Indexing in MongoDB

Google slide deck available here

What is an Index in MongoDB?

Indexes hold a **small portion of the collection's data** in a form that's easy to traverse.

Speed up queries and updates

Avoid disk I/O as queries otherwise **collection scans** are used

Overall **computation is reduced**

Indexes are designed to track a small portion of a collection’s data in a format that is quick and easy to traverse.
Their goal is to speed up queries and updates.
They help to avoid disk input/output as without indexes every document in a collection must be read to ensure the criteria of the query are met. This is called a collection scan.
Indexed help reduce the overall computation in the system.

When should Indexes be used?

Whenever you **query or update data** in a collection.

Developers and the **most frequent queries** should be used to determine which indexes should be used.

4 indexes is a good rule of thumb for the **ideal number for a given collection.**

64 indexes are the maximum per collection, **however above 20** and performances renders the system almost unusable for workloads.

An index should support any query that updates or queries data in a collection, if possible.
They should be present for the most frequent queries.
4 indexes are a good rule of thumb for the ideal number of indexes on a collection.
More than this and you should determine if the schema and indexing strategy shouldn't be refactored.

You can have at maximum 64 indexes per collection, however typically once there are 20 or more indexes on a collection then it can become highly unresponsive for a large number of workloads and use cases.

Indexes have associated costs

Indexes require **RAM.**

Avoid **unnecessary indexes at all cost**, otherwise the write. performance will suffer. Each index **adds 10% overhead**.

When does an **index entry get modified**?

- Data is inserted (applies to all indexes).
- Data is deleted (applies to all indexes).
- Data is updated in such a way that its indexed field changes.

It is important to keep in mind that every index that you have should be used by a query, otherwise an unused index will sit there, using up precious RAM space and slowing down writes.

Unnecessary indexes should be avoided as they will impact write performance. Each index adds approximately 10% overhead. How come an index slows down writes? Because it gets modified every time a field that is an index key is modified.

When does an index entry get modified?

An index is modified any time a document:

- Is inserted (applies to all indexes)
- Is deleted (applies to all indexes)
- Is updated in such a way that its indexed field changes

Quiz

Quiz

Which of the true for indexing in MongoDB? More than 1 answer choice can be correct.

- A. Supports query or update operators
- B. Necessary and required for querying
- C. All fields should be part of one or more indexes
- D. Indexes should be used on the most frequent queries
- E. Indexes do not add any performance impact on the database

Quiz

Which of the true for indexing in MongoDB?

- ✓ A. Supports query or update operators
- ✗ B. Necessary and required for querying
- ✗ C. All fields should be part of one or more indexes
- ✓ D. Indexes should be used on the most frequent queries
- ✗ E. Indexes do not add any performance impact on the database

CORRECT: Supports query or update operators. - This is correct. Indexes are designed to help these operators by more quickly returning the documents being acted on than having to perform a collection scan for all documents.
INCORRECT: Necessary and required for querying. - This is incorrect, in the sense that indexes apart from the _id index are not required. If there is no index then the query will use a collection scan to read all the documents to service the query.
INCORRECT: All fields should be part of one or more indexes. - This is incorrect, indexing should be limited to the fields that best support the queries so that it may not even be necessary to add all the fields used in a query to an index.
CORRECT: Indexes should be used on the most frequent queries. - This is correct and with a well designed indexing strategy significant performance improvements can be seen.
INCORRECT: Indexes do not add any performance impact on the database. - This is incorrect. Indexes do add an impact to performance, the index(es) must be updated whenever the fields indexes are changed or updated or deleted.

Quiz

Which of the true for indexing in MongoDB?

- ✓ A. Supports query or update operators
- ✗ B. Necessary and required for querying
- ✗ C. All fields should be part of one or more indexes
- ✓ D. Indexes should be used on the most frequent queries
- ✗ E. Indexes do not add any performance impact on the database

This is correct. Indexes are designed to help these operators by more quickly returning the documents being acted on than having to perform a collection scan for all documents.

CORRECT: Supports query or update operators. - This is correct. Indexes are designed to help these operators by more quickly returning the documents being acted on than having to perform a collection scan for all documents.

Quiz

Which of the true for indexing in MongoDB?

- ✓ A. Supports query or update operators
- ✗ B. Necessary and required for querying
- ✗ C. All fields should be part of one or more indexes
- ✓ D. Indexes should be used on the most frequent queries
- ✗ E. Indexes do not add any performance impact on the database

This incorrect, in the sense that indexes apart from the _id index are not required. If there is no index then the query will use a collection scan to read all the documents to service the query.

INCORRECT: Necessary and required for querying. - This is incorrect, in the sense that indexes apart from the _id index are not required. If there is no index then the query will use a collection scan to read all the documents to service the query.

Quiz

Which of the true for indexing in MongoDB?

- ✓ A. Supports query or update operators
- ✗ B. Necessary and required for querying
- ✗ C. **All fields should be part of one or more indexes**
- ✓ D. Indexes should be used on the most frequent queries
- ✗ E. Indexes do not add any performance impact on the database

This incorrect, indexing should be limited to the fields that best support the queries so that it may not even be necessary to add all the fields used in a query to an index.

INCORRECT: All fields should be part of one or more indexes. - This is incorrect, indexing should be limited to the fields that best support the queries so that it may not even be necessary to add all the fields used in a query to an index.

Quiz

Which of the true for indexing in MongoDB?

- ✓ A. Supports query or update operators
- ✗ B. Necessary and required for querying
- ✗ C. All fields should be part of one or more indexes
- ✓ D. **Indexes should be used on the most frequent queries**
- ✗ E. Indexes do not add any performance impact on the database

This is correct and with a well designed indexing strategy significant performance improvements can be seen.

CORRECT: Indexes should be used on the most frequent queries. - This is correct and with a well designed indexing strategy significant performance improvements can be seen.

Quiz

Which of the true for indexing in MongoDB?

- ✓ A. Supports query or update operators
- ✗ B. Necessary and required for querying
- ✗ C. All fields should be part of one or more indexes
- ✓ D. Indexes should be used on the most frequent queries
- ✗ E. Indexes do not add any performance impact on the database

This incorrect. Indexes do add an impact to performance, the index(es) must be updated whenever the fields indexes are changed or updated or deleted.

INCORRECT: Indexes do not add any performance impact on the database. - This is incorrect. Indexes do add an impact to performance, the index(es) must be updated whenever the fields indexes are changed or updated or deleted.

Types of indexes in MongoDB

What types of Indexes are available?

Single Field

Compound Index

Other types of specialized indexes including:

- Multikey Index
- Geospatial Index
- Text Index
- Hashed Index
- Time-To-Live (TTL) Index
- Hidden Index
- Partial Index
- Wildcard Index

Single-field indexes are the most common index type.

Compound indexes are used to support queries using two or more fields.

Multikey indexes allow arrays to be indexed by adding an index key per element in the array.

Geospatial indexes support queries on location fields, in MongoDB location fields can be GeoJSON objects or coordinate pairs. If you wanted to find the closest location within a specified geographical area then this index can support your query.

Text indexes allow for string fields or arrays with string elements to be effectively queried. If you had product descriptions or review fields then this type of index would help effectively support your query. A richer type of text index is available in MongoDB Atlas that allows for much greater support of these types of queries, it is the Atlas Search feature and uses Apache Lucene.

Hashed indexes are used to support querying in sharding. The hash of a value is indexed rather then the value itself.

Time-To-Live (TTL) indexes are used primarily for data expiration. They allow documents to be removed after a period of time or at a specific clock time. They are a specialised use of a single-field index but are worth discussing separately.

Hidden index are an optimisation feature for indexes in MongoDB. A index that is set to hidden will not be visible to the query planner and as such cannot be used to support a query. This type of index can be done on any index type with the exception of the _id index. The purpose of a hidden index is to all you to see what happens if you remove an index but gives the advantage that you haven't deleted it. If the index turns out to be used and important to your queries then it can be unhidden. Previously, if an index was removed it had to be completely deleted and if it was needed again then it would have to be rebuilt from scratch.

Partial indexes allow you to define a specific filter expression and only index the subset of documents in a collection that match the expression. This can reduce the storage requirements and reduce the performance costs for index creation and maintenance.

Wildcard indexes can be used to support queries against unknown or arbitrary fields. MongoDB supports dynamic schemas allowing applications to query against fields whose names cannot be known in advance or which may be arbitrary. This type of index can support querying this type of dynamic schema. This type of index has some limitations and is not designed to replace single or compound indexes so please refer to the MongoDB documentation for more details on how and when to specifically use this index type.

Quiz

Quiz

Fill in the blank for each of these questions from what we have just learnt.

A. The _____ index is primarily used for data expiration

B. The _____ index can be used to index a subset of documents within a collection using an expression

C. The _____ index supports dynamic schema where fields and their names may not be known in advance

D. A compound index supports queries on _____ fields

The first is:

A. The _____ index is primarily used for data expiration

Quiz

A. The **Time-To-Live (TTL)** index is primarily used for data expiration

B. The _____ index can be used to index a subset of documents within a collection using an expression

C. The _____ index supports dynamic schema where fields and their names may not be known in advance

D. A compound index supports queries on _____ fields

The Time-To-Live (TTL) index is primarily used for data expiration.

Quiz

A. The **Time-To-Live (TTL)** index is primarily used for data expiration

B. The _____ index can be used to index a subset of documents within a collection using an expression

C. The _____ index supports dynamic schema where fields and their names may not be known in advance

D. A compound index supports queries on _____ fields

The second question is:

A. The _____ index can be used to index a subset of documents within a collection using an expression.

Quiz

A. The **Time-To-Live (TTL)** index is primarily used for data expiration

B. The **Partial** index can be used to index a subset of documents within a collection using an expression

C. The _____ index supports dynamic schema where fields and their names may not be known in advance

D. A compound index supports queries on _____ fields

The Partial index can be used to index a subset of documents within a collection using an expression.

Quiz

A. The **Time-To-Live (TTL)** index is primarily used for data expiration

B. The **Partial** index can be used to index a subset of documents within a collection using an expression

C. The _____ index supports dynamic schema where fields and their names may not be known in advance

D. A compound index supports queries on _____ fields

The third question is:

A. The _____ supports dynamic schema where fields and their names may not be known in advance.

Quiz

A. The **Time-To-Live (TTL)** index is primarily used for data expiration

B. The **Partial** index can be used to index a subset of documents within a collection using an expression

C. The **Wildcard** index supports dynamic schema where fields and their names may not be known in advance

D. A compound index supports queries on _____ fields

The Wildcard index supports dynamic schema where fields and their names may not be known in advance.

Quiz

A. The **Time-To-Live (TTL)** index is primarily used for data expiration

B. The **Partial** index can be used to index a subset of documents within a collection using an expression

C. The **Wildcard** index supports dynamic schema where fields and their names may not be known in advance

D. A compound index supports queries on _____ fields

The fourth question is:

A. A compound index supports queries on _____ fields.

Quiz

A. The **Time-To-Live (TTL)** index is primarily used for data expiration

B. The **Partial** index can be used to index a subset of documents within a collection using an expression

C. The **Wildcard** index supports dynamic schema where fields and their names may not be known in advance

D. A compound index supports queries on **two or more** fields

A compound index supports queries on two or more fields.

Single Field Indexes

Let’s move to look at Single Field Indexes.

Single Field Indexes

Vast majority of all indexes created/used.

Optimise to help search for values within a given field.

Can be on **any data type**.

Caveat, the indexing an object type field does not index all the values separately.

The most common type of index created and used in MongoDB is the single field index.

It is designed to help optimise searching for values within a given field.

Single field indexes can be used on any data type.

Single field indexes can index an object type field but you should note that it does not index all the values separately.

Let's take a walkthrough of how to add an index but first let's look at what happens without an index.

Walkthrough of a normal db.<col>.find()

```
>>> db.inventory.drop()
```

To ensure we don’t have any old data that might interfere with our example, let’s drop this collection as we will add data into it in the next step.

Walkthrough of a normal db.<col>.find()

```
>>> db.inventory.drop()
>>> db.inventory.insertMany([
   { item: "journal", qty: 25, tags: ["blank", "red"], dim_cm: [ 14, 21 ] },
   { item: "notebook", qty: 50, tags: ["red", "blank"], dim_cm: [ 14, 21 ] },
   { item: "paper", qty: 100, tags: ["red", "blank", "plain"], dim_cm: [ 14, 21 ]
},
   { item: "planner", qty: 75, tags: ["blank", "red"], dim_cm: [ 22.85, 30 ] },
   { item: "postcard", qty: 45, tags: ["blue"], dim_cm: [ 10, 15.25 ] }
]);
```

In order to have some data, let's insert some documents into the inventory collection. We've highlighted the qty 75 in the planner document as we'll search for that in the next step.

Walkthrough of a normal db.<col>.find()

```
>>> db.inventory.drop()
>>> db.inventory.insertMany([
   { item: "journal", qty: 25, tags: ["blank", "red"], dim_cm: [ 14, 21 ] },
   { item: "notebook", qty: 50, tags: ["red", "blank"], dim_cm: [ 14, 21 ] },
   { item: "paper", qty: 100, tags: ["red", "blank", "plain"], dim_cm: [ 14, 21 ]
},
   { item: "planner", qty: 75, tags: ["blank", "red"], dim_cm: [ 22.85, 30 ] },
   { item: "postcard", qty: 45, tags: ["blue"], dim_cm: [ 10, 15.25 ] }
]);
>>> db.inventory.find({qty:75})
{ "_id" : ObjectId("5f5f8e0d789a1e68da1c0d7d"), "item" : "planner", "qty" : 75,
"tags" : [ "blank", "red" ], "dim_cm" : [ 22.85, 30 ] }
```

Let's now search for all documents where the quantity in the inventory is exactly 75 and we find the planner document.

```
db.inventory.drop();
db.inventory.insertMany( [
  { item: "journal", status: "A", size: { h: 14, w: 21,
uom: "cm" }, instock: [ { warehouse: "A", qty: 5 } ] },
  { item: "notebook", status: "A",  size: { h: 8.5, w: 11,
uom: "in" }, instock: [ { warehouse: "C", qty: 5 } ] },
  { item: "paper", status: "D", size: { h: 8.5, w: 11,
uom: "in" }, instock: [ { warehouse: "A", qty: 60 } ] },
  { item: "planner", status: "D", size: { h: 22.85, w: 30,
uom: "cm" }, instock: [ { warehouse: "A", qty: 40 } ] },
  { item: "postcard", status: "A", size: { h: 10, w:
15.25, uom: "cm" }, instock: [ { warehouse: "B", qty: 15
}, { warehouse: "C", qty: 35 } ] }
]);
db.inventory.find({qty:75});
```

See: https://docs.mongodb.com/manual/tutorial/project-fields-from-query-results/

Let's see how this query is interpreted by the DB

```
>>> db.inventory.find({qty:75}).explain()
```

An explain is the method that allows us to see how the database interprets and processes the query.
We can simply rerun the last query and add explain() to the end of the query to see how the DB interprets and processes the query.

For more details on explain, checkout the documentation page at https://docs.mongodb.com/manual/reference/method/db.collection.explain/index.html

The important point in this output is the COLLSCAN, which is short for Collection Scan. This means that to fulfil this query every document in the collection was looked at. This is not a very efficient approach and as the number of documents grow this will have significant performance impacts.

Let's see how this query is interpreted by the DB

```
>>> db.inventory.find({qty:75}).explain()
{
      "queryPlanner" : {
            "plannerVersion" : 1,
            "namespace" : "test.inventory",
            "indexFilterSet" : false,
            "parsedQuery" : { "qty" : { "$eq" : 75 } },
            "queryHash" : "36DB8386",
            "planCacheKey" : "36DB8386",
            "winningPlan" : {
                  "stage" : "COLLSCAN",
                  "filter" : { "qty" : { "$eq" : 75 } },
                  "direction" : "forward"
            },
            "rejectedPlans" : [ ]
      },
      "serverInfo" : {
```

The explain() command provides a lot of output, specifically we want to look at the winningPlan.
The important point in this output is the COLLSCAN, which is short for Collection Scan. This means that to fulfil this query every document in the collection was looked at. This is not a very efficient approach and as the number of documents grow this will have significant performance impacts.
This indicates that the query did not have a suitable index that could be used to service the query.

For more details on explain, checkout the documentation page at https://docs.mongodb.com/manual/reference/method/db.collection.explain/index.html

Explaining **explain()**

`explain()` verbosity can be adjusted:

- **default**: determines the winning query plan but does not execute query
- **executionStats**: executes query and gathers more statistics
- **allPlansExecution**: runs all candidate plans to completion and gathers statistics on all of these

Explain() is the function which provides details on how the query planner interpreted the query and what method it used to find and return the documents.

It has three levels of verbosity:

the default level which we have just covered and that only highlights the winning query plan but does not execute the query.

The next level of verbosity in terms of more information is executionStats. This level executes the query and gathers more detailed statistics than the default level.

The highest level of verbosity is the allPlansExecution level. This runs all of the candidate plans to completion and then provides detailed statistics on all of these.

Let's look at the query in more depth with the executionStats, firstly without an index and then by adding an index to compare the differences.

Let’s dig deeper into the explain statistics

```
>>> db.inventory.find({qty:75}).explain("executionStats")
{
      "queryPlanner" : {
                  "plannerVersion" : 1,
                  "namespace" : "test.inventory",
                  "indexFilterSet" : false,
                  "parsedQuery" : {
                        "qty" : {
                              "$eq" : 75
                        }
                  },
                  "winningPlan" : {
                        "stage" : "COLLSCAN",
                        "filter" : {
                              "qty" : {
                                    "$eq" : 75
                              }
                        },
                  "direction" : "forward"
            },
            "rejectedPlans" : [ ]
      },
```

This output shows similar results to the default setting, however let’s look at the next slide to see some more detailed statistics in the output

Let's dig deeper into the explain statistics

```
>>> db.inventory.find({qty:75}).explain("executionStats")
{....
      "executionStats" : {
            "executionSuccess" : true,
            "nReturned" : 1,
            "executionTimeMillis" : 0,
            "totalKeysExamined" : 0,
            "totalDocsExamined" : 5,
            "executionStages" : {
                  "stage" : "COLLSCAN",
                  "filter" : { "qty" : { "$eq" : 75 } },
                  "nReturned" : 1,
                  "executionTimeMillisEstimate" : 0,
                  "works" : 7, "advanced" : 1,
                  "needTime" : 5, "needYield" : 0,
                  "saveState" : 0, "restoreState" : 0,
                  "isEOF" : 1, "direction" : "forward",
                  "docsExamined" : 5
            }
      },

            "rejectedPlans" : [ ]
      },
```

We can see more details here on the query, that it needed to examine 5 documents in order to return a single document to satisfy the query criteria. It also didn't examine any keys as there are no indexes applicable for this query.

totalDocsExamined is the statistic that indicates how many documents were examined by the query, 5 in this example.
totalKeysExamined is the statistic that indicates how many index keys were examined by the query, 0 in this example as there is no suitable index available.
nReturned is the number of documents return by the query that satisfied the query criteria.

What does adding an index do?

```
>>> db.inventory.createIndex({qty:1})
```

Before re-running the query, let’s add a single field index to the collection to support the query we want to run.

What does adding an index do?

```
>>> db.inventory.createIndex({qty:1})
>>> db.inventory.find({qty:75}).explain("executionStats")
{....
      "executionStats" : {
            "executionSuccess" : true,
            "nReturned" : 1,
            "executionTimeMillis" : 0,
            "totalKeysExamined" : 1,
            "totalDocsExamined" : 1, ...
            "executionStages" : {
                  "stage" : "FETCH", ...
                  "inputStage" : {
                        "stage" : "IXSCAN",
                        "nReturned" : 1,
                        "keyPattern" : { "qty" : 1},
                        "indexName" : "qty_1",
                        "keysExamined" : 1, ...
            }
      },

            "rejectedPlans" : [ ]
      },
```

After adding the index, let's re-run the query from our previous example and again use the explain function to explore how the query is being processed by the database

We can see even more details here on the query with the explain after adding the index. We can see that it now needed to examine 1 key in order to return a single document to satisfy the query criteria.

The 'qty' index we added was able to service the query and hence it was an index rather than collection scan (IXSCAN) which was used. This is much more efficient and scalable than the previous query without an index.

Acronyms used by **explain()**

`explain()` uses several acronyms to highlight the various aspects:

- **COLLSCAN** (Looks through the documents in a collection)
- **IXSCAN** (Looks through index)
- **FETCH** (Reads a known document) retrieval of the document
- **totalKeysExamined/totalDocsExamined** how many of the keys or documents needed to be read as part of the query.
- **nReturned** how many documents were returned by the query.

Explain uses a number of acronyms that it's worth repeating the definitions of here to ensure these key concepts are clear. These are the main items that you should look for when visually scanning the output from the explain() to help you pinpoint the key statistics on your query.

COLLSCAN - this means the query must look through all of the documents in the collection to fulfill the query
IXSCAN - this means the query must look through an index to fulfill the query.
FETCH - this is where the query planner reads and returns a known document
totalKeysExamined - this is the number of index keys examined in servicing the query
totalDocsExamined - this is the number of documents examined in servicing the query
nReturned - this is how many documents that this query returned to fulfil the query's criteria

Quiz

Quiz

Which of the true for explain() in MongoDB? More than 1 answer choice can be correct.

- A. Supports various levels of verbosity of detail.
- B. executionStats is the default level of verbosity.
- C. All calls to explain() will run all the candidate plans for a query.
- D. Explain() provides output on how many index keys and how many documents were read to service the query.

Quiz

Which of the true for explain() in MongoDB?

- ✓ A. Supports various levels of verbosity of detail
- ✗ B. executionStats is the default level of verbosity
- ✗ C. All calls to explain() will run all the candidate plans for a query
- ✓ D. Explain() provides output on how many index keys and how many documents were read to service the query

CORRECT: Supports various levels of verbosity of detail. - It supports three levels of verbosity
INCORRECT: executionStats is the default level of verbosity. - Default is the default level of verbosity
INCORRECT: All calls to explain() will run all the candidate plans for a query. - Only allPlansExecuted performs this.
CORRECT: Explain() provides output on how many index keys and how many documents were read to service the query. - This is correct.

Quiz

Which of the true for explain() in MongoDB?

- ✓ A. **Supports various levels of verbosity of detail**
- ✗ B. executionStats is the default level of verbosity
- ✗ C. All calls to explain() will run all the candidate plans for a query
- ✓ D. Explain() provides output on how many index keys and how many documents were read to service the query

This is correct. It supports three levels of verbosity.

CORRECT: Supports various levels of verbosity of detail. - This is correct. It supports three levels of verbosity

Quiz

Which of the true for explain() in MongoDB?

- A. Supports various levels of verbosity of detail
- B. executionStats is the default level of verbosity
- C. All calls to explain() will run all the candidate plans for a query
- D. Explain() provides output on how many index keys and how many documents were read to service the query

This incorrect. Default is the default level of verbosity.

INCORRECT: executionStats is the default level of verbosity. - This is incorrect. Default is the default level of verbosity

Quiz

Which of the true for explain() in MongoDB?

- ✓ A. Supports various levels of verbosity of detail
- ✗ B. executionStats is the default level of verbosity
- ✗ C. **All calls to explain() will run all the candidate plans for a query**
- ✓ D. Explain() provides output on how many index keys and how many documents were read to service the query

This incorrect. Only allPlansExecuted performs this.

INCORRECT: All calls to explain() will run all the candidate plans for a query. - This is incorrect. Only allPlansExecuted performs this.

Quiz

Which of the true for explain() in MongoDB?

- ✓ A. Supports various levels of verbosity of detail
- ✗ B. executionStats is the default level of verbosity
- ✗ C. All calls to explain() will run all the candidate plans for a query
- ✓ D. **Explain() provides output on how many index keys and how many documents were read to service the query**

This is correct. Explain provides these details.

CORRECT: Explain() provides output on how many index keys and how many documents were read to service the query. - This is correct. Explain provides these details

Multikey Indexes

Multikey Indexes

A multikey index provides on index entry for **each value in an array**.

- Supports primitives, documents, or sub-arrays.

Created with the **same syntax** as normal indexes.

A compound multikey index can only have **at most one indexed field whose value is an array**.

Each **array element is given one entry in the index**. This can create very large indexes in comparison to single field indexes.

A multikey index provides one index entry per value in an array. It can support documents, primitives, or sub-arrays.

It uses the normal/standard syntax for normal indexes.

A compound multikey index can have only one indexed field whose value is an array.

Every and each array element is assigned an index entry, this can create very large indexes in comparison to single field indexes.

Your turn to try out indexing and explains.

Change **<A>** and **<B>** as necessary below to create the index and then to the fields you have created the index on.

How many keys were examined? How many documents were examined?

```
>>> db.inventory.drop()
>>> db.inventory.insertMany( [ { _id: 1, item: "abc", stock: [ { size: "S", color:
"red", quantity: 25 }, { size: "S", color: "blue", quantity: 10 }, { size: "M", color:
"blue", quantity: 50 } ] }, { _id: 2, item: "def", stock: [ { size: "S", color: "blue",
quantity: 20 }, { size: "M", color: "blue", quantity: 5 }, { size: "M", color: "black",
quantity: 10 }, { size: "L", color: "red", quantity: 2 } ] }, { _id: 3, item: "ijk",
stock: [ { size: "M", color: "blue", quantity: 15 }, { size: "L", color: "blue",
quantity: 100 }, { size: "L", color: "red", quantity: 25 } ] } ])
>>> db.inventory.createIndex( { "stock.size": <A>, "stock.quantity": <B> } )
>>> db.inventory.find( { <A>: "S", <B>: { $gt: 20 } } ).explain("executionStats")
```

Let's dive into a hands-on, in this exercise you will use the example code to drop the existing inventory collection and recreate it with some documents. This will give us the baseline from which you can then try creating an indexing and interpreting the explain output for a query.

Here is the complete code for this exercise with the solution below:

```
db.inventory.drop();
db.inventory.insertMany( [ { _id: 1, item: "abc", stock: [ {
size: "S", color: "red", quantity: 25 }, { size: "S", color:
"blue", quantity: 10 }, { size: "M", color: "blue", quantity:
50 } ] }, { _id: 2, item: "def", stock: [ { size: "S", color:
"blue", quantity: 20 }, { size: "M", color: "blue", quantity: 5
}, { size: "M", color: "black", quantity: 10 }, { size: "L",
color: "red", quantity: 2 } ] }, { _id: 3, item: "ijk", stock:
[ { size: "M", color: "blue", quantity: 15 }, { size: "L",
color: "blue", quantity: 100 }, { size: "L", color: "red",
quantity: 25 } ] } ]);
db.inventory.createIndex( { "stock.size": 1, "stock.quantity":
1 } );
db.inventory.find( { "stock.size": "S", "stock.quantity": {
$gt: 20 } } ).explain("executionStats");
```

The answers are:

- 3 index keys were examined

- 2 documents were examined

Results for indexing and explains.

For the fields to the query **<A>** should be **"stock.size"** and **<B>** should be **"stock.quality"**.

```
>>> db.inventory.find( { “stock.size” : "S", “stock.quality”: { $gt: 20 } }
).explain("executionStats")
```

In this case, let’s set <A> to “stock.size” and <B> to “stock.quality” as well as adding the “executionStats” option to the explain function. We’ll now run the query.

Results for indexing and explains.

For the fields to the query <A> should be "stock.size" and <B> should be "stock.quality".

3 keys were examined and 2 documents were examined to return 1 document.

```
>>> db.inventory.find( { “stock.size” : "S", “stock.quality”: { $gt: 20 } }
).explain("executionStats")
…
      "executionStats" : {
            "executionSuccess" : true,
            "nReturned" : 1,
            "executionTimeMillis" : 0,
            "totalKeysExamined" : 3,
            "totalDocsExamined" : 2,
…
```

Let’s break down the results from the “executionStats” output, firstly the query examined 3 index keys indicated by “totalKeysExamined.,

Results for indexing and explains.

For the fields to the query <A> should be `"stock.size"` and <B> should be `"stock.quality"`.

3 keys were examined and **2 documents were examined** to return 1 document.

```
>>> db.inventory.find( { “stock.size” : "S", “stock.quality”: { $gt: 20 } }
).explain("executionStats")
…
      "executionStats" : {
            "executionSuccess" : true,
            "nReturned" : 1,
            "executionTimeMillis" : 0,
            "totalKeysExamined" : 3,
            "totalDocsExamined" : 2,
…
```

The query looked at 2 documents as indicated by the “totalDocsExamined” field in the “executionStats” output.

Results for indexing and explains.

For the fields to the query <A> should be `"stock.size"` and <B> should be `"stock.quality"`.

3 keys were examined and 2 documents were examined **to return 1 document**.

```
>>> db.inventory.find( { “stock.size” : "S", “stock.quality”: { $gt: 20 } }
).explain("executionStats")
…
      "executionStats" : {
            "executionSuccess" : true,
            "nReturned" : 1,
            "executionTimeMillis" : 0,
            "totalKeysExamined" : 3,
            "totalDocsExamined" : 2,
…
```

The final piece of information of interest from this piece of the “executionStats” output is that the query returned a single / one document.

Compound Indexes

Compound Indexes

Indexes created on **two or more field**, with a maximum of **32 fields**.

Direction and **field order** are of critical importance.

A compound index can be used as long as **the first field in index is in the query** then the other index fields do not need to be part of the query.

ESR - **Equality** then **Range** or **Sort** (usually sort)

A limit of **one array field per compound index**.

Compound indexes are created on two or more fields, they have a limit of using a maximum of 32 fields.

In creating these indexes, the direction and field ordering are of critical importance to ensure the maximum utility of the index.

A compound index can be used in any query once the first field in the index is in the query, this is regardless of whether or not other fields from the index are present or not in the query.

Equality-Sort-Range This highlights Equality First then narrow down the amount of documents that will need to be queried by range or sorted. Fewer documents will mean less work required which ranks this index higher in the query planner's assessment, making it more likely to be the index that is chosen to support the query.

ESR applies to most but not to all situations, so it's a guide rather than a rule.

There is a limit of one array field per compound index.

An example of field ordering

```
find({country:"IE",city:"Dublin"})

createIndex({country:1,state:1,city:1})

createIndex({country:1,city:1,state:1})
```

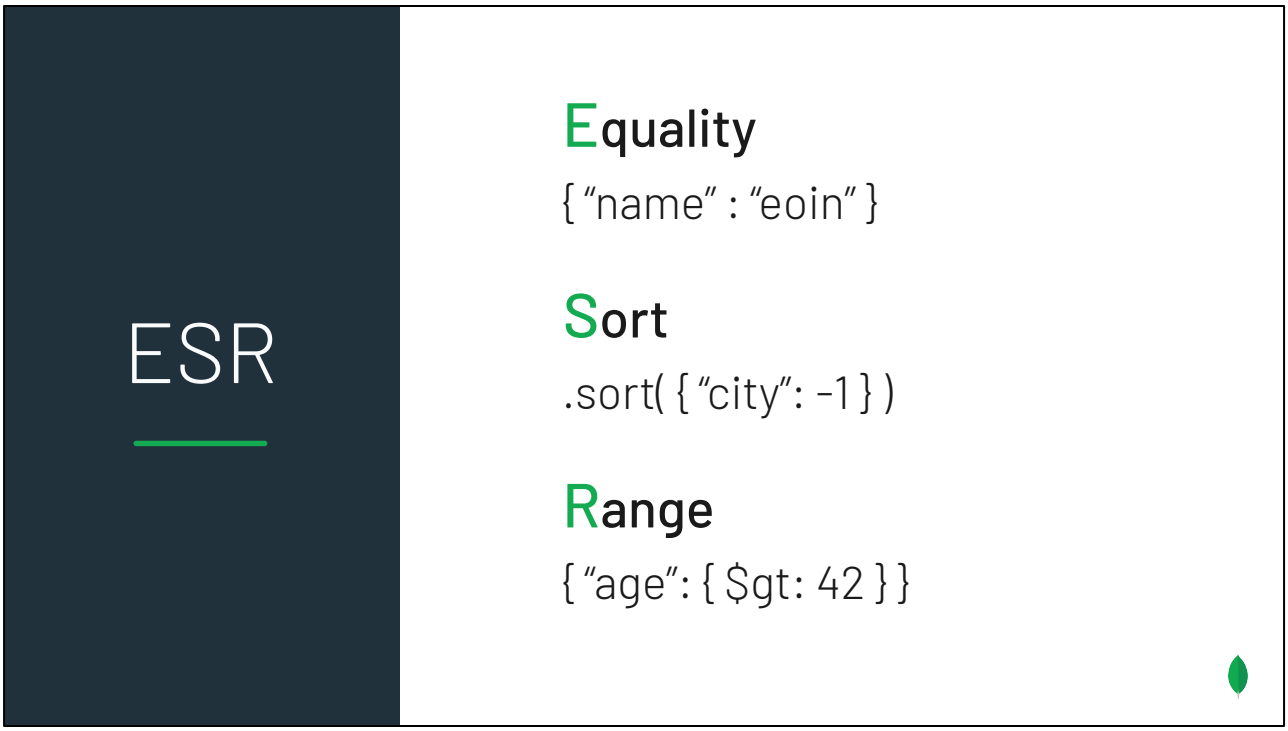
The latter index `{country:1,city:1,state:1}` is the better choice. It doesn't requiring examining all the keys for that country and for every state in that country.

It goes directly to the country and to the city.

Given this example query for location with a city, county and where the document also contains the state geographical information.
We can create the index on country, state, and city.
We can equally create the index on country, city, and state.

The ordering is important as the latter index on country, city and state is a better choice. The ordering prevents the query from needing to examine all of the state keys for the country.

Instead it can query directly to the country and to the city required.



Taking an aside for a moment, let's look at the ESR guide. Let's first define ESR.

E is for Equality

S is for Sort

R is for Range

Cardinality

Attempts to quantify how unique the data values are for the field and the given number of values possible for the field.

Distinct Values / Number of Values

High Cardinality: _id

Medium Cardinality: name

Low Cardinality: isEmployee

Cardinality is a metric that attempts to quantify how unique the data values are for a given field and the given number of possible values for that field.

_id is an example of a high cardinality field, name would be a medium cardinality field and the boolean isEmployee would be a low cardinality field.

Selectivity

Selectivity defines how much an index can reduce the items to examined

Effective indexes are more selective

Compound indexes **improve the selectivity** of low-selectivity fields

Include the low-selectivity field (name) and another field (age)

Selectivity defines how much an index helps in reducing the number of items to be examined when fulfilling the query.

More selectivity indexes are more effective.

A compound index can help improve the selectivity of low-selectivity fields (say combing name with another field such as age).

Selectivity and Compound Indexes

If we are creating a compound index and the query matches on two fields Name and Age, then creating an index {Name, Age} or {Age, Name} will have same performance irrespective of the selectivity of each individual field.

{ Name, Age } **is equally selective as** { Age, Name }

It's important to note that is we have a compound index with a query which is a match on both fields say Name and Age. The ordering of these fields is irrelevant as either ordering is equally selective.

Quiz

Quiz

Which of the following are true for compound indexes in MongoDB?
More than 1 answer choice can be correct.

- A. Field direction and field ordering are not important
- B. It can be used for a query only if all the fields in the query are in the index
- C. 64 is the maximum number of fields be used in this type of index
- D. A limit of one array field per compound index

Quiz

Which of the following are true for compound indexes in MongoDB?

- ✗ A. Field direction and field ordering are not important
- ✗ B. It can be used for a query only if all the fields in the query are in the index
- ✗ C. 64 is the maximum number of fields be used in this type of index
- ✓ D. A limit of one array field per compound index

INCORRECT: Field direction and field ordering are not important. - This is false as the ordering of fields and their direction are very important to the performance of a compound index.
INCORRECT: It can be used for a query only if all the fields in the query are in the index. - This is not true, you can use a compound index so long as the first field of the index is in the query.
INCORRECT: 64 is the maximum number of fields be used in this type of index. - 64 is the number of indexes you can have in a collection, 32 is the number of fields you can index with a compound index.
CORRECT: A limit of one array field per compound index. - You can use up to 32 fields in this type of index but only one may be an array field.

Quiz

Which of the following are true for compound indexes in MongoDB?

- A. Field direction and field ordering are not important
- B. It can be used for a query only if all the fields in the query are in the index
- C. 64 is the maximum number of fields be used in this type of index
- D. A limit of one array field per compound index

This incorrect. The ordering of fields and their direction are very important to the performance of a compound index.

INCORRECT: Field direction and field ordering are not important. - This is incorrect. The ordering of fields and their direction are very important to the performance of a compound index.

Quiz

Which of the following are true for compound indexes in MongoDB?

- A. Field direction and field ordering are not important
- B. It can be used for a query only if all the fields in the query are in the index
- C. 64 is the maximum number of fields be used in this type of index
- D. A limit of one array field per compound index

This incorrect. You can use a compound index so long as the first field of the index is in the query.

INCORRECT: It can be used for a query only if all the fields in the query are in the index. - This is incorrect. You can use a compound index so long as the first field of the index is in the query.

Quiz

Which of the following are true for compound indexes in MongoDB?

- A. Field direction and field ordering are not important
- B. It can be used for a query only if all the fields in the query are in the index
- C. 64 is the maximum number of fields be used in this type of index
- D. A limit of one array field per compound index

This incorrect. 64 is the number of indexes you can have in a collection, 32 is the number of fields you can index with a compound index.

INCORRECT: 64 is the maximum number of fields be used in this type of index. - This is incorrect. 64 is the number of indexes you can have in a collection, 32 is the number of fields you can index with a compound index.

Quiz

Which of the following are true for compound indexes in MongoDB?

- A. Field direction and field ordering are not important
- B. It can be used for a query only if all the fields in the query are in the index
- C. 64 is the maximum number of fields be used in this type of index
- D. A limit of one array field per compound index

This is correct. You can use up to 32 fields in this type of index but only one may be an array field.

CORRECT: A limit of one array field per compound index. - This is correct. You can use up to 32 fields in this type of index but only one may be an array field.

ESR - Message Board Example

In this section of the lesson, we'll talk about the equality sort range guidance in the context of a message board example application.

ESR Message Board Example: Let's add data

```
>>> db.messages.drop()
>>> a = [
{ "timestamp": 1, "username": "anonymous", "rating": 3 },
{ "timestamp": 2, "username": "anonymous", "rating": 5 },
{ "timestamp": 3, "username": "sam", "rating": 1 },
{ "timestamp": 4, "username": "anonymous", "rating": 2 },
{ "timestamp": 5, "username": "martha", "rating": 5 } ]
>>> db.messages.insertMany(a)
//Index on timestamp
>>> db.messages.createIndex( { timestamp: 1 } )
```

In this example of a message board application, let's firstly add some data to help us explore equality, sort and range.

As good practice, we'll clear the collection before we add data to it to ensure we start from a known good state / set of documents. Let's start with an index on just the timestamp, that is our first criteria after all.

Here's the code that you can cut and paste to add the data:

```
db.messages.drop()
a = [
{ "timestamp": 1, "username": "anonymous", "rating": 3 },
{ "timestamp": 2, "username": "anonymous", "rating": 5 },
{ "timestamp": 3, "username": "sam", "rating": 1 },
{ "timestamp": 4, "username": "anonymous", "rating": 2 },
{ "timestamp": 5, "username": "martha", "rating": 5 } ]
db.messages.insertMany(a)
db.messages.createIndex( { timestamp: 1 } )
```

ESR Message Board Example: Let's explain it!

```
>>> db.messages.find({timestamp: { $gte: 2, $lte: 4 } } )
                                .explain("executionStats")
//"nReturned" : 3,
//"executionTimeMillis" : 0,
//"totalKeysExamined" : 3,
//"totalDocsExamined" : 3,
// Looks great. nReturned = totalKeysExamined
// How will it hold up when we add the {username: "anonymous"}
criteria?
```

the explain results look good:
"executionSuccess" : true,
"nReturned" : 3,
"executionTimeMillis" : 0,
"totalKeysExamined" : 3,
"totalDocsExamined" : 3,

Let's add the {username: "anonymous"} criteria to the query

Let's updating the query with `username`

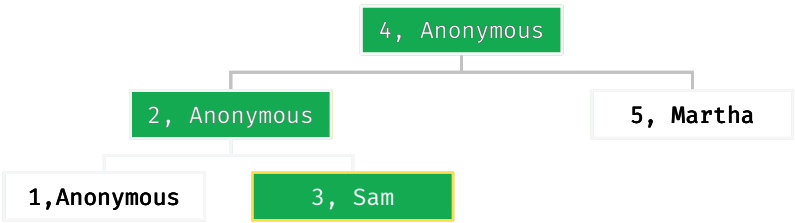
```
>>> db.messages.find({timestamp: { $gte: 2, $lte: 4 },
                      username: "anonymous" }).explain("executionStats")
// Less great. nReturned < totalKeysExamined
// Should we add username to the index?

db.messages.createIndex( { timestamp: 1, username: 1 } )
// Still nReturned < totalKeysExamined :(
```

How does the explain result change with this additional query requirement?
What happens when we traverse the existing index?

Index in wrong order

```
Query: {timestamp:{$gte:2, $lte:4}, username:"anonymous"}
Index: { timestamp: 1, username: 1 }
```



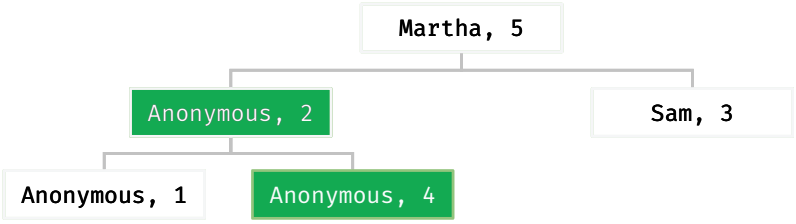
1. Must do range part first as timestamp is first in index
2. Start at first timestamp >= 2
3. Walk tree L to R until timestamp not <= 4 (3 nodes) check each if 'anonymous'
4. Return only 2 of the three nodes visited (2 and 4)

Keeping the same query and looking at our first index on timestamp and username, let's dive into how it is represented (see the graphic which represents how it is stored) and how we need to walk the index to find the documents that match the query criteria.

Firstly, we need to do a a range part as the first index field is the timestamp, so we start at first timestamp which is greater than or equal to 2 and then from left to right we walk the tree until the timestamp is not less than or equal to 4. This means we walk 3 nodes checking if each has a username equal to "anonymous". We find only 2 match the username so we return these.

Index in correct order

```
Query: {timestamp:{$gte:2, $lte:4}, username:"anonymous"}
Index: { username: 1, timestamp: 1 }
```



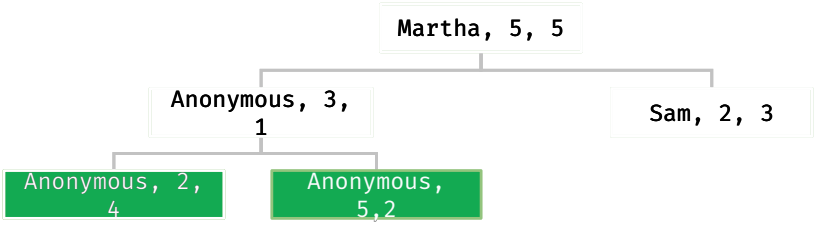
1. Exact Match at start filters down the tree to walk (Just Anonymous).
2. Find first Anonymous where timestamp >= 2
3. Walk tree whilst Anonymous & timestamp <= 4
4. Visits only 2 index nodes in total (2 and 4)

If we reorder the index to be username and then timestamp, we can improve the performance of the index to service the query.

We find the first ‘anonymous’ index record where the timestamp is greater than or equal to 2. We then walk the tree for username being ‘anonymous’ and where the timestamp is less than or equal to 4. This means we’ll only need to visit two index nodes, 2 and 4.

Add rating to the query and sort before range

```
Query: { timestamp:{$gte:2, $lte:4}, username:"anonymous" }.sort( { rating: 1} )
Index: { username: 1, rating: 1, timestamp: 1 }
```



1. Exact Match at start filters down the tree to walk (Just Anonymous).
2. Find first Anonymous where timestamp >= 2
3. Walk tree whilst Anonymous & timestamp <= 4
4. Visits only 2 index nodes in total (2 and 4)

If we change the query as we also want to sort the results, we can use the index to actually perform the sort. The index should then be username, then rating and then timestamp. This allows the index to sort the query results rather than needing to do this separately with the complete results of the query.

We find the first ‘anonymous’ index record where the timestamp is greater than or equal to 2. We then walk the tree for username being ‘anonymous’ and where the timestamp is less than or equal to 4. This means we’ll only need to visit two index nodes where the timestamp values are 2 and 4.

Indexing commands summary

Action	Command
Create an index	`db.<coll>.createIndex()`

To create, view, analyze our indexes for space and for how they are interpreted as well as to remove an index we can use these commands.

To create an index use the command db.<coll>.createIndex()

Indexing commands summary

Action	Command
Create an index	`db.<coll>.createIndex()`
List all the indexes	`db.<coll>.getIndexes()`

To list the indexes of a collection use the command db.<coll>.getIndexes()

Indexing commands summary

Action	Command
Create an index	`db.<coll>.createIndex()`
List all the indexes	`db.<coll>.getIndexes()`
See how much space they take up	`db.<coll>.stats().indexSizes`

To see how much space the indexes of a collection use then use the command db.<coll>.stats().indexSizes

Indexing commands summary

Action	Command
Create an index	`db.<coll>.createIndex()`
List all the indexes	`db.<coll>.getIndexes()`
See how much space they take up	`db.<coll>.stats().indexSizes`
See how they are used	`db.<coll>.find().explain()`

To see how the indexes of a collection are being used then use the command db.<coll>.find().explain()

Indexing commands summary

Action	Command
Create an index	`db.<coll>.createIndex()`
List all the indexes	`db.<coll>.getIndexes()`
See how much space they take up	`db.<coll>.stats().indexSizes`
See how they are used	`db.<coll>.find().explain()`
Remove an index	`db.<coll>.dropIndex()`

To drop an index then use the command db.<coll>.dropIndex()

Quiz

Quiz

Fill in the blank for each of these questions from what we have just learnt.

A. The _____ command creates an index

B. The _____ command removes an index

C. The _____ command can be used to show how the query planner interprets the query and the indexes able to service it

D. The _____ Sort Range rule of thumb is helpful in ordering the fields of an index for better performance

Fill in the blanks for each of these questions on indexing.
The first is:

A. The _____ command creates an index

Quiz

A. The **createIndex()** command creates an index

B. The _____ command removes an index

C. The _____ command can be used to show how the query planner interprets the query and the indexes able to service it

D. The _____ Sort Range rule of thumb is helpful in ordering the fields of an index for better performance

The createIndex() command creates an index.

Quiz

A. The **createIndex()** command creates an index

B. The _____ command removes an index

C. The _____ command can be used to show how the query planner interprets the query and the indexes able to service it

D. The _____ Sort Range rule of thumb is helpful in ordering the fields of an index for better performance

The second question is:

A. The _____ command removes an index

Quiz

A. The **createIndex()** command creates an index

B. The **dropIndex()** command removes an index

C. The _____ command can be used to show how the query planner interprets the query and the indexes able to service it

D. The _____ Sort Range rule of thumb is helpful in ordering the fields of an index for better performance

The dropIndex() command removes an index.

Quiz

A. The **createIndex()** command creates an index

B. The **dropIndex()** command removes an index

C. The _____ command can be used to show how the query planner interprets the query and the indexes able to service it

D. The _____ Sort Range rule of thumb is helpful in ordering the fields of an index for better performance

The third question is:

A. The _____ command can be used to show how the query planner interprets the query and the indexes able to service it.

Quiz

A. The **createIndex()** command creates an index

B. The **dropIndex()** command removes an index

C. The **explain()** command can be used to show how the query planner interprets the query and the indexes able to service it

D. The _____ Sort Range rule of thumb is helpful in ordering the fields of an index for better performance

The explain() command can be used to show how the query planner interprets the query and the possible indexes to service it.

Quiz

A. The **createIndex()** command creates an index

B. The **dropIndex()** command removes an index

C. The **explain()** command can be used to show how the query planner interprets the query and the indexes able to service it

D. The _____ Sort Range rule of thumb is helpful in ordering the fields of an index for better performance

The fourth question is:

A. The _____ Sort Range rule of thumb is helpful in ordering the fields of an index for better performance

Quiz

A. The **createIndex()** command creates an index

B. The **dropIndex()** command removes an index

C. The **explain()** command can be used to show how the query planner interprets the query and the indexes able to service it

D. The **Equality Sort Range** rule of thumb is helpful in ordering the fields of an index for better performance

The Equality Sort Range rule of thumb is helpful in ordering the fields of an index for better performance.

Continue Learning!

MongoDB University has free self-paced courses and labs ranging from beginner to advanced levels.

Github Student Developer Pack

Sign up for the MongoDB Student Pack to receive $50 in Atlas credits and free certification!

This concludes the material for this lesson. However, there are many more ways to learn about MongoDB and non-relational databases, and they are all free! Check out MongoDB's University page to find free courses that go into more depth about everything MongoDB and non-relational. For students and educators alike, MongoDB for Academia is here to offer support in many forms. Check out our educator resources and join the Educator Community. Students can receive $50 in Atlas credits and free certification through the Github Student Developer Pack.