

# Unit - 05

## **Recap of NoSQL databases :**

What is NoSQL?

NoSQL encompasses a wide variety of different database technologies that were developed in response to the demands presented in building modern applications:

- Developers are working with applications that create massive volumes of new, rapidly changing data types — structured, semi-structured, unstructured and polymorphic data.
- Long gone is the twelve-to-eighteen month waterfall development cycle. Now small teams work in agile sprints, iterating quickly and pushing code every week or two, some even multiple times every day.
- Applications that once served a finite audience are now delivered as services that must be always-on, accessible from many different devices and scaled globally to millions of users.
- Organizations are now turning to scale-out architectures using open source software, commodity servers and cloud computing instead of large monolithic servers and storage infrastructure.

Relational databases were not designed to cope with the scale and agility challenges that face modern applications, nor were they built to take advantage of the commodity storage and processing power available today.

Read our free white paper: Top 5 Considerations When Evaluating NoSQL Databases and learn about:

- Selecting the appropriate data model: document, key-value & wide column, or graph model
- The pros and cons of consistent and eventually consistent systems
- Why idiomatic drivers minimize onboarding time for new developers and simplify application development

Download the White Paper

NoSQL Database Types

- Document databases pair each key with a complex data structure known as a document. Documents can contain many different key-value pairs, or key-array pairs, or even nested documents.
- Graph stores are used to store information about networks of data, such as social connections. Graph stores include Neo4J and Giraph.
- Key-value stores are the simplest NoSQL databases. Every single item in the database is stored as an attribute name (or 'key'), together with its value. Examples of key-value stores are Riak and Berkeley DB. Some key-value stores, such as Redis, allow each value to have a type, such as 'integer', which adds functionality.
- Wide-column stores such as Cassandra and HBase are optimized for queries over large datasets, and store columns of data together, instead of rows.

### The Benefits of NoSQL

When compared to relational databases, NoSQL databases are more scalable and provide superior performance, and their data model addresses several issues that the relational model is not designed to address:

- Large volumes of rapidly changing structured, semi-structured, and unstructured data
- Agile sprints, quick schema iteration, and frequent code pushes
- Object-oriented programming that is easy to use and flexible
- Geographically distributed scale-out architecture instead of expensive, monolithic architecture

### Dynamic Schemas

Relational databases require that schemas be defined before you can add data. For example, you might want to store data about your customers such as phone numbers, first and last name, address, city and state – a SQL database needs to know what you are storing in advance.

This fits poorly with agile development approaches, because each time you complete new features, the schema of your database often needs to change. So if you decide, a few iterations into development, that you'd like to store customers' favorite items in addition to their addresses and phone numbers, you'll need to add that column to the database, and then migrate the entire database to the new schema.

If the database is large, this is a very slow process that involves significant downtime. If you are frequently changing the data your application stores – because you are iterating rapidly – this downtime may also be frequent. There's also no way, using a relational database, to effectively address data that's completely unstructured or unknown in advance.

NoSQL databases are built to allow the insertion of data without a predefined schema. That makes it easy to make significant application changes in real-time, without worrying about service interruptions – which means development is faster, code integration is more reliable, and less database administrator time is needed. Developers have typically had to add application-side code to enforce data quality controls, such as mandating the presence of specific fields, data types or permissible values. More sophisticated NoSQL databases allow validation rules to be applied within the database, allowing users to enforce governance across data, while maintaining the agility benefits of a dynamic schema.

### Auto-sharding

Because of the way they are structured, relational databases usually scale vertically – a single server has to host the entire database to ensure acceptable performance for cross-table joins and transactions. This gets expensive quickly, places limits on scale, and creates a relatively small number of failure points for database infrastructure. The solution to support rapidly growing applications is to scale horizontally, by adding servers instead of concentrating more capacity in a single server.

'Sharding' a database across many server instances can be achieved with SQL databases, but usually is accomplished through SANs and other complex arrangements for making hardware act as a single server. Because the database does not provide this ability natively, development teams take on the work of deploying multiple relational databases across a number of machines. Data is stored in each database instance autonomously. Application code is

developed to distribute the data, distribute queries, and aggregate the results of data across all of the database instances. Additional code must be developed to handle resource failures, to perform joins across the different databases, for data rebalancing, replication, and other

	SQL Databases	NOSQL Databases
Types	One type (SQL database) with minor variations	Many different types including key-value stores, document databases, wide-column stores, and graph databases
Development History	Developed in 1970s to deal with first wave of data storage applications	Developed in late 2000s to deal with limitations of SQL databases, especially scalability, multi-structured data, geo-distribution and agile development sprints
Examples	MySQL, Postgres, Microsoft SQL Server, Oracle Database	MongoDB, Cassandra, HBase, Neo4j
Data Storage Model	Individual records (e.g., 'employees') are stored as rows in tables, with each column storing a specific piece of data about that record (e.g., 'manager,' 'date hired,' etc.), much like a spreadsheet. Related data is stored in separate tables, and then joined together when more complex queries are executed. For example, 'offices' might be stored in one table, and 'employees' in another. When a user wants to find the work address of an employee, the database engine joins the 'employee' and 'office' tables together to get all the information necessary.	Varies based on database type. For example, key-value stores function similarly to SQL databases, but have only two columns ('key' and 'value'), with more complex information sometimes stored as BLOBs within the 'value' columns. Document databases do away with the table-and-row model altogether, storing all relevant data together in single 'document' in JSON, XML, or another format, which can nest values hierarchically.
Schemas	Structure and data types are fixed in advance. To store information about a new data item, the entire database must be altered, during which time the database must be taken offline.	Typically dynamic, with some enforcing data validation rules. Applications can add new fields on the fly, and unlike SQL table rows, dissimilar data can be stored together as necessary. For some databases (e.g., wide-column stores), it is somewhat more challenging to add new fields dynamically.
Scaling	Vertically, meaning a single server must be made increasingly powerful in order to deal with increased demand. It is possible to spread SQL databases over many servers, but significant additional engineering is generally required, and core relational features such as JOINS, referential integrity and transactions are typically lost.	Horizontally, meaning that to add capacity, a database administrator can simply add more commodity servers or cloud instances. The database automatically spreads data across servers as necessary.
Development Model	Mix of open-source (e.g., Postgres, MySQL) and closed source (e.g., Oracle Database)	Open-source

Supports Transactions	Yes, updates can be configured to complete entirely or not at all	In certain circumstances and at certain levels (e.g., document level vs. database level)
Data Manipulation	Specific language using Select, Insert, and Update statements, e.g. SELECT fields FROM table WHERE...	Through object-oriented APIs
Consistency	Can be configured for strong consistency	Depends on product. Some provide strong consistency (e.g., MongoDB, with tunable consistency for reads) whereas others offer eventual consistency (e.g., Cassandra).

requirements. Furthermore, many benefits of the relational database, such as transactional integrity, are compromised or eliminated when employing manual sharding.

NoSQL databases, on the other hand, usually support auto-sharding, meaning that they natively and automatically spread data across an arbitrary number of servers, without requiring the application to even be aware of the composition of the server pool. Data and query load are automatically balanced across servers, and when a server goes down, it can be quickly and transparently replaced with no application disruption.

Cloud computing makes this significantly easier, with providers such as Amazon Web Services providing virtually unlimited capacity on demand, and taking care of all the necessary infrastructure administration tasks. Developers no longer need to construct complex, expensive platforms to support their applications, and can concentrate on writing application code. Commodity servers can provide the same processing and storage capabilities as a single high-end server for a fraction of the price.

### Replication

Most NoSQL databases also support automatic database replication to maintain availability in the event of outages or planned maintenance events. More sophisticated NoSQL databases are fully self-healing, offering automated failover and recovery, as well as the ability to distribute the database across multiple geographic regions to withstand regional failures and enable data localization. Unlike relational databases, NoSQL databases generally have no requirement for separate applications or expensive add-ons to implement replication.

### Integrated Caching

A number of products provide a caching tier for SQL database systems. These systems can improve read performance substantially, but they do not improve write performance, and they add operational complexity to system deployments. If your application is dominated by reads then a distributed cache could be considered, but if your application has just a modest write volume, then a distributed cache may not improve the overall experience of your end users, and will add complexity in managing cache invalidation.

Many NoSQL database technologies have excellent integrated caching capabilities, keeping frequently-used data in system memory as much as possible and removing the need for a separate caching layer. Some NoSQL databases also offer fully managed, integrated in-memory database management layer for workloads demanding the highest throughput and lowest latency.

## NoSQL vs. SQL Summary

### Implementing a NoSQL Database

Often, organizations will begin with a small-scale trial of a NoSQL database in their organization, which makes it possible to develop an understanding of the technology in a low-stakes way. Most NoSQL databases are also open-source, meaning that they can be downloaded, implemented and scaled at little cost. Because development cycles are faster, organizations can also innovate more quickly and deliver superior customer experience at a lower cost.

As you consider alternatives to legacy infrastructures, you may have several motivations: to scale or perform beyond the capabilities of your existing system, identify viable alternatives to expensive proprietary software, or increase the speed and agility of development. When selecting the right database for your business and application, there are five important dimensions to consider.

### Free White Paper

Read our free white paper: *Top 5 Considerations When Evaluating NoSQL Databases* and learn about:

- Selecting the appropriate data model: document, key-value & wide column, or graph model
- The pros and cons of consistent and eventually consistent systems
- Why idiomatic drivers minimize onboarding time for new developers and simplify application development

## **MongoDB – CRUD :**

### **CRUD Operations in MongoDB**

CRUD operations refer to the basic Insert, Read, Update and Delete operations. In the previous chapter, we learnt about how to create a database and drop the database in MongoDB. Now, let us learn how to perform CRUD (Create/Read/Update/Delete) operations in MongoDB. <sup>[1]</sup><sub>SEP</sub>

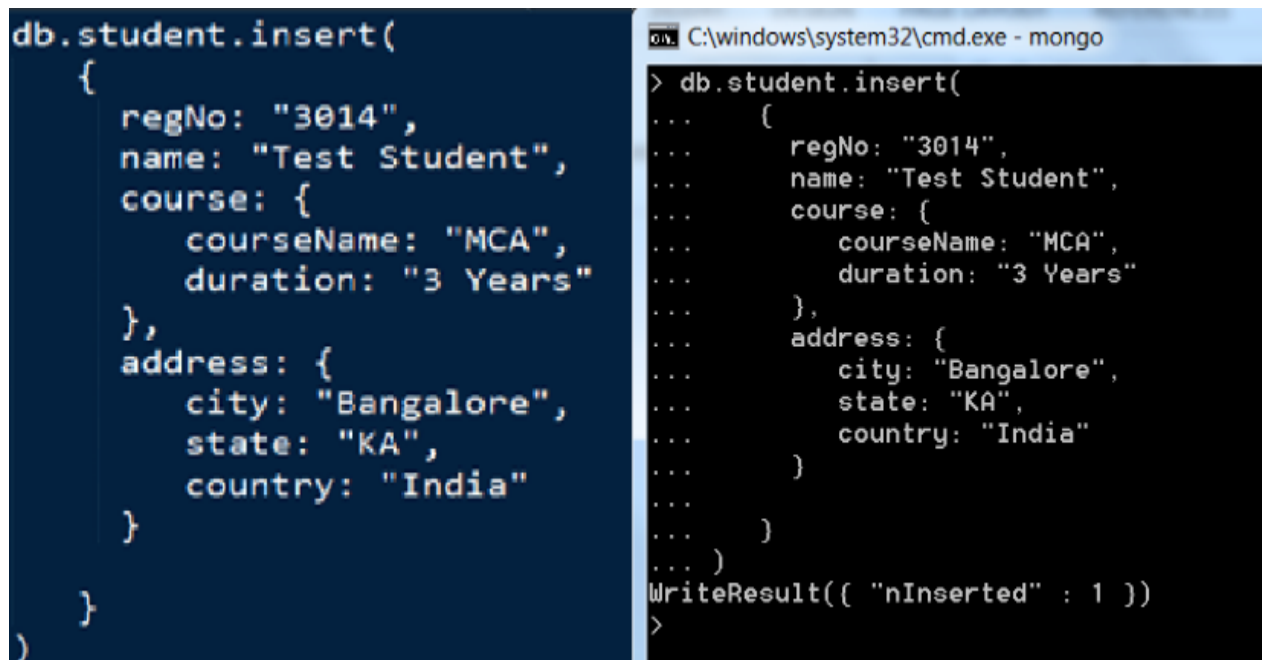
---

#### *Inserting a document into a collection (Create)*

The command `db.collection.insert()` will perform an insert operation into a collection of a document.

Let us insert a document to a student collection. You must be connected to a database for doing any insert. It is done as follows:

```
db.student.insert({  
    regNo: "3014",  
    name: "Test Student",  
    course: {  
        courseName: "MCA",  
        duration: "3 Years"  
    },  
    address: {  
        city: "Bangalore",  
        state: "KA",  
        country: "India"  
    }  
})
```



The image contains two side-by-side screenshots of a MongoDB command prompt window. The left screenshot shows the command `db.student.insert({` followed by a document structure with fields `regNo`, `name`, `course`, and `address`. The right screenshot shows the same command being executed, followed by the output `WriteResult({ "nInserted" : 1 })`, indicating that one document was successfully inserted into the collection.

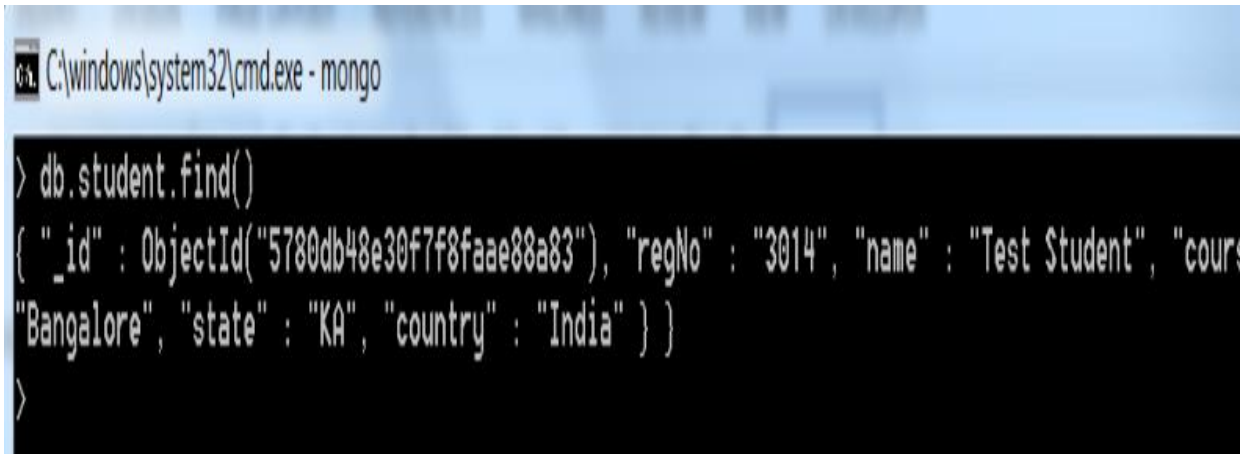
Note that an entry has been made into the collection called student.

---

#### Querying a document from a collection (Read)

To retrieve (Select) the inserted document, run the below command. The `find()` command will retrieve all the documents of the given collection.

```
db.collection_name.find()
```



```
C:\windows\system32\cmd.exe - mongo
> db.student.find()
{ "_id" : ObjectId("5780db48e30f7f8faae88a83"), "regNo" : "3014", "name" : "Test Student", "course" : "Bangalore", "state" : "KA", "country" : "India" } }
```

**NOTE :** Please observe that the record retrieved contains an attribute called `_id` with some unique identifier value called **ObjectId** which acts as a document identifier.

If a record is to be retrieved based on some criteria, the `find()` method should be called passing parameters, then the record will be retrieved based on the attributes specified.

```
db.collection_name.find({"fieldname":"value"})
```

For Example : Let us retrieve the record from the **student** collection where the attribute **regNo** is **3014** and the query for the same is as shown below:

```
db.students.find({"regNo":"3014"})
```

### *Updating a document in a collection (Update)*

In order to update specific field values of a collection in MongoDB, run the below query.

```
db.collection_name.update()
```

`update()` method specified above will take the fieldname and the new value as argument to update a document.

Let us update the attribute **name** of the collection **student** for the document with **regNo** 3014.

```
db.student.update({
    "regNo": "3014"
},
$update:
{
    "name":"Viraj"
})
```

```
db.student.update(  
  {  
    "regNo": "3014"  
  },  
  {  
    $set:  
      {"name": "Keerthi Kumar N"}  
  })
```

You will see the following in the Command Prompt :

```
> db.student.update(  
... {  
... "regNo": "3014"  
... },  
... {  
... $set:  
... ("name": "Keerthi Kumar N")  
... })  
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })  
> db.student.find()  
( { "_id" : ObjectId("57820660e30f7f8faae88a86"), "regNo" : "3014", "name" : "Keerthi Kumar N", "course" : { "courseName" : "MCA", "duration" : "3 Years" }, "address"  
: "Bangalore", "state" : "KA", "country" : "India" } )  
>
```

---

### *Removing an entry from the collection (Delete)*

Let us now look into the deleting an entry from a collection. In order to delete an entry from a collection, run the command as shown below :

**db.collection\_name.remove({"fieldname": "value"})**

For Example : **db.student.remove({"regNo": "3014"})**

```
db.student.remove({"regNo" : "3014"})  
> db.student.remove({"regNo" : "3014"})  
WriteResult({ "nRemoved" : 1 })  
> db.student.find()  
>
```

Note that after running the `remove()` method, the entry has been deleted from the student collection.



## **MongoDB- Arrays :**

### **Array Update Operators**

#### **On this page**

- [Update Operators](#)
- [Update Operator Modifiers](#)

#### **NOTE**

For details on specific operator, including syntax and examples, click on the specific operator to go to its reference page.

#### Update Operators

Name	Description
\$	Acts as a placeholder to update the first element that matches the query condition in an update.
\$addToSet	Adds elements to an array only if they do not already exist in the set.
\$pop	Removes the first or last item of an array.
\$pullAll	Removes all matching values from an array.
\$pull	Removes all array elements that match a specified query.

Name	Description
\$pushAll	<i>Deprecated.</i> Adds several items to an array.
\$push	Adds an item to an array.

## Update Operator Modifiers

Name	Description
\$each	Modifies the \$push and \$addToSet operators to append multiple items for array updates.
\$slice	Modifies the \$push operator to limit the size of updated arrays.
\$sort	Modifies the \$push operator to reorder documents stored in an array.
\$position	Modifies the \$push operator to specify the position in the array to add elements.

**Java Scripts:**

## Write Scripts for the mongo Shell

### On this page

- [Opening New Connections](#)
- [Differences Between Interactive and Scripted mongo](#)
- [Scripting](#)

You can write scripts for the mongo shell in JavaScript that manipulate data in MongoDB or perform administrative operation. For more information about the mongo shell, see the Running .js files via a mongo shell Instance on the Server section for more information about using these mongo script.

This tutorial provides an introduction to writing JavaScript that uses the mongo shell to access MongoDB.

### Opening New Connections

From the mongo shell or from a JavaScript file, you can instantiate database connections using the `Mongo()` constructor:

```
new Mongo()
```

```
new Mongo(<host>)
```

```
new Mongo(<host:port>)
```

Consider the following example that instantiates a new connection to the MongoDB instance running on localhost on the default port and sets the global db variable to myDatabase using the `getDB()` method:

```
conn = new Mongo();
```

```
db = conn.getDB("myDatabase");
```

If connecting to a MongoDB instance that enforces access control, you can use the `db.auth()` method to authenticate.

Additionally, you can use the `connect()` method to connect to the MongoDB instance. The following example connects to the MongoDB instance that is running on localhost with the non-default port 27020 and set the global db variable:

```
db = connect("localhost:27020/myDatabase");
```

### SEE ALSO

## mongo Shell Methods

## Differences Between Interactive and Scripted mongo

When writing scripts for the mongo shell, consider the following:

- To set the db global variable, use the `getDB()` method or the `connect()` method. You can assign the database reference to a variable other than `db`.
- Write operations in the mongo shell use a write concern of `{ w: 1 }` by default. If performing bulk operations, use the `Bulk()` methods. See [Write Method Acknowledgements](#) for more information.

*Changed in version 2.6:* Before MongoDB 2.6, call `db.getLastError()` explicitly to wait for the result of write operations.

- You **cannot** use any shell helper (e.g. use `<dbname>`, `show dbs`, etc.) inside the JavaScript file because they are not valid JavaScript. The following table maps the most common mongo shell helpers to their JavaScript equivalents.

Shell Helpers	JavaScript Equivalents
<code>show dbs</code> , <code>show databases</code>	<code>db.adminCommand('listDatabases')</code>
<code>use &lt;db&gt;</code>	<code>db = db.getSiblingDB('&lt;db&gt;')</code>
<code>show collections</code>	<code>db.getCollectionNames()</code>
<code>show users</code>	<code>db.getUsers()</code>
<code>show roles</code>	<code>db.getRoles({showBuiltinRoles: <b>true</b>})</code>
<code>show log &lt;logname&gt;</code>	<code>db.adminCommand({ 'getLog' : '&lt;logname&gt;' })</code>

Shell Helpers	JavaScript Equivalents
show logs	db.adminCommand({ 'getLog' : '*' })
it	<pre> cursor = db.collection.find() <b>if</b> ( cursor.hasNext() ){     cursor.next(); } </pre>

- In interactive mode, mongo prints the results of operations including the content of all cursors. In scripts, either use the JavaScript print() function or the mongo specific printjson() function which returns formatted JSON.

**EXAMPLE**

To print all items in a result cursor in mongo shell scripts, use the following idiom:

```

cursor = db.collection.find();
while ( cursor.hasNext() ) {
    printjson( cursor.next() );
}

```

## Scripting

From the system prompt, use mongo to evaluate JavaScript.

**--eval option**

Use the *--eval* option to mongo to pass the shell a JavaScript fragment, as in the following:

```
mongo test --eval "printjson(db.getCollectionNames())"
```

This returns the output of db.getCollectionNames() using the mongo shell connected to the mongod or mongos instance running on port 27017 on the localhost interface.

**Execute a JavaScript file**

You can specify a .js file to the mongo shell, and mongo will execute the JavaScript directly. Consider the following example:

```
mongo localhost:27017/test myjsfile.js
```

This operation executes the myjsfile.js script in a mongo shell that connects to the test database on the mongod instance accessible via the localhost interface on port 27017.

Alternately, you can specify the mongodb connection parameters inside of the javascript file using the `Mongo()` constructor. See [Opening New Connections](#) for more information.

You can execute a .js file from within the mongo shell, using the `load()` function, as in the following:

```
load("myjstest.js")
```

This function loads and executes the myjstest.js file.

The `load()` method accepts relative and absolute paths. If the current working directory of the mongo shell is `/data/db`, and the `myjstest.js` resides in the `/data/db/scripts` directory, then the following calls within the mongo shell would be equivalent:

```
load("scripts/myjstest.js")
```

```
load("/data/db/scripts/myjstest.js")
```

## **Cursors :**

In the mongo shell, the primary method for the read operation is the `db.collection.find()` method. This method queries a collection and returns a cursor to the returning documents.

To access the documents, you need to iterate the cursor. However, in the mongo shell, if the returned cursor is not assigned to a variable using the `var` keyword, then the cursor is automatically iterated up to 20 times [\[1\]](#) to print up to the first 20 documents in the results.

For example, in the mongo shell, the following read operation queries the inventory collection for documents that have type equal to 'food' and automatically print up to the first 20 matching documents:

```
db.inventory.find( { type: 'food' } );
```

To manually iterate the cursor to access the documents, see [Iterate a Cursor in the mongo Shell](#).

[\[1\]](#) You can use the `DBQuery.shellBatchSize` to change the number of iteration from the

default value 20. See [Executing Queries](#) for more information.

## Cursor Behaviors

### Closure of Inactive Cursors

By default, the server will automatically close the cursor after 10 minutes of inactivity or if client has exhausted the cursor. To override this behavior, you can specify the `noTimeout` flag in your query using `cursor.addOption()`; however, you should either close the cursor manually or exhaust the cursor. In the mongo shell, you can set the `noTimeout` flag:

```
var myCursor = db.inventory.find().addOption(DBQuery.Option.noTimeout);
```

See your driver documentation for information on setting the `noTimeout` flag. For the mongo shell, see `cursor.addOption()` for a complete list of available cursor flags.

### Cursor Isolation

As a cursor returns documents, other operations may interleave with the query. For the MMAPv1 storage engine, intervening write operations on a document may result in a cursor that returns a document more than once if that document has changed. To handle this situation, see the information on snapshot mode.

### Cursor Batches

The MongoDB server returns the query results in batches. Batch size will not exceed the maximum BSON document size. For most queries, the *first* batch returns 101 documents or just enough documents to exceed 1 megabyte. Subsequent batch size is 4 megabytes. To override the default size of the batch, see `batchSize()` and `limit()`.

For queries that include a sort operation *without* an index, the server must load all the documents in memory to perform the sort before returning any results.

As you iterate through the cursor and reach the end of the returned batch, if there are more results, `cursor.next()` will perform a `getmore` operation to retrieve the next batch. To see how many documents remain in the batch as you iterate the cursor, you can use the `objLeftInBatch()` method, as in the following example:

```
var myCursor = db.inventory.find();
```

```
var myFirstDocument = myCursor.hasNext() ? myCursor.next() : null;
```

```
myCursor.objsLeftInBatch();
```

### Cursor Information

The `db.serverStatus()` method returns a document that includes a `metrics` field. The `metrics` field contains a `cursor` field with the following information:

- number of timed out cursors since the last server restart
- number of open cursors with the option `DBQuery.Option.noTimeout` set to prevent timeout after a period of inactivity
- number of “pinned” open cursors
- total number of open cursors

Consider the following example which calls the `db.serverStatus()` method and accesses the `metrics` field from the results and then the `cursor` field from the `metrics` field:

```
db.serverStatus().metrics.cursor
```

The result is the following document:

```
{
  "timedOut" : <number>
  "open" : {
    "noTimeout" : <number>,
    "pinned" : <number>,
    "total" : <number>
  }
}
```

```
db.serverStatus()
```

Name	Description



Name	Description
<code>cursor.addOption()</code>	Adds special wire protocol flags that modify the behavior of the query.'
<code>cursor.batchSize()</code>	Controls the number of documents MongoDB will return to the client in a single network message.
<code>cursor.close()</code>	Close a cursor and free associated server resources.
<code>cursor.collation()</code>	Specifies the collation for the cursor returned by the <code>db.collection.find()</code> .
<code>cursor.comment()</code>	Attaches a comment to the query to allow for traceability in the logs and the <code>system.profile</code> collection.
<code>cursor.count()</code>	Modifies the cursor to return the number of documents in the result set rather than the documents themselves.
<code>cursor.explain()</code>	Reports on the query execution plan for a cursor.
<code>cursor.forEach()</code>	Applies a JavaScript function for every document in a cursor.

Name	Description
<code>cursor.hasNext()</code>	Returns true if the cursor has documents and can be iterated.
<code>cursor.hint()</code>	Forces MongoDB to use a specific index for a query.
<code>cursor.itcount()</code>	Computes the total number of documents in the cursor client-side by fetching and iterating the result set.
<code>cursor.limit()</code>	Constrains the size of a cursor's result set.
<code>cursor.map()</code>	Applies a function to each document in a cursor and collects the return values in an array.
<code>cursor.maxScan()</code>	Specifies the maximum number of items to scan; documents for collection scans, keys for index scans.
<code>cursor.maxTimeMS()</code>	Specifies a cumulative time limit in milliseconds for processing operations on a cursor.
<code>cursor.max()</code>	Specifies an exclusive upper index bound for a cursor. For use with <code>cursor.hint()</code>

Name	Description
<code>cursor.min()</code>	Specifies an inclusive lower index bound for a cursor. For use with <code>cursor.hint()</code>
<code>cursor.next()</code>	Returns the next document in a cursor.
<code>cursor.noCursorTimeout()</code>	Instructs the server to avoid closing a cursor automatically after a period of inactivity.
<code>cursor.objsLeftInBatch()</code>	Returns the number of documents left in the current cursor batch.
<code>cursor.pretty()</code>	Configures the cursor to display results in an easy-to-read format.
<code>cursor.readConcern()</code>	Specifies a read concern for a <code>find()</code> operation.
<code>cursor.readPref()</code>	Specifies a read preference to a cursor to control how the client directs queries to a replica set.
<code>cursor.returnKey()</code>	Modifies the cursor to return index keys rather than the documents.

Name	Description
<code>cursor.showRecordId()</code>	Adds an internal storage engine ID field to each document returned by the cursor.
<code>cursor.size()</code>	Returns a count of the documents in the cursor after applying <code>skip()</code> and <code>limit()</code> methods.
<code>cursor.skip()</code>	Returns a cursor that begins returning results only after passing or skipping a number of documents.
<code>cursor.snapshot()</code>	Forces the cursor to use the index on the <code>_id</code> field. Ensures that the cursor returns each document, with regards to the value of the <code>_id</code> field, only once.
<code>cursor.sort()</code>	Returns results ordered according to a sort specification.
<code>cursor.tailable()</code>	Marks the cursor as tailable. Only valid for cursors over capped collections.
<code>cursor.toArray()</code>	Returns an array that contains all documents returned by the cursor.

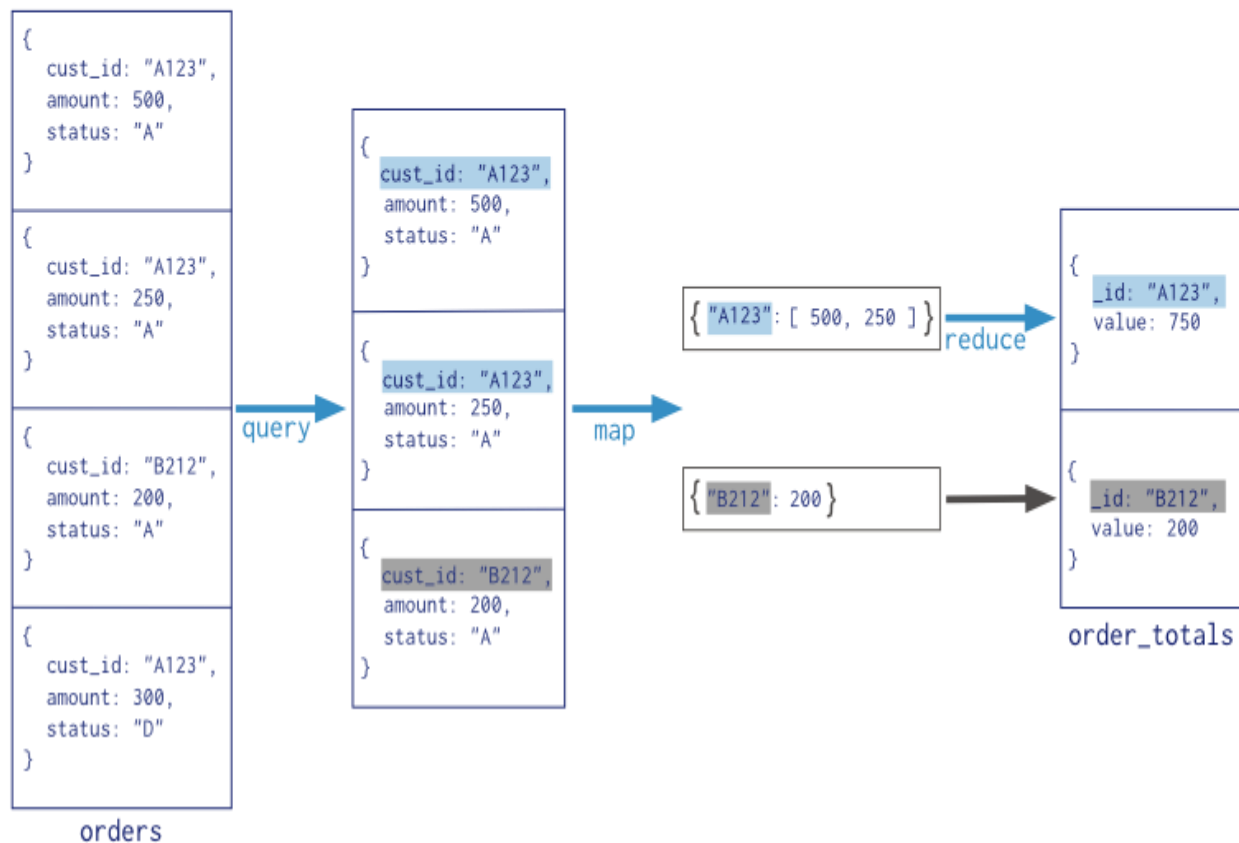
## Map Reduce Programming:

Map-reduce is a data processing paradigm for condensing large volumes of data into useful *aggregated* results. For map-reduce operations, MongoDB provides the `mapReduce` database command.

Consider the following map-reduce operation:

```

Collection
  ↓
db.orders.mapReduce(
  map   → function() { emit( this.cust_id, this.amount ); },
  reduce → function(key, values) { return Array.sum( values ) },
  {
    query: { status: "A" },
    out: "order_totals"
  }
)
  
```



In this map-reduce operation, MongoDB applies the *map* phase to each input document (i.e. the documents in the collection that match the query condition). The map function emits key-value pairs. For those keys that have multiple values, MongoDB applies the *reduce* phase,

which collects and condenses the aggregated data. MongoDB then stores the results in a collection. Optionally, the output of the reduce function may pass through a *finalize* function to further condense or process the results of the aggregation.

All map-reduce functions in MongoDB are JavaScript and run within the mongod process. Map-reduce operations take the documents of a single collection as the *input* and can perform any arbitrary sorting and limiting before beginning the map stage. `mapReduce` can return the results of a map-reduce operation as a document, or may write the results to collections. The input and the output collections may be sharded.

**NOTE**

For most aggregation operations, the Aggregation Pipeline provides better performance and more coherent interface. However, map-reduce operations provide some flexibility that is not presently available in the aggregation pipeline.

### Map-Reduce JavaScript Functions

In MongoDB, map-reduce operations use custom JavaScript functions to *map*, or associate, values to a key. If a key has multiple values mapped to it, the operation *reduces* the values for the key to a single object.

The use of custom JavaScript functions provide flexibility to map-reduce operations. For instance, when processing a document, the map function can create more than one key and value mapping or no mapping. Map-reduce operations can also use a custom JavaScript function to make final modifications to the results at the end of the map and reduce operation, such as perform additional calculations.

### Map-Reduce Behavior

In MongoDB, the map-reduce operation can write results to a collection or return the results inline. If you write map-reduce output to a collection, you can perform subsequent map-reduce operations on the same input collection that merge replace, merge, or reduce new results with previous results. See `mapReduce` and `Perform Incremental Map-Reduce` for details and examples.

When returning the results of a map reduce operation *inline*, the result documents must be within the BSONDocument Size limit, which is currently 16 megabytes. For additional information on limits and restrictions on map-reduce operations, see the `mapReduce` reference page.

MongoDB supports map-reduce operations on sharded collections. Map-reduce operations can also output the results to a sharded collection. See `Map-Reduce and Sharded Collections`.

## **Aggregations :**

Aggregations operations process data records and return computed results. Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result. In SQL count(\*) and with group by is an equivalent of mongodb aggregation.

#### The aggregate() Method

For the aggregation in MongoDB, you should use **aggregate()** method.

#### Syntax

Basic syntax of **aggregate()** method is as follows –

```
>db.COLLECTION_NAME.aggregate(AGGREGATE_OPERATION)
```

#### Example

In the collection you have the following data –

```
{
  _id: ObjectId(7df78ad8902c)
  title: 'MongoDB Overview',
  description: 'MongoDB is no sql database',
  by_user: 'tutorials point',
  url: 'http://www.tutorialspoint.com',
  tags: ['mongodb', 'database', 'NoSQL'],
  likes: 100
},
{
  _id: ObjectId(7df78ad8902d)
  title: 'NoSQL Overview',
  description: 'No sql database is very fast',
  by_user: 'tutorials point',
  url: 'http://www.tutorialspoint.com',
  tags: ['mongodb', 'database', 'NoSQL'],
  likes: 10
},
{
  _id: ObjectId(7df78ad8902e)
  title: 'Neo4j Overview',
  description: 'Neo4j is no sql database',
  by_user: 'Neo4j',
```

```
url: 'http://www.neo4j.com',
tags: ['neo4j', 'database', 'NoSQL'],
likes: 750
},
```

Now from the above collection, if you want to display a list stating how many tutorials are written by each user, then you will use the following **aggregate()** method –

```
> db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$sum : 1}}}]
{
  "result" : [
    {
      "_id" : "tutorials point",
      "num_tutorial" : 2
    },
    {
      "_id" : "Neo4j",
      "num_tutorial" : 1
    }
  ],
  "ok" : 1
}
>
```

Sql equivalent query for the above use case will be **select by\_user, count(\*) from mycol group by by\_user**.

In the above example, we have grouped documents by field **by\_user** and on each occurrence of by\_user previous value of sum is incremented. Following is a list of available aggregation expressions.

Expression	Description	Example
\$sum	Sums up the defined value from all documents in the collection.	db.mycol.aggregate([{\$group : {_id : "\$by_user", num_tutorial : {\$sum : "\$likes"}}}])



\$avg	Calculates the average of all given values from all documents in the collection.	db.mycol.aggregate([{\$group : { _id : "\$by_user", num_tutorial : {\$avg : "\$likes"}}}])
\$min	Gets the minimum of the corresponding values from all documents in the collection.	db.mycol.aggregate([{\$group : { _id : "\$by_user", num_tutorial : {\$min : "\$likes"}}}])
\$max	Gets the maximum of the corresponding values from all documents in the collection.	db.mycol.aggregate([{\$group : { _id : "\$by_user", num_tutorial : {\$max : "\$likes"}}}])
\$push	Inserts the value to an array in the resulting document.	db.mycol.aggregate([{\$group : { _id : "\$by_user", url : {\$push: "\$url"}}}])
\$addToSet	Inserts the value to an array in the resulting document but does not create duplicates.	db.mycol.aggregate([{\$group : { _id : "\$by_user", url : {\$addToSet : "\$url"}}}])
\$first	Gets the first document from the source documents according to the grouping. Typically this makes only sense together with some previously applied "\$sort"-stage.	db.mycol.aggregate([{\$group : { _id : "\$by_user", first_url : {\$first : "\$url"}}}])
\$last	Gets the last document from the source documents according to the grouping. Typically this makes only sense together with some previously applied "\$sort"-stage.	db.mycol.aggregate([{\$group : { _id : "\$by_user", last_url : {\$last : "\$url"}}}])

#### Pipeline Concept

In UNIX command, shell pipeline means the possibility to execute an operation on some input and use the output as the input for the next command and so on. MongoDB also supports same concept in aggregation framework. There is a set of possible stages and each of those is taken as a set of documents as an input and produces a resulting set of documents (or the final resulting JSON document at the end of the pipeline). This can then in turn be used for the next stage and so on.

Following are the possible stages in aggregation framework –

- **\$project** – Used to select some specific fields from a collection.

- **\$match** – This is a filtering operation and thus this can reduce the amount of documents that are given as input to the next stage.
- **\$group** – This does the actual aggregation as discussed above.
- **\$sort** – Sorts the documents.
- **\$skip** – With this, it is possible to skip forward in the list of documents for a given amount of documents.
- **\$limit** – This limits the amount of documents to look at, by the given number starting from the current positions.
- **\$unwind** – This is used to unwind document that are using arrays. When using an array, the data is kind of pre-joined and this operation will be undone with this to have individual documents again