of reads to writes, in the application and
- Memory consumption and availability on the system.

We must keep in mind these scenarios before choosing to create indexes:

4.3.1 Create Indexes to Support Queries

For a query to run with greater efficiency and increase performance, we must identify the fields that the query is scanning and then the index must be captured on those fields.

For example, for the Zovia database, the product smartphones have been searched multiple times. Hence, that will be made a part of indexing.

```
db.product_catalog.createIndex({category.main:"smartphone": 1})
```

4.3.2 Usage of Sorted Indexes

To make most of indexing, we must use the sorting of fields (ascending or descending order) to our maximum benefit. Hence we mention the 1 or -1 order in which the index must be created.

For example, to fetch the product with the highest price, the catalog can be indexed on the price field and sorted in descending order. In this way, the query will find match document in the first row itself and execution time will be much faster.

```
db.product_catalog.createIndex({price:1})
```

4.3.3 Use RAM as Indexing memory

If the indexing fields and their mapping is stored in RAM, the system can avoid reading the index from secondary memory or any disk. This will enable faster processing.

For example, if the index for product_catalog is created in the manufacturer field. The retrieval will be much faster in terms of accessing RAM then secondary memory.

4.3.4 Use Selectivity to frame queries

Selectivity is the capability of MongoDB to make use of an Index to quickly process that query. For every time a query executes, MongoDB will ensure that indexing is applied to process faster results.

For example, if the backend code interacts will MongoDB for retrieval of all products. Then that query must be identified, the field used must be added to the indexing logic.

5. Aggregation

Aggregation is a method used to process data and return computed results.

You can perform aggregation in three ways in MongoDB:

5.1 Single purpose aggregation methods

5.1.1 count

Returns the count of documents that would match a find() query for the collection or view. The

db.collection.count() method does not perform the find() operation but instead counts and returns the number of results that match a query.

```
db.collection.count(query, options)
```

Count the number of the documents in the orders collection with the field ord_dt greater than new

Date('01/01/2012'):

```
db.orders.count( { ord_dt: { $gt: new Date('01/01/2012') } } )
```

The query is equivalent to the following:

```
db.orders.find( { ord_dt: { $gt: new Date('01/01/2012') } } ).count()
```

5.1.2 distinct

Finds the distinct values for a specified field across a single collection or view and returns the results in an array.

```
db.collection.distinct(field, query, options)
```

```
{ "_id": 1, "dept": "A", "item": { "sku": "111", "color": "red" }, "sizes": [ "S", "M" ] }
{ "_id": 2, "dept": "A", "item": { "sku": "111", "color": "blue" }, "sizes": [ "M", "L" ] }
{ "_id": 3, "dept": "B", "item": { "sku": "222", "color": "blue" }, "sizes": "S" }
{ "_id": 4, "dept": "A", "item": { "sku": "333", "color": "black" }, "sizes": [ "S" ] }
```

```
db.inventory.distinct( "item.sku", { dept: "A" } )
```

5.2 Aggregation Pipeline

An aggregation pipeline consists of one or more stages that process documents:

- Each stage performs an operation on the input documents. For example, a stage can filter documents, group documents, and calculate values.
- The documents that are output from a stage are passed to the next stage.
- An aggregation pipeline can return results for groups of documents. For example, return the total, average, maximum, and minimum values.

```
db.orders.aggregate( [
  {stages}
] )
```

5.2.1 Stage Operators

5.2.1.1 $match

Filters the documents to pass only the documents that match the specified condition(s) to the next pipeline stage.The $match stage has the following prototype form:

```
{ $match: { <query> } }
```

```
{ "_id" : ObjectId("512bc95fe835e68f199c8686"), "author" : "dave", "score" : 80, "views" : 100 }
{ "_id" : ObjectId("512bc962e835e68f199c8687"), "author" : "dave", "score" : 85, "views" : 521 }
{ "_id" : ObjectId("55f5a192d4bede9ac365b257"), "author" : "ahn", "score" : 60, "views" : 1000 }
{ "_id" : ObjectId("55f5a192d4bede9ac365b258"), "author" : "li", "score" : 55, "views" : 5000 }
{ "_id" : ObjectId("55f5a1d3d4bede9ac365b259"), "author" : "annT", "score" : 60, "views" : 50 }
{ "_id" : ObjectId("55f5a1d3d4bede9ac365b25a"), "author" : "li", "score" : 94, "views" : 999 }
{ "_id" : ObjectId("55f5a1d3d4bede9ac365b25b"), "author" : "ty", "score" : 95, "views" : 1000 }
```

```
db.articles.aggregate(
   [ { $match : { author : "dave" } } ]
);
```

```
{ "_id" : ObjectId("512bc95fe835e68f199c8686"), "author" : "dave", "score" : 80, "views" : 100 }
{ "_id" : ObjectId("512bc962e835e68f199c8687"), "author" : "dave", "score" : 85, "views" : 521 }
```

5.2.1.2 $project

Passes along the documents with the requested fields to the next stage in the pipeline. The specified fields can be existing fields from the input documents or newly computed fields.

The $project stage has the following prototype form:

```
{ $project: { <specification(s)> } }
```

```
{ $project: { "<field1>": 0/1 or true/false } }

{ $project: { <array>:[$field1,$field2] } }
```

```
{
  "_id" : 1,
  title: "abc123",
  isbn: "0001122223334",
  author: { last: "zzz", first: "aaa" },
  copies: 5,
  lastModified: "2016-07-28"
}
```

```
db.books.aggregate( [ { $project : { "author.first" : 0, "lastModified" : 0 } } ] )
db.bookmarks.aggregate( [ { $project: { "author": { "first": 0}, "lastModified" : 0 } } ] )
```

5.2.1.3 $count

Passes a document to the next stage that contains a count of the number of documents input to the stage.

```
{ $count: <string> }
```

<string> is the name of the output field which has the count as its value. <string> must be a non-empty string, must not start with $ and must not contain the . character.

```
{ "_id" : 1, "subject" : "History", "score" : 88 }
{ "_id" : 2, "subject" : "History", "score" : 92 }
{ "_id" : 3, "subject" : "History", "score" : 97 }
{ "_id" : 4, "subject" : "History", "score" : 71 }
{ "_id" : 5, "subject" : "History", "score" : 79 }
{ "_id" : 6, "subject" : "History", "score" : 83 }
```

```
db.scores.aggregate(
  [
    {
      $match: {
        score: {
          $gt: 80
        }
      }
    },
    {
      $count: "passing_scores"
    }
  ]
)
```

```
{ "passing_scores" : 4 }
```

5.2.1.4 $group

The $group stage separates documents into groups according to a "group key". The output is one document for each unique group key.
A group key is often a field, or group of fields. The group key can also be the result of an expression. Use the _id field in the $group pipeline stage to set the group key. See below for usage examples.

In the $group stage output, the _id field is set to the group key for that document. The output documents can also contain additional fields that are set using accumulator expressions.

```
{
  $group:
    {
      _id: <expression>, // Group key
      <field1>: { <accumulator1> : <expression1> },
      ...
    }
}
```

```
db.sales.insertMany([
  { "_id" : 1, "item" : "abc", "price" : NumberDecimal("10"), "quantity" : NumberInt("2"), "date"
: ISODate("2014-03-01T08:00:00Z") },
  { "_id" : 2, "item" : "jkl", "price" : NumberDecimal("20"), "quantity" : NumberInt("1"), "date" :
ISODate("2014-03-01T09:00:00Z") },
  { "_id" : 3, "item" : "xyz", "price" : NumberDecimal("5"), "quantity" : NumberInt( "10"), "date"
: ISODate("2014-03-15T09:00:00Z") },
  { "_id" : 4, "item" : "xyz", "price" : NumberDecimal("5"), "quantity" :  NumberInt("20") ,
"date" : ISODate("2014-04-04T11:21:39.736Z") },
  { "_id" : 5, "item" : "abc", "price" : NumberDecimal("10"), "quantity" : NumberInt("10") ,
"date" : ISODate("2014-04-04T21:23:13.331Z") },
  { "_id" : 6, "item" : "def", "price" : NumberDecimal("7.5"), "quantity": NumberInt("5" ) ,
"date" : ISODate("2015-06-04T05:08:13Z") },
  { "_id" : 7, "item" : "def", "price" : NumberDecimal("7.5"), "quantity": NumberInt("10") ,
"date" : ISODate("2015-09-10T08:43:00Z") },
  { "_id" : 8, "item" : "abc", "price" : NumberDecimal("10"), "quantity" : NumberInt("5" ) ,
"date" : ISODate("2016-02-06T20:20:13Z") },
])
```

```
db.sales.aggregate( [ { $group : { _id : "$item" } } ] )
```

```
{ "_id" : "abc" }
{ "_id" : "jkl" }
{ "_id" : "def" }
{ "_id" : "xyz" }
```

5.2.1.5 $sort

Sorts all input documents and returns them to the pipeline in sorted order.

```
{ $sort: { <field1>: <sort order>, <field2>: <sort order> ... } }
```

```
db.restaurants.insertMany( [
  { "_id" : 1, "name" : "Central Park Cafe", "borough" : "Manhattan"},
  { "_id" : 2, "name" : "Rock A Feller Bar and Grill", "borough" : "Queens"},
  { "_id" : 3, "name" : "Empire State Pub", "borough" : "Brooklyn"},
  { "_id" : 4, "name" : "Stan's Pizzaria", "borough" : "Manhattan"},
  { "_id" : 5, "name" : "Jane's Deli", "borough" : "Brooklyn"},
] );
```

- 1 Sort ascending.
- -1 Sort descending.
- { $meta: "textScore" } Sort by the computed textScore metadata in descending order

```
db.restaurants.aggregate(
  [
    { $sort : { borough : 1, _id: 1 } }
  ]
)
```

5.2.1.6 $skip

Skips over the specified number of documents that pass into the stage and passes the remaining documents to the next stage in the pipeline.

```
{ $skip: <positive 64-bit integer> }
```

```
db.article.aggregate([
{ $skip : 5 }
]);
```

This operation skips the first 5 documents passed to it by the pipeline.

5.2.1.7 $limit

Limits the number of documents passed to the next stage in the pipeline.

```
{ $limit: <positive 64-bit integer> }
```

```
db.article.aggregate([
  { $limit : 5 }
]);
```

This operation returns only the first 5 documents passed to it by the pipeline.

5.2.1.8 $out

Takes the documents returned by the aggregation pipeline and writes them to a specified collection. Starting in MongoDB 4.4, you can specify the output database.

The $out stage must be the last stage in the pipeline.

The $out operator lets the aggregation framework return result sets of any size.

```
{ $out: { db: "<output-db>", coll: "<output-collection>" } }

{ $out: "<output-collection>" } // Output collection is in the same database
```

```
db.getSiblingDB("test").books.insertMany([
   { "_id" : 8751, "title" : "The Banquet", "author" : "Dante", "copies" : 2 },
   { "_id" : 8752, "title" : "Divine Comedy", "author" : "Dante", "copies" : 1 },
   { "_id" : 8645, "title" : "Eclogues", "author" : "Dante", "copies" : 2 },
   { "_id" : 7000, "title" : "The Odyssey", "author" : "Homer", "copies" : 10 },
   { "_id" : 7020, "title" : "Iliad", "author" : "Homer", "copies" : 10 }
])
```

```
db.getSiblingDB("test").books.aggregate( [
   { $group : { _id : "$author", books: { $push: "$title" } } },
   { $out : "authors" }
  ] )
```

The collection contains the following documents:

```
{ "_id" : "Homer", "books" : [ "The Odyssey", "Iliad" ] }
{ "_id" : "Dante", "books" : [ "The Banquet", "Divine Comedy", "Eclogues" ] }
```

5.2.1.9 $bucket

Categorizes incoming documents into groups, called buckets, based on a specified expression and bucket boundaries and outputs a document per each bucket. Each output document contains an _id field whose value specifies the inclusive lower bound of the bucket. The output option specifies the fields included in each output document.

$bucket only produces output documents for buckets that contain at least one input document.

```
{
  $bucket: {
     groupBy: <expression>,
     boundaries: [ <lowerbound1>, <lowerbound2>, ... ],
     default: <literal>,
     output: {
        <output1>: { <$accumulator expression> },
        ...
        <outputN>: { <$accumulator expression> }
     }
   }
}
```

```
db.artists.insertMany([
  { "_id" : 1, "last_name" : "Bernard", "first_name" : "Emil", "year_born" : 1868, "year_died" : 1941, "nationality" : "France" },
  { "_id" : 2, "last_name" : "Rippl-Ronai", "first_name" : "Joszef", "year_born" : 1861, "year_died" : 1927, "nationality" : "Hungary" },
  { "_id" : 3, "last_name" : "Ostroumova", "first_name" : "Anna", "year_born" : 1871, "year_died" : 1955, "nationality" : "Russia" },
  { "_id" : 4, "last_name" : "Van Gogh", "first_name" : "Vincent", "year_born" : 1853, "year_died" : 1890, "nationality" : "Holland" },
  { "_id" : 5, "last_name" : "Maurer", "first_name" : "Alfred", "year_born" : 1868, "year_died" : 1932, "nationality" : "USA" },
  { "_id" : 6, "last_name" : "Munch", "first_name" : "Edvard", "year_born" : 1863, "year_died" : 1944, "nationality" : "Norway" },
  { "_id" : 7, "last_name" : "Redon", "first_name" : "Odilon", "year_born" : 1840, "year_died" : 1916, "nationality" : "France" },
  { "_id" : 8, "last_name" : "Diriks", "first_name" : "Edvard", "year_born" : 1855, "year_died" : 1930, "nationality" : "Norway" }
])
```

```
db.artists.aggregate( [
  // First Stage
  {
    $bucket: {
      groupBy: "$year_born",                        // Field to group by
      boundaries: [ 1840, 1850, 1860, 1870, 1880 ], // Boundaries for the buckets
      default: "Other",                             // Bucket id for documents which do not fall into a bucket
      output: {                                     // Output for each bucket
        "count": { $sum: 1 },
        "artists" :
          {
            $push: {
              "name": { $concat: [ "$first_name", " ", "$last_name"] },
              "year_born": "$year_born"
            }
          }
      }
    }
  },
  // Second Stage
  {
    $match: { count: {$gt: 3} }
  }
] )
```

```
{ "_id" : 1860, "count" : 4, "artists" :
  [
    { "name" : "Emil Bernard", "year_born" : 1868 },
    { "name" : "Joszef Rippl-Ronai", "year_born" : 1861 },
    { "name" : "Alfred Maurer", "year_born" : 1868 },
    { "name" : "Edvard Munch", "year_born" : 1863 }
  ]
}
```

5.2.1.10 $unwind

Deconstructs an array field from the input documents to output a document for each element.

Each output document is the input document with the value of the array field replaced by the element.

```
{ $unwind: <field path> }
```

$unwind treats the sizes field as a single element array if:

- the field is present,
- the value is not null, and
- the value is not an empty array.

```
db.clothing.insertMany([
  { "_id" : 1, "item" : "Shirt", "sizes": [ "S", "M", "L"] },
  { "_id" : 2, "item" : "Shorts", "sizes" : [ ] },
  { "_id" : 3, "item" : "Hat", "sizes": "M" },
  { "_id" : 4, "item" : "Gloves" },
  { "_id" : 5, "item" : "Scarf", "sizes" : null }
])
```

```
db.clothing.aggregate( [ { $unwind: { path: "$sizes" } } ] )
```

```
{ _id: 1, item: 'Shirt', sizes: 'S' },
{ _id: 1, item: 'Shirt', sizes: 'M' },
{ _id: 1, item: 'Shirt', sizes: 'L' },
{ _id: 3, item: 'Hat', sizes: 'M' }
```

5.2.1.11 $lookup

The $lookup stage adds a new array field to each input document. The new array field contains the matching documents from the "joined" collection. The

$lookup stage passes these reshaped documents to the next stage.

- **Equality Match with a Single Join Condition**

To perform an equality match between a field from the input documents with a field from the documents of the "joined" collection, the

$lookup stage has this syntax:

```
{
  $lookup:
    {
      from: <collection to join>,
      localField: <field from the input documents>,
      foreignField: <field from the documents of the "from" collection>,
      as: <output array field>
    }
}
```

- **Join Conditions and Subqueries on a Joined Collection**

MongoDB supports:

- Executing a pipeline on a joined collection.
- Multiple join conditions.
- Correlated and uncorrelated subqueries.

```
{
  $lookup:
    {
      from: <joined collection>,
      let: { <var_1>: <expression>, …, <var_n>: <expression> },
      pipeline: [ <pipeline to run on joined collection> ],
      as: <output array field>
    }
}
```

5.2.2 Accumulators

5.2.2.1 $sum

Calculates and returns the collective sum of numeric values.

$sum ignores non-numeric values.

```
{ $sum: <expression> }

{ $sum: [ <expression1>, <expression2> ... ]  }
```

```
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-01-
01T08:00:00Z") }
{ "_id" : 2, "item" : "jkl", "price" : 20, "quantity" : 1, "date" : ISODate("2014-02-03T09:00:00Z")
}
{ "_id" : 3, "item" : "xyz", "price" : 5, "quantity" : 5, "date" : ISODate("2014-02-03T09:05:00Z")
}
{ "_id" : 4, "item" : "abc", "price" : 10, "quantity" : 10, "date" : ISODate("2014-02-
15T08:00:00Z") }
{ "_id" : 5, "item" : "xyz", "price" : 5, "quantity" : 10, "date" : ISODate("2014-02-
15T09:05:00Z") }
```

```
db.sales.aggregate(
  [
    {
     $group:
       {
         _id: { day: { $dayOfYear: "$date"}, year: { $year: "$date" } },
         totalAmount: { $sum: { $multiply: [ "$price", "$quantity" ] } },
         count: { $sum: 1 }
       }
    }
  ]
)
```

```
{ "_id" : { "day" : 46, "year" : 2014 }, "totalAmount" : 150, "count" : 2 }
{ "_id" : { "day" : 34, "year" : 2014 }, "totalAmount" : 45, "count" : 2 }
{ "_id" : { "day" : 1, "year" : 2014 }, "totalAmount" : 20, "count" : 1 }
```

5.2.2.2 $avg

Returns the average value of the numeric values. $avg ignores non-numeric values.

```
{ $avg: <expression> }
```

```
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-01-01T08:00:00Z") }
{ "_id" : 2, "item" : "jkl", "price" : 20, "quantity" : 1, "date" : ISODate("2014-02-03T09:00:00Z") }
{ "_id" : 3, "item" : "xyz", "price" : 5, "quantity" : 5, "date" : ISODate("2014-02-03T09:05:00Z") }
{ "_id" : 4, "item" : "abc", "price" : 10, "quantity" : 10, "date" : ISODate("2014-02-15T08:00:00Z") }
{ "_id" : 5, "item" : "xyz", "price" : 5, "quantity" : 10, "date" : ISODate("2014-02-15T09:12:00Z") }
```

```
db.sales.aggregate(
  [
    {
      $group:
        {
          _id: "$item",
          avgAmount: { $avg: { $multiply: [ "$price", "$quantity" ] } },
          avgQuantity: { $avg: "$quantity" }
        }
    }
  ]
)
```

```
{ "_id" : "xyz", "avgAmount" : 37.5, "avgQuantity" : 7.5 }
{ "_id" : "jkl", "avgAmount" : 20, "avgQuantity" : 1 }
{ "_id" : "abc", "avgAmount" : 60, "avgQuantity" : 6 }
```

5.2.2.3 $subtract

Subtracts two numbers to return the difference, or two dates to return the difference in milliseconds, or a date and a number in milliseconds to return the resulting date.

```
{ $subtract: [ <expression1>, <expression2> ] }
```

```
db.sales.insertMany([
  { "_id" : 1, "item" : "abc", "price" : 10, "fee" : 2, "discount" : 5, "date" : ISODate("2014-03-01T08:00:00Z") },
  { "_id" : 2, "item" : "jkl", "price" : 20, "fee" : 1, "discount" : 2, "date" : ISODate("2014-03-01T09:00:00Z") }
])
```

```
db.sales.aggregate( [ { $project: { item: 1, total: { $subtract: [ { $add: [ "$price", "$fee" ] }, "$discount" ] } } } ] )
```

```
{ "_id" : 1, "item" : "abc", "total" : 7 }
{“_id" : 2, "item" : "jkl", "total" : 19 }
```

5.2.2.4 $divide

Divides one number by another and returns the result. Pass the arguments to $divide in an array.

```
{ $divide: [ <expression1>, <expression2> ] }
```

```
{ "_id" : 1, "name" : "A", "hours" : 80, "resources" : 7 },
{ "_id" : 2, "name" : "B", "hours" : 40, "resources" : 4 }
```

```
db.planning.aggregate(
  [
    { $project: { name: 1, workdays: { $divide: [ "$hours", 8 ] } } }
  ]
)
```

```
{ "_id" : 1, "name" : "A", "workdays" : 10 }
{ "_id" : 2, "name" : "B", "workdays" : 5 }
```

5.2.2.5 $multiply

Multiplies numbers together and returns the result. Pass the arguments to $multiply in an array.

```
{ $multiply: [ <expression1>, <expression2>, ... ] }
```

```
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity": 2, date: ISODate("2014-03-01T08:00:00Z") }
{ "_id" : 2, "item" : "jkl", "price" : 20, "quantity": 1, date: ISODate("2014-03-01T09:00:00Z") }
{ "_id" : 3, "item" : "xyz", "price" : 5, "quantity": 10, date: ISODate("2014-03-15T09:00:00Z") }
```

```
db.sales.aggregate(
  [
    { $project: { date: 1, item: 1, total: { $multiply: [ "$price", "$quantity" ] } } }
  ]
)
```

```
{ "_id" : 1, "item" : "abc", "date" : ISODate("2014-03-01T08:00:00Z"), "total" : 20 }
{ "_id" : 2, "item" : "jkl", "date" : ISODate("2014-03-01T09:00:00Z"), "total" : 20 }
{ "_id" : 3, "item" : "xyz", "date" : ISODate("2014-03-15T09:00:00Z"), "total" : 50 }
```

5.2.2.6 $add

Adds numbers together or adds numbers and a date. If one of the arguments is a date, $add treats the other arguments as milliseconds to add to the date.

```
{ $add: [ <expression1>, <expression2>, ... ] }
```

```
{ "_id" : 1, "item" : "abc", "price" : 10, "fee" : 2, date: ISODate("2014-03-01T08:00:00Z") }
{ "_id" : 2, "item" : "jkl", "price" : 20, "fee" : 1, date: ISODate("2014-03-01T09:00:00Z") }
{ "_id" : 3, "item" : "xyz", "price" : 5,  "fee" : 0, date: ISODate("2014-03-15T09:00:00Z") }
```

```
db.sales.aggregate(
  [
    { $project: { item: 1, total: { $add: [ "$price", "$fee" ] } } }
  ]
)
```

```
{ "_id" : 1, "item" : "abc", "total" : 12 }
{ "_id" : 2, "item" : "jkl", "total" : 21 }
{ "_id" : 3, "item" : "xyz", "total" : 5 }
```

5.3 Map Reduce

Map-Reduce is a data processing mechanism in which a large amount of data is transformed into beneficial aggregated results by applying mainly two functions on the collection: the mapper function and the reducer function.

- Mapper function– It builds a map over the data read from the collection based on the fields specified and group them into an array. This is provided as input to the Reducer function.

Map phase: filter/convert/transform data

- Reducer function – This function will transform the values i.e., it collects the key-value pair from the mapper function and condenses the aggregated data into a collection or as a document.

Reduce phase: performs aggregation over the data

```
db.collection.mapReduce(
                        function() {emit(key,value);},  //map function
                        function(key,values) {return reduceFunction}, {
                        out: collection,
                        query: document,
                        sort: document,
                         limit: number
                    }
      )
```

- map() - a function that performs mapping of value with a key and emitting a key-value pair corresponding to it.
- reduce() - a function that clubs all the documents having the identical key.
- out() - it is used to mention the name of the new collection where the documents will be stored
- query() - acts as the specific fields filter that need to be part of the output
- sort() – needed to order the fields in a specific format (asc, desc).
- limit() - one can choose to limit the number of documents to be added as the resultant set.

```
{
      "_id" : ObjectId("5e81c21e3abf9ba929f49691"),
      "name" : "Anwar",
      "age" : 29
}
{
      "_id" : ObjectId("5e81c21e3abf9ba929f49692"),
      "name" : "Visak",
      "age" : 26
}
{
      "_id" : ObjectId("5e81c21e3abf9ba929f49693"),
```

```
        "name" : "Anwar",
        "age" : 34
}
{
        "_id" : ObjectId("5e81c21e3abf9ba929f49694"),
        "name" : "Kiran",
        "age" : 29
}
{
        "_id" : ObjectId("5e81c21e3abf9ba929f49695"),
        "name" : "Swasthik",
        "age" : 30
}
{
        "_id" : ObjectId("5e81c21e3abf9ba929f49696"),
        "name" : "Titus",
        "age" : 29
}
{
        "_id" : ObjectId("5e81c21e3abf9ba929f49697"),
        "name" : "Alex",
        "age" : 30
}
{
        "_id" : ObjectId("5e81c21e3abf9ba929f49698"),
        "name" : "Visak",
        "age" : 32
}
```

```
db.userDetail. mapReduce ({
   map:function()
   {
     emit(this.name,this.age)
   },
   reduce:function(name,counters)
   {
      return Array.sum(counters);
     }
   ,out:"SumOfAges"
 })
```

```
{ "_id" : "Alex", "value" : 30 }
{ "_id" : "Anwar", "value" : 63 }
{ "_id" : "Kiran", "value" : 29 }
{ "_id" : "Swasthik", "value" : 30 }
{ "_id" : "Titus", "value" : 29 }
{ "_id" : "Visak", "value" : 58 }
```