**INDEXES**

**Table of Contents:**

1. **Introduction.**
2. **Index Table Example (structure).**
3. **Examining the Query Behaviour.**
4. **Covered Queries.**
5. **Type of Indexes.**
6. **Index Methods.**

**Introduction:**

Indexes in MongoDB enhance query performance by allowing the database to locate specific documents more efficiently. Instead of scanning the entire collection, queries can directly access the relevant documents through the sorted index fields (in either ascending or descending order). However, excessive indexing can impact performance, as MongoDB must update the index table with every data mutation (create, update, or delete operation). It is essential to use indexes judiciously to balance query speed and write performance.

Important Points:

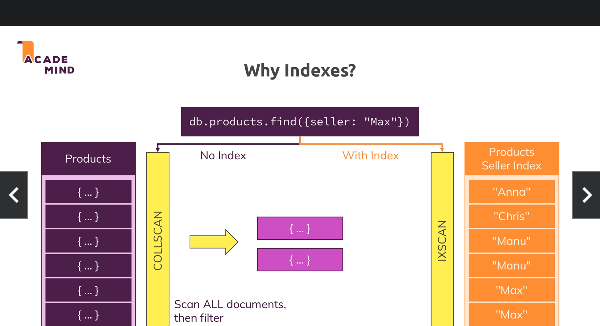
* Indexes can be in either of two order ascending or descending.
* Indexes are only useful when our goal is to fetch some fraction of data.
* Indexes must be used carefully, not to create too much indexes.
* Without index mongo-db performs collection scan (COLLSCAN) and with index it performs index scan (IXSCAN).
* The number of indexes is equals to the number of separate index tables the database has to maintain.
* With each Mutation operation (Create/Update/Delete) mongo-db recreates the index table, so incase of more indexes it will be performance costing.
* No. of Indexes ∝ 1 / mutation query performance
* Indexes makes the query faster by, not scanning the entire collection, but finding the matched value in the index table and returning the related documents.

**Index Table Example:**

When we create a index on a field, mongodb maintains a Hash table like this for that particular field as index table.

e.g. we have created index on the field age as { age: -1 }

|  |  |
| --- | --- |
| **Sorted Values** | **Memory Address of Document** |
| 90 => | Sjdsadjklsdjeowodfje |
| 87 => | Klsdlfjlfjkldfkljdfkljdklfkl |
| 72 => | Kfjkjklsdjfjkldfkljdsklfkjlsdfkl |
| ….. => | ……………………………………………. |



**Examining the Query behavior:**

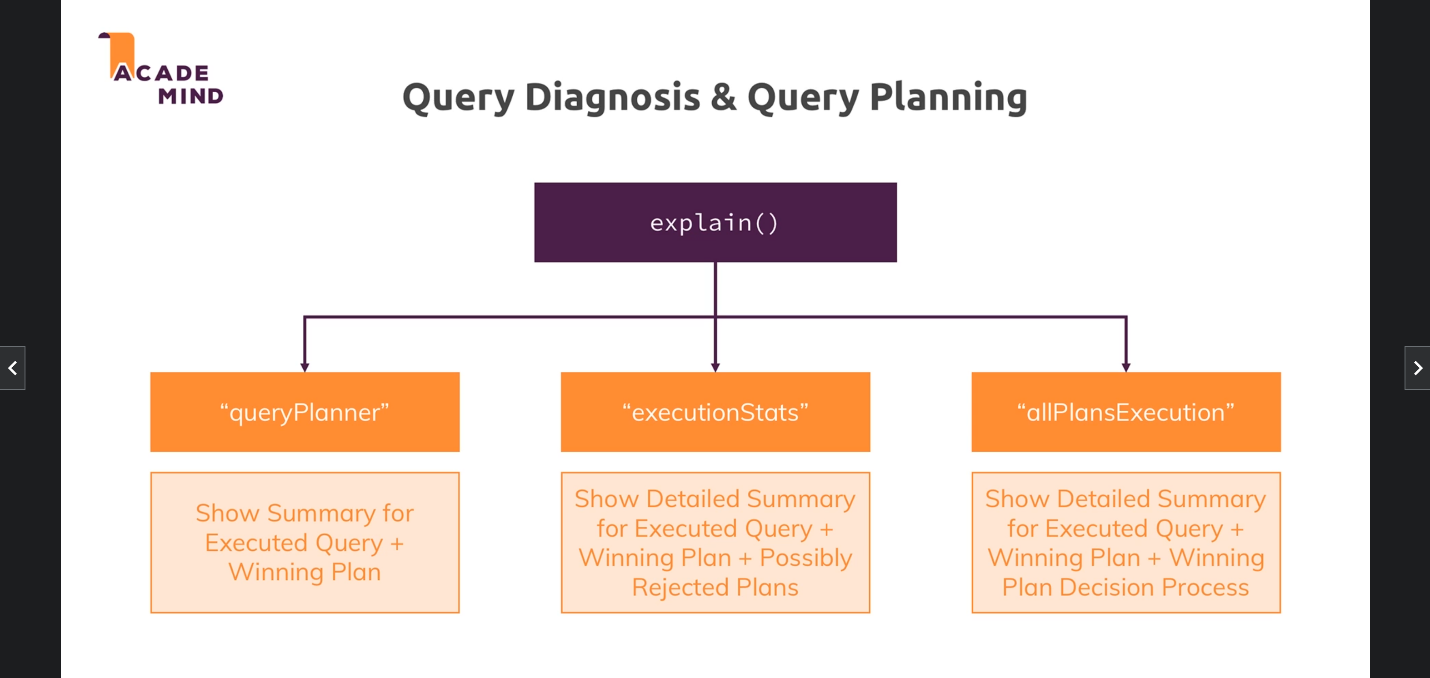
The .explain() and .explain(‘executionStats’) chaining methods are used to get insights abut the query execution like scan type, time taken, document scanned etc.

* db.<collection>.explain().find(<query>)

[returns the detailed stats of the query executions as per mongodb.]

* db.<collection>.explain(‘executionStats’).find(<query>)

[returns the detailed stats of how mongodb planned and executed the query.]



**Covered Queries:**

The Covered queries in MongoDB is a kind of indexed query where the query has no need to examine the document in order to return the result. i.e. the value is directly returned from the index table.

E.g. We have a person collection, with having index on the email field. Our covered query will look like :

* db.Person.find({ email: { $in: [‘h@gmail.co’, ‘s@gmail.co’] } }, { email: 1, \_id: 0 })

[This query will return only the email value from the index table, without having to examine the document to return other fields]

Note: You can also utilize the compound index to write covered queries that return multiple values.

**Types of indexes:**

Here we will list down the types of indexes in mongodb:

* **Primary Index:**  This is the default index of the mongodb reserved for \_id generated by mongodb itself and is always unique.
* **Secondary Index:**  This are the indexes that we can create manually on the collection, by default they are not unique.
  + - db.<collection>.createIndex( { field : 1/-1 }, { option1: value 1} )
* **Secondary Unique Index:**  This are the indexes that we can create manually, but they are unique in nature. i.e. not repeated values allowed.
  + - db.<collection>.createIndex({ ‘email’: -1 }, { unique: true })
* **Compound Index:**  This types of indexes can be created manually, and has combination of two or more fields to create a compound index.
  + When is index is made up of two or more field it is called compound index.
  + In compound index sorting occurres based on the left most field.
  + The left most field of the compound index can utilize the index standalone, other field need to be in combination.
    - db.<collection>.createIndex({field1: 1, field2: 1});
* **Partial Index:**  This types of index can be created manually, instead of creating index of all the documents in the table, it will create index table for specific documents in the table as per the condition.
  + Partial index is only made up on some specific set of documents.
  + A COLLSCAN is performed if we search value outside partial index, a IXSCAN is performed is search value inside the partial index.
  + These types of indexes can be used to speed up the mutation queries cause all mutations are not going to affect the partial index table.
    - db.<collection>.createIndex({‘age’: -1}, {partialFilterExpression: {“age”: {$gt: 50}}});
* **TTL(Time To Live) Index:**  This types of indexes can be made manually, these indexes define the time for which a documents should stay in the collection.
  + TTL indexes can only be made on the date fields (mostly createdAt).
  + TTL indexes are used to define the time for which the document stay in the mongodb collection.
    - db.<collection>.createIndex({ createdAt: 1 }, { expireAfterSeconds: 3600 })
* **Multi-Key Index :**  This types of indexes can be made manually, a index is called a multi key index if it made upon array field of the document.
  + Multi-key indexes are performance costing, cause on each mutation operation

Mongodb has to recreated the indexes with multiple values for a single document.

* + Multi Key index table is made by unwinding the array values to place one value at a time in the array table.
    - db.<collection>.createIndex({ <arrayField>: 1\-1 })
* **Text-Index:**  This type of indexes are made up on text fields, but any index made up on text field is not text index.

Real World Example : When we search a song by its lyrics on youtube, it performs the text search on the lyrics (text index) to get the matched songs back.

* + Text-index is made up on text field, but all indexes on text field is not text-index.
  + Text-index does not create index on the entire value, but on each words of the value.
  + One Collection can have only one text index (single field text index or compound text index).
  + A special keyword ‘text’ is used to create a text index.
    - db.<collection>.createIndex({description: ‘text’})
  + We can also create a single text compound index by merging the keywords of two or more fields and its still be treated as a single text index.
    - db.<collection>.createIndex({ title: ’text’, description: ‘text’ })

[It will create a single compound text index on by combining the values of title and description]

* + We can also assign the **default\_language** and **weights** for the text index fields during index creation.
    - db.<collection>.createIndex({ title:’text’, description: ‘text’ }, {default\_language:’english’, weights: {title: 10, description: 1}})
      * **default\_language**: This option is used to define the language that is stored in our text index (defaults to english).
      * **weights**: This option is used to define that which field contains our most important keywords basis of which the score will be calculated. Higher the weight, more important the keywords.
  + A special syntax is used to search through in text index, no field name is required cause ref: point 3.
    - db.<collection>.find({ $text: { $search: ‘random search text’ } }, { score: { $meta: ‘textScore’ } })

[This query will see through the text index and return the documents with score field that define the match score]

* + To get the highest matchScore document sorting can be done like:
    - db.<collection>.find({ $text: { $search: ‘random search text’ } }, { score: { $meta: ‘textScore’ } }).sort({ score: { $meta: ‘textScore’ } })
  + To exclude some words from matching use ‘**-**’ in front of them:
    - db.<collection>.find({ $text: { $search: ‘random -text’ } }, { score: { $meta: ‘textScore’ } }).sort({ score: { $meta: ‘textScore’ } })

[This query will return all the documents having random in their text index field but not text.]

* + To make case sensitive matching use **$caseSensitive**  operator which is defaults to false.
    - db.<collection>.find({ $text: { $search: ‘random -text’, $caseSensitive: true } }, { score: { $meta: ‘textScore’ } }).sort({ score: { $meta: ‘textScore’ } })

**Index Methods:**

Here we list down the methods used to create/See/Drop the indexes on the collections.

* **createIndex(keyObject, options) method**: This method is used to create a single index on the collection.

Syntax:

db.<collection>.createIndex( { field : 1/-1 }, { option1: value 1} )

Example:

db.<collection>.createIndex({ ‘email’: -1 }, { unique: true })

[will create secondary unique index on the field ‘email’ the documents.]

db.<collection>.createIndex({field1: 1, field2: 1});

[will create compound index on the fields ‘field1’ and ‘field2’ in ascending order as per field 1]

db.<collection>.createIndex({‘age’: -1}, {partialFilterExpression: {“age”: {$gt: 50}}});

[will create the index table for the documents where age is greater than 50.]

db.<collection>.createIndex({ createdAt: 1 }, { expireAfterSeconds: 3600 })

[Will create a TTL index on the createdAt field to delete documents automatically after an hour.]

db.<collection>.createIndex({‘arrayField’: 1\-1});

[Will create a multi-key index on the arrayField, cause we have created index on array]

db.<collection>.createIndex({ description: ‘text’ });

[Will create a text index on the description field]

* **getIndexes()**:  this method is used to get the list or indexes on the collection.

db.<collection>.getIndexes();

[returns the list of indexes on the collection]

* **dropIndex()**:  this method is used to drop an already existing index on the collection.

db.<collection>.dropIndex({field: 1 \ -1})

[drops the index from the field aka drops the index table.]

**Index Creation (Blocking/Non-Blocking):**

Generally during the creation of the indexes the collection is locked for the interval the index table is being created for the collection for obvious reasons. But in production locking a collection even for a fraction of second will create a issue here the **background: boolean (default false)** option comes to play, it will create the index table in background without blocking the collection.

* + Blocking index creation

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

[It will create a index in for-ground by blocking the entire collection]

* + Non-Blocking index creation

db.<collection>.createIndex({ field: 1 }, {background: true})

[It will create a index in background without blocking the entire collection]