Chapter 5. Index

References

- MongoDB The Definitive Guide: Powerful and Scalable Data Storage 3rd Edition
- https://docs.mongodb.com/
- https://www.mongodb.com/docs/manual/

Learning objectives

- Understand why and how to use indexes

Contents

1. Overview
Analyze query performance

```
db.store.insertMany([
    { "_id" : 1, "item" : "f1", type: "food", quantity: 600 },
    { "_id" : 2, "item" : "f2", type: "food", quantity: 100 },
    { "_id" : 3, "item" : "p1", type: "paper", quantity: 200 },
    { "_id" : 4, "item" : "p2", type: "paper", quantity: 150 },
    { "_id" : 5, "item" : "f3", type: "food", quantity: 300 },
    { "_id" : 6, "item" : "t1", type: "toys", quantity: 500 },
    { "_id" : 7, "item" : "a1", type: "apparel", quantity: 250 },
    { "_id" : 8, "item" : "a2", type: "apparel", quantity: 400 },
    { "_id" : 9, "item" : "t2", type: "toys", quantity: 50 },
    { "_id" : 10, "item" : "f4", type: "food", quantity: 75 }])
```

Without the index, the query scans the whole collection of 10 documents to return 3 matching documents. The query also had to scan the entirety of each document, potentially pulling them into memory.
→ expensive and slow

When run with an index, the query scanned 3 index entries and 3 documents to return 3 matching documents.
→ very efficiently

No index:

```
db.store.find( { quantity: { $gte: 100, $lte: 200 } } )
```

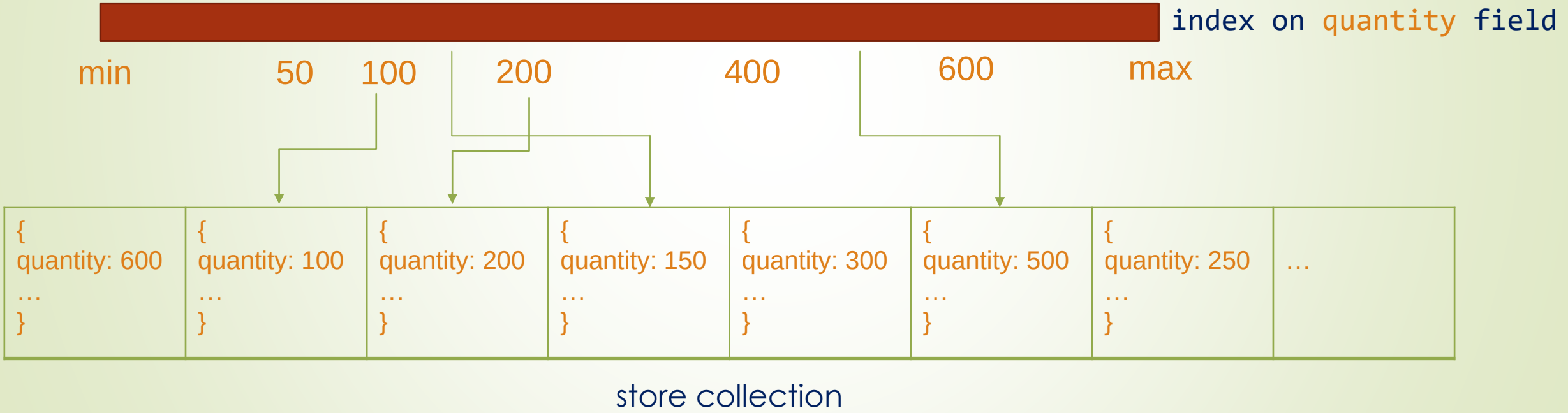
With index:

```
db.store.createIndex( { quantity: 1 } )
db.store.find( { quantity: { $gte: 100, $lte: 200 } } )
```

1. Overview
Analyze query performance

```
db.store.find( { quantity: { $gte: 100, $lte: 200 } } )
```

```
db.store.createIndex( { quantity: 1 } )
```



store collection

```
db.store.find( { quantity: { $gte: 100, $lte: 200 } } ).explain("executionStats")
```

1. Overview
Analyze query performance

```
queryPlanner: {
        ...
        winningPlan: {
            queryPlan: {
                stage: 'COLLSCAN',
                ...
            }
        }
},
executionStats: {
    executionSuccess: true,
    nReturned: 3,
    executionTimeMillis: 0,
    totalKeysExamined: 0,
    totalDocsExamined: 10,
    executionStages: {
        stage: 'COLLSCAN',
```

collection scan

number of documents returned

not using an index

number of documents to scan

1. Overview
What is index in MongoDB?

- Indexes support the efficient execution of queries in MongoDB. Without indexes, MongoDB must scan every document in a collection, to select those documents that match the query statement. If an appropriate index exists for a query, MongoDB can use the index to limit the number of documents it must inspect.
- Indexes are special data structures, that store a small portion of the data set in an easy to-traverse form. The index stores the *value of a specific field or set of fields*, *ordered by the value* of the field as specified in the index.
- MongoDB creates a *unique index* on the _id field during the creation of a collection. The _id index prevents clients from inserting two documents with the same value for the _id field. You cannot drop this index on the _id field.

Contents

2. Manage index
Create indexes

- Syntax:

db.collection.createIndex(<key and index type specification>, <options>)

Parameter	Type	Description
key	document	1 specifies an index that orders items in ascending order. -1 specifies an index that orders items in descending order.
options	document	Optional. A document that contains a set of options that controls the creation of the index.

  - Example: Create a single key ascending index on the quantity field

```
db.store.createIndex( { quantity: -1 } )
```

2. Manage index
Create indexes with names

- The default name for an index is the concatenation of the indexed keys and each key's direction in the index (1 or -1) using underscores as a separator.
- Example:

```
db.products.createIndex( {item: 1, quantity: -1} )
```

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

- A custom name can be created with indexes, and cannot be renamed once created. Example:

```
db.products.createIndex(
   {  item: 1, quantity: -1  } ,
   {  name: "query for inventory"  }
)
```

2. Manage index
Other methods

- List all indexes: db.collection.getIndexes()
- Drop an index: db.collection.dropIndex("index_name")
- Drop all indexes: db.collection.dropIndexes()
- View information about the query plan and execution statistics when the query is run:

  db.collection.find(find_fields).explain();

  - Example:

```
db.store.find( { quantity: { $gte: 100, $lte: 200 } }).explain("executionStats")
```

Contents

3. Index types

- Single Field Indexes
- Compound Indexes
- Multikey Indexes
- Text Indexes
- Unique Indexes

3. Index types
Single Field Index

- A single field index means index on a single field of a document. This index is helpful for fetching data in ascending or descending order.
- Syntax: `db.collection.createIndex({"<fieldName>" : <1 or -1>})`
- The sort order of the index key does not matter because MongoDB can traverse the index in either direction.
- Example:

```
{
  "_id": ObjectId("570c04a4ad233577f97dc459"),
  "score": 1034,
  "location": { state: "NY", city: "New York" }
}
```

```
db.records.createIndex({ score: 1 })
```

→ The created index will support queries that select on the field score:

```
db.records.find({ score: 2 })
db.records.find({ score: { $gt: 10 } })
```

3. Index types
Single Field Index

- Create an index on an embedded field. Example:

```
db.records.createIndex({ "location.state": 1 })
```

→ The created index on the field location.state

```
db.records.find({ "location.state": "CA" })
db.records.find({ "location.city": "Albany", "location.state": "NY" })
```

- Create an index on embedded document. Example:

```
db.records.createIndex({ location: 1 })
```

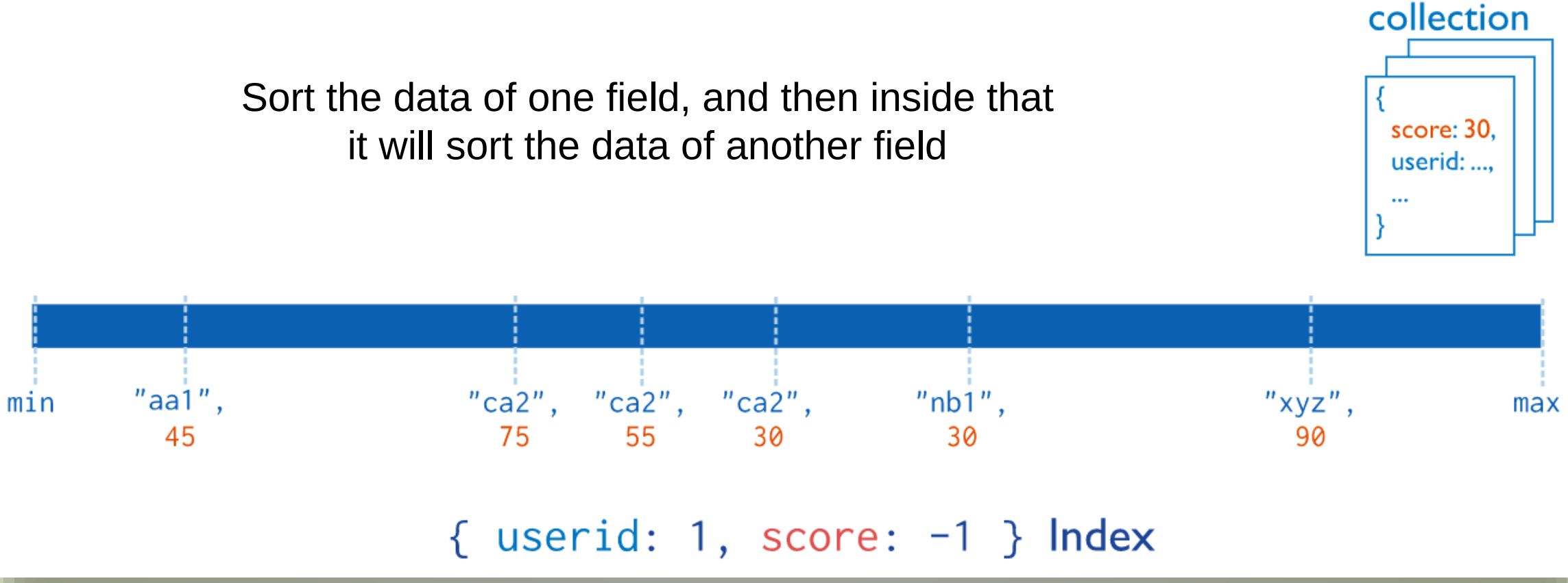
→ The created index on the field location of city and state

```
db.records.find({location: { city: "New York", state: "NY" } } )
```

3. Index types
Compound Index

- A compound index is a single structure holds **references to multiple fields** within a collection's documents.

Sort the data of one field, and then inside that it will sort the data of another field



3. Index types
Compound Index

- Syntax:

```
db.collection.createIndex(
  {
     <field1>: <type>,
     <field2>: <type2>,
     ...
  }
)
```

- The **order of the fields** listed in a compound index is important because it can matter in determining whether the index can support a sort operation.

3. Index types
Compound Index

- Example:

products collection:

```
{
    "_id": ObjectId(...),
    "item": "Banana",
    "category": ["food", "produce", "grocery"],
    "location": "4th Street Store",
    "stock": 4,
}
```

```
db.products.createIndex({ "item": 1, "stock": 1 })
```

→ the index will contain references to documents sorted first by the values of the item field and, within each value of the item field, sorted by values of the stock field.

```
db.products.find( { item: "Banana" } )
db.products.find( { item: "Banana", stock: { $gt: 5 } } )
```

→ this index supports queries on the item field as well as both item and stock fields

3. Index types
Compound Index

- Example:

```
db.events.createIndex({ "username": 1, "date": -1 })
```

This index can support both these sort operations:

```
db.events.find().sort({ username: 1, date: -1 })
db.events.find().sort({ username: -1, date: 1 })
```

But this index **cannot** support:

```
db.events.find().sort({ username: 1, date: 1 })
```

3. Index types
Compound Index

- For a compound index, MongoDB can use the index to support queries on the **index prefixes**. Index prefixes are the beginning subsets of indexed fields.
- Example, consider the following compound index:

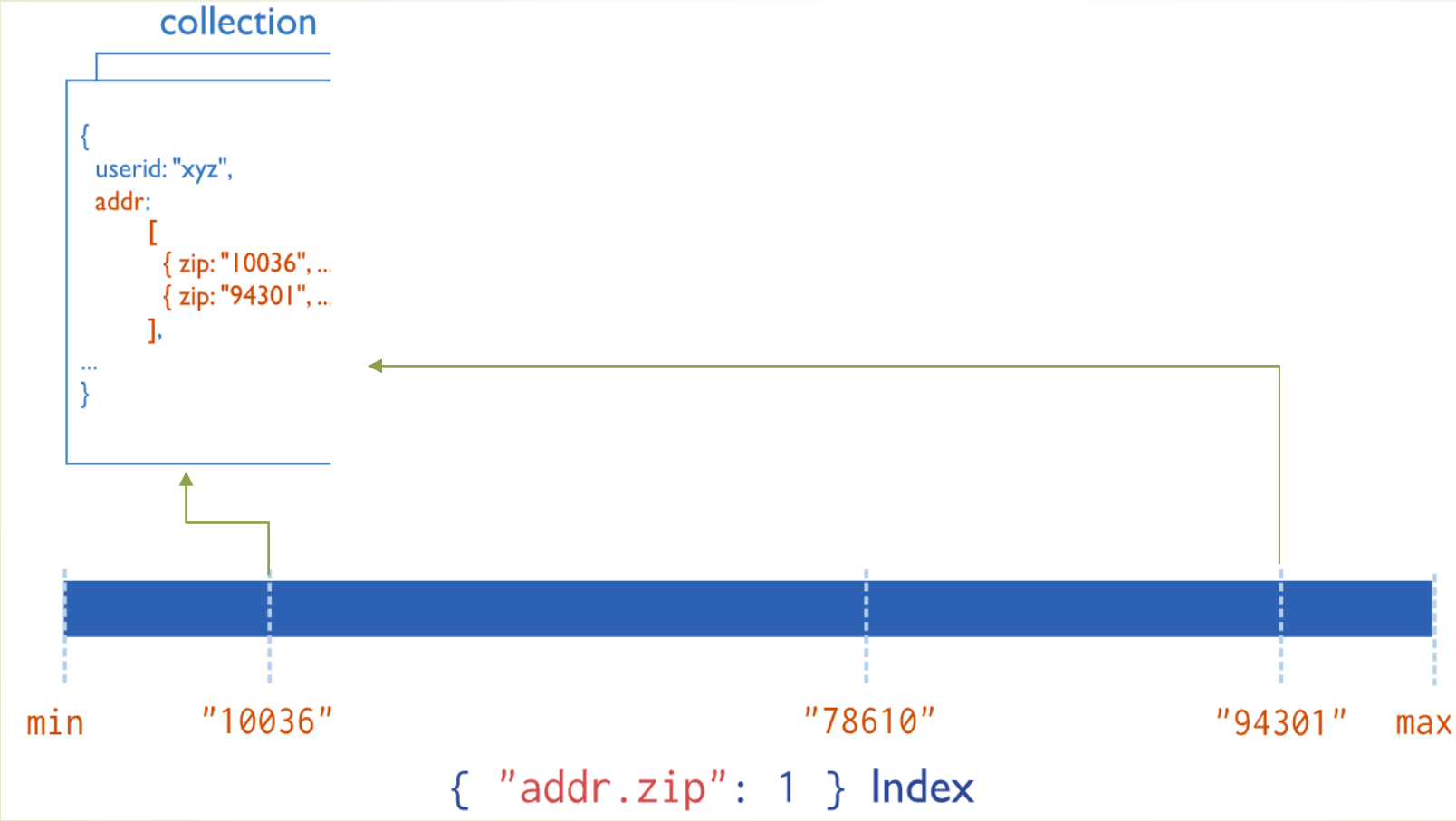
```
{ "item": 1, "location": 1, "stock": 1 }
```

We can use the index for queries on the following fields:

- The *item* field,
- The *item* field and the *location* field,
- The *item* field and the *location* field and the *stock* field.

3. Index types
Multikey Index

- Multikey index allows to **index a field that holds an array value** by creating an index key for each element in the array.



3. Index types
Multikey Index

- Example:

```
db.survey.insertOne( { _id: 1, item: "ABC", ratings: [ 2, 5, 9 ] })

db.survey.createIndex( { ratings: 1 } )
```

→ The multikey index contains the following three index keys, each pointing to the same document:

- 2,
- 5, and
- 9

3. Index types
Multikey Index

- Example:

```
{ _id: 5, type: "food", item: "aaa", ratings: [ 5, 8, 9 ] }
{ _id: 6, type: "food", item: "bbb", ratings: [ 5, 9 ] }
{ _id: 7, type: "food", item: "ccc", ratings: [ 9, 5, 8 ] }
{ _id: 8, type: "food", item: "ddd", ratings: [ 9, 5 ] }
{ _id: 9, type: "food", item: "eee", ratings: [ 5, 9, 5 ] }
```

```
db.inventory.createIndex( { ratings: 1 } )
```

```
db.inventory.find( { ratings: [ 5, 9 ] } )
```

MongoDB can use the multikey index to find documents that have 5 at any position in the ratings array. Then, MongoDB retrieves these documents and filters for documents whose ratings array equals the query array [5, 9].

3. Index types
Text Index

- Text indexes can include any field whose value is a string or an array of string elements.
- Text index supports text search queries on string content.
- A collection can only have one text index, but that index can cover multiple fields.
- Syntax:

```
db.collection.createIndex( { field1 : "text", field2 : "text",…} )
```

- The text index supports $text query operations.

3. Index types
Text Index

- $text performs a text search on the content of the fields indexed with a text index.
- Syntax:

```
{
  $text:
    {
        $search: <string>,
        $language: <string>,
        $caseSensitive: <boolean>,
        $diacriticSensitive: <boolean>
    }
}
```

$search: A string of terms that MongoDB parses and uses to query the text index.

$language: Optional. The language for the search.

$caseSensitive: Optional. A boolean flag to enable or disable case sensitive search. Defaults to false.

$diacriticSensitive: Optional. A boolean flag to enable or disable diacritic sensitive search. Defaults to false.

3. Index types
Text Index

- Examples:

```
db.articles.insertMany( [
   { _id: 1, subject: "coffee", author: "xyz", views: 50 },
   { _id: 2, subject: "Coffee Shopping", author: "efg", views: 5 },
   { _id: 3, subject: "Baking a cake", author: "abc", views: 90 },
   { _id: 4, subject: "baking", author: "xyz", views: 100 },
   { _id: 5, subject: "Café Con Leche", author: "abc", views: 200 },
   { _id: 6, subject: "Сырники", author: "jkl", views: 80 },
   { _id: 7, subject: "coffee and cream", author: "efg", views: 10 },
   { _id: 8, subject: "Cafe con Leche", author: "xyz", views: 10 }]
)
```

```
db.articles.createIndex( { subject: "text" } )
```

3. Index types
Text Index

- Examples:
  - Search for a single word:

```
dbtest> db.articles.find( { $text: { $search: "coffee" } } )
[
  { _id: 1, subject: 'coffee', author: 'xyz', views: 50 },
  { _id: 7, subject: 'coffee and cream', author: 'efg', views: 10 },
  { _id: 2, subject: 'Coffee Shopping', author: 'efg', views: 5 }
]
dbtest> db.articles.find( { $text: { $search: "coffee" ,$caseSensitive:true} } )
[
  { _id: 1, subject: 'coffee', author: 'xyz', views: 50 },
  { _id: 7, subject: 'coffee and cream', author: 'efg', views: 10 }
]
```

3. Index types
Text Index

- Match any of the search terms:

```
db.articles.find( { $text: { $search: "bake coffee cake" } } )
```

This query returns documents that contain either bake **or** coffee **or** cake in the indexed subject field

- Search for a phrase:

```
db.articles.find( { $text: { $search: "\"coffee shop\"" } } )
```

- Exclude documents that contain a term:

```
db.articles.find( { $text: { $search: "coffee -shop" } } )
```

This example searches for documents that contain the words coffee but do **not contain** the term *shop*

Contents

4. Index properties
Unique index

- A unique index ensures that the indexed fields do not store duplicate values.
- By default, MongoDB creates a unique index on the _id field during the creation of a collection.
- Syntax:

```
db.collection.createIndex(
        <key and index type specification>,
        { unique: true }
 )
```

- If you use the unique constraint on a **compound index**, then MongoDB will enforce uniqueness on the combination of the index key values.

4. Index properties
Unique index

- Example:
  - Create a unique index on the *score* field of the scoreHistory collection:

```
db.scoreHistory.createIndex( { score : 1 }, { unique: true } )
db.scoreHistory.insertMany([ { score : 1 } , { score : 2 }, { score : 3 } ])
db.scoreHistory.insert( { score : 2 } )
```

  → Try to insert a duplicate score document that fails because of the unique index

4. Index properties
Unique index

- Example:
  - Create a unique compound multikey index on *a.loc* and *a.qty*:

    db.collection.createIndex({ "a.loc": 1, "a.qty": 1 }, { unique: true })

    Try to insert these documents:

```
{ _id: 1, a: [ { loc: "A", qty: 5 }, { qty: 10 } ] }
{ _id: 2, a: [ { loc: "A" }, { qty: 5 } ] }
{ _id: 3, a: [ { loc: "A", qty: 10 } ] }
{ _id: 4, a: [ { loc: "B" }, { loc: "B" } ] }
{ _id: 5, a: [ { loc: "A" } ] }
{ _id: 6, a: [ { qty: 5 } ] }
{ _id: 7, a: [ { qty: 6 } ] }
```

4. Index properties
Partial index

- Partial indexes only index the documents in a collection that meet a specified **filter** expression → lower storage requirements and reduced performance costs for index creation and maintenance.
- Syntax:

```
db.collection.createIndex(
        <key and index type specification>,
        { partialFilterExpression : { filter } }
 )
```

  - filter: a document that specifies the filter condition using $eq, $and, $or, $in, $exists: true, $gt, $gte, $lt, $lte, $type.

- Example:

```
db.restaurants.createIndex(
        { cuisine: 1, name: 1 },
        { partialFilterExpression: { rating: { $gt: 5 } } }
)
```

Contents

5. Indexing strategies

- Note to make the best for the index:
  - the kinds of queries you expect,
  - the ratio of reads to writes,
  - the amount of free memory on your system
- The best overall strategy for designing indexes is to **profile a variety of index configurations** with data sets similar to the ones you'll be running in production to see which configurations perform best. Inspect the current indexes created for your collections to ensure they are supporting your current and planned queries. If an index is no longer used, drop the index.

5. Indexing strategies

- Use indexes to sort query results
- Use the ESR (Equality, Sort, Range) rule
- Create indexes to support your queries
- Ensure indexes fit in RAM

5. Indexing strategies
Use indexes to sort query results

- If the query planner cannot obtain the sort order from an index, it will sort the results in memory.
- In addition, sort operations that do not use an index will abort when they use 32 megabytes of memory.
- Sort operations can obtain the sort order by retrieving documents based on the ordering in an index.
- Sort operations that use an index often have better performance than blocking sorts.

5. Indexing strategies
Use indexes to sort query results

- Sort with a single field index:
  - If an ascending or a descending index is on a single field, the sort operation on the field can be in either direction.
  - Example:
    - Create an ascending index on the field ‘a’ for a collection records

```
db.records.createIndex( { "user": 1 } )
```

    - This index can support an ascending sort on a:

```
db.records.find().sort( {"user": 1 } )
db.records.find().sort( {"user": -1 } )
```

5. Indexing strategies
Use indexes to sort query results

- Sort on multiple fields:
  - Create a compound index to support sorting on multiple fields.
  - We can specify a sort on all the keys of the index or on a subset; however, the sort keys must be listed in the same order as they appear in the index.
  - For a query to use a compound index for a sort, the specified sort direction for all keys in the cursor.sort() document must match the index key pattern or match the inverse of the index key pattern.
  - Example: An index key pattern { a: 1, b: 1 }
    - can support a sort on { a: 1, b: -1 } and { a: -1, b: 1 }
    - **but not** on { b: 1, a: 1 }.

5. Indexing strategies
Use the ESR (Equality, Sort, Range) rule

- Applying the ESR (Equality, Sort, Range) rule to arrange the index keys helps to create a more efficient compound index.
  - Place fields that require exact matches first in your index to limit the number of documents that need to be examined.
  - Sorting after the equality matches allows MongoDB to do a non-blocking sort.
  - Place the range filter after the sort predicate so MongoDB can use a non-blocking index sort.
- Example:

```
db.cars.find( {
                    manufacturer: 'Ford',
                    cost: { $gt: 10000 }
                } ).sort( { model: 1 } )
```

Following the ESR rule, the optimal index for the example query is:

```
{ manufacturer: 1, model: 1, cost: 1 }
```

5. Indexing strategies
Create indexes to support your queries

- An index supports a query when the index contains all the fields scanned by the query.
  - Create a single-key index if all queries use the same, single key
  - Create compound indexes to support several different queries
  - Create indexes to support text search
  - Index use and collation

5. Indexing strategies
Ensure indexes fit in RAM

- For the fastest processing, ensure that your indexes fit entirely in RAM so that the system can avoid reading the index from disk.
- To check the size of your indexes, use the **db.collection.totalIndexSize()** helper, which returns data in bytes.
- Example: db.articles.totalIndexSize()

Question?