NIIT

Performance

Objectives

- By the end of this session, you should be able to:
  - Query Analysis
  - Query Execution Stages in Mongo DB
  - Explaining the Query
  - Introduction to Indexes
  - Creating and listing indexes
  - Hiding and dropping indexes
  - Types of indexes
  - Properties of Indexes

Query Analysis

- In order to write efficient queries, it is important to analyze them, find any possible performance issues, and fix them.
- This technique is called performance optimization.
- There are many factors that can negatively affect the performance of a query, such as incorrect scaling, incorrectly structured collections, and inadequate resources such as RAM and CPU.
- However, the biggest and most common factor is the difference between the number of records scanned and the number of records returned during the query execution.
- The greater the difference is, the slower the query will be.
- In MongoDB, this factor is the easiest to address and is done using indexes

Query Execution Stages in Mongo DB

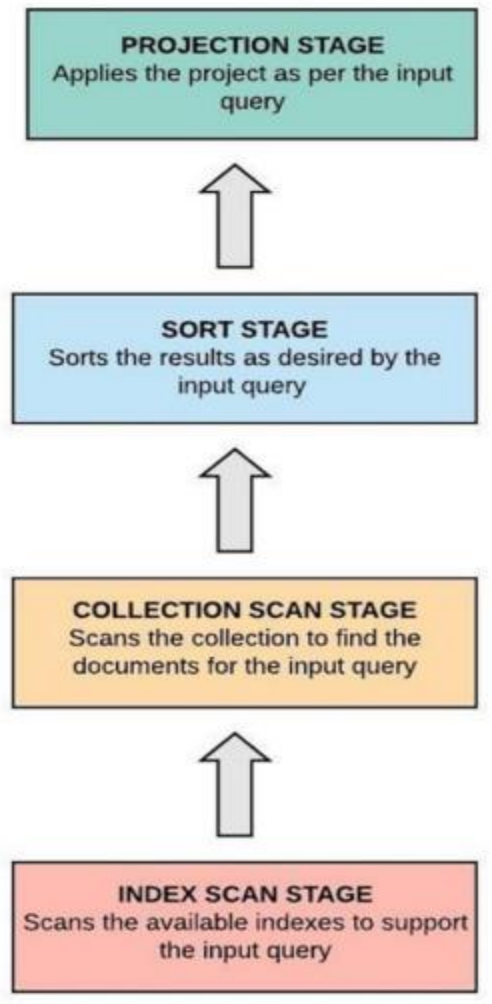

PROJECTION STAGE
Applies the project as per the input query
SORT STAGE
Sorts the results as desired by the input query
COLLECTION SCAN STAGE
Scans the collection to find the documents for the input query
INDEX SCAN STAGE
Scans the available indexes to support the input query

Figure 9.1: Query execution stages

Explaining the Query

- The explain() function is extremely useful for exploring the internal workings of a query.
- The most important metrics it can give us are as follows:
    - Query execution time
    - Number of documents scanned
    - Number of documents returned
    - The index that was use

Explaining the Query (cont..)

- explain() function be used with
  - find()
  - remove()
  - update()
  - count()
  - aggregate()
  - distinct()
  - findAndModify()

Viewing Execution Stats

- In order to view the execution stats, you need to pass executionStats as an argument to the explain() function.
- The execution stats provide useful metrics pertinent to each execution phase, along with some top-level fields where some metrics are aggregated over the total execution of the query.

1. executionTimeMillis: This is the total time (in milliseconds) taken for query execution.
2. totalKeysExamined: This indicates the number of indexed keys that were scanned.
3. totalDocsExamined: This indicates the number of documents examined against the given query condition.
4. nReturned: This is the total number of records returned in the query output.

Identifying Problems

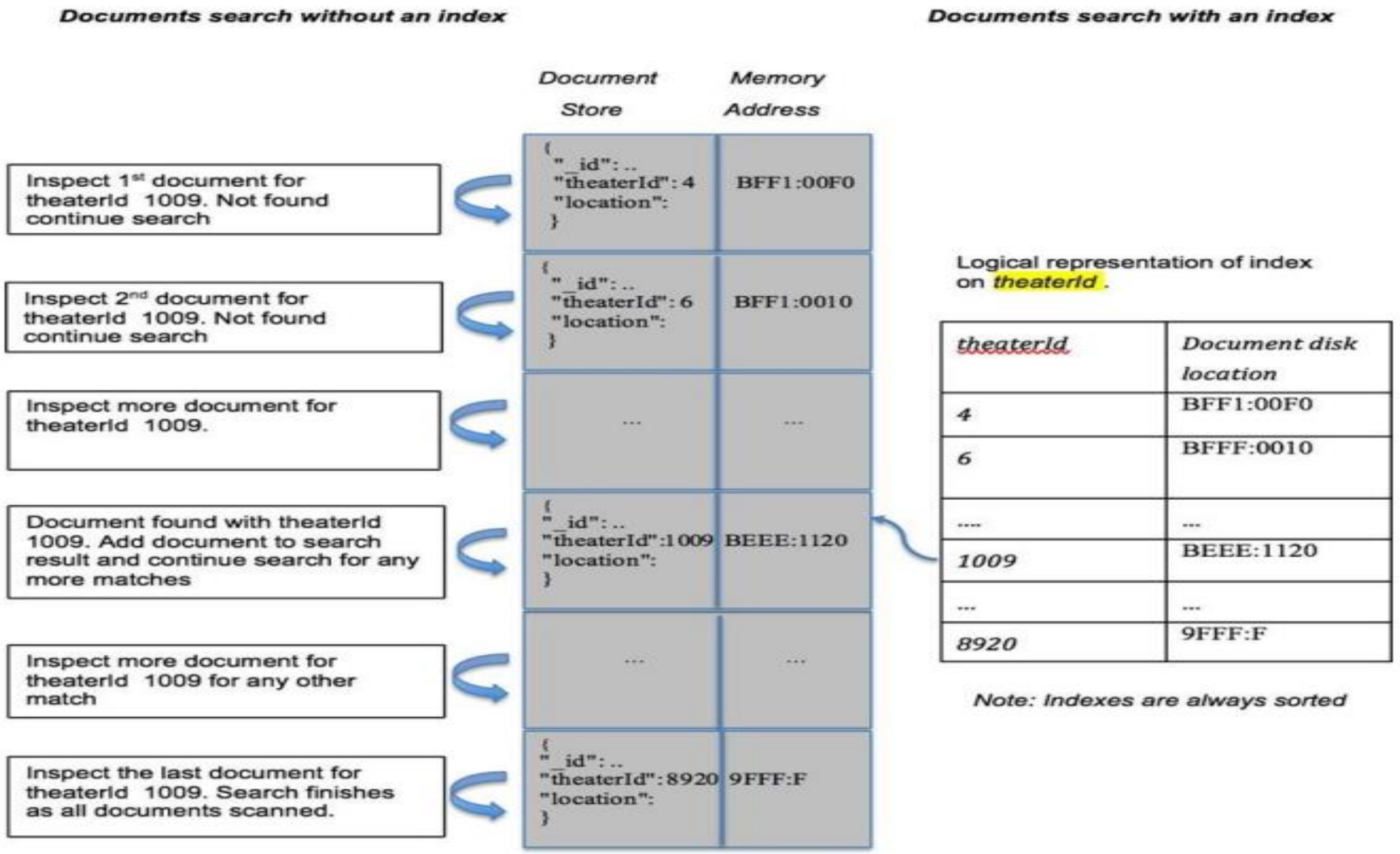
- To return few matching records, the query has to examine almost all the documents.
- Having to scan a large number of documents slows down the query execution.
- However, the query execution time can vary based on the network traffic, the RAM and CPU loads on the server, and the number of records getting scanned.

Reason for Slow Performance - Linear Search

- When we execute a find query with a search criterion on a collection, the database search engine picks the first record in the collection and checks whether it matches the given criteria. If no match is found, the search engine moves on to the next record to find a match, and the process is repeated till a search is found.
- This search technique is called a sequential or linear search.
- Linear searches perform better when they are applied to a small amount of data, or in the best-case scenarios, where the required term is found within the first search.
- Linear searches can be avoided or at least limited by creating indexes on specific fields of a collection.

Introduction to Indexes

- Databases can maintain and use indexes to make searches more efficient. In MongoDB, indexes are created on a field or a combination of fields.
- The database maintains a special registry of indexed fields and some of their data.
- The registry is easily searchable, as it maintains a logical link between the value of an indexed field and the respective documents in the collection.
- During a search operation, the database first locates the value in the registry and identifies the matching documents in the collection accordingly.
- The values in a registry are always sorted in ascending or descending order of the values, which helps during a range search and also while sorting the results.
- Databases support indexes for the faster retrieval of data and how the index registry helps avoid complete collection scans.
- Ex: db.theaters.find({"theaterId" : 1009})


Documents search without an index
Documents search with an index
Document Store
Memory Address
{ "_id": .. "theaterId": 4 "location": }
BFF1:00F0
{ "_id": .. "theaterId": 6 "location": }
BFF1:0010
{ "_id": .. "theaterId":1009 "location": }
BEEE:1120
{ "_id": .. "theaterId":8920 "location": }
9FFF:F
Logical representation of index on theaterId .

Inspect 1st document for theaterId 1009. Not found continue search

Inspect 2nd document for theaterId 1009. Not found continue search

Inspect more document for theaterId 1009.

Document found with theaterId 1009. Add document to search result and continue search for any more matches

Inspect more document for theaterId 1009 for any other match

Inspect the last document for theaterId 1009. Search finishes as all documents scanned.

theaterId	*Document disk location*
4	BFF1:00F0
6	BFFF:0010
....	...
1009	BEEE:1120
...	...
8920	9FFF:F

Note: Indexes are always sorted

Figure 9.2: Data search with and without an index

Creating and Listing Indexes

- Indexes can be created by executing a createIndex() command on a collection.

```
db.collection.createIndex(
keys,
options
)
```

- You can list the indexes of a collection by using the getIndexes() command.
- This command does not take any parameters. It simply returns an array of indexes with some basic details.
- Ex:

```
db.movies.getIndexes()
```

Index Names

- ❑ MongoDB assigns a default name to an index if a name is not provided explicitly.
- ❑ The default name of an index consists of the field name and the sort order, separated by underscores.
- ❑ If there is more than one key in the index (known as a compound index), all the keys are concatenated in the same manner.
- ❑ You can also create an index with a specific name.
- ❑ You can use the “name” attribute to provide a custom name to the index.

Hiding and Dropping Indexes

- Dropping an index means removing the values of the fields from the index registry.
- Thus, any searches on the related fields will be performed in a linear fashion, provided there are no other indexes present on the field.
- To fix an incorrectly created index, we need to drop it and recreate it correctly.
- An index is deleted using the dropIndex function

```
db.collection.dropIndex(indexNameOrSpecification)
```

Dropping Multiple Indexes

- You can also drop multiple indexes using the dropIndexes command.

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

- This command can be used to drop all the indexes on a collection except the default _id index.
- You can also use the command to delete a group of indexes by passing an array of index names.

Hiding an Index

- To hide an index, the hideIndex() command can be used on the collection.

```
db.collection.hideIndex(indexNameOrSpecification)
```

- Hidden indexes appear only on the getIndexes() function call.
- You can restore or unhide indexes by using the unhideIndex()

```
db.collection.unhideIndex(indexNameOrSpecification)
```

Type of Indexes

- **Default Indexes**: Each document in a collection has a primary key (namely, the _id field) and is indexed by default.
- **Single-Key Indexes:** An index created using a single field from a collection is called a single-key index.

```
db.collection.createIndex({ field1: type}, {options})
```

- **Compound Indexes:** Sometimes single-key indexes are not sufficient to reduce the collection scans. This typically happens when the query is based on more than one field

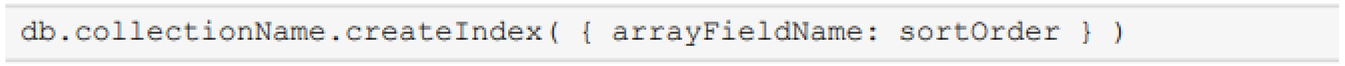
  The createIndex command can be used to create a compound index.

```
db.collection.createIndex({ field1: type, field2: type, ...}, {options})
```

Type of Indexes (cont..)

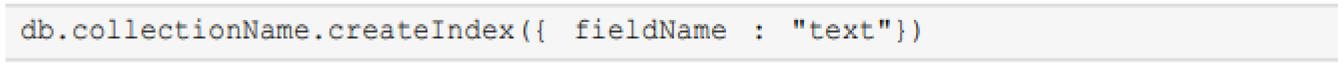
- **Multikey Indexes:** An index created on the fields of an array type is called a multikey index.

When an array field is passed as an argument to the createIndex function, MongoDB creates an index entry for each element of the array.



```
db.collectionName.createIndex( { arrayFieldName: sortOrder } )
```

- **Text Indexes:** An index defined on a string field or an array of string elements is called a text index. Text indexes are not sorted, meaning that they are faster than normal indexes.

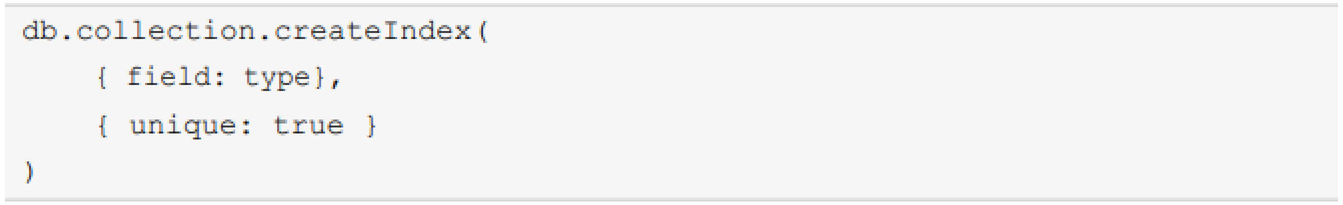


```
db.collectionName.createIndex({  fieldName  :  "text"})
```

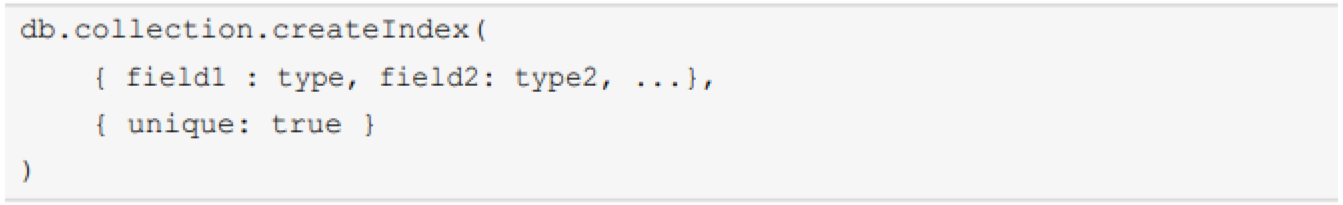
- **Indexes on Nested Documents :** A document can contain nested objects to group a few attributes. Using a dot (.) notation, you can create an index on any of the nested document fields, just like any other field in the collection
- **Wildcard Indexes:** wildcard characters (eg: $**) can be used to create indexes on the specificied field.

Properties of Indexes

- **Unique Indexes**: A unique index property restricts the duplication of the index key.



```
db.collection.createIndex(
    { field: type},
    { unique: true }
)
```



```
db.collection.createIndex(
    { field1 : type, field2: type2, ...},
    { unique: true }
)
```

Properties of Indexes(cont…)

- **TTL Indexes:** TTL (or Time to Live) indexes put an expiry on documents.

  Once the documents have expired, they are deleted.

  This index can only be created on a field of the date type.

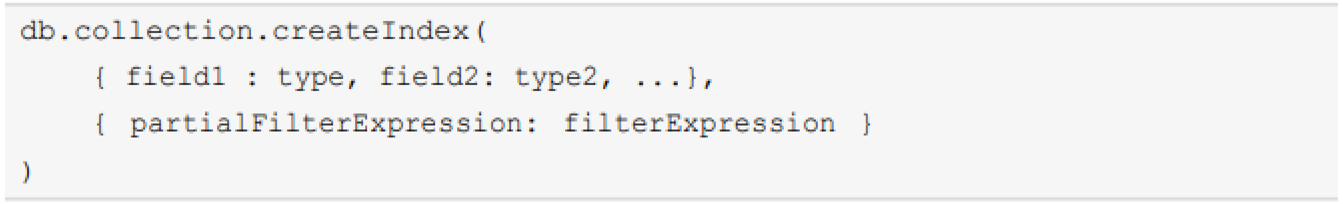
```
db.collection.createIndex({ field: type}, { expireAfterSeconds: seconds })
```

- **Sparse Indexes:** if an index is marked as sparse, then only those documents are registered in which the given field exists with some value including null.

```
db.collection.createIndex({ field1 : type, field2: type2, ...}, { sparse:
true })
```

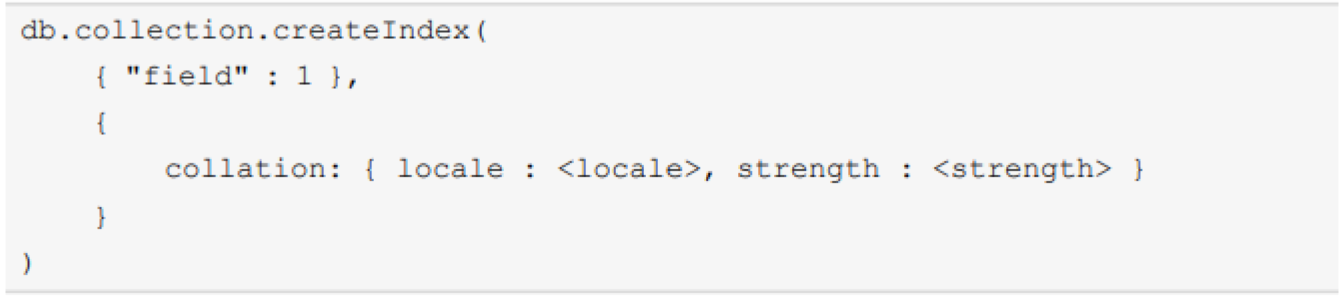
Properties of Indexes(cont…)

- **Partial Indexes**: An index can be created to maintain documents that match a given filter expression. Such an index is called a partial index.



```
db.collection.createIndex(
    { field1 : type, field2: type2, ...},
    { partialFilterExpression: filterExpression }
)
```

- **Case-Insensitive Indexes:** Case-insensitive indexes allow you to find data using indexes in a case-insensitive manner.



```
db.collection.createIndex(
    { "field" : 1 },
    {
        collation: { locale : <locale>, strength : <strength> }
    }
)
```

Other Query Optimization Techniques

- **Fetch Only What You Need**
- **Correct Query Condition and Projection**
- **Pagination**
- **Sorting Using Indexes**
- **Fitting Indexes in the RAM :** To fit indexes in memory, you can use the totalIndexSize function on a collection

```
db.collection.totalIndexSize()
```

- **Index Selectivity**
- **Providing Hints** : we can use a hint() function to specify which index should be used for the execution

```
db.users.find().hint(
    { index }
)
```

Summary

- ❑ In this chapter, you practiced improving query performance.
- ❑ the concept of indexes; how they solve performance issues for a query; various ways to create, list, and delete indexes; different types of indexes; and their properties.
- ❑ you studied query optimization techniques