Indexing in MongoDB

MongoDB uses indexing in order to make the query processing more efficient. If there is no indexing, then the MongoDB must scan every document in the collection and retrieve only those documents that match the query. Indexes are special data structures that stores some information related to the documents such that it becomes easy for MongoDB to find the right data file. The indexes are order by the value of the field specified in the index.

When not to create indexing:

- When collection is small
- When collection is updated frequently
- When queries are complex
- When collection is large and multiple indexes are applied.

If your application is repeatedly running queries on the same fields, you can create an index on those fields to improve performance. For example, consider the following scenarios:

Scenario	Index Type
A human resources department often needs to look up employees by employee ID. You can create an index on the employee ID field to improve query performance.	**Single Field Index**
A store manager often needs to look up inventory items by name and quantity to determine which items are low stock. You can create a single index on both the item and quantity fields to improve query performance.	**Compound Index**

- Indexes are special data structures that store a small portion of the collection's data set in an easy-to-traverse form.
- MongoDB indexes use a B-tree data structure.
- The index stores the value of a specific field or set of fields, ordered by the value of the field.

Default Index
MongoDB creates a unique index on the _id field during the creation of a collection. The _id index prevents clients from inserting two documents with the same value for the _id field. You cannot drop this index.

Index Names
The default name for an index is the combination of the **index keys** and each key's **direction** in the index **(1 or -1)** with **underscores** as a separator.

For example, an index created on { item : 1, quantity: -1 } has the name item_1_quantity_-1.

explain() method

Provides information on the query plan for the db.collection.find() method.
The explain() method has the following form:
db.collection.find().explain()
The following example runs cursor.explain() in "executionStats" verbosity mode to return the query planning and execution information for the specified db.collection.find() operation:

```
db.students.find({age:{$gt:15}}).explain("executionStats")
```

Explain(executionStats) is a method that you can apply to simple queries or to cursors to investigate the query execution plan. The execution plan is how MongoDB resolves a query. Looking at all the information returned by explain():

- **totalDocsExamined**: how many documents were examined
- **nReturned**:how many documents were returned
- **stage:** Inside the winningPlan -> queryPlan -> stage is defined. It is used to defined the stage of input scanning.
    - COLLSCAN for a collection scan
    - IXSCAN for scanning index keys

Creating an Index

MongoDB provides a method called createIndex() that allows user to create an index.

Syntax: db.**collection_name**.createIndex({KEY:1})

MongoDB only creates the index if an index of the same specification does not exist.
The key determines the field on the basis of which you want to create an index and 1 (or -1) determines the order in which these indexes will be arranged

1 for ascending order and -1 for descending order

1. Create an Index on a Single Field

We can create an index on any one of the fields of collection.
Example:

```
db.table1.createIndex({age:1})
```

This will create an index named “age_1”.

***Read documents using Index:**

Syntax: db.collection_name.find().explain(“executionStats”)

2. Compound indexing

Compound indexes are indexes that contain references to multiple fields. Compound indexes improve performance for queries on exactly the fields in the index or fields in the index prefix.
To create a compound index, use the db.collection.createIndex() method:
db.<collection>.createIndex({<field1>:<sortOrder>,<field2>:<sortOrder>,...,<fieldN>:<sortOrder>})

Example:

```
db.table1.createIndex({age:1,name:-1})
```

An index created on { age : 1, name: -1 } has the name age_1_name_-1.

In this example:

- The index on age is ascending (1).
- The index on name is descending (-1).

The created index supports queries that applies on:

- Both age and name fields.
- Only the age field, because age is a prefix of the compound index.

For example, the index supports these queries: (Perform IXSCAN)

```
db.students.find( { name: "Abc", age: 36 } )
db.students.find( { age: 26 } )
```

The index does not support queries on only the name field, because name is not part of the index prefix. For example, the index does not support this query:

```
db.students.find( { name: "Abc" } ) →perform CoLLSCAN
```

- **getIndexes()**

  To confirm that the index was created, use below command:

```
db.collection.getIndexes()
```

Output:

```
[{ v: 2, key: { _id: 1 }, name: '_id_' },{ v: 2, key: { age: 1 }, name: 'age_1' }]
```

- **dropIndex()/dropIndexes()**

  dropIndex() : Drops or removes the specified index from a collection.

```
db.collection.dropIndex(index)
```

dropIndexes() : It is used to drop all indexes except the _id index from a collection.

```
db.collection.dropIndexes()
```

Drop a specified index from a collection. To specify the index, you can pass the method either:

- **The index specification document**

  db.collection.dropIndexes({ a: 1, b: 1 })
- **The index name:**

  db.collection.dropIndexes("a_1_b_1")
- **Drop specified indexes from a collection. To specify multiple indexes to drop, pass the method an array of index names:**

  db.collection.dropIndexes(["a_1_b_1", "a_1", "a_1__id_-1"])

If the array of index names includes a non-existent index, the method errors without dropping any of the specified indexes

To get the names of the indexes, use the db.collection.getIndexes() method.

Let’s understand the concept with following example.

Suppose, we have a collection named student and we want to apply query to find students who **have age greater than 12**.

Student Collection

_id	Name	Age
1	ABC	10
2	XYZ	12
3	ABC	15
4	PQR	13
5	XYZ	15
6	ABC	8

- **Find students without creating an index.**

```
db.student.find({age:{$gt:12}}).explain("executionStats")
```

It will examine 6 documents and return 3 documents by performing COLLSCAN.

totalDocsExamined: 6

nReturned:3

stage: COLLSCAN

- **Create an index on the age field to improve performance for those queries:**

```
db.students.createIndex( { age: 1 } )
```

index name: age_1

- **Now, find the students with age greater than 12**

```
db.student.find({age:{$gt:12}}).explain("executionStats")
```

It will examine 3 documents and return 3 documents by performing IXSCAN.

totalDocsExamined: 3

nReturned: 3

stage: IXSCAN

- **Create an index on the age and name fields to improve performance.(Compound Indexing)**

Note: Before creating any other indexing on same fields. Drop the created indexing

db.students.createIndex({ age: 1,name:-1 })
index name: age_1_name_-1

_id	name	age
1	ABC	8
2	ABC	10
3	XYZ	12
4	PQR	13
5	XYZ	15
6	ABC	15

- So here on name field indexing is applied based on age field.
- In this example we have two students with age 15. For age field values 15 the name field values are arranged in descending order.
- name field indexing is depending on the age field. So if we apply query on name field then it will perform the collscan.

Now, consider below three scenarios

1. Display student with name “ABC” and age 15

db.student.find({age:15,name:”ABC”}).explain("executionStats")
It will examine 1 documents and return 1 documents by performing IXSCAN.
totalDocsExamined: 1
nReturned: 1
stage: IXSCAN

2. Display students whose age is greater than 12

db.student.find({age:{$gt:12}).explain("executionStats")
It will examine 3 documents and return 3 documents by performing IXSCAN.
totalDocsExamined: 3
nReturned: 3
stage: IXSCAN

3. Display students whose name is “ABC”

db.student.find({name:”ABC”}).explain("executionStats")
It will examine 6 documents and return 3 documents by performing COLLSCAN.
totalDocsExamined: 6
nReturned: 3
stage: COLLSCAN

3. Partial Indexing

The partial index functionality allows users to create indexes that match a certain filter condition. Partial indexes use the partialFilterExpression option to specify the filter condition. The partialFilterExpression option accepts a document that specifies the filter condition using:

- equality expressions (i.e. field: value or using the $eq operator),
- $exists: true expression,
- $gt, $gte, $lt, $lte expressions,
- $type expressions,
- $and operator,
- $or operator,
- $in operator

Example:
Consider student collection with 6 documents

- **Create index on age where age is greater than 15**

```
db.student.createIndex({age:1},{partialFilterExpression:{age:{$gt:15}}})
```

- **Display student/s whose age is 16.**

```
db.student.find({age:16}).explain("executionStats")
Output: stage: 'IXSCAN',
        nReturned: 1,
        docsExamined: 1
```

Suppose we have only one document available with age value 16.

- **Display student/s whose age is 13.**

```
db.student.find({age:13}).explain("executionStats")
Output: stage: 'COLLSCAN',
        nReturned: 1,
        docsExamined: 6
```

Suppose we have only one document available with age value 13.

We have applied index on age field where age is greater than 15. we are looking for age 13 which is less than 15. Then here it will perform the COLLSCAN.

- **Display student/s whose age is 17.**

```
db.student.find({age:17}).explain("executionStats")
Output: stage: 'IXSCAN',
        nReturned: 2,
        docsExamined: 2
```

Suppose we have only 2 documents available with age value 17.

We have applied index on age field where age is greater than 15. we are looking for age 17 which is greater than 15. Then here it will perform the IXSCAN.

Winning plan:

If we have applied same field indexing on multiple times, then searching can be done using one type of indexing only. In this scenario, it races with one indexing strategy and only one indexing strategy wins, the other strategy is stored in loosing plan (rejected plan).

The winning plan is stored in cache till 1000 write operations and next time it does not race with matching index. So, this index is stored in winning plan and the other index is stored in rejected plan.

Let's say we have already one index on age and one more index apply on age like:

Code:
```
db.DATA.createIndex({age:1,name:1})
db.DATA.find({age:19}).explain("executionStats")
```

so, two indexes have been created on age field, then it selects only one index for searching called inside winning plan, rest of the index is called as rejected plan.

Example:

Consider a collection student having documents like this:

```
[
{_id:123433,name: "DDD",age:32},
{_id:123434,name: "BBB",age:20},
{_id:123435,name: "AAA",age:10},
]
```

Do as directed:

(1) Create an index & fire a command to retrieve a document having age>15 and name is "BBB". Stats must return values nReturned=1, docExamined=1, stage="IXSCAN". Perform required indexing.

(2) Create an index on subset of a collection having age>30. Also write a command to get a stats "IXSCAN" for age>30.

Solution:

1) db.student.createIndex({age:1,name:1})
   db.student.find({age:{$gt:15},name:'DDD'}).explain('executionStats')

2) db.student.createIndex({age:1},{partialFilterExpression:{age:{$gt:30}}})
   db.student.find({age:{$gt:30}}).explain('executionStats')

MongoDB Replication

Replication is the process of synchronizing data across multiple servers. It provides redundancy and increase data availability with multiple copies of data on diff. DB servers.

Replication protect a Database from loss of a single server. It also allows you to recover from Hardware failure and service interruption.

A replica set in MongoDB is a group of mongod processes that maintain the same data set. Replica sets provide redundancy and high availability, and are the basis for all production deployments

Redundancy and Data Availability

Replication provides redundancy and increases data availability. With multiple copies of data on different database servers, replication provides a level of fault tolerance against the loss of a single database server.

In some cases, replication can provide increased read capacity as clients can send read operations to different servers. Maintaining copies of data in different data centres can increase data locality and availability for distributed applications. You can also maintain additional copies for dedicated purposes, such as disaster recovery, reporting, or backup.

Replication Key Features:

- Replica sets are the clusters of N different nodes that maintain the same copy of the data set.
- The primary server receives all write operations and record all the changes to the data/
- The secondary members then copy and apply these changes in an asynchronous process.
- All the secondary nodes are connected with the primary nodes. there is one heartbeat signal from the primary nodes. If the primary server goes down an eligible secondary will hold the new primary.

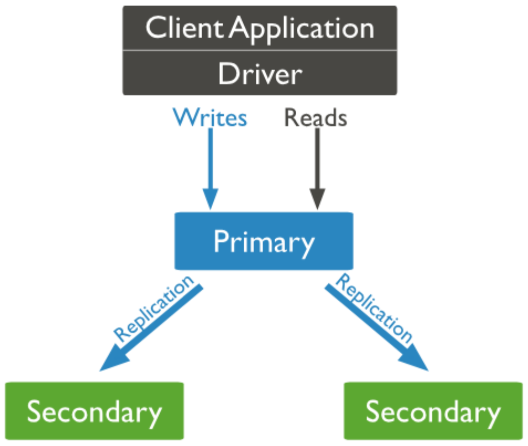
Why Replication?

- High Availability of data disasters recovery
- No downtime for maintenance (like backups index rebuilds and compaction)
- Read Scaling (Extra copies to read from)

How replication works

Replica set is a group of two or more nodes.

In a replica set, one node is Primary node and remaining nodes are secondary.



- Data of primary server is copied to secondary. This duplication is done asynchronously.

Sharding

Sharding is a method for allocating data across multiple machines. MongoDB used sharding to help deployment with very big data sets and large throughput the operation. By sharding, you combine more devices to carry data extension and the needs of read and write operations

Sharding solve the the problem of horizontal scaling.

The vertical scaling approach

The vertical scaling approach, sometimes referred to as "scaling up," focuses on adding more resources or more processing power to a single machine. These additions may include CPU and RAM resources upgrades which will increase the processing speed of a single server or increase the storage capacity of a single machine to address increasing data requirements.

The horizontal scaling approach

The horizontal scaling approach, sometimes referred to as "scaling out," entails adding more machines to further distribute the load of the database and increase overall storage and/or processing power. There are two common ways to perform horizontal scaling — they include sharding, which increases the overall capacity of the system, and replication, which increases the availability and reliability of the system.

Automatic Failover

When a primary does not communicate with the other members of the set for more than the configured electionTimeoutMillis period (10 seconds by default), an eligible secondary calls for an election to nominate itself as the new primary. The cluster attempts to complete the election of a new primary and resume normal operations.

In a three member replica set with two secondaries, the primary becomes unreachable. The loss of a primary triggers an election where one of the secondaries becomes the new primary.

Steps for Replication

Primary server on port 27018

Secondary server on port 27019

Steps	**Command**
1	Prepare new folder data inside that another folder named db. Inside that make two sub folder db1 and db2.
2	Open **command prompt(cmd 1)** and run below command to create mongodb server on port number 27020 and store data at mentioned database path. mongod --port 27018 --dbpath “D:\data\db\db1” --replSet rs1
3	Open **command prompt(cmd 2)** and run below command to create mongodb server on port number 27021 and store data at mentioned database path. mongod --port 27019 --dbpath “D:\data\db\db2” --replSet rs1
4	Open **command prompt (cmd no.3**) to check replica created or not by using below command mongosh --port 27018 (if it opens “test>” then created successfully)
5	To initiate replica set **cmd no.3** The following example initiates a new replica set with two members. rs.initiate({ _id:”rs1”, members:[{_id:0, host:”127.0.0.1:27018”}, {_id:1, host:”127.0.0.1:27019”}] })
6	**rs.status():** on Cmd no.3 **rs.status()** – Returns the replica set status from the point of view of the member where the method is run. Shows that 127.0.0.1:27018 is the primary server and 127.0.0.1:27019 is the Secondary serve
7	To lookout present database on server port 27018 hit below command in cmd no.3 rs1 [direct: primary] test> show dbs 3 Dbs named admin,config and local. On New server port :27018
8	Switch database by using below command in Cmd no 3:

	rs1 [direct: primary] test> use kppdata Insert record on primary server rs1 [direct: primary] kppdata> db.people.insertMany([{ name:”ABC”,age:29,dept:”IT”},{ name:”PQR”,age:29,dept:”IT”}]) This will also create a replica of the mydb database on 27019 port. Open mondoDB compass. Connect with port 27019 and check the database and collection.
9	Data automatically inserted in secondary server. Permission required to perform read operation. **Open cmd 4 and hit below to command to reach specific database** • mongosh --port 27019 • test>use kppdata To specify Read Preference Mode The following operation sets the read preference mode to target the read to a primary member. This implicitly allows reads from primary. **Command to hit: db.getMongo().setReadPref(“primaryPreferred”)** **Read Preference Mode**s 1. **primary:** Default mode. All operations read from the current replica set primary. 2. **primaryPreferred:** In most situations, operations read from the primary but if it is unavailable, operations read from secondary members. 3. **secondary:** All operations read from the secondary members of the replica set. 4. **secondaryPreferred:** Operations typically read data from secondary members of the replica set. If the replica set has only one single primary member and no other members, operations read data from the primary member. 5. **nearest:** Operations read from a random eligible replica set member, irrespective of whether that member is a primary or secondary, based on a specified latency threshold.
10	Run following command to read documents of people collection from secondary server 27019. **db.people.find()**
11	Try to enter data in secondary server db.people.insertOne({name:”a1”}). It will not allow and gives message not primary.