

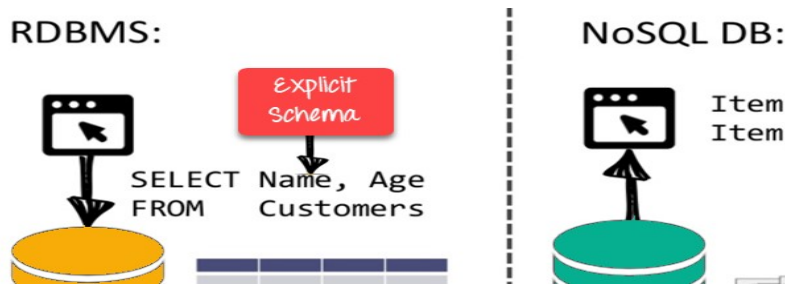
1. concepts of NoSQL. Advantages and feature

- Databases can be divided in 3 types:
 - RDBMS (Relational Database Management System)
 - OLAP (Online Analytical Processing)
 - NoSQL (recently developed database)
- **NoSQL database** stands for “Not Only SQL” or “Not SQL.”
- Traditional RDBMS uses SQL syntax to store and retrieve data for further insights. Instead, a NoSQL database system encompasses a wide range of database technologies that can store structured, semi-structured, unstructured and polymorphic data with huge volumes of data.
- The system response time becomes slow when you use RDBMS for massive volumes of data.
To resolve this problem, we could “scale up” our systems by upgrading our existing hardware. This process is expensive.
- The alternative for this issue is to distribute database load on multiple hosts whenever the load increases. This method is known as “scaling out.”
- It provides a mechanism for storage and retrieval of data other than tabular relations model used in relational databases. NoSQL database doesn't use tables for storing data. It is generally used to store big data and real-time web applications.
 - NoSQL relies upon a softer model known as the BASE model. BASE (Basically Available, Soft state, Eventual consistency).
 - Basically Available: Guarantees the availability of the data . There will be a response to any request (can be failure too).
 - Soft state: The state of the system could change over time.
 - Eventual consistency: The system will eventually become consistent once it stops receiving input.

Unit – 1 Concept of NoSQL : MongoDB

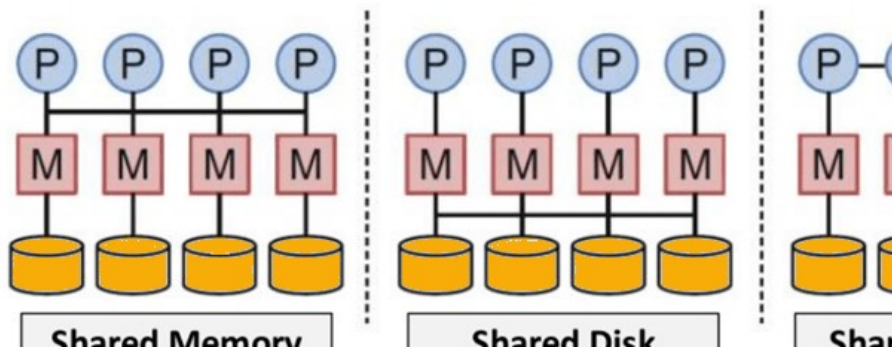
Features of NoSQL

- Non-relational
 - NoSQL databases never follow the relational model
 - Never provide tables with flat fixed-column records
 - Work with self-contained aggregates or BLOBs
 - Doesn't require object-relational mapping and data normalization
 - No complex features like query languages, query planners, referential integrity joins, ACID
- Schema-free
 - NoSQL databases are either schema-free or have relaxed schemas
 - Do not require any sort of definition of the schema of the data
 - Offers heterogeneous structures of data in the same domain



- Simple API
 - Offers easy to use interfaces for storage and querying data provided
 - APIs allow low-level data manipulation & selection methods
 - Text-based protocols mostly used with HTTP REST with JSON
 - Mostly used no standard based NoSQL query language
 - Web-enabled databases running as internet-facing services
- Distributed
 - Multiple NoSQL databases can be executed in a distributed fashion
 - Offers auto-scaling and fail-over capabilities
 - Often ACID concept can be sacrificed for scalability and throughput
 - Mostly no synchronous replication between distributed nodes Asynchronous Multi-Master Replication, peer-to-peer, HDFS Replication
 - Only providing eventual consistency
 - Shared Nothing Architecture. This enables less coordination and higher distribution.

Unit – 1 Concept of NoSQL : MongoDB



- Highscalability
NoSQL database use sharding for horizontal scaling.
- High availability –
Auto replication feature in NoSQL databases makes it highly available because in case of any failure data replicates itself to the previous consistent state.
- Distributed Computing
- Lower cost
- Schema flexibility
- Un/semi-structured data
- No complex relationships

Disadvantages of NoSQL

- No standardization rules
- Limited query capabilities
- RDBMS databases and tools are comparatively mature
- It does not offer any traditional database capabilities, like consistency when multiple transactions are performed simultaneously.
- When the volume of data increases it is difficult to maintain unique values as keys become difficult
- Doesn't work as well with relational data
- The learning curve is stiff for new developers
- Open source options so not so popular for enterprises.

Unit – 1 Concept of NoSQL : MongoDB

SQL	NoSQL
RELATIONAL DATABASE MANAGEMENT SYSTEM (RDBMS)	Non-relational or distributed database system.
These databases have fixed or static or predefined schema	They have dynamic schema
These databases are not suited for hierarchical data storage.	These databases are best suited for hierarchical data storage.
These databases are best suited for complex queries	These databases are not so good for complex queries
Vertically Scalable	Horizontally scalable
Follows ACID property	Follows CAP(consistency, availability, partition tolerance)
Examples: MySQL, PostgreSQL, Oracle, MS-SQL Server etc	Examples: MongoDB, GraphQL, HBase, Neo4j, Cassandra etc

• Types of NoSQL Databases

NoSQL Databases are mainly categorized into four types: Key-value pair, Column-oriented, Graph-based and Document-oriented. Every category has its unique attributes and limitations. None of the above-specified database is better to solve all the problems. Users should select the database based on their product needs.

Types of NoSQL Databases:

- Key-value Pair Based
- Column-oriented Graph
- Graphs based
- Document-oriented

Unit – 1 Concept of NoSQL : MongoDB

Key Value Pair Based

Data is stored in key/value pairs. It is designed in such a way to handle lots of data and heavy load.

Key-value pair storage databases store data as a hash table where each key is unique, and the value can be a JSON, BLOB(Binary Large Objects), string, etc.

It is one of the most basic NoSQL database example. This kind of NoSQL database is used as a collection, dictionaries, associative arrays, etc. Key value stores help the developer to store schema-less data. They work best for shopping cart contents.

Key	Value
Name	Joe Bloggs
Age	42
Occupation	Stunt Doubl
Height	175cm

Column-based

Column-oriented databases work on columns and are based on BigTable paper by Google.

Every column is treated separately. Values of single column databases are stored contiguously.

Column-based NoSQL databases are widely used to manage data warehouses, business intelligence, CRM, Library card catalogs.

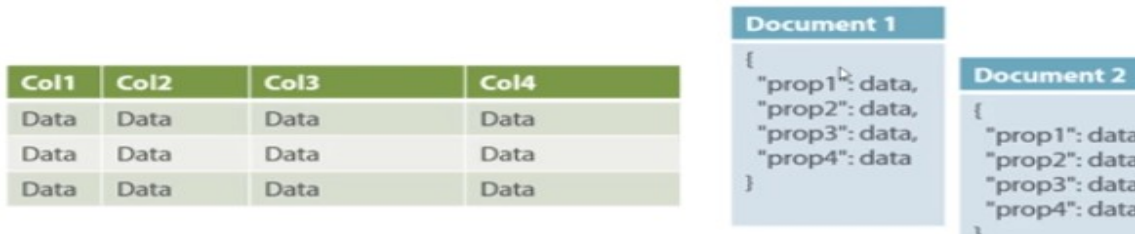
ColumnFamily			
Row Key	Column Name		
	Key	Key	k
	Value	Value	v
	Column Name		
	Key	Key	k

Document-Oriented

Document-Oriented NoSQL DB stores and retrieves data as a key value pair but the value part is stored as a document. The document is stored in JSON or XML formats. The value is understood by the DB and can be queried

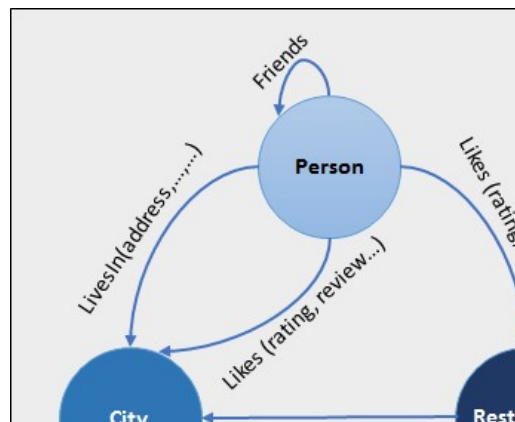
Unit – 1 Concept of NoSQL : MongoDB

The document type is mostly used for CMS systems, blogging platforms, real-time analytics & e-commerce applications. It should not use for complex transactions which require multiple operations or queries against varying aggregate structures.



Graph-Based

A graph type database stores entities as well the relations amongst those entities. The entity is stored as a node with the relationship as edges. An edge gives a relationship between nodes. Every node and edge has a unique identifier.



Graph base database mostly used for social networks, logistics, spatial data.

• When to Use NoSQL Databases

A few cases where NoSQL databases should be used instead of a relational database below:

- You don't know what kind of data will be stored in an application.
- Data may change frequently in an application.
- You need to develop an application with as little overhead or planning as possible.
- You need to develop an application as fast as possible.
- You can't relationally store data affordably.
- You need to be able to scale your database in size quickly.
- You anticipate your application evolving rapidly.

Unit – 1 Concept of NoSQL : MongoDB

MongoDB is a document oriented database. It is a key feature of MongoDB. It offers a document oriented storage.

MongoDB, the most popular NoSQL database, is an open-source document-oriented database. It means that MongoDB isn't based on the table-like relational database structure but provides an altogether different mechanism for storage and retrieval of data.

Relational Database Management System(RDBMS) is **not the correct choice when it comes to handling big data by the virtue of their design since they are not horizontally scalable**. If the database runs on a single server, then it will reach a scaling limit. NoSQL databases are more scalable and provide superior performance. MongoDB is such a NoSQL database that scales by adding more and more servers and increases productivity with its flexible document model.

MongoDB Advantages

- MongoDB is schema less. It is a document database in which one collection holds different documents.
- There may be difference between number of fields, content and size of the document from one to other.
- Structure of a single object is clear in MongoDB.
- There are no complex joins in MongoDB.
- MongoDB provides the facility of deep query because it supports a powerful dynamic query on documents.
- It is very easy to scale.
- It uses internal memory for storing working sets and this is the reason of its fast access.

Distinctive features of MongoDB

- Easy to use
- Light Weight
- Extremely faster than RDBMS

➤ MongoDB data types

Unit – 1 Concept of NoSQL : MongoDB

Data Types	Description
String	String is the most commonly used datatype. It is used to store data. A string must be UTF 8 valid in mongodb.
Integer	Integer is used to store the numeric value. It can be 32 bit or 64 bit depending on the server you are using.
Boolean	This datatype is used to store boolean values. It just shows YES/NO values.
Double	Double datatype stores floating point values.
Min/Max Keys	This datatype compare a value against the lowest and highest bson elements.
Arrays	This datatype is used to store a list or multiple values into a single key.
Object	Object datatype is used for embedded documents.
Null	It is used to store null values.
Symbol	It is generally used for languages that use a specific type.
Date	This datatype stores the current date or time in unix time format. It makes you possible to specify your own date time by creating object of date and pass the value of date, month, year into it.

➤ Database creation and dropping database

Creation of Database in MongoDB

There is no **create database** command in MongoDB. Actually, MongoDB do not provide any command to create database.

It may be look like a weird concept, if you are from traditional SQL background where you need to create a database, table and insert values in the table manually.

Here, in MongoDB you don't need to create a database manually because MongoDB will create it automatically when you save the value into the defined collection at first time.

You also don't need to mention what you want to create, it will be automatically created at the time you save the value into the defined collection.

Unit – 1 Concept of NoSQL : MongoDB

How and when to create database ?

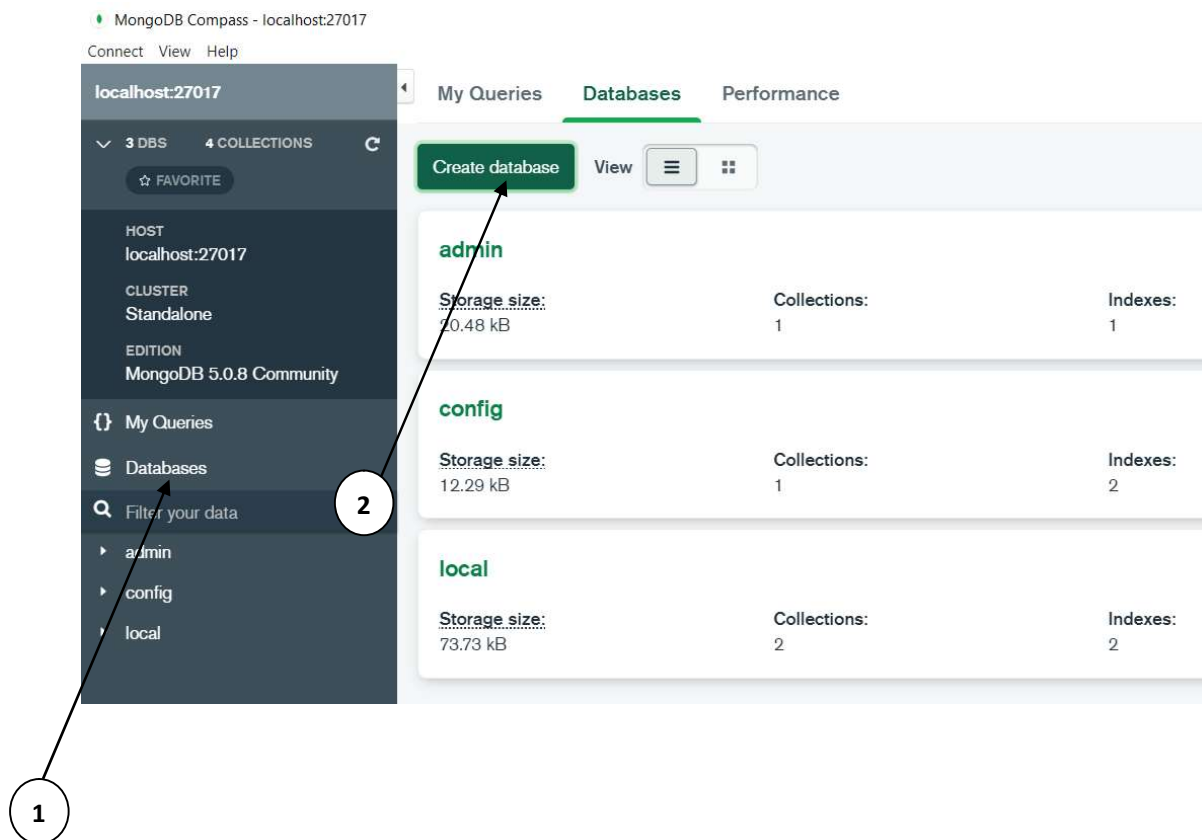
If there is no existing database, the following command is used to create a new database.

Syntax:

use DATABASE_NAME

If the database already exists, it will return the existing database.

For example, we are going to create database studentdb using user interface in MongoDB.



1. Click on Database option.
2. Click on create database.
3. One dialog box will appear. It asked you to enter database name and collection name in particular database.
4. Enter database name “studntdb” and collection name “sem1” as shown below.
5. Click on **create database** button.
6. Your database is created and located at left hand side of the screen.

Unit – 1 Concept of NoSQL : MongoDB

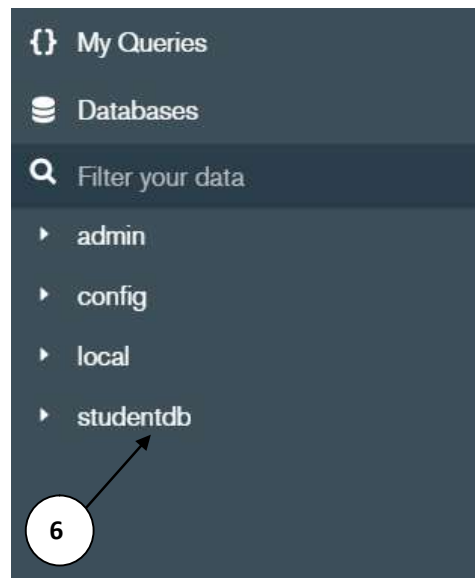
Create Database


Database Name
studentdb

Collection Name
sem1

> Advanced Collection Options (e.g. Time-Series, Capped, Clustered collections)

Cancel Create Database



Now to do this things using command line, click the arrow button  given below on the screen. Command line screen will appear. Type the following command.

Unit – 1 Concept of NoSQL : MongoDB

```
>_MONGOSH
> use studentinfo
< 'switched to db studentinfo'
studentinfo>
```

Database is created using command line, named as studentinfo.

How to create collection ?

In MongoDB, `db.createCollection(name, options)` is used to create collection. But usually you don't need to create collection. MongoDB creates collection automatically when you insert some documents. First see how to create collection:

Syntax:

`db.createCollection(name, options)`

Here,

Name: is a string type, specifies the name of the collection to be created.

Options: is a document type, specifies the memory size and indexing of the collection. It is an optional parameter.

Following is the list of options that can be used.

Field	Type	Description
Capped	Boolean	(Optional) If it is set to true, enables a capped collection. Capped collection is a fixed size collection that automatically overwrites its oldest entries when it reaches its maximum size. If you specify true, you need to specify size parameter also.
AutoIndexID	Boolean	(Optional) If it is set to true, automatically create index on ID field. Its default value is false.
Size	Number	(Optional) It specifies a maximum size in bytes for a capped collection. If capped is true, then you need to specify this field also.
Max	Number	(Optional) It specifies the maximum number of documents allowed

Unit – 1 Concept of NoSQL : MongoDB

		in the capped collection.
--	--	---------------------------

During the creation of database **studentdb**, we have created a collection named as **sem1**. This creation is done using user interface. Now we will create collection name as **sem2**. Type the following command into command window,

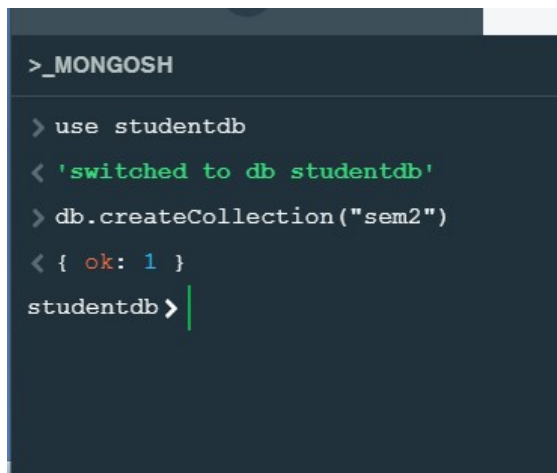
Use studentdb

Output is shown in below screenshot.

To create collection, type the following command,

```
Db.createcollection("sem2")
```

Sem2 collection is created.



```
>_MONGOSH
> use studentdb
< 'switched to db studentdb'
> db.createCollection("sem2")
< { ok: 1 }
studentdb> |
```

To **check the created collection**, use the command "show collections".

```
> _MONGOSH
> use studentdb
< 'switched to db studentdb'
> db.createCollection("sem2")
< { ok: 1 }
> show collections
< sem1
   sem2
studentdb> |
```

Collections listed above.

How to drop collection ?

In MongoDB, `db.collection.drop()` method is used to drop a collection from a database. It completely removes a collection from the database and does not leave any indexes associated with the dropped collections.

The `db.collection.drop()` method does not take any argument and produce an error when it is called with an argument. This method removes all the indexes associated with the dropped collection.

Syntax:

`db.COLLECTION_NAME.drop()`

use the database from which you want to remove the collection. Follow these commands.

```
> use studentdb
< 'switched to db studentdb'
> show collections
< sem1
   sem2
> db.sem2.drop()
< true
studentdb> |
```

Now again type show collections command,

```
> show collections
< sem1
   sem2
> db.sem2.drop()
< true
> show collections
< sem1
studentdb> |
```

How to drop database ?

The **drop Database** command is used to drop a database. It also deletes the associated data files. It operates on the current database.

Syntax:

`db.dropDatabase()`

This syntax will delete the selected database. In the case you have not selected any database, it will delete default "test" database. See the following screenshot.

```
> _MONGOSH
> db.dropDatabase()
< { ok: 1, dropped: 'test' }
test>
```

To **check the database list**, use the command `show dbs`:

Syntax: `show databases`

Unit – 1 Concept of NoSQL : MongoDB

```
> _MONGOSH
> db.dropDatabase()
< { ok: 1, dropped: 'test' }
> show databases
< admin      40.00 KiB
  config     108.00 KiB
  local      128.00 KiB
  studentdb   8.00 KiB
test>
```

List of databases are display on the screen. Now we want to remove studentdb database from the list. First check that how many collections are there in studentdb. Then remove the database. Consider following commands to perform the deletion of database.

```
> show databases
< admin      40.00 KiB
  config     36.00 KiB
  local      128.00 KiB
  studentdb   8.00 KiB
> use studentdb
< 'switched to db studentdb'
studentdb> |
```

Command:

Show databases ->> display all the database.

Use <<database_name>> -> Current database in use.

```
> use studentdb
< 'switched to db studentdb'
> db.dropDatabase()
< { ok: 1, dropped: 'studentdb' }
> show dbs
< admin      40.00 KiB
  config     72.00 KiB
  local      128.00 KiB
studentdb>
```

Unit – 1 Concept of NoSQL : MongoDB

Db.dropdatabase() → remove database.

MongoDB Query and Projection Operator

The MongoDB query operator includes comparison, logical, element, evaluation, Geospatial, array, bitwise, and comment operators.

To understand the operators, we will consider the collection toys.

MongoDB Comparison Operators

\$eq

The \$eq specifies the equality condition. It matches documents where the value of a field equals the specified value.

Syntax:

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

Example:

```
db.toys.find ( { price: { $eq: 300 } } )
```

The above example queries the toys collection to select all documents where the value of the price field equals 300.

\$gt

The \$gt chooses a document where the value of the field is greater than the specified value.

Syntax:

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

Example:

```
db.toys.find ( { price: { $gt: 200 } } )
```

\$gte

The \$gte choose the documents where the field value is greater than or equal to a specified value.

Unit – 1 Concept of NoSQL : MongoDB

Syntax

{ field: { \$gte: value } }

Example:

```
db.toys.find ( { price: { $gte: 250 } } )
```

\$in

The \$in operator choose the documents where the value of a field equals any value in the specified array.

Syntax:

{ field: { \$in: [<value1>, <value2>,] } }

Example:

```
db.toys.find( { price: { $in: [100, 200] } } )
```

\$lt

The \$lt operator chooses the documents where the value of the field is less than the specified value.

Syntax:

{ field: { \$lt: value } }

Example:

```
db.toys.find ( { price: { $lt: 20 } } )
```

\$lte

The \$lte operator chooses the documents where the field value is less than or equal to a specified value.

Syntax:

{ field: { \$lte: value } }

Example:

Unit – 1 Concept of NoSQL : MongoDB

```
db.toys.find ( { price: { $lte: 250 } } )
```

\$ne

The \$ne operator chooses the documents where the field value is not equal to the specified value.

Syntax:

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

Example:

```
db.toys.find ( { price: { $ne: 500 } } )
```

\$nin

The \$nin operator chooses the documents where the field value is not in the specified array or does not exist.

Syntax:

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

Example:

```
Db.toys.find ( { price: { $nin: [ 50, 150, 200 ] } } )
```

MongoDB Logical Operator

\$and

The \$and operator works as a logical AND operation on an array. The array should be of one or more expressions and chooses the documents that satisfy all the expressions in the array.

Syntax:

```
{ $and: [ { <exp1> }, { <exp2> }, ....] }
```

Example:

```
db.toys.find ( { $and: [ { price: { $ne: 500 } }, { price: { $exists: true } } ] } )
```

\$not

Unit – 1 Concept of NoSQL : MongoDB

The \$not operator works as a logical NOT on the specified expression and chooses the documents that are not related to the expression.

Syntax:

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

Example:

```
db.toys.find ( { price: { $not: { $gt: 200 } } } )
```

\$nor

The \$nor operator works as logical NOR on an array of one or more query expression and chooses the documents that fail all the query expression in the array.

Syntax:

```
{ $nor: [ { <expression1> } , { <expresion2> } , ..... ] }
```

Example:

```
db.toys.find ( { $nor: [ { price: 200 } , { sale: true } ] } )
```

\$or

It works as a logical OR operation on an array of two or more expressions and chooses documents that meet the expectation at least one of the expressions.

Syntax:

```
{ $or: [ { <exp_1> } , { <exp_2> } , ... , { <exp_n> } ] }
```

Example:

```
db.toys.find ( { $or: [ { quantity: { $lt: 200 } } , { price: 500 } ] } )
```

MongoDB Element Operator

\$exists

Unit – 1 Concept of NoSQL : MongoDB

The exists operator matches the documents that contain the field when Boolean is true. It also matches the document where the field value is null.

Syntax:

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

Example:

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

\$type

The type operator chooses documents where the value of the field is an instance of the specified BSON type.

Syntax:

```
{ field: { $type: <BSON type> } }
```

Example:

```
db.toys.find ( { "toyid" : { $type : 2 } } );
```

MongoDB Evaluation Operator

\$expr

The expr operator allows the use of aggregation expressions within the query language.

Syntax:

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

Example:

```
db.store.find( { $expr: { $gt: [ "$product", "price" ] } } )
```

\$jsonSchema

It matches the documents that satisfy the specified JSON Schema.

Syntax:

Unit – 1 Concept of NoSQL : MongoDB

{ \$jsonSchema: <JSON **schema** object> }

\$mod

The mod operator selects the document where the value of a field is divided by a divisor has the specified remainder.

Syntax:

{ field: { \$mod: [divisor, remainder] } }

Example:

```
db.toys.find ( { qty: { $mod: [ 200, 0] } } )
```

\$regex

It provides regular expression abilities for pattern matching strings in queries. The MongoDB uses regular expressions that are compatible with Perl.

Syntax:

{ <field>: /pattern/<options> }

Example:

```
db.toys.find( { price: { $regex: /789$/ } } )
```

\$text

The \$text operator searches a text on the content of the field, indexed with a text index.

Syntax:

```
{
  $text:
  {
    $search: <string>,
    $language: <string>,
    $caseSensitive: <boolean>,

```

```
$diacriticSensitive: <boolean>
}
}
```

Example:

```
db.toys.find( { $text: { $search: "toytrain" } } )
```

\$where

The "where" operator is used for passing either a string containing a JavaScript expression or a full JavaScript function to the query system.

Example:

```
db.toys.find( { $where: function() {

    return (hex_md5(this.name)=="9b53e667f30cd329dca1ec9e6a8")

} } );
```

MongoDB Geospatial Operator

\$geoIntersects

It selects only those documents whose geospatial data intersects with the given GeoJSON object.

Syntax:

```
{
  <location field>: {
    $geoIntersects: {
      $geometry: {
        type: "<object type>",
        coordinates: [ <coordinates> ]
      }
    }
  }
}
```

Example:

```
db.places.find(
```

```
{
  loc: {
    $geoIntersects: {
      $geometry: {
        type: "Triangle" ,
        coordinates: [
          [ [ 0, 0 ], [ 3, 6 ], [ 6, 1 ] ]
        ]
      }
    }
  }
}
```

\$geoWithin

The geoWithin operator chooses the document with geospatial data that exists entirely within a specified shape.

Syntax:

```
{
  <location field>: {
    $geoWithin: {
      $geometry: {
        type: <"Triangle" or "Rectangle"> ,
        coordinates: [ <coordinates> ]
      }
    }
  }
}
```

\$near

The near operator defines a point for which a geospatial query returns the documents from close to far.

Syntax:

```
{
  <location field>: {
    $near: {
      $geometry: {
        type: "Point" ,
        coordinates: [ <longitude> , <latitude> ]
      },

```

```
$maxDistance: <distance in meters>,  
$minDistance: <distance in meters>  
}  
}
```

Example:

```
db.places.find(  
{  
  location:  
    { $near :  
      {  
        $geometry: { type: "Point", coordinates: [ -73.9667, 40.78 ] },  
        $minDistance: 1000,  
        $maxDistance: 5000  
      }  
    }  
}  
)
```

\$nearSphere

The nearsphere operator specifies a point for which the geospatial query returns the document from nearest to farthest.

Syntax:

```
{  
  $nearSphere: [ <x>, <y> ],  
  $minDistance: <distance in radians>,  
  $maxDistance: <distance in radians>  
}
```

Example:

```
db.legacyPlaces.find(  
  { location : { $nearSphere : [ -73.9667, 40.78 ], $maxDistance: 0.10 } }  
)
```

\$all

It chooses the document where the value of a field is an array that contains all the specified elements.

Syntax:

Unit – 1 Concept of NoSQL : MongoDB

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

Example:

```
db.toys.find( { tags: { $all: [ "toytrain", "cube", "rope" ] } } )
```

\$elemMatch

The operator relates documents that contain an array field with at least one element that matches with all the given query criteria.

Syntax:

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

Example:

```
db.books.find(
  { results: { $elemMatch: { $gte: 500, $lt: 400 } } }
)
```

\$size

It selects any array with the number of the element specified by the argument.

Syntax:

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

MongoDB Bitwise Operator

\$bitsAllClear

It matches the documents where all the bit positions given by the query are clear in field.

Syntax:

{ <field>: { \$bitsAllClear: <numeric bitmask> } }

Example:

```
db.inventory.find( { a: { $bitsAllClear: [ 1, 5 ] } } )
```

\$bitsAllSet

Unit – 1 Concept of NoSQL : MongoDB

The `bitallset` operator matches the documents where all the bit positions given by the query are set in the field.

Syntax:

```
{ <field>: { $bitsAllSet: <numeric bitmask> } }
```

Example:

```
db.inventory.find( { a: { $bitsAllClear: [ 1, 5 ] } } )
```

`$bitsAnyClear`

The `bitAnyClear` operator matches the document where any bit of positions given by the query is clear in the field.

Syntax:

```
{ <field>: { $bitsAnyClear: <numeric bitmask> } }
```

Example:

```
db.inventory.find( { a: { $bitsAnyClear: [ 5, 10 ] } } )
```

`$bitsAnySet`

It matches the document where any of the bit positions given by the query are set in the field.

Syntax:

```
{ <field>: { $bitsAnySet: <numeric bitmask> } }
```

Example:

```
db.inventory.find( { a: { $bitsAnySet: [ 1, 5 ] } } )
```

MongoDB comments operator

`$comment`

The `$comment` operator associates a comment to any expression taking a query predicate.

Syntax:

```
db.inventory.find( { <query>, $comment: <comment> } )
```

Example:

```
db.inventory.find(
{
  x: { $mod: [ 1, 0 ] },
  $comment: "Find Odd values."
}
```

MongoDB Projection Operator

\$

The \$ operator limits the contents of an array from the query results to contain only the first element matching the query document.

Syntax:

```
db.books.find( { <array>: <value> ... },
  { "<array>.$": 1 } )
db.books.find( { <array.field>: <value> ... },
  { "<array>.$": 1 } )
```

\$elemMatch

The content of the array field made limited using this operator from the query result to contain only the first element matching the element \$elemMatch condition.

Syntax:

```
db.library.find( { bookcode: "63109" },
  { students: { $elemMatch: { roll: 102 } } } )
```

\$meta

The meta operator returns the result for each matching document where the metadata associated with the query.

Syntax:

```
{ $meta: <metaDataKeyword> }
```

Example:

```
db.books.find(
```

```
<query>,  
{ score: { $meta: "textScore" } }
```

\$slice

It controls the number of values in an array that a query returns.

Syntax:

```
db.books.find( { field: value }, { array: { $slice: count } } );
```

Example:

```
db.books.find( {}, { comments: { $slice: [ 200, 100 ] } } )
```

MongoDB Update Operator

The following modifiers are available to update operations. For example - in `db.collection.update()` and `db.collection.findAndModify()`.

Defines the operator expression in the document of the form:

1. {
2. <operator1>: { <field1>: <value1>, ... },
3. <operator2>: { <field2>: <value2>, ... },
4. }

Field Operator

\$currentDate

It updates the elements of a field to the current date, either as a Date or a timestamp. The default data type of this operator is the date.

Syntax:

1. { \$currentDate: { <field1>: <typeSpecification1>, ... } }

Example:

1. db.books.insertOne(

Unit – 1 Concept of NoSQL : MongoDB

2. { _id: 1, status: "a", lastModified: purchaseDate("2013-10-02T01:11:18.965Z") }
3.)

\$inc

It increases a field by the specified value.

Syntax:

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

Example:

1. {
2. _id: 000438,
3. sku: "MongoDB",
4. quantity: 1,
5. metrics: {
6. orders: 2,
7. ratings: 3.5
8. }
9. }

\$min

It changes the value of the field to a specified value if the specified value is less than the current value of the field.

Syntax:

1. { \$min: { <field1>: <value1>, ... } }

Example:

1. { _id: 0021, highprice: 800, lowprice: 200 }
2. db.books.update({ _id: 0021 }, { \$min: { highprice: 500 } })

\$max

Unit – 1 Concept of NoSQL : MongoDB

It changes the value of the field to a specified value if the specified value is greater than the current value of the field.

Syntax:

1. { **\$max**: { <field1>: <value1>, ... } }

Example:

1. { _id: 0021, highprice: 800, lowprice: 200 }
2. db.books.**update**({ _id: 0021 }, { **\$max**: { highprice: 950 } })

\$mul

It multiplies the value of a field by a number.

Syntax:

1. { **\$mul**: { <field1>: <number1>, ... } }

Example:

1. db.books.**update**(
2. { _id: 1 },
3. { **\$mul**: { price: NumberDecimal("180.25"), qty: 2 } }
4.)

\$rename

The rename operator changes the name of a field.

Syntax:

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

Example:

db.books.updateMany({}, { **\$rename**: { "nmae": "name" } })

\$set

The set operator changes the value of a field with the specified value.

Syntax:

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

Example:

1. {
2. _id: 100,
3. sku: "abc123",
4. quantity: 50,
5. instock: true,
6. reorder: false,
7. details: { model: "14Q2", make: "xyz" },
8. tags: ["technical", "non technical"],
9. ratings: [{ by: "ijk", rating: 4 }]

\$setOnInsert

If the upsert is set to true, then it results in an insert of a document, then setOnInsert operator assigns the specified values to the field in the document.

Syntax:

1. db.collection.update(
2. <query>,
3. { \$setOnInsert: { <field1>: <value1>, ... } },
4. { upsert: true }
5.)

\$unset

It removes a specified field.

Syntax:

1. { \$unset: { <field1>: "", ... } }

Example:

1. db.products.update(
2. { sku: "unknown" },
3. { \$unset: { quantity: "", instock: "" } }

Array Operators

\$

We can update an element in an array without explicitly specifying the position of the element.

Syntax:

1. { "<array>.\$" : value }

Example:

1. db.collection.update(
2. { <array>: value ... },
3. { <update operator>: { "<array>.\$" : value } }

\$[]

The positional operator indicates that the update operator should change all the elements in the given array field.

Syntax:

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

Example:

1. db.collection.updateMany(
2. { <query conditions> },
3. { <update operator>: { "<array>.\$[]" : value } }

\$[<identifier>]

It is called a filtered positional operator that identifies the array elements.

Syntax:

Unit – 1 Concept of NoSQL : MongoDB

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

Example:

1. db.collection.updateMany({ <query conditions> },
2. { <update operator>: { "<array>.\$[<identifier>]" : value } },
3. { arrayFilters: [{ <identifier>: <condition> }] })

\$addToSet

It adds an element to an array unless the element is already present, in which case this operator does nothing to that array.

Syntax:

1. { \$addToSet: { <field1>: <value1>, ... } }

Example:

1. db.books.update(
2. { _id: 1 },
3. { \$addToSet: { tags: "MongoDB" } }

\$pop

We can remove the first or last element of an array using the pop operator. We need to pass the value of pop as -1 to remove the first element of an array and 1 to remove the last element in an array.

Syntax:

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

Example:

1. db.books.update({ _id: 1 }, { \$pop: { mongoDB: -1 } })

\$pull

Using a pull operator, we can remove all the instances of a value in an array that matches the specified condition.

Unit – 1 Concept of NoSQL : MongoDB

Syntax:

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

Example:

1. db.books.update({ }, { \$pull: { Development: { \$in:["Java", "RDBMS"] }, Tech: "Cybersecurity" } },
2. { multi: true }
3.)

\$push

It appends a specified value to an array.

Syntax:

1. { \$push: { <field1>: <value1>, ... } }

Example:

1. db.students.update({ _id: 9 }, { \$push: { scores: 91 } })

\$pullAll

We can remove all instances of the specified value from an existing array using the pullAll operator. It removes elements that match the listed value.

Syntax:

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

Example:

1. db.survey.update({ _id: 1 }, { \$pullAll: { scores: [0, 5] } })

Modifiers

Seach

It is used with the \$addToSet operator and the \$push operator. It is used with the addToSet operator to add multiple values to an array if the value does not exist in the field.

Unit – 1 Concept of NoSQL : MongoDB

Syntax:

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

It is used with the push operator to append multiple values to an array.

Syntax:

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

Example:

1. db.students.update({ name: "Akki" }, { \$push: { scores: { \$each: [90, 92, 85] } } })

\$position

It specifies the location where the push operator inserts elements inside an array.

Syntax:

1. {
2. \$push: {
3. <field>: {
4. \$each: [<value1>, <value2>, ...],
5. \$position: <num>
6. }
7. }

Example:

1. db.students.update(
2. { _id: 1 },
3. {
4. \$push: {
5. scores: {
6. \$each: [50, 60, 70],
7. \$position: 0
8. }

9. }
10. }
11.)

\$slice

This modifier is used to limit the number of array elements during the push operation.

Syntax:

1. {
2. \$push: {
3. <field>: {
4. \$each: [<value1>, <value2>, ...],
5. \$slice: <num>
6. }
7. }

Example:

1. db.students.update(
2. { _id: 1 },
3. {
4. \$push: {
5. scores: {
6. \$each: [80, 78, 86],
7. \$slice: -5
8. }
9. }
10. }
11.)

\$sort

The sort modifier arranges the values of an array during the push operation.

Syntax:

1. {
2. \$push: {
3. <field>: {
4. \$each: [<value1>, <value2>, ...],
5. \$sort: <sort specification>
6. }
7. }

Example:

1. db.students.update(
2. { _id: 1 },
3. {
4. \$push: {
5. quizzes: {
6. \$each: [{ id: 3, score: 8 }, { id: 4, score: 7 }, { id: 5, score: 6 }],
7. \$sort: { score: 1 }
8. }
9. }
10. }
11.)

Bitwise Operator

\$bit

The bit operator updates a field using a bitwise operation. It supports bitwise AND, bitwise OR, and bitwise XOR operations.

Syntax:

1. { \$bit: { <field>: { <and|or|xor>: <int> } } }

Example:

1. db.books.update({ _id: 1 }, { \$bit: { expdata: { and: price(100) } } }
2.)

Aggregation Pipeline Operators

The aggregation pipeline operators construct expressions for use in the aggregation pipeline stages. The following are the list of Aggregation Pipeline Operators.

Arithmetic Expression Operators

It is used to perform arithmetic operations on numbers. Some arithmetic expression also supports data arithmetic.

\$abs

The abs operator returns the absolute value of a number.

Syntax:

1. { **\$abs**: <number> }

Example:

1. db.score.aggregate([
2. {
3. \$school: { marks: { **\$abs**: { \$subtract: ["\$max", "\$min"] } } }
4. }
5.])

\$add

It adds two or more numbers and a date. If one of the arguments is a date, then the date treats the other argument as milliseconds to add to the date.

Syntax:

1. { **\$add**: [<expression1>, <expression2>, ...] }

Example:

1. db.books.aggregate(
2. [
3. { \$project: { item: 1, total: { **\$add**: ["\$price", "\$tax"] } } }
4.]
5.)

Unit – 1 Concept of NoSQL : MongoDB

\$ceil

The ceil operator returns the smallest integer that is greater than or equal to the specified number.

Syntax:

1. { \$ceil: <number> }

Example:

1. db.samples.aggregate([{ \$project: { value: 1, ceilingValue: { \$ceil: "\$value" } } }])

\$divide

It divides one or more numbers by another and returns the result.

Syntax:

1. { \$divide: [<expression1>, <expression2>] }

Example:

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

\$exp

The exp operator is used to raise Euler's number to the specified exponent and returns the result.

Syntax:

1. { \$exp: <exponent> }

Example:

1. db.accounts.aggregate([{ \$project: { effectiveRate: { \$subtract: [{ \$exp: "\$rate" }, 1] } } }])

\$floor

The floor operator returns the greatest integer less than or equal to the specified number.

Syntax:

Unit – 1 Concept of NoSQL : MongoDB

1. { \$floor: <number> }

Example:

1. db.samples.aggregate([{ \$project: { value: 1, floorValue: { \$floor: "\$value" } } }])

\$ln

The ln operator calculates the natural logarithm of a number and returns the result as a double.

Syntax:

1. { \$ln: <number> }

Example:

1. db.sales.aggregate([{ \$project: { x: "\$year", y: { \$ln: "\$sales" } } }])

\$log

The log operator calculates the log of a number for the specified base and returns the result as double.

Syntax:

1. { \$log: [<number>, <base>] }

Example:

1. db.examples.aggregate([
2. { \$project: { bitsNeeded:
3. {
4. \$floor: { \$add: [1, { \$log: ["\$positiveInt", 2] } }] }
5. }
6.])

\$log10

The log10 operator calculates the log base 10 of a number and returns the result as a double.

Syntax:

Unit – 1 Concept of NoSQL : MongoDB

1. { \$log10: <number> }

Example:

1. db.samples.aggregate([{ \$project: { pH: { \$multiply: [-1, { \$log10: "\$H3O" }] } } }])

\$mod

The mod operator divides one number with another and returns the remainder.

Syntax:

1. { \$mod: [<expression1>, <expression2>] }

Example:

1. db.planning.aggregate(
2. [
3. { \$project: { remainder: { \$mod: ["\$hours", "\$tasks"] } } }
4.]
5.)

\$multiply

The multiply operator gives the product of two or more numbers.

Syntax:

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

Example:

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

\$pow

The pow operator raises the number to the given exponent and returns the result.

Syntax:

1. { \$pow: [<number>, <exponent>] }

Unit – 1 Concept of NoSQL : MongoDB

Example:

1. `db.quizzes.aggregate([{ $project: { variance: { $pow: [{ $stdDevPop: "$scores.score" }, 2] } } }])`

\$round

The round operator rounds a number to a whole integer or a specified decimal place.

Syntax:

1. `{ $round : [<number>, <place>] }`

Example:

1. `db.samples.aggregate([{ $project: { roundedValue: { $round: ["$value", 1] } } }])`

\$sqrt

The sqrt operator returns the square root of a positive number as double.

Syntax:

1. `{ $sqrt: <number> }`

Example:

1. `db.points.aggregate([`
2. `{`
3. `$project: {`
4. `distance: {`
5. `$sqrt: {`
6. `$add: [`
7. `{ $pow: [{ $subtract: ["$p2.y", "$p1.y"] }, 2] },`
8. `{ $pow: [{ $subtract: ["$p2.x", "$p1.x"] }, 2] }`
9. `]`
10. `}`
11. `}`
12. `}`

13. }

14.])

\$subtract

The subtract operator subtracts two or more numbers to return the difference of the number.

Syntax:

1. { \$subtract: [<expression1>, <expression2>] }

Example:

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

\$trunc

The trunc command deletes the data from the specified decimal place.

Syntax:

1. { \$trunc : [<number>, <place>] }

Example:

1. db.samples.aggregate([{ \$project: { truncatedValue: { \$trunc: ["\$value", 1] } } }])

Array Expression Operator

\$arrayElemAt

It returns the element at the specified array index.

Syntax:

1. { \$arrayElemAt: [<array>, <idx>] }

Example:

1. db.users.aggregate([

2. {

```
3.   $project:
4.   {
5.     name: 1,
6.     first: { $arrayElemAt: [ "$favorites", 0 ] },
7.     last: { $arrayElemAt: [ "$favorites", -1 ] }
8.   }
9. }
10. )
```

\$arrayToObject

The arrayToObject operator converts an array into a single document.

Syntax:

```
1.   [ [ "item", "abc123"], [ "qty", 25 ] ]
```

Example:

```
1. db.inventory.aggregate(
2.   [
3.     {
4.       $project: {
5.         item: 1,
6.         dimensions: { $arrayToObject: "$dimensions" }
7.       }
8.     }
9.   ]
10. )
```

\$concatArrays

The concatArrays operator joins the array to return the concatenated array.

Syntax:

```
1. { $concatArrays: [ <array1>, <array2>, ... ] }
```

Unit – 1 Concept of NoSQL : MongoDB

Example:

1. db.warehouses.aggregate([
2. { \$project: { items: { \$concatArrays: ["\$instock", "\$ordered"] } } }
3.])

\$filter

The filter operator selects a subset of an array to return the result based on the specified condition.

Syntax:

1. { \$filter: { input: <array>, as: <string>, cond: <expression> } }

Example:

1. db.sales.aggregate([
2. {
3. \$project: {
4. items: {
5. \$filter: {
6. input: "\$items",
7. as: "item",
8. cond: { \$gte: ["\$\$item.price", 100] }
9. }
10. }
11. }
12. }
13.])

\$in

The in operator returns a boolean indicating that the specified value is in the array or not.

Syntax:

1. { \$in: [<expression>, <array expression>] }

Unit – 1 Concept of NoSQL : MongoDB

Example:

```
1. db.fruit.aggregate([
2.   {
3.     $project: {
4.       "store location" : "$location",
5.       "has bananas" : {
6.         $in: [ "bananas", "$in_stock" ]
7.       }
8.     }
9.   }
10. ])
```

\$indexOfArray

The `indexOfArray` operator searches the array for the occurrence of a specified value and returns the array index of the first occurrence.

Syntax:

```
1. { $indexOfArray: [ <array expression>, <search expression>, <start>, <end> ] }
```

Example:

```
1. db.inventory.aggregate(
2.   [
3.     {
4.       $project:
5.         {
6.           index: { $indexOfArray: [ "$items", 2 ] },
7.         }
8.     }
9.   ]
10. )
```

\$isArray

Unit – 1 Concept of NoSQL : MongoDB

It determines and returns a Boolean value if the operand is an Array.

Syntax:

1. { \$isArray: [<expression>] }

Example:

1. db.shop.aggregate([{ \$project: { items: { \$cond:
2. { if: { \$and: [{ \$isArray: "\$instock" }, { \$isArray: "\$ordered" }] }, then: { \$concatArrays: ["\$instock", "\$ordered"] },
3. else: "One or more fields is not an array." } } } })

\$map

The map operator attaches value to each item in an array and returns an array with the applied result.

Syntax:

1. { \$map: { input: <expression>, as: <string>, in: <expression> } }

Example:

1. db.grades.aggregate(
2. [
3. { \$project:
4. { adjustedGrades:
5. {
6. \$map:
7. {
8. input: "\$quizzes",
9. as: "grade",
10. in: { \$add: ["\$grade", 2] }
11. }
12. }
13. }

14. }

15.]

16. }

\$objectToArray

This operator converts a document to an array.

Syntax:

1. { \$objectToArray: <object> }

Example:

```
1. db.inventory.aggregate(  
2.   [  
3.     {  
4.       $project: {  
5.         item: 1,  
6.         dimensions: { $objectToArray: "$dimensions" }  
7.       }  
8.     }  
9.   ]  
10. )
```

\$range

The range operator returns an array whose elements are a generated sequence of numbers.

Syntax:

1. { \$range: [<start>, <end>, <non-zero step>] }

Example:

```
1. db.distances.aggregate([ {  
2.   $project: {  
3.     _id: 0,
```


Unit – 1 Concept of NoSQL : MongoDB

4. city: 1,
5. "Rest stops": { \$range: [0, "\$distance", 25] } }])

\$reduce

The reduce operator applies an expression to each element in an array and combines them into a single value.

Syntax:

1. {
2. \$reduce: {
3. input: <array>,
4. initialValue: <expression>,
5. in: <expression>
6. }
7. }

Example:

1. db.clothes.aggregate([{ \$project: { "discountedPrice": {
2. \$reduce: { input: "\$discounts", initialValue: "\$price", in: { \$multiply: ["\$\$value", { \$subtract: [1, "\$\$this"] }] } } } }])

\$reverseArray

It returns an array with the element in reverse order.

Syntax:

1. { \$reverseArray: <array expression> }

Example:

1. db.users.aggregate([{ \$project: { name: 1,
2. reverseFavorites: { \$reverseArray: "\$favorites" } }])

\$size

The size operator counts and returns the total number of items in an array.

Unit – 1 Concept of NoSQL : MongoDB

Syntax:

1. { \$size: <expression> }

Example:

1. db.books.aggregate([{ \$project: { item: 1, numberOfColors: { \$cond: { if: { \$isArray: "\$colors" }, then: { \$size: "\$colors" }, else: "NA" } } } }])

\$slice

The slice operator results in a subset of an array.

Syntax:

1. { \$slice: [<array>, <n>] }

Example:

1. db.books.aggregate([{ \$project: { name: 1, threeFavorites: { \$slice: ["\$favorites", 3] } } }])
2.])

\$zip

The zip operator transposes an array so that the first element of the output array would be an array containing the first element of the first input array.

Syntax:

1. {
2. \$zip: { inputs: [<array expression1>, ...], useLongestLength: <boolean>, defaults: <array expression> } }

Example:

1. db.matrices.aggregate([{ \$project: { _id: false, transposed: { \$zip: { inputs: [
2. { \$arrayElemAt: ["\$matrix", 0] },
3. { \$arrayElemAt: ["\$matrix", 1] },
4. { \$arrayElemAt: ["\$matrix", 2] },
5.] } } } }])

Unit – 1 Concept of NoSQL : MongoDB

MongoDB limit() Method

In MongoDB, limit() method is used to limit the fields of document that you want to show. Sometimes, you have a lot of fields in collection of your database and have to retrieve only 1 or 2. In such case, limit() method is used.

The MongoDB limit() method is used with find() method.

Syntax:

1. `db.COLLECTION_NAME.find().limit(NUMBER)`

This collection has following fields within it.

1. [
2. {
3. Course: "Java",
4. details: { Duration: "6 months", Trainer: "Sonoo Jaiswal" },
5. Batch: [{ size: "Medium", qty: 25 }],
6. category: "Programming Language"
7. },
8. {
9. Course: ".Net",
10. details: { Duration: "6 months", Trainer: "Prashant Verma" },
11. Batch: [{ size: "Small", qty: 5 }, { size: "Medium", qty: 10 }],
12. category: "Programming Language"
13. },
14. {
15. Course: "Web Designing",
16. details: { Duration: "3 months", Trainer: "Rashmi Desai" },
17. Batch: [{ size: "Small", qty: 5 }, { size: "Large", qty: 10 }],
18. category: "Programming Language"
19. }
20.];

Here, you have to display only one field by using limit() method.

Unit – 1 Concept of NoSQL : MongoDB

Example

1. `db.javatpoint.find().limit(1)`

After the execution, you will get the following result

Output:

Output:

```
{  "_id"   : ObjectId("564dbced8e2c097d15fbb601"),  "Course"  : "Java",
  "details" : {
    "Duration" : "6 months", "Trainer" : "Sonoo Jaiswal" }, "Batch" : [ {
    "size" :
    "Medium", "qty" : 25 } ], "category" : "Programming Language" }
```

MongoDB skip() method

In MongoDB, skip() method is used to skip the document. It is used with find() and limit() methods.

Syntax

1. `db.COLLECTION_NAME.find().limit(NUMBER).skip(NUMBER)`

Scenario:

Consider here also the above discussed example. The collection javatpoint has three documents.

1. [
2. {
3. Course: "Java",
4. details: { Duration: "6 months", Trainer: "Sonoo Jaiswal" },
5. Batch: [{ size: "Medium", qty: 25 }],
6. category: "Programming Language"
7. },
8. {
9. Course: ".Net",
10. details: { Duration: "6 months", Trainer: "Prashant Verma" },
11. Batch: [{ size: "Small", qty: 5 }, { size: "Medium", qty: 10 }],
12. category: "Programming Language"
13. },
14. {
15. Course: "Web Designing",

Unit – 1 Concept of NoSQL : MongoDB

16. details: { Duration: "3 months", Trainer: "Rashmi Desai" },
17. Batch: [{ size: "Small", qty: 5 }, { size: "Large", qty: 10 }],
18. category: "Programming Language"
19. }
20.];

Execute the following query to retrieve only one document and skip 2 documents.

Example

1. db.javatpoint.find().limit(1).skip(2)

After the execution, you will get the following result

Output:

```
{ "_id" : ObjectId("564dbced8e2c097d15fbb603"), "Course" : "Web Designing",
"details" : { "Duration" : "3 months", "Trainer" : "Rashmi Desai" }, "Batch" :
[ { "size" : "Small", "qty" : 5 }, { "size" : "Large", "qty" : 10 } ],
"category" : "Programming Language" }
```

As you can see, the skip() method has skipped first and second documents and shows only third document.

MongoDB sort() method

In MongoDB, sort() method is used to sort the documents in the collection. This method accepts a document containing list of fields along with their sorting order.

The sorting order is specified as 1 or -1.

- 1 is used for ascending order sorting.
- -1 is used for descending order sorting.

Syntax:

1. db.COLLECTION_NAME.find().sort({KEY:1})

This collection has following fields within it.

1. [
2. {

Unit – 1 Concept of NoSQL : MongoDB

3. Course: "Java",
4. details: { Duration: "6 months", Trainer: "Sonoo Jaiswal" },
5. Batch: [{ size: "Medium", qty: 25 }],
6. category: "Programming Language"
7. },
8. {
9. Course: ".Net",
10. details: { Duration: "6 months", Trainer: "Prashant Verma" },
11. Batch: [{ size: "Small", qty: 5 }, { size: "Medium", qty: 10 },],
12. category: "Programming Language"
13. },
14. {
15. Course: "Web Designing",
16. details: { Duration: "3 months", Trainer: "Rashmi Desai" },
17. Batch: [{ size: "Small", qty: 5 }, { size: "Large", qty: 10 }],
18. category: "Programming Language"
19. }
20.];

Execute the following query to display the documents in descending order.

1. `db.javatpoint.find().sort({"Course":-1})`

This will show the documents in descending order.

```
{ "_id" : ObjectId("564dbced8e2c097d15fbb603"), "Course" : "Web Designing",
"details" : { "Duration" : "3 months", "Trainer" : "Rashmi Desai" }, "Batch" :
[ { "size" : "Small", "qty" : 5 }, { "size" : "Large", "qty" : 10 } ],
"category" : "Programming Language" }
{ "_id" : ObjectId("564dbced8e2c097d15fbb601"), "Course" : "Java",
"details" : { "Duration" : "6 months", "Trainer" : "Sonoo Jaiswal" }, "Batch" : [ {
"size" : "Medium", "qty" : 25 } ], "category" : "Programming Language" }
{ "_id" : ObjectId("564dbced8e2c097d15fbb602"), "Course" : ".Net",
"details" : { "Duration" : "6 months", "Trainer" : "Prashant Verma" }, "Batch" : [ {
"size" : "Medium", "qty" : 25 } ], "category" : "Programming Language" }
```

Unit – 1 Concept of NoSQL : MongoDB

```
"Small", "qty" : 5 }, { "size" : "Medium", "qty" : 10 } ], "category" :  
"Programming Language" }
```

Note: By default sort() method displays the documents in ascending order. If you don't specify the sorting preference, it will display documents in ascending order.

Query Modifiers

We have number of meta operation in addition to the MongoDB Query Operators to modify the output or behaviour of a query.

1. `db.collection.find({ <query> })._addSpecial(<option>)`
2. `db.collection.find({ $query: { <query> }, <option> })`

Modifiers

\$comment

The comment operator makes it possible to add a comment to a query in any context.

Syntax:

1. `db.collection.find({ <query> })._addSpecial("$comment", <comment>)`

\$explain

The explain modifier provides details about the query plan. It returns a file that describes the process and indexes used to return the query. It may give useful insight when attempting to optimize a query.

Syntax:

1. `db.example.find({ $query: {}, $explain: 1 })`

\$hint

This operator is deprecated in the mongo shell now. The hint operator attaches the optimizer to use the declared index to fulfill the query. It is also used for testing query performance and indexing strategies.

Syntax:

1. `db.users.find().hint({ age: 1 })`

Unit – 1 Concept of NoSQL : MongoDB

\$max

The max operator is deprecated in mongo shell since v3.2. It defines a max value to specify the exclusive upper bound for the given index in order to limit the results of find ().

Syntax:

1. `db.example.find({ <query> }).max({ field1: <max value>, ... fieldN: <max valueN> })`

\$maxTimeMS

It is also deprecated since v3.2. It defines a cumulative time in ms for processing operations on the cursor.

Syntax:

1. `db.collection.find().maxTimeMS(100)`

\$min

The min operator is used to find a minimum value to declare the included lower bound for a specified index to constrain the results of find ().

Syntax:

1. `db.collection.find({ <query> }).min({ field1: <min value>, ... fieldN: <min valueN> })`

\$orderby

The orderby operator arranges the results of a query in ascending or descending order.

Syntax:

1. `db.collection.find().sort({ age: -1 })`

\$query

It forcefully interprets an expression as a query using MongoDB.

Syntax:

1. `db.collection.find({ $query: { age : 25 } })`
2. `db.collection.find({ age : 25 })`

\$returnKey

Unit – 1 Concept of NoSQL : MongoDB

The return key returns the index fields for the results of the query. If returnkey operator is set to true then the returned documents will not contain any fields.

Syntax:

1. `db.collection.find({ <query> })._addSpecial("$returnKey", true)`
2. `db.collection.find({ $query: { <query> }, $returnKey: true })`

\$showDiskLoc

The showDiskLoc operator adds a field to the resultant documents. The value of the added diskLoc field is a document that contains the disk location details.

Syntax:

1. `"$diskLoc": {`
2. `"file": <int>,`
3. `"offset": <int>`

\$natural

The natural operator is a special sort order operator that arranges the documents using the order of documents on disk using the `cursor.hint ()`.

CRUD: Documents

MongoDB insert documents

In MongoDB, the `db.collection.insert()` method is used to add or insert new documents into a collection in your database.

Upsert

There are also two methods "`db.collection.update()`" method and "`db.collection.save()`" method used for the same purpose. These methods add new documents through an operation called upsert.

Upsert is an operation that performs either an update of existing document or an insert of new document if the document to modify does not exist.

Syntax

1. `>db.COLLECTION_NAME.insert(document)`

Unit – 1 Concept of NoSQL : MongoDB

Let's take an example to demonstrate how to insert a document into a collection. Create a database named **studentdb**. In this example we insert a document into a collection named **coursedetail**. (Method for creation of database and collection is given in beginning of this unit.) This operation will automatically create a collection if the collection does not currently exist.

Example

```
1. db. coursedetail.insert(  
2.   {  
3.     course: "java",  
4.     details: {  
5.       duration: "6 months",  
6.       Trainer: "Sonoo jaiswal"  
7.     },  
8.     Batch: [ { size: "Small", qty: 15 }, { size: "Medium", qty: 25 } ],  
9.     category: "Programming language"  
10.  }  
11.)
```

After the successful insertion of the document, the operation will return a **WriteResult** object with its status.

Output:

```
WriteResult({ "nInserted" : 1 })
```

Here the **nInserted** field specifies the number of documents inserted. If an error is occurred then the **WriteResult** will specify the error information.

Check the inserted documents

If the insertion is successful, you can view the inserted document by the following query.

```
1. >db. coursedetail.find()
```

You will get the inserted document in return.

Output:

```
{ "_id" : ObjectId("56482d3e27e53d2dbc93cef8"), "course" : "java",  
  "details" :  
    { "duration" : "6 months", "Trainer" : "Sonoo jaiswal" }, "Batch" :
```

Unit – 1 Concept of NoSQL : MongoDB

```
[ { "size" : "Small", "qty" : 15 }, { "size" : "Medium", "qty" : 25 } ],  
  "category" : "Programming language" }
```

Note: Here, the ObjectId value is generated by MongoDB itself. It may differ from the one shown.

MongoDB insert multiple documents

If you want to insert multiple documents in a collection, you have to pass an array of documents to the `db.collection.insert()` method.

Create an array of documents

Define a variable named `Allcourses` that hold an array of documents to insert.

```
1. var Allcourses =  
2.   [  
3.     {  
4.       Course: "Java",  
5.       details: { Duration: "6 months", Trainer: "Sonoo Jaiswal" },  
6.       Batch: [ { size: "Medium", qty: 25 } ],  
7.       category: "Programming Language"  
8.     },  
9.     {  
10.      Course: ".Net",  
11.      details: { Duration: "6 months", Trainer: "Prashant Verma" },  
12.      Batch: [ { size: "Small", qty: 5 }, { size: "Medium", qty: 10 } ],  
13.      category: "Programming Language"  
14.    },  
15.    {  
16.      Course: "Web Designing",  
17.      details: { Duration: "3 months", Trainer: "Rashmi Desai" },  
18.      Batch: [ { size: "Small", qty: 5 }, { size: "Large", qty: 10 } ],  
19.      category: "Programming Language"  
20.    }  
21.  ];
```

Unit – 1 Concept of NoSQL : MongoDB

Inserts the documents

Pass this Allcourses array to the db.collection.insert() method to perform a bulk insert.

1. > db. **coursedetail.insert**(Allcourses);

```
BulkWriteResult({
  "writeErrors" : [ ],
  "writeConcernErrors" : [ ],
  "nInserted" : 3,
  "nUpserted" : 0,
  "nMatched" : 0,
  "nModified" : 0,
  "nRemoved" : 0,
  "upserted" : [ ]
})
```

Note: Here the nInserted field specifies the number of documents inserted. In the case of any error during the operation, the **BulkWriteResult** will specify that error.

You can check the inserted documents by using the following query:

1. >db. **coursedetail.find**()

Insert multiple documents with Bulk

In its latest version of MongoDB (MongoDB 2.6) provides a Bulk() API that can be used to perform multiple write operations in bulk.

You should follow these steps to insert a group of documents into a MongoDB collection.

Initialize a bulk operation builder

First initialize a bulk operation builder for the collection javatpoint.

1. var bulk = db. **coursedetail.initializeUnorderedBulkOp**();

This operation returns an unordered operations builder which maintains a list of operations to perform .

Add insert operations to the bulk object

1. bulk.**insert**(
2. {
3. course: "java",
4. details: {

Unit – 1 Concept of NoSQL : MongoDB

5. duration: "6 months",
6. Trainer: "Sonoo jaiswal"
7. },
8. Batch: [{ size: "Small", qty: 15 }, { size: "Medium", qty: 25 }],
9. category: "Programming language"
10. }
11.);

Execute the bulk operation

Call the execute() method on the bulk object to execute the operations in the list.

1. bulk.execute();

After the successful insertion of the documents, this method will return a **BulkWriteResult** object with its status.

```
BulkWriteResult({
  "writeErrors" : [ ],
  "writeConcernErrors" : [ ],
  "nInserted" : 1,
  "nUpserted" : 0,
  "nMatched" : 0,
  "nModified" : 0,
  "nRemoved" : 0,
  "upserted" : [ ]
})
```

Here the nInserted field specifies the number of documents inserted. In the case of any error during the operation, the **BulkWriteResult** will specify that error.

MongoDB update documents

In MongoDB, update() method is used to update or modify the existing documents of a collection.

Syntax:

1. db.COLLECTION_NAME.update(SELECTIOIN_CRITERIA, UPDATED_DATA)

Example

Consider an example which has a collection name javatpoint. Insert the following documents in collection:

Unit – 1 Concept of NoSQL : MongoDB

1. `db. coursedetail.insert(`
2. `{`
3. `course: "java",`
4. `details: {`
5. `duration: "6 months",`
6. `Trainer: "Sonoo jaiswal"`
7. `},`
8. `Batch: [{ size: "Small", qty: 15 }, { size: "Medium", qty: 25 }],`
9. `category: "Programming language"`
10. `}`
11. `)`

After successful insertion, check the documents by following query:

1. `>db. coursedetail.find()`

Output:

```
{ "_id" : ObjectId("56482d3e27e53d2dbc93cef8"), "course" : "java",
  "details" :
  { "duration" : "6 months", "Trainer" : "Sonoo jaiswal" }, "Batch" :
  [ { "size" : "Small", "qty" : 15 }, { "size" : "Medium", "qty" : 25 } ],
  "category" : "Programming language" }
```

Update the existing course "java" into "android":

1. `>db. coursedetail.update({'course':'java'},{$set:{'course':'android'}})`

Check the updated document in the collection:

1. `>db. coursedetail.find()`

Output:

```
{ "_id" : ObjectId("56482d3e27e53d2dbc93cef8"), "course" : "android",
  "details" :
  { "duration" : "6 months", "Trainer" : "Sonoo jaiswal" }, "Batch" :
  [ { "size" : "Small", "qty" : 15 }, { "size" : "Medium", "qty" : 25 } ],
  "category" : "Programming language" }
```

MongoDB Delete documents

Unit – 1 Concept of NoSQL : MongoDB

In MongoDB, the `db.collection.remove()` method is used to delete documents from a collection. The `remove()` method works on two parameters.

1. Deletion criteria: With the use of its syntax you can remove the documents from the collection.

2. JustOne: It removes only one document when set to true or 1.

Syntax:

1. `db.collection_name.remove (DELETION_CRITERIA)`

Remove all documents

If you want to remove all documents from a collection, pass an empty query document `{}` to the `remove()` method. The `remove()` method does not remove the indexes.

Let's take an example to demonstrate the `remove()` method. In this example, we remove all documents from the "javatpoint" collection.

1. `db. coursedetail.remove({})`

Remove all documents that match a condition

If you want to remove a document that match a specific condition, call the `remove()` method with the `<query>` parameter.

The following example will remove all documents from the javatpoint collection where the type field is equal to programming language.

1. `db. coursedetail.remove({ type : "programming language" })`

Remove a single document that match a condition

If you want to remove a single document that match a specific condition, call the `remove()` method with just One parameter set to true or 1.

The following example will remove a single document from the javatpoint collection where the type field is equal to programming language.

1. `db. coursedetail.remove({ type : "programming language" }, 1)`

Summary :

Database

Unit – 1 Concept of NoSQL : MongoDB

- Single collection or more collections
- 1 or more collection is called database

Collections

- Collections do NOT enforce any schema, No join concept.
- We can join multiple collections using Aggregation.
- It's a set of documents
- Can have any number of document
- Documents can have any dynamic schema. They can be same or different.

Documents

- Is a simple key value pair data
- MongoDB valid document example
- Schema can be different for different documents
- Document can have any data type- as long as it is valid MongoDB data type
- User defined schema in MongoDB and they are NOT static or Fixed.
- MongoDB will add a key automatically for each document called “_id”.

```
{
  “_id” : “<unique_value>”,
  “firstname” : ”Vivekanand College”,
  “Lastname” : “For BCA”,
  “Email” : “vivekanandbca@gmail.com”
}, /* document1*/

{
  “_id” : “<unique_value>”,
  “firstname” : ”Vivekanand College”,
  “Lastname” : “For BCA”,
  “Email” : “vivekanandbca@gmail.com”,
  “Address” : “Jahangirpura,surat”
} /*document2*/
```