SESSION 7
MONGODB INDEXING

Learning Objectives

In this session, students will learn to:

- Explain indexes
- Describe the types of indexes
- List the advantages of using the indexes
- Explain how to create and drop indexes

In a book, the first page usually contains a table of contents that lists the chapters, topics, and subtopics with page numbers. This helps in navigating to a particular chapter, topic, or subtopic in the book instead of iterating through all the pages to find the required piece of information. Similarly, indexes are ready reckoners for the database. They help in reaching the required data in a short duration of time with less overhead.

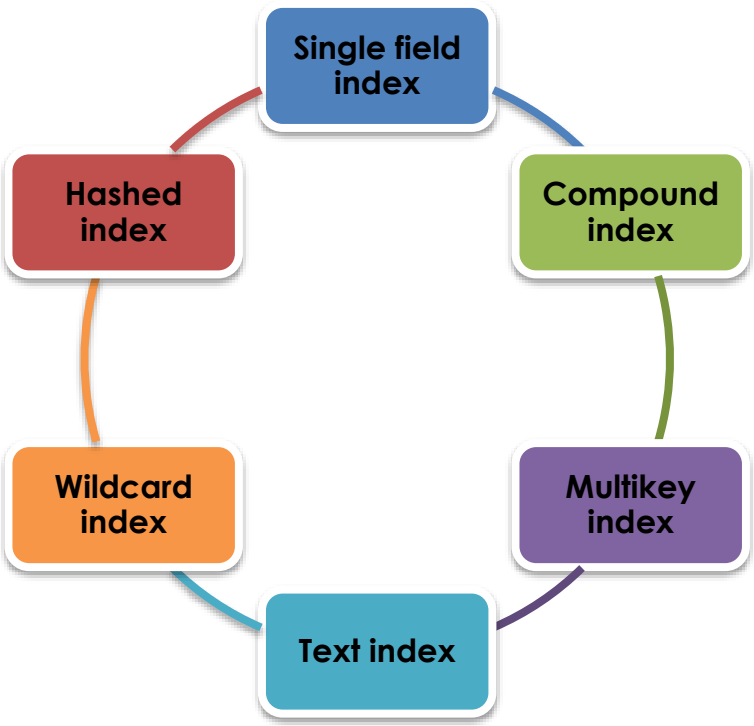
This session will explain indexes and different types of indexes in MongoDB. It will also explain the benefits of using indexes. Finally, it will explain how to create and drop indexes.

7.1 Introduction to Indexes

Indexes are nothing but data structures that store the values of a field or multiple fields in ascending or descending order. After the index is created, the user can easily query for a specific document with a low response time. This is because the database engine looks up the index to find a particular document instead of traversing through the entire collection to locate a specific document. This improves the query performance and reduces the overhead on the database engine of traversing through whole collections. The ordering of the index helps in easily accessing data in equality-based and range-based queries.

By default, MongoDB adds the `_id` field for each document in the collection. This field is unique for each document. It prevents duplicate documents from getting inserted into the collection. An index is created by default on the `_id` field and it cannot be dropped. It is known as the Default index.

MongoDB supports different types of indexes:



7.2 Single Field Index

MongoDB allows users to create an index on a single field of documents in a collection. The field on which index is created must be a prominent field that is used when searching for documents in that collection. Such an index created on a single field is called as single field index.

The general syntax for creating a single field index is:

```
db.Collection_name.createIndex(key,option,commitQuorum)
```

The `createIndex` command takes the parameters described in Table 7.1.

Parameter	Description
`key`	It is a key-value pair, where the key specifies the field on which the index must be created, and the value specifies the order of the index. It takes the value `1` for ascending order and the value `-1` for descending order. This is a required parameter.
`option`	It is a document that contains options, which specify how the index should be created. This is an optional parameter.
`commitQuorum`	It specifies how many replica sets, which contain data and participate in voting, must respond with successful index creation before the primary marks the index as ready to use. This is an optional parameter.

Table 7.1: Parameters of the `createIndex` Command

The `option` parameter in the `createIndex` command takes the parameters described in Table 7.2.

Parameter	Description
`background`	This is a `boolean` value. If set to `true`, it specifies that the index must be built in the background without interfering with the other database activities. By default, this parameter is set to `false`.
`unique`	This is a `boolean` value. If set to `true`, it creates a unique index that does not allow insertion of more than one document with same index key. By default, this parameter is set to `false`.

Table 7.2: List of Options for Creating Single Field Index

name	This is a string value that specifies a name for the index. If a name is not specified, MongoDB assigns a name which is a combination of indexed field names and the sort order.
sparse	This is a boolean value. If set to true, it includes only those documents in the index that have the specified field. By default, this parameter is set to false.
expireAfterSeconds	This is an integer value that specifies time in seconds. This time is Time-To-Live (TTL) that controls how long the documents in the collection will be retained by MongoDB.

To understand the importance of indexing, in the sample_analytics database, let us find a document that has account_id as 781775 in the grades collection. To view the execution details of the query, let us also use the explain("executionStats") method. To do this, the user can run the query:

```
db.accounts.find({"account_id":781775}).explain
("executionStats")
```

Figure 7.1 shows the output of this query.

```
sample_analytics> db.accounts.find({"account_id":781775}).explain("executionStats"
)
{
  explainVersion: '1',
  queryPlanner: {
    namespace: 'sample_analytics.accounts',
    indexFilterSet: false,
    parsedQuery: { account_id: { '$eq': 781775 } },
    queryHash: 'C7AD3977',
    planCacheKey: 'C7AD3977',
    maxIndexedOrSolutionsReached: false,
    maxIndexedAndSolutionsReached: false,
    maxScansToExplodeReached: false,
    winningPlan: {
      stage: 'COLLSCAN',
      filter: { account_id: { '$eq': 781775 } },
      direction: 'forward'
    },
    rejectedPlans: []
  },
  executionStats: {
    executionSuccess: true,
    nReturned: 1,
    executionTimeMillis: 9,
    totalKeysExamined: 0,
    totalDocsExamined: 1746,
    executionStages: {
      stage: 'COLLSCAN',
      filter: { account_id: { '$eq': 781775 } },
      nReturned: 1,
      executionTimeMillisEstimate: 6,
      works: 1748,
      advanced: 1,
      needTime: 1746,
      needYield: 0,
      saveState: 1,
      restoreState: 1,
      isEOF: 1,
      direction: 'forward',
```

Figure 7.1: Output of `explain("executionStats")` Method Before Indexing

Note that in the execution status, the total number of documents examined is shown as 1746.

Now, let us create an index for the `accounts` collection in the `sample_analytics` database based on the `account_id` field in the ascending order. To do this, the query to run is:

```
db.accounts.createIndex({"account_id":1})
```

The index is created as specified in the query.

Let us now run the previous query again. Figure 7.2 shows the output of the query.

Figure 7.2: Output of `explain("executionStats")` Method After Indexing

Note that in the output the number of documents examined is `1`. This indicates that after the index is created, the number of documents examined reduces, thereby, reducing the time required to complete the query execution.

Now, let us create another index for the `accounts` collection based on the `limit` field in the descending order. To do this, the query to run is:

```
db.accounts.createIndex({"limit":-1})
```

The index is created as specified in the query. To view all the indexes created for the `accounts` collection, run the query as:

```
db.accounts.getIndexes()
```

Figure 7.3 shows the output of this query.

```
sample_analytics> db.accounts.getIndexes()
[
  { v: 2, key: { _id: 1 }, name: '_id_' },
  { v: 2, key: { account_id: 1 }, name: 'account_id_1' },
  { v: 2, key: { limit: -1 }, name: 'limit_-1' }
]
```

Figure 7.3: Indexes Created for the `accounts` Collection

Note that the index on the `_id` field is created by default when the collection is created. The other two indexes are user created indexes.

7.3 Compound Index

In MongoDB, users can create indexes based on multiple fields. These types of indexes are known as compound indexes. Users can create indexes that reference a maximum of 32 fields. For each field, the sort order must be specified. The syntax for creating a compound index is:

```
db.collections.createIndex({fieldName1:ord1,
            fieldName2:ord2, ...} )
```

The order in which the fields are specified in the syntax is important for the index to function efficiently. The documents will be sorted on `fieldName1`, then within each value of `fieldName1`, the documents are sorted on `fieldName2`, and so on. Changing the order of the fields in the `createIndex` command will produce different results in terms of efficiency of the index.

In the `sample_training` database, consider that the user wants to create a compound index named `compoundIndex` for the `grades` collection. The index must first be created on the `class_id` field in the ascending order and then on the `student_id` field in the descending order. To do this, the user can execute the query as:

```
db.grades.createIndex({class_id: 1, student_id:-1},
            {name: "compoundIndex"})
```

Figure 7.4 shows the output of this query.

```
sample_training> db.grades.createIndex({class_id: 1, student_id:-1},
 {name: "compoundIndex"})
compoundIndex
```

Figure 7.4: Compound Index Created

To view the indexes created for the `grades` collection, run the query as:

```
db.grades.getIndexes()
```

Figure 7.5 shows the output of this query.

```
sample_training> db.grades.getIndexes()
[
  { v: 2, key: { _id: 1 }, name: '_id_' },
  { v: 2, key: { class_id: 1, student_id: -1 }, name: 'compoundIndex' }
]
```

Figure 7.5: Indexes Created for the `grades` Collection

7.4 Multikey Index

Indexes created on array fields are multikey indexes. In this case, an index is created on each element in the array. Multikey indexes facilitate efficient querying of the arrays in documents. The syntax for creating a multikey index is:

```
db.collections.createIndex(arrayname : type)
```

In the syntax, `arrayname` specifies the name of the array field on which the index must be created, and `type` specifies the sort order. The index must not be specified as a multikey index because if the specified field is an array, MongoDB will automatically create a multikey index.

Consider the `accounts` collection of the `sample_analytics` database. The user wants to create a multikey index named `multiKeyIndex` on the `products` field, which is an array, in the ascending order. To do so, the user can execute the query as:

```
db.accounts.createIndex({products:1},
        {name: "multiKeyIndex"})
```

Multikey index is created for the `products` field.

To view the indexes created on the `accounts` collection, the user can execute the query as:

```
db.accounts.getIndexes()
```

Figure 7.6 shows the output of this query.

```
sample_analytics> db.accounts.getIndexes()
[
  { v: 2, key: { _id: 1 }, name: '_id_' },
  { v: 2, key: { account_id: 1 }, name: 'account_id_1' },
  { v: 2, key: { limit: -1 }, name: 'limit_-1' },
  { v: 2, key: { products: 1 }, name: 'multiKeyIndex' }
]
```

Figure 7.6: Indexes Created for the `accounts` Collection

7.4.1 Compound Multikey Index

Users can also create a compound multikey index. This index can have only one array field as an indexed field. If the documents in a collection have two array fields A and B, the index can include field A or field B, but not both the fields. If some documents in a collection have field A as an array and others have field B as an array, then the index can include both the fields.

If a compound multikey index is created on a collection, any document that violates the index cannot be inserted into the collection. That is, say an index has been created to include field A, which exists for some documents. The index includes field B, which exists for the other documents. In this case, a document that includes both array fields A and B cannot be inserted into the collection.

7.4.2 Index on Fields Embedded in Arrays

An array field may have other fields embedded in it. Indexes can be created on these embedded fields. In the `sample_training` database, the `grades` collection has an array field named `scores`. This array has `type` and `score` as embedded fields, as shown in Figure 7.7.

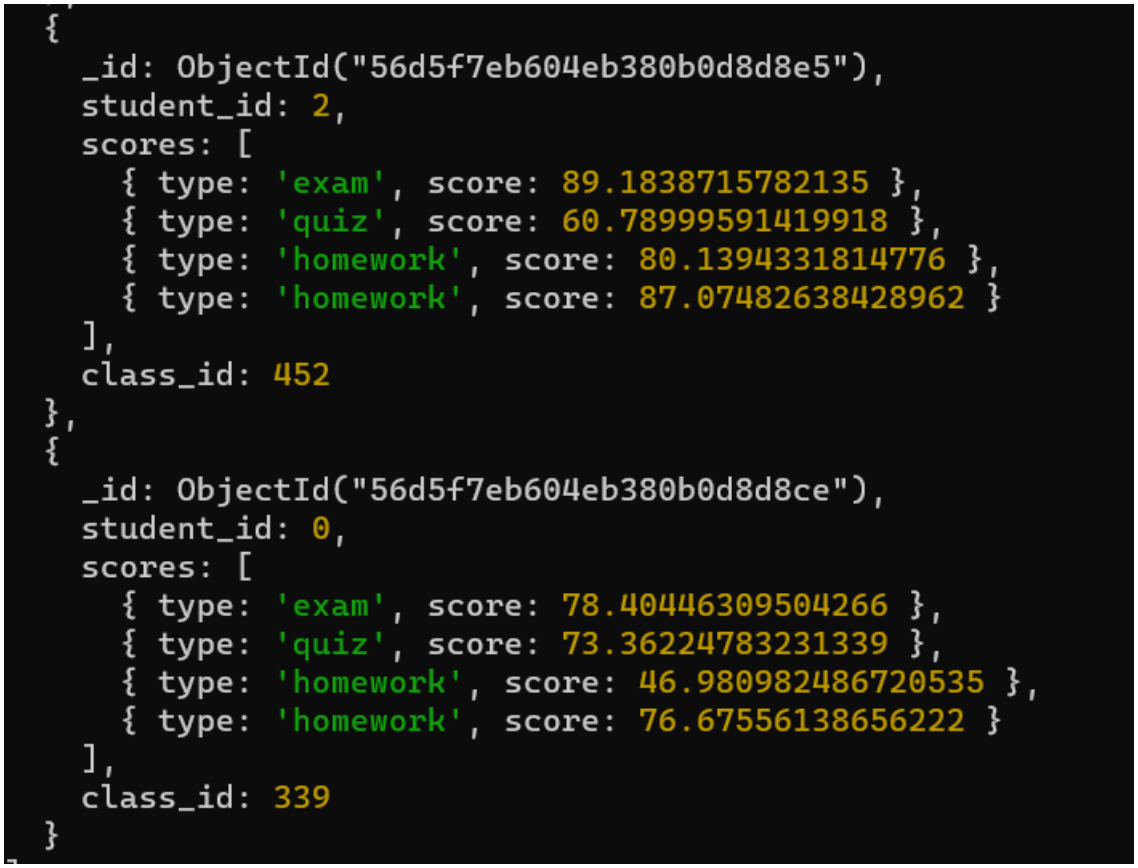


Figure 7.7: Data in the `grades` Collection

To create a multikey index on the `score.type` and `score.score` fields, the user can execute the query as:

```
db.grades.createIndex({"scores.type":1, "scores.score":1},
                {name: "embeddedFieldIndex"})
```

Figure 7.8 shows the output of this query.

```
sample_training> db.grades.createIndex({"scores.type":1, "scores.score":1},
{name: "embeddedFieldIndex"})
embeddedFieldIndex
```

Figure 7.8: Multikey Index on Embedded Fields in an Array

Indexes created on fields embedded in the array help users to run queries that include one of the indexed fields or both the indexed fields. For example, the user can view the documents where `scores.type` is `exam`. To do so, the user can use the query as:

```
db.grades.find( { "scores.type": "exam" } )
```

The user can also view the documents where `scores.type` is `quiz` and `scores.score` is greater than `50`. To do so, the user can use the query as:

```
db.grades.find( { "scores.type": "quiz",
    "scores.score": { $gt: 50 } } )
```

The indexes can also be used to sort the data. Consider that the user wants to view the documents where `scores.type` is `exam` and sort `scores.score` for each type of value in the ascending order. The user can use the query as:

```
db.grades.find( { "scores.type": "exam" }
    ).sort( { "scores.score": 1 } )
```

7.5 Text Index

MongoDB allows users to create indexes on texts or strings. These types of indexes are called text indexes. Text indexes can be created on a text field or on array of text elements and help in executing text-based searches. A document can have only one text index, but the index can include many fields. The syntax for creating a text Index is:

```
db.collections.createIndex(key, options, commitQuorum)
```

In this syntax,

- `key` specifies the field name on which the index must be created
- `option` specifies how the index must be created
- `commitQuorum` specifies the smallest number of replica sets that must report the successful creation of the index before the index is marked as ready for use

The `option` parameter takes various parameters as described in Table 7.3.

Parameter	Description
`weights`	This is a document that takes values ranging from `1` to `99999`. These values indicate the significance of an indexed field with respect to the other fields in terms of score.

default_language	This is a string value that specifies the language, which is used to determine rules for the stemmer and tokenizer and the list of stop words. By default, this parameter is set to English.
language_override	This is a string value that specifies the language to use to override the default language. By default, this parameter is set to language.
textIndexVersion	This is an integer value which is optional and is used to specify a version number for the index. If this parameter is used, the specified version number will replace the default version number of the index.

Table 7.3: List of Options for Creating Text Index

Consider the inspections collection of the sample_training database. The user wants to create a text index on the business_name field. To do this, the user can execute the query as:

```
db.inspections.createIndex({"business_name":1})
```

7.6 Wildcard Index

When creating any of the different types of indexes discussed until now, the field on which the index must be created is specified as a parameter in the createIndex command. Now, since MongoDB allows the schema to be flexible or unstructured, it could happen that all documents do not have similar fields on which the index can be created. In such cases, wildcard indexes are created. Wildcard indexes can be created on:

- Unknown fields
- Unstructured schema

7.6.1 Wildcard Index on a Specific Field

The syntax for creating a wildcard index on a specific field is:

```
db.collections.createIndex({"fieldA.$**" : 1})
```

This syntax is used to create an index on all the values in the fieldA field. If this field is a document or an array, the index iterates through the document or array and creates an index for each field in the document or array.

For example, the `inspections` collection in the `sample_training` database contains a field named `address`, which contains nested fields such as `city`, `zip`, `street`, and `number` as shown in Figure 7.9.

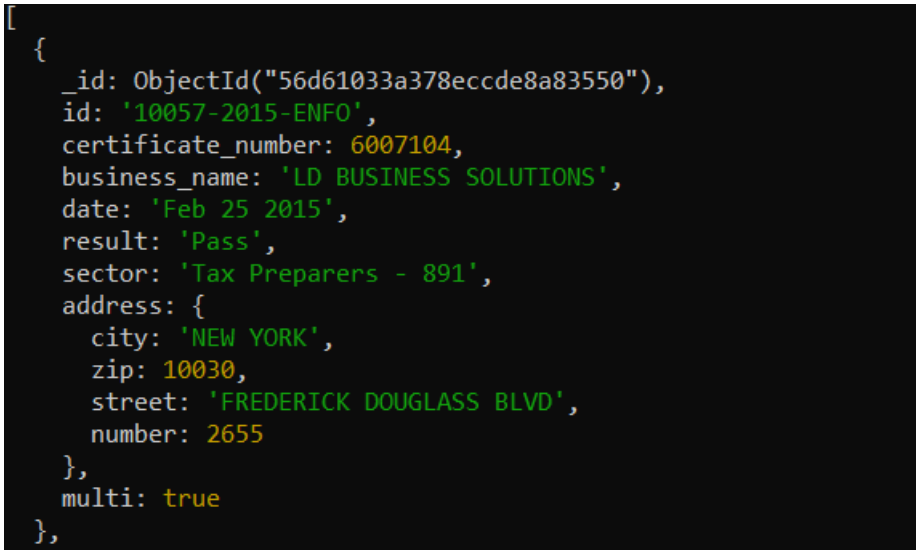


Figure 7.9: Fields in the `inspection` Collection

For creating an index on the `address` field, the query to use is:

```
db.inspections.createIndex( { "address.$**" : 1 } )
```

This index can now help with queries such as:

```
db.inspections.find({ "address.city" : "NEW YORK" })
```

```
db.inspections.find({ "address.zip" : {$gt:10029} })
```

7.6.2 Wildcard Index on All Fields

The syntax for creating a wildcard index on all fields in a document is:

```
db.collection.createIndex( { "$**" : 1 } )
```

This syntax is used to create an index on the all the fields of the documents in the specified collection. If this field is a document or an array, the index iterates through the document or array and creates an index for each field in the document or array.

For example, to create an index on all the fields in the `inspections` collection, the user can run the query as:

```
db.inspections.createIndex( { "$**" : 1 } )
```

7.6.3 Wildcard Indexes Including or Excluding Multiple Fields

Users can specify multiple fields for indexing when creating wildcard indexes. The syntax for creating this type of wildcard index is:

```
db.collection.createIndex({ "$**" : 1 },
 { "wildcardProjection" : { "fieldA" : 1,
       "fieldB.fieldC" : 1 } } )
```

In this syntax, `fieldA` and `fieldB.fieldC` are the specified fields on which the index must be created. For example, for the `transactions` collection of the `sample_analytics` database, the user wants to create an index on all the values in the `name` and `customer.gender` fields. The user can run the query as:

```
db.sales.createIndex({ "$**" : 1 },
{"wildcardProjection" : { "name" : 1,
"customer.gender" : 1 } })
```

Similarly, users can specify multiple fields that should not be included in the index when creating wildcard indexes.

The syntax for creating this type of wildcard index is:

```
db.collection.createIndex({ "$**" : 1 },
 { "wildcardProjection" : { "fieldA" : 0,
       "fieldB.fieldC" : 0 } } )
```

In this syntax, `fieldA` and `fieldB.fieldC` are the specified fields which must not be included in the index.

For example, for the `grades` collection, to create an index on all the values except the values in the `class_id` and `scores.score` fields, the user can run the query as:

```
db.grades.createIndex( { "$**" : 1 }, {
"wildcardProjection" : { "class_id" : 0,
"scores.score" : 0 } } )
```

7.7 Hashed Index

Hashing involves using an algorithm to compute the hash value for each of the values in the specified field to be indexed. A hash index is created using these hashed values. Hash indexes cannot be created for an array field or a document field. The syntax for creating a hashed index field is:

```
db.collections.createIndex({fieldname: "hashed"})
```

For example, to create a hashed index for the `student_id` field of the `grades` collection in the `sample_training` database, users can run the query as:

```
db.inspections.createIndex({student_id: "hashed"})
```

Hashed indexes can also be included in compound indexes. The syntax for creating a compound index with a hashed index is:

```
db.collections.createIndex ({"fieldA" :
1, "fieldB" : "hashed", "fieldC" : -1 })
```

Consider that the user wants to create a compound index with a hashed index for the `grades` collection. The index must sort the `student_id` field in the ascending order, hash the `class_id` field, and sort the `scores` field in the descending order. The users can run the query as:

```
db.inspections.createIndex({"student_id" : 1,
"class_id" : "hashed", "scores" : -1})
```

Hashed indexes are exceedingly used in database sharding.

7.8 Drop Indexes

Created indexes can be dropped using the `dropIndexes` command. The syntax for `dropIndexes` command is:

```
db.collection.dropIndexes({indexes})
```

The `indexes` parameter specifies the index or indexes to be dropped. If the parameter is omitted, all the indexes created for the specified collection are dropped, except the default index created on the `_id` field.

To drop indexes created on the `inspection` collection in the `sample_training` database, let us first view the indexes created using the query as:

```
db.inspections.getIndexes()
```

Figure 7.10 shows the output of this query.

```
sample_training> db.inspections.getIndexes()
[
  { v: 2, key: { _id: 1 }, name: '_id_' },
  { v: 2, key: { business_name: 1 }, name: 'business_name_1' },
  { v: 2, key: { 'address.$**': 1 }, name: 'address.$**_1' },
  { v: 2, key: { '$**': 1 }, name: '$**_1' },
  { v: 2, key: { student_id: 'hashed' }, name: 'student_id_hashed' },
  {
    v: 2,
    key: { student_id: 1, class_id: 'hashed', scores: -1 },
    name: 'student_id_1_class_id_hashed_scores_-1'
  }
]
```

Figure 7.10: List of Indexes Created for the `inspections` Collection

To drop the ascending index created on the `business_name` field, the user can run the query as:

```
db.inspections.dropIndexes("business_name_1")
```

MongoDB drops the specified index.

To drop multiple indexes from the `inspections` collection, an array of the index names to be dropped must be passed to the `dropIndexes` command. To do so, the user can run the query as:

```
db.inspections.dropIndexes(["address.$**_1", "$**_1"])
```

To view the remaining indexes on the `inspections` collection, the user can run the query as:

```
db.inspections.getIndexes()
```

Figure 7.11 shows the output of this query.

```
sample_training> db.inspections.getIndexes()
[
  { v: 2, key: { _id: 1 }, name: '_id_' },
  { v: 2, key: { student_id: 'hashed' }, name: 'student_id_hashed' },
  {
    v: 2,
    key: { student_id: 1, class_id: 'hashed', scores: -1 },
    name: 'student_id_1_class_id_hashed_scores_-1'
  }
]
```

Figure 7.11: List of Indexes Created for the `inspections` Collection

To drop all the indexes created for the `inspections` collection, the user can run the query as:

```
db.inspections.dropIndexes()
```

To view the remaining indexes on the `inspections` collection, the user can run the query as:

```
db.inspections.getIndexes()
```

Figure 7.12 shows the output of this query.

```
sample_training> db.inspections.getIndexes()
[ { v: 2, key: { _id: 1 }, name: '_id_' } ]
```

Figure 7.12: List of Indexes Created for the `inspections` Collection

7.9 Summary

- Indexes are data structures that store the values of a field or multiple fields in ascending or descending order.
- Single key index is created on a single field by specifying the order of sorting.
- Compound index is the index created on more than one field.
- Multikey index is an index created on an array field.
- Wildcard index helps in building indexes on unknown fields.
- Hashed indexes are the indexes created by hashing the value in the indexed field.
- All indexes created for a collection can be dropped except the default index created on the `_id` field.

Test Your Knowledge

1. Which of the following statements are true about creating indexes in MongoDB?

    a. It provides a default index named `_id` for each collection.
    b. Default indexes cannot be dropped.
    c. An index cannot hold references to more than one field of the documents.
    d. Indexes store the references in ascending or descending order.

2. Which of the following option is the correct syntax for creating multikey index?

    a. `db.collections.createIndex(<fieldname,type>)`
    b. `db.collections.createIndex(<arrayname,type>)`
    c. `db.collections.createIndex(<indexname>)`
    d. `db.collections.createIndex(<key,type>)`

3. Which of the following is the correct syntax for creating a wildcard index on all fields in a document except the `_id` field?

    a. `db.collection.createIndex( { "#**" : 1 } )`
    b. `db.collection.createIndex( { "&**" : 1 } )`
    c. `db.collection.createIndex( { "$**" : 1 } )`
    d. `db.collection.createIndex( { "***" : 1 } )`

4. Which of the following statements are true about wildcard index?

    a. Users can create indexes on unknown fields.
    b. Users can create indexes on unstructured schema.
    c. Users can create the index either using `createIndex` or `createIndexes` commands.
    d. Wildcard index does not omit the `_id` field by default.

5. Which of the following data type is not supported by hash index?

    a. `strings`
    b. `integer`
    c. `double`
    d. `arrays`

Answers to Test Your Knowledge

1	a, b, d
2	b
3	c
4	a, b, c
5	d

Try it Yourself

1. Create a database named `Customer` and a collection named `customer_purchase`.
2. Insert the following four documents into the `customer_purchase` collection.

```
[
{
customer_id: 1231,
customer_name:"Alexander",
city:"Atlanta",
purchased:[
{type:'diary',amount:2000},
{type:'dried_goods',amount:1500},
{type:'snacks',amount:0},
{type:'care_products', amount:200},
{type:'vegetable_fruits',amount:1000}
]},
{
customer_id: 1267,
customer_name:"Daniel",
city:"Boston",
purchased:[
{type:'diary',amount:1000},
{type:'dried_goods',amount:1200},
{type:'snacks',amount:200},
{type:'care_products', amount:700},
{type:'vegetable_fruits',amount:500}
]},
{
customer_id: 1342,
customer_name:"Jackson",
city:"Atlanta",
purchased:[
{type:'diary',amount:600},
{type:'dried_goods',amount:700},
{type:'snacks',amount:100},
{type:'care_products', amount:0},
{type:'vegetable_fruits',amount:400}
]},
{
customer_id: 1346,
customer_name:"Gabriel",
```

```
city:"Texas",
purchased:[
{type:'diary',amount:700},
{type:'dried_goods',amount:300},
{type:'snacks',amount:400},
{type:'care_products', amount:800},
{type:'vegetable_fruits',amount:0}
]},
customer_id: 1323,
customer_name:"Nicholas",
city:"Texas",
purchased:[
{type:'diary',amount:300},
{type:'dried_goods',amount:100},
{type:'snacks',amount:0},
{type:'care_products', amount:800},
{type:'vegetable_fruits',amount:200}
]}
]
```

Using the `customer_purchase` collection, perform the following tasks:

3. Write a query to find the number of customers from `Boston` city and use the execution statistics command to view the number of documents examined to find the result of the query.
4. Create an index on the `city` field.
5. Create a compound index on the `customer_id` and `customer_name` fields.
6. Create a multikey index on the `purchased` array field.
7. View the indexes created on the `customer_purchase` collection.
8. Drop the index created on the `customer_name` field.
9. Write a query to find the number of customers from `Boston` city and use the execution statistics command to view the number of documents examined to find the result of the query. Compare the number of documents reviewed before and after indexing.
10. Create a mutikey index on the embedded array fields `purchased.type` and `purchased.amount`.
11. Create a text index on the `customer_name` field.
12. Drop the multikey index created on the `purchased.amount` and `purchased.type` fields.
13. Create the wildcard indexes on all the array elements of the `purchased` field.
14. View the indexes created on the `customer_puchase` collection.