**MongoDB**

MongoDB is an open source, document-oriented, and cross-platform database. It is primarily written in C++. It is also the leading NoSQL database and tied with the SQL database in fifth position after PostgreSQL. It provides high performance, high availability, and easy scalability. MongoDB uses JSON-like documents with schema. MongoDB, developed by MongoDB Inc., is free to use. It is published under a combination of the GNU Affero General Public License and the Apache License.

Let's go through the MongoDB features:

* **Rich query support**: We can query the database as we do with SQL databases. It has a large query set that supports insert, update, delete and select operations. MongoDB supports fields, range queries, and regular expressions. Queries also support the projection where they return a value for specific keys.
* **Indexing**: MongoDB supports primary and secondary indices in its fields.
* **Replication**: Replication means providing more than one copy of data. MongoDB provides multiple copies of data with multiple servers. It provides fault tolerance, if one database server goes down, the application uses other database servers.
* **Load balancing**: Replica sets provide multiple copies of data. MongoDB can scale read operation by client request directly to the secondary node. This divides loads across multiple servers.
* **File storage**: We can store documents up to 6 MB directly to the MongoDB JSON field. For documents exceeding the size limit of 16 MB, MongoDB provides GridFS to store in chunks.
* **Aggregation**: The aggregate function takes a number of records and calculates single results like sum, min, and max. MongoDB provides a data pipeline and multistage pipeline to move large data to the aggregate function which improves performance.

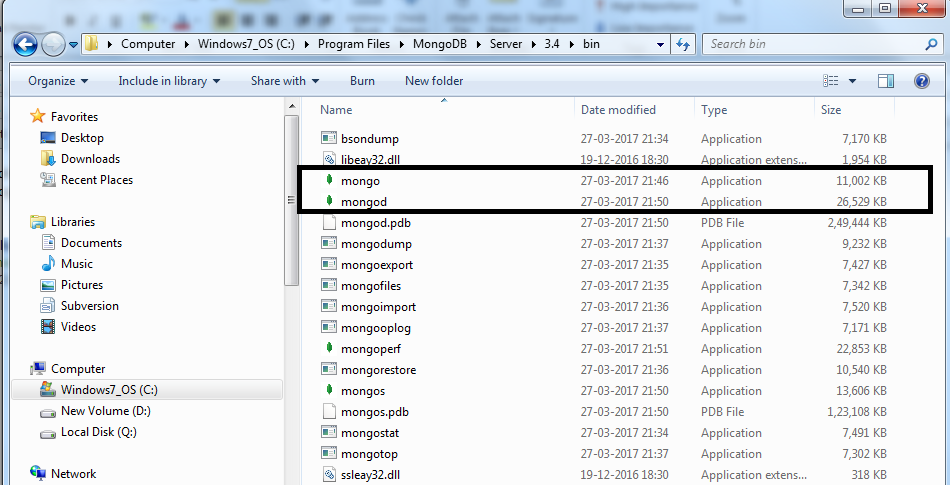
# Installing of MongoDB

You can download the latest version of MongoDB here: <https://www.mongodb.com/download-center#community>. Follow the setup instructions to install it.

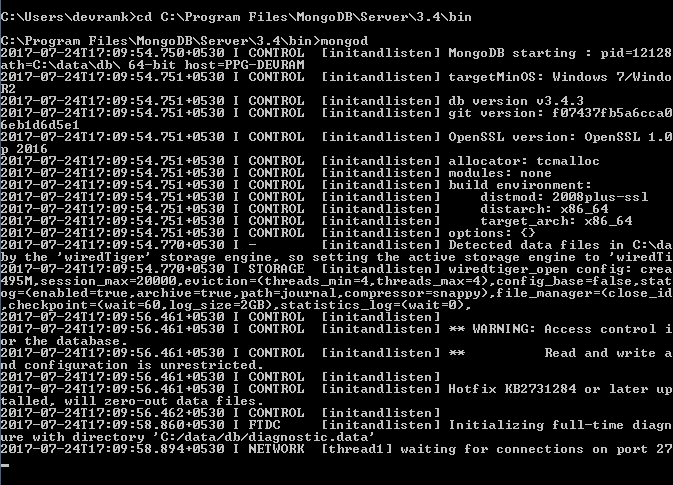
Once MongoDB is installed on your Windows PC, you have to create the following directory:

**Data directory C:\data\db**

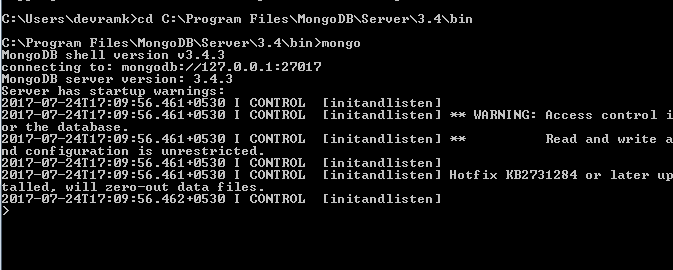
Once you have successfully installed MongoDB, you will be able to see the following executable:



We have to start the mongod instances to begin working with MongoDB. To start the mongod instance, execute it from the command prompt, as shown in the following screenshot:



Once mongod has started, we have to connect this instance using the **mongo** client with the mongo executable:



Once we are connected to the database, we can start working on the database operations.

**MongoDB data types**

Documents in MongoDB are *JSON-like* objects. JSON is a simple representation of data. It supports the following data types:

* null: The null data type is used to represent the null value as well as a value that does not exist:

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781787288867/files/assets/9da8122f-857d-4fdb-8c0c-6f984b59c3d9.png

* boolean: The boolean type is used to represent true and false values:

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781787288867/files/assets/bf76a31e-92da-4ffe-aca9-5d1451b5491a.png

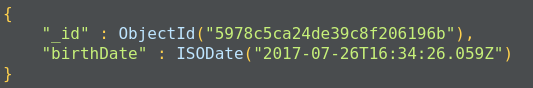
* number: In MongoDB, the shell default supports 64-bit floating-point numbers. To process long and integer numbers, MongoDB provides NumberLong and NumberInt, which represent 4 bytes and 8 bytes, respectively.
* string: The string data type represents the collection of characters. The MongoDB default supports UTF-\* character encoding:

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781787288867/files/assets/1327079f-1809-43bd-a9e1-21505c9f362d.png

* date: MongoDB stores dates in milliseconds since the epoch. The time zone information is not saved:



* After inserting a date using the preceding way in the document, when we query using find it returns a document with a date in the following format:



* array: A set or list of values represents arrays. Also, multiple JSON objects represent an array of elements. The following example shows an array of city values:



* Embedded document: Another MongoDB document-like structure, which can also be used as a key. In the following screenshot, we are storing address fields as an array of addresses, instead of creating a separate collection of addresses:

>

# The MongoDB database

Data is stored in a database in the form of collections. It is a container for collection, just like in SQL databases where the database is a container for tables.

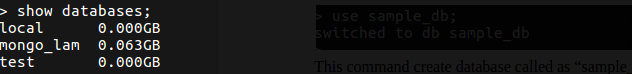
To create a database in MongoDB, we use the following command:

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781787288867/files/assets/2137c894-c9c8-4ab8-af13-d7e364351ef2.png

This command creates a database called sample\_db, which can be used as a container for storing collections.

The default database for mongo is test. If we do not specify a database before storing our collection, MongoDB will store the collection in the test database.

Each database has its own set of files on the filesystem. A MongoDB server can have multiple databases. We can see the list of all the databases using the following command:



# MongoDB collections

The collection is a container for MongoDB documents. It is equivalent to SQL tables, which store the data in rows. The collection should only store related documents. For example, the user\_profiles collection should only store data related to user profiles. It should not contain a user's friend list as this should not be a part of a user's profile; instead, this should fall under the users\_friend collection.

To create a new collection, you can use the following command:

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781787288867/files/assets/a229a052-cb43-41ae-b707-6315dff479ea.png

Here, db represents the database in which we are storing a collection and users\_profile is the new collection we are creating.

Documents in a collection should have a similar or related purpose. A database cannot have multiple collections with the same name, they are unique in the given database.

Collections do not force the user to define a schema and are thus known as schemaless. Documents within the collection have different fields. For example, in one document, we can have user\_address, but in another document, it is not mandatory to have the user\_address field.

This is suitable for an agile approach.

# MongoDB documents

Data in MongoDB is actually stored in the form of documents. The document is a collection of key-value pairs. The key is also known as an **attribute**.

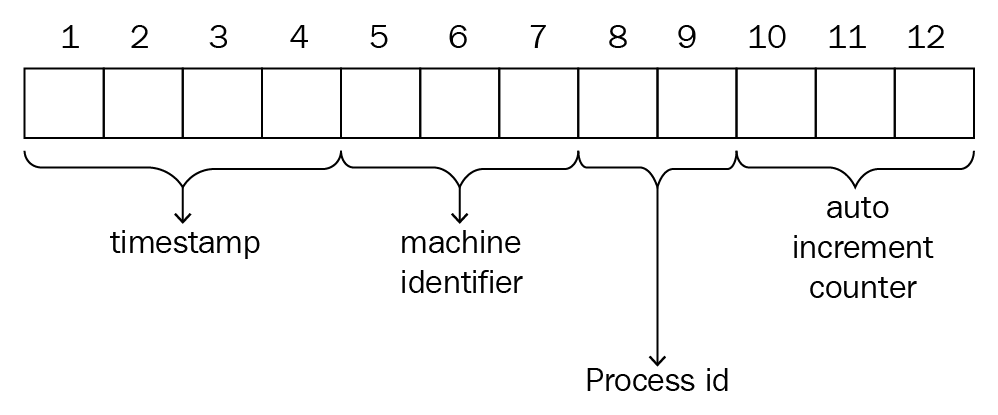
Documents have a dynamic schema and documents in the same collection may vary in field set.

Here is the structure of a MongoDB document:



MongoDB documents have a special field called \_id.\_id is a 12-byte hexadecimal number that ensures the uniqueness of the document. It is generated by MongoDB if not provided by the developer.

Of these 12 bytes, the first 4 bytes represent the current time stamp, the next 3 bytes represent the machine ID, the next 2 bytes represent the process ID on the MongoDB server, and the remaining 3 bytes are simple auto-increment values, as shown in the following diagram:



\_id also represents the primary key for MongoDB documents.

Let's look at a quick comparison between SQL and MongoDB:

|  |  |
| --- | --- |
| **SQL** | **MongoDB** |
| Database | Database |
| Table | Collection |
| Row | Document |
| Column | Field |

Here are some of MongoDB's advantages over RDBMS:

* Collections in the MongoDB database are schemaless. Documents inserted in collections can have different sets of fields.
* The structure of a single-document object is simple and clear.
* Complex joins are not required as the MongoDB database supports an embedded document structure.
* MongoDB has rich query support. MongoDB supports dynamic queries on a database.
* MongoDB is easy to scale.
* Conversion or mapping between database objects and application objects is simple as most of the application supports JSON mapping with database objects.
* Integrated memory support allows the user to access data in a much faster way.

The following are the uses of MongoDB:

* Document-oriented storage
* Index on any attribute
* Replication and high availability
* Auto-sharding
* Rich query support
* Professional support from MongoDB

The applications of MongoDB are:

* **User profiles**: Authentication tools and LDAP are good for authentication and authorization, but data, such as rewards, criminal records, promotions, phone numbers, and addresses are added day by day. Other databases are not able to adopt such quick-changing data. We can use MongoDB dynamic documents to store such data over time in the document.
* **Product and catalog data**: In e-commerce companies or chemical companies, many new products are getting added every day. Each time a new product is added, it is not easy to change schema quickly. In these scenarios, using a document-based database is easier than using any other traditional database.
* **Metadata**: We often require metadata that describes our data. In such scenarios, a graph-based database is a good choice, but we can also use MongoDB for these applications.
* **Content**: MongoDB is mainly a document database. It is great for serving text as well as HTML documents. Also, it provides fine control over storing and indexing contents.

The limitations of MongoDB are:

* The maximum document size supported by MongoDB is 16 MB.
* The maximum document-nesting level supported by MongoDB is 100.
* The maximum namespace (*database + collection name*) supported by MongoDB is 123 characters.
* The database name is limited to 64 characters.
* If we apply an index on any field, that field value cannot contain more than 1024 bytes.
* A maximum of 64 indexes are allowed per collection and a maximum of 34 fields are allowed in compound indexes.
* A hashed index cannot be unique.
* A maximum of 12 nodes are allowed in a replica set.
* A maximum of 512 bytes are allowed for shard keys.
* You cannot rollback automatically if data is more than 300 MB. Manual intervention is needed in such cases.

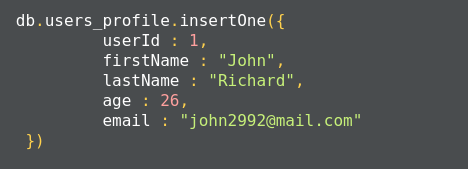
Now we will go through the MongoDB CRUD operations.

**The create operation**

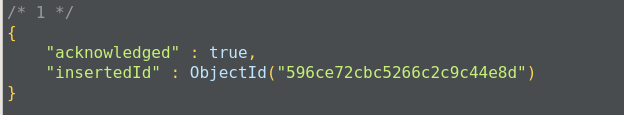
The create operation inserts the new document into the collection. If a collection does not exist, MongoDB will create a new collection and insert a document in it. MongoDB provides the following methods to insert a document into the database:

* db.collection.insertOne();
* db.collection.insertMany();

The MongoDB insert operation will target single collections. Also, mongo preserves atomicity at the document level:



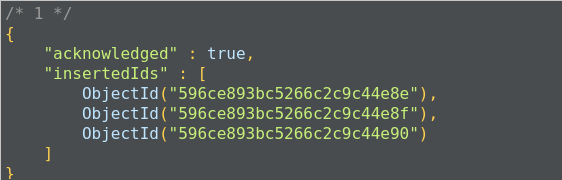
This method returns a document that contains the newly added documents \_id value:



The method insertMany(), can insert multiple documents into the collection at a time. We have to pass an array of documents to the method:



In MongoDB, each document requires an \_id field that uniquely identifies the document that acts as a **primary key**. If a user does not insert the \_id field during the insert operation, MongoDB will automatically generate and insert an ID for each document:



The following is the list of methods that can also be used to insert documents into the collection:

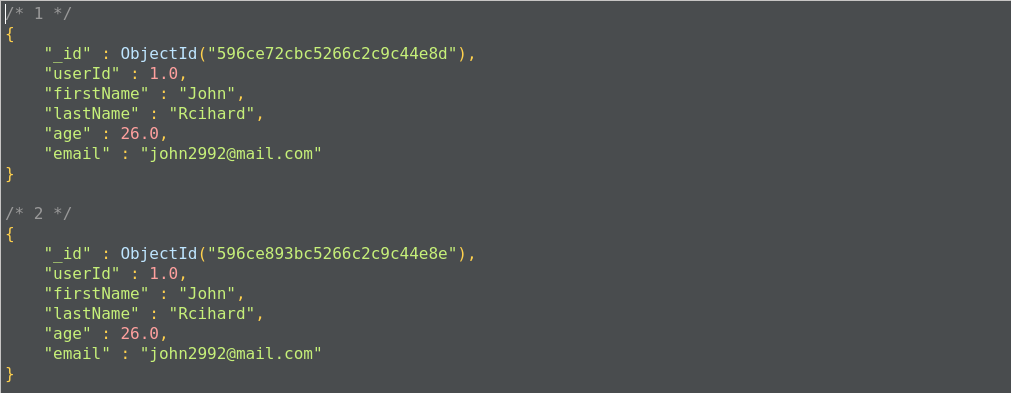
* db.collection.update();
* db.collection.updateOne();
* db.collection.updateMany();
* db.collection.findAndModify();
* db.collection.findOneAndUpdate();
* db.collection.findOneAndReplace();
* db.collection.save();
* db.collection.bulkWrite();

# The read operation

The read operation retrieves documents or data from documents in the collection. To retrieve all documents from a given collection, pass an empty document as a filter. We have to pass the query filter parameter in order to apply our criteria for the selection of documents:

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781787288867/files/assets/48adc834-463c-462a-a07a-cab6cdceb478.png

The preceding code returns all the documents in the given collection:



This operation will return all the documents from the user\_profiles collection. It is also equivalent to the following SQL operation, where user\_profiles is the table, and the query will return all the rows from the table:

**SELECT \* FROM user\_profiles;**

7h 43m remaining

**Applying conditional and logical operators on the filter parameter**

We can apply conditional parameters while retrieving data from collections, such as IN, AND, and OR with less than and greater than conditions:

* Apply the IN condition: We can set filter parameters so the query can match values from a given set and retrieve values from collections where documents match values from a given set. The syntax for the in parameter is the following:

{<field 1> : {<operator 1> : <value 1>},....}

The following query will return all the documents from user\_profiles where the first name is John or Kedar:



Here, we get all the documents with a firstName of either John or Kedar:

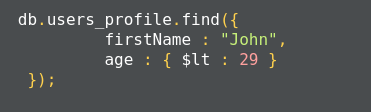


The preceding operation corresponds to the following SQL operation query:

**SELECT \* FROM user\_profiles WHERE firstName in('John', 'Kedar');**

* Apply the AND condition: If the implicit query applies to more than two fields, it covers the AND condition and does not need to apply it separately. This compound query matches all the conditions specified and returns documents where all these conditions are met.

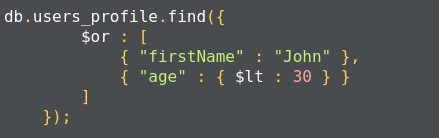
For example, the following query returns the result where the firstname matches John and the age of the user is less than 29:



This corresponds to the following SQL query:

**SELECT \* FROM user\_profiles WHERE firstName='John' AND age<30;**

* Apply the OR condition: Using the $or operator, we can specify the compound query that can apply to two or more fields. The following is the query example where we retrieve any user's profile that has either the first name of John or an age value of less than 30:



This corresponds to the following SQL query:

**SELECT \* FROM user\_profiles WHERE firstName='John' or age<30;**

* Apply the AND/OR condition in the combine. The following is a query example where we apply both the AND and OR operators in the combine to retrieve documents. This query returns user's profiles where the firstName is John and the age is less than 30 or the lastName starts with s:



This operation corresponds to the following SQL operation:

**SELECT \* FROM user\_profiles WHERE firstName='John'**  
**AND age<30 OR lastName like 's%';**

MongoDB also supports $regex to perform string-pattern matching. Here is a list of the comparison query operators:

|  |  |
| --- | --- |
| **Operator** | **Description** |
| $eq | Matches values that equals a specified value |
| $gt | Matches values that are greater than a specified value |
| $gte | Matches values that are greater than or equal to a specified value |
| $lt | Matches values that are less than a specified value |
| $lte | Matches values that are less than or equal to a specified value |
| $ne | Matches values that are not equal to a specified value |
| $in | Matches values that are specified in a set of values |
| $nin | Matches values that are not specified in a set of values |

Here is a list of logical operators:

|  |  |
| --- | --- |
| **Operator** | **Description** |
| $or | Joins query clauses with the OR operator and returns all the documents that match any condition of the clause. |
| $and | Joins query clauses with the AND operator and returns documents that match all the conditions of all the clauses. |
| $not | Inverts the effects of the query expression and returns all the documents that do not match the given criteria. |
| $nor | Joins query clauses with the logical NOR operator and returns all the documents that match the criteria specified in the clauses. |

MongoDB also uses the findOne method to retrieve documents from the mongo collection. It internally calls the find method with a limit of 1. findOne matches all the documents with the filter criteria and returns the first document from the result set.

**The update operation**

The following is a list of methods MongoDB uses to update the document information:

\* db.collection.updateOne(<filter>, <update>, <options>);  
\* db.collection.updateMany(<filter>, <update>, <options>);  
\* db.collection.replaceOne(<filter>, <update>, <options>);

If the update operation increases the size of the document while updating, the update operation relocates the document on the disk:

* **Update single document**: The following example uses the db.collection.updateOne() method to update a single document. The following query finds the userId 1 document in user\_profiles and updates the age to 30:



Here, the query uses the $set operator to update the value of the age field, where the userId matches 1:



* **Update multiple documents**: The following example uses the db.collection.updateMany() method to update multiple documents where the condition matches.

The following example updates the age of all users between the ages of 30 to 35:



The output gives us acknowledgement of how many documents have been updated:



* **Replace document**: The db.collection.replaceOne() method replaces an entire document with a new document, except for the \_id field. The \_id field has the same value as the current document. The following example replaces a user's document with that of userId : 1:



The following are additional methods that can be used for the update operation:

* db.collection.findOneAndReplace();
* db.collection.findOneAndUpdate();
* db.collection.findAndModify();
* db.collection.save();
* db.collection.bulkWrite();

**The delete operation**

MongoDB provides the following methods to remove documents from the collection:

* db.collection.deleteOne();
* db.collection.deleteMany();

The db.collection.deleteMany() method is used to delete all the documents that match the given criteria. If you want to delete all the documents, you can pass an empty document criterion:

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781787288867/files/assets/3c66b6e8-d945-4d4d-b5fd-2e35207f55ca.png

This query will delete all the documents contained in the user\_profiles collection.

To delete the single most document which matched the given criteria, we use the db.collection.deleteOne() method. The following query deletes the record where the userId is equal to 1:



The following methods can also delete documents from the collection:

* db.collection.findOneAndDelete()
* db.collection.findAndModify()
* db.collection.bulkWrite()

**Data models in MongoDB**

The MongoDB collection does not enforce structure on the document. This allows the document to map to objects or entities easily. Each document can match a data field of the entity. In practice, documents in collections share the same structure.

When deciding on data modeling, we have to consider the requirements of the application, the performance characteristics of the database's design, and data retrieval patterns. When designing data models, we have to focus on the application's usage of data and the inherent structure of the data.

While deciding the data model, we have to consider the structure of the document and how documents relate to each other. There are two key data models that show these relationships:

* The reference document data model
* The embedded data model

**The references document data model**

In this model, the relationship is maintained using links between documents. References from one document is stored in another document. This process is also called normalization; it establishes the relationship between different collections and defines collections for a more specific purpose:



We use the normalized data approach in the following scenarios:

* When embedding data, it will create duplicate data but not provide significant read performance
* To represent more complex many-to-many relationships
* To model large hierarchical data

**The embedded data model**

In this document model, the relationships between data are maintained by storing data in a single document. Here, we do not create a separate document to define a relationship. We can embed the document structure in a field or array within the document. These documents are denormalized. This allows us to retrieve data in a single operation, but it unnecessarily increases the size of the document.

The embedded document structure allows us to store related pieces of data in the same document. This also allows us to update a single document without worrying about data consistency.

The embedded document structure is used in two cases:

* When there is a **one-to-one relationship** with the embedded document. We can store the embedded document as an object field.
* When there is a **one-to-many relationship** with the embedded document. Here we can store the embedded document as an array of the object field.

The embedded structure provides better performance for read operations, requests and retrieves related data in a single database operation, and updates the data in a single atomic operation. But this approach can lead to an increase in the size of the document, and MongoDB will store such documents in a fragment, which leads to poor write performance.

Modeling the application data for MongoDB depends on the data as well as the characteristics of MongoDB itself. When creating data models for applications, analyze all of the read and write operations with the following operations and MongoDB features:

* **Document size**: The update operation on a document may increase the size of the document as MongoBD documents are schemaless. An update may include adding more elements to an array or adding new elements to the document. If the document size exceeds the maximum limit, MongoDB automatically relocates the document on to the disk.
* **Atomicity**: In MongoDB, each operation is atomic at the document level. A single operation can change only one document at a time. So, the operation that modified more than one document needed multiple write operations. The embedded document structure is more suitable in such a scenario where all related data is stored in a single document.
* **Sharding**: Sharding provides horizontal scaling in MongoDB. This enables deployment with a large dataset and high throughput for operations. Sharding allows us to partition a collection and store documents from collections across multiple instances of mongod or clusters. MongoDB uses shard keys to select data. The sharding key has an effect on performance and can prevent query isolation and increased write capacity. So be careful when choosing the shard key.
* **Indexes**: We use indexes to improve performance for common queries. Normally, we build indexes on a field that is often used in filter criteria and can be in sorted order so that searching will use effective algorithms, such as mid-search. MongoDB will automatically create an index on the \_id field. While creating indexes, consider the following points:
  + Each index requires at least 8 KB of space.
  + Indexes have a negative impact on write operations. For collection with a high write-to-read ratio, indexes are much more expensive as each insert operation leads to some update operations.
  + Collections with a high read-to-write ratio often benefit from indexes. Indexes do not affect read operations on non-index fields.
  + Active indexes use disk space and memory. This usage can be significant and we should analyze it for performance considerations.
* **Large numbers of collections**: In some use cases, we may decide to store data over multiple collections instead of a single one. If the number of documents in the collection is low, then we can group the documents by type. For example, by maintaining a separate collection for the dev, prod, and debug logs, instead of using three collections named dev\_log, prod\_log, and deug\_log, we can maintain a single collection called **log**. Having a large number of collections decreases the performance of operations. When adding collections, consider the following points:
  + Each collection has an overhead of a few kilobytes.
  + Each index on \_id requires at least 8 KB of data space.
  + Each database single namespace stores the metadata. And each index and collection have an entry in the namespace file.
* **Data lifecycle management**: The **Time to live** (**TTL**) feature of a collection expires documents after a certain period of time. We can consider using the TTL feature if certain data in the collection is not useful after a specific period of time. If an application used only recently-inserted documents, use **capped collections**. The capped collection provides **first in, first out** (**FIFO**) management of documents that supports insert and read operations based on insertion order.

# Introduction to MongoDB indexing

Indexes allow efficient execution of MongoDB queries. If we don't have indexes, MongoDB has to scan all the documents in the collection to select those documents that match the criteria. If proper indexing is used, MongoDB can limit the scanning of documents and select documents efficiently. Indexes are a special data structure that store some field values of documents in an easy-to-traverse way.

Indexes store the values of specific fields or sets of fields, ordered by the values of fields. The ordering of field values allows us to apply effective algorithms of traversing, such as the mid-search algorithm, and also supports range-based operations effectively. In addition, MongoDB can return sorted results easily.

Indexes in MongoDB are the same as indexes in other database systems. MongoDB defines indexes at the collection level and supports indexes on fields and sub-fields of documents.

**The default \_id index**

MongoDB creates the default \_id index when creating a document. The \_id index prevents users from inserting two documents with the same \_id value. You cannot drop an index on an \_id field.

The following syntax is used to create an index in MongoDB:

>db.collection.createIndex(<key and index type specification>, <options>);

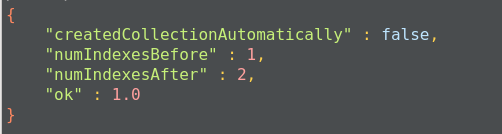
The preceding method creates an index only if an index with the same specification does not exist. MongoDB indexes use the B-tree data structure.

The following are the different types of indexes:

* **Single field**: In addition to the \_id field index, MongoDB allows the creation of an index on any single field in ascending or descending order. For a single field index, the order of the index does not matter as MongoDB can traverse indexes in any order. The following is an example of creating an index on the single field where we are creating an index on the firstName field of the user\_profiles collection:



The query gives acknowledgment after creating the index:



This will create an ascending index on the firstName field. To create a descending index, we have to provide -1 instead of 1.

* **Compound index**: MongoDB also supports user-defined indexes on multiple fields. The order of fields defined while creating an index has a significant effect. For example, a compound index defined as {firstName:1, age:-1} will sort data by firstName first and then each firstName with age.
* **Multikey index**: MongoDB uses multi-key indexes to index the content in the array. If you index the field that contains the array values, MongoDB creates an index for each field in the object of an array. These indexes allow queries to select the document by matching the element or set of elements of the array. MongoDB automatically decides whether to create multi-key indexes or not.
* **Text indexes**: MongoDB provides text indexes that support the searching of string contents in the MongoDB collection. To create text indexes, we have to use the db.collection.createIndex() method, but we need to pass a *text* string literal in the query:

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781787288867/files/assets/9584c14e-fc2c-43dc-8b77-b2f473fbb662.png

You can also create text indexes on multiple fields, for example:



Once the index is created, we get an acknowledgment:

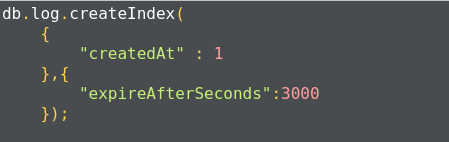


Compound indexes can be used with text indexes to define an ascending or descending order of the index.

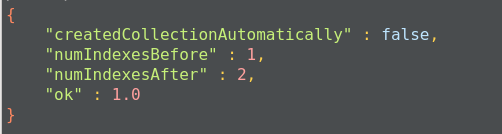
* **Hashed index**: To support hash-based sharding, MongoDB supports hashed indexes. In this approach, indexes store the hash value and query, and the select operation checks the hashed indexes. Hashed indexes can support only equality-based operations. They are limited in their performance of range-based operations.

Indexes have the following properties:

* **Unique indexes**: Indexes should maintain uniqueness. This makes MongoDB drop the duplicate value from indexes.
* **Partial Indexes**: Partial indexes apply the index on documents of a collection that match a specified condition. By applying an index on the subset of documents in the collection, partial indexes have a lower storage requirement as well as a reduced performance cost.
* **Sparse index**: In the sparse index, MongoDB includes only those documents in the index in which the index field is present, other documents are discarded. We can combine unique indexes with a sparse index to reject documents that have duplicate values but ignore documents that have an indexed key.
* **TTL index**: TTL indexes are a special type of indexes where MongoDB will automatically remove the document from the collection after a certain amount of time. Such indexes are ideal to remove machine-generated data, logs, and session information that we need for a finite duration. The following TTL index will automatically delete data from the log table after 3000 seconds:



Once the index is created, we get an acknowledgment message:



The limitations of indexes:

* A single collection can have up to 64 indexes only.
* The qualified index name is <database-name>.<collection-name>.$<index-name> and cannot have more than 128 characters. By default, the index name is a combination of index type and field name. You can specify an index name while using the createIndex() method to ensure that the fully-qualified name does not exceed the limit.
* There can be no more than 31 fields in the compound index.
* The query cannot use both text and geospatial indexes. You cannot combine the $text operator, which requires text indexes, with some other query operator required for special indexes. For example, you cannot combine the $text operator with the $near operator.
* Fields with 2d sphere indexes can only hold geometry data. 2d sphere indexes are specially provided for geometric data operations. For example, to perform operations on co-ordinate, we have to provide data as points on a planer co-ordinate system, [*x*, *y*]. For non-geometries, the data query operation will fail.

The limitation on data:

* The maximum number of documents in a capped collection must be less than 2^32. We should define it by the max parameter while creating it. If you do not specify, the capped collection can have any number of documents, which will slow down the queries.
* The MMAPv1 storage engine will allow 16,000 data files per database, which means it provides the maximum size of 32 TB.  
  We can set the storage.mmapv1.smallfile parameter to reduce the size of the database to 8 TB only.
* Replica sets can have up to 50 members.
* Shard keys cannot exceed 512 bytes.

# Replication

A replica set is a group of MongoDB instances that store the same set of data. Replicas are basically used in production to ensure a high availability of data.

**Redundancy and data availability**: because of replication, we have redundant data across the MongoDB instances. We are using replication to provide a high availability of data to the application. If one instance of MongoDB is unavailable, we can serve data from another instance. Replication also increases the read capacity of applications as reading operations can be sent to different servers and retrieve data faster. By maintaining data on different servers, we can increase the locality of data and increase the availability of data for distributed applications. We can use the replica copy for backup, reporting, as well as disaster recovery.

# Replication in MongoDB

A replica set is a group of MongoDB instances that have the same dataset. A replica set has one arbiter node and multiple data-bearing nodes. In data-bearing nodes, one node is considered the primary node while the other nodes are considered the secondary nodes.

All write operations happen at the primary node. Once a write occurs at the primary node, the data is replicated across the secondary nodes internally to make copies of the data available to all nodes and to avoid data inconsistency.

If a primary node is not available for the operation, secondary nodes use election algorithms to select one of their nodes as a primary node.

A special node, called an arbiter node, is added in the replica set. This arbiter node does not store any data. The arbiter is used to maintain a quorum in the replica set by responding to a heartbeat and election request sent by the secondary nodes in replica sets. As an arbiter does not store data, it is a cost-effective resource used in the election process. If votes in the election process are even, the arbiter adds a voice to choose a primary node. The arbiter node is always the arbiter, it will not change its behavior, unlike a primary or secondary node. The primary node can step down and work as secondary node, while secondary nodes can be elected to perform as primary nodes.

Secondary nodes apply read/write operations from a primary node to secondary nodes asynchronously.

# Automatic failover in replication

Primary nodes always communicate with other members every 10 seconds. If it fails to communicate with the others in 10 seconds, other eligible secondary nodes hold an election to choose a primary-acting node among them. The first secondary node that holds the election and receives the majority of votes is elected as a primary node. If there is an arbiter node, its vote is taken into consideration while choosing primary nodes.

# Read operations

Basically, the read operation happens at the primary node only, but we can specify the read operation to be carried out from secondary nodes also. A read from a secondary node does not affect data at the primary node. Reading from secondary nodes can also give inconsistent data.

# Sharding

Sharding is a methodology to distribute data across multiple machines. Sharding is basically used for deployment with a large dataset and high throughput operations. The single database cannot handle a database with large datasets as it requires larger storage, and bulk query operations can use most of the CPU cycles, which slows down processing. For such scenarios, we need more powerful systems.

One approach is to add more capacity to a single server, such as adding more memory and processing units or adding more RAM on the single server, this is also called vertical scaling. Another approach is to divide a large dataset across multiple systems and serve a data application to query data from multiple servers. This approach is called horizontal scaling. MongoDB handles horizontal scaling through sharding.

**Sharded clusters**

MongoDB's sharding consists of the following components:

* **Shard**: Each shard stores a subset of sharded data. Also, each shard can be deployed as a replica set.
* **Mongos**: Mongos provide an interface between a client application and sharded cluster to route the query.
* **Config server**: The configuration server stores the metadata and configuration settings for the cluster. The MongoDB data is sharded at the collection level and distributed across sharded clusters.
* **Shard keys**: To distribute documents in collections, MongoDB partitions the collection using the shard key. MongoDB shards data into chunks. These chunks are distributed across shards in sharded clusters.

**Advantages of sharding**

Here are some of the advantages of sharding:

* When we use sharding, the load of the read/write operations gets distributed across sharded clusters.
* As sharding is used to distribute data across a shard cluster, we can increase the storage capacity by adding shards horizontally.
* MongoDB allows continuing the read/write operation even if one of the shards is unavailable. In the production environment, shards should deploy with a replication mechanism to maintain high availability and add fault tolerance in a system.

**Storing large data in MongoDB**

MongoDB is document-based database, data is stored in JSON and XML documents. MongoDB has a document size limit of 16 MB. If the size of a JSON document exceeds 16 MB, instead of storing data as a single file, MongoDB divides the file into chunks and each chunk is stored as a document in the system. MongoDB creates a chunk of 255 KB to divide files and only the last chuck can have less than 255 KB.

MongoDB uses two collections to work with gridfs. One collection is used to store the chunk data and another collection is used to store the metadata. When you query MongoDB for the operation of the gridfs file, MongoDB uses the metadata collection to perform the query and collect data from different chunks. GridFS stores data in two collections:

* chunks: Stores binary chunks.
* files:: Stores the file's metadata.

# Summary

In this chapter, we learned about MongoDB, which is one of the most popular NoSQL databases. It is widely used in projects where requirements change frequently and is suitable for agile projects. It is a highly fault-tolerant and robust database.