mongoDB

Bộ môn: Kỹ Thuật Phần Mềm

Giáo viên: **Trần Thế Trung.**
Email: tranthetrung@iuh.edu.vn

MongoDB
Indexes

1. Overview
2. Single field Indexes.
3. Compound Indexes.
4. Multikey indexes
5. Unique indexes
6. Text Indexes

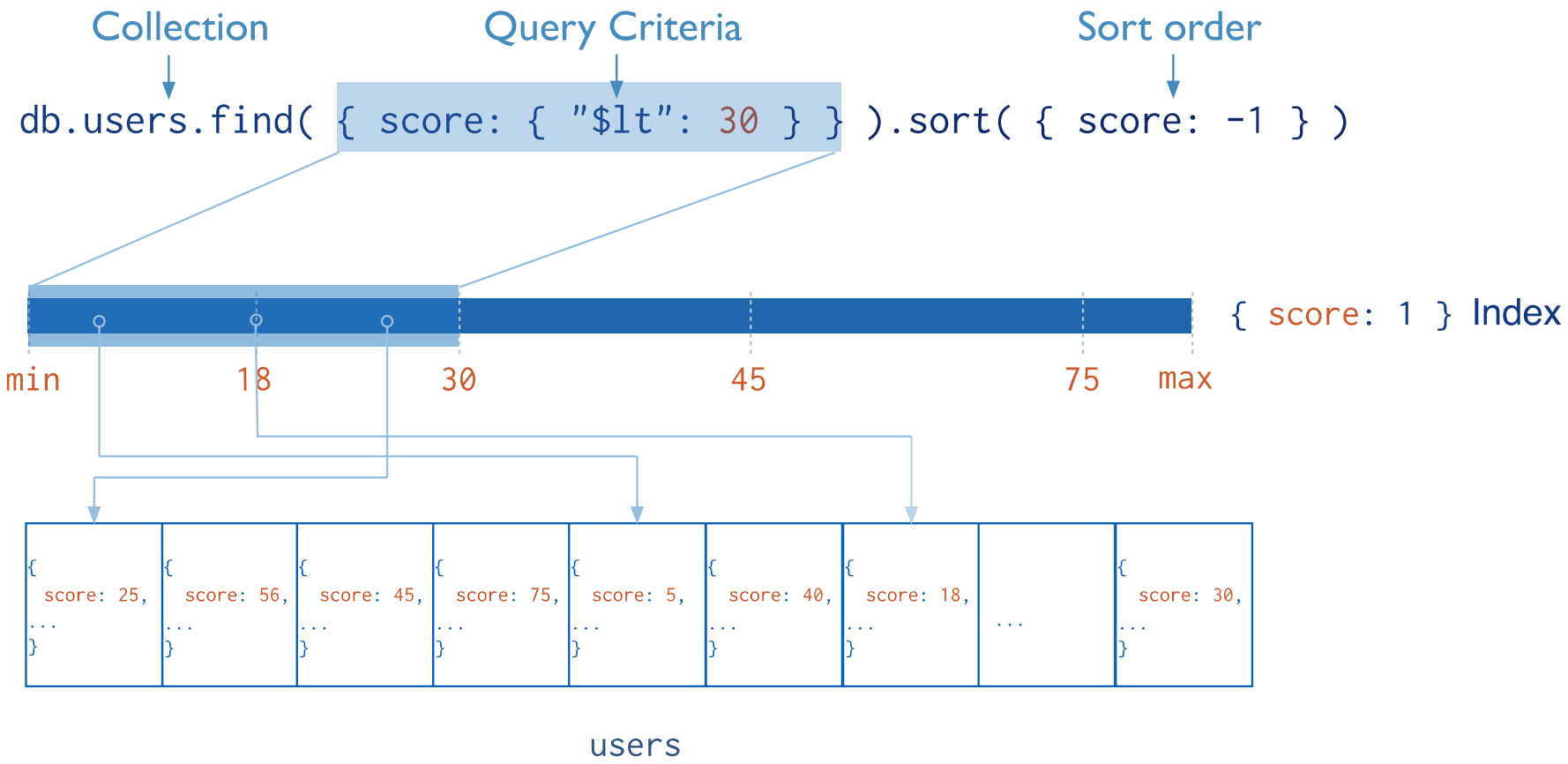
1. Overview

Overview

- **Indexes** support the efficient execution of queries in MongoDB. Without indexes, MongoDB must perform scan every document in a collection, to select those documents that match the query statement. 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.

Overview


Collection
Query Criteria
Sort order
db.users.find({ score: { "$lt": 30 } }).sort({ score: -1 })
{ score: 1 } Index
min
18
30
45
75
max
{ score: 25, ... }
{ score: 56, ... }
{ score: 45, ... }
{ score: 75, ... }
{ score: 5, ... }
{ score: 40, ... }
{ score: 18, ... }
...
{ score: 30, ... }
users

Create an Index

- **Syntax**: db.collection.createIndex({key and index type specification}, {option})

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

Create an Index

- **Index 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 } ,
   { name: "query for inventory" }
)
```

- an index { item : 1, quantity: -1 } has the name **item_1**, **quantity_-1**.

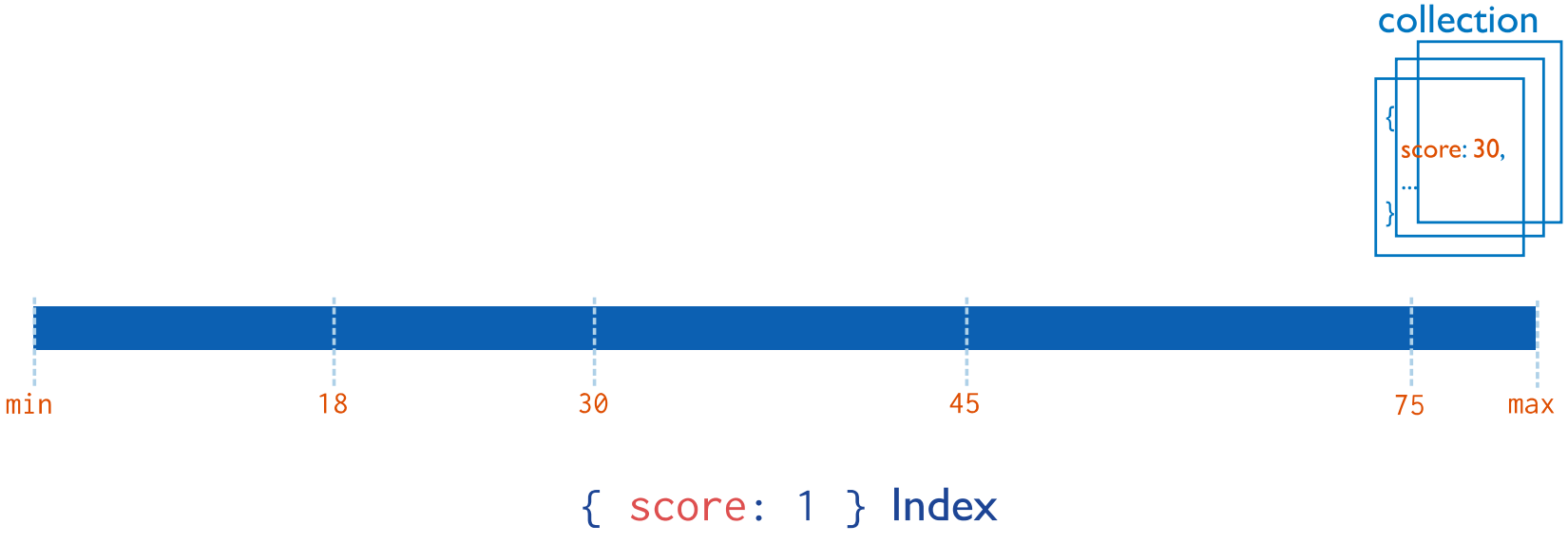
Index Methods

Method	Description
createIndex(indexed_fields, [options]);	Create Indexes
getIndexes();	List all Indexes on a Collection
dropIndex(indexed_fields); dropIndexes();	Delete Index Delete All Index
find(find_fields).explain();	to return the query planning and execution information for the specified find() operation

2. Single Fields

Single Field index

- **Syntax**: db.collection.createIndex({“<fieldName>” : <1 or -1>});
- 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.



Single Field index

Example: collection *records*

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

- Create an Ascending Index on a Single Field

```
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 } } )
```

Single Field index

- **Create an Index on an Embedded Field:**

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

Example:

The created index on the field *location.state*

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

Single Field index

- **Create an Index on Embedded Document**

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

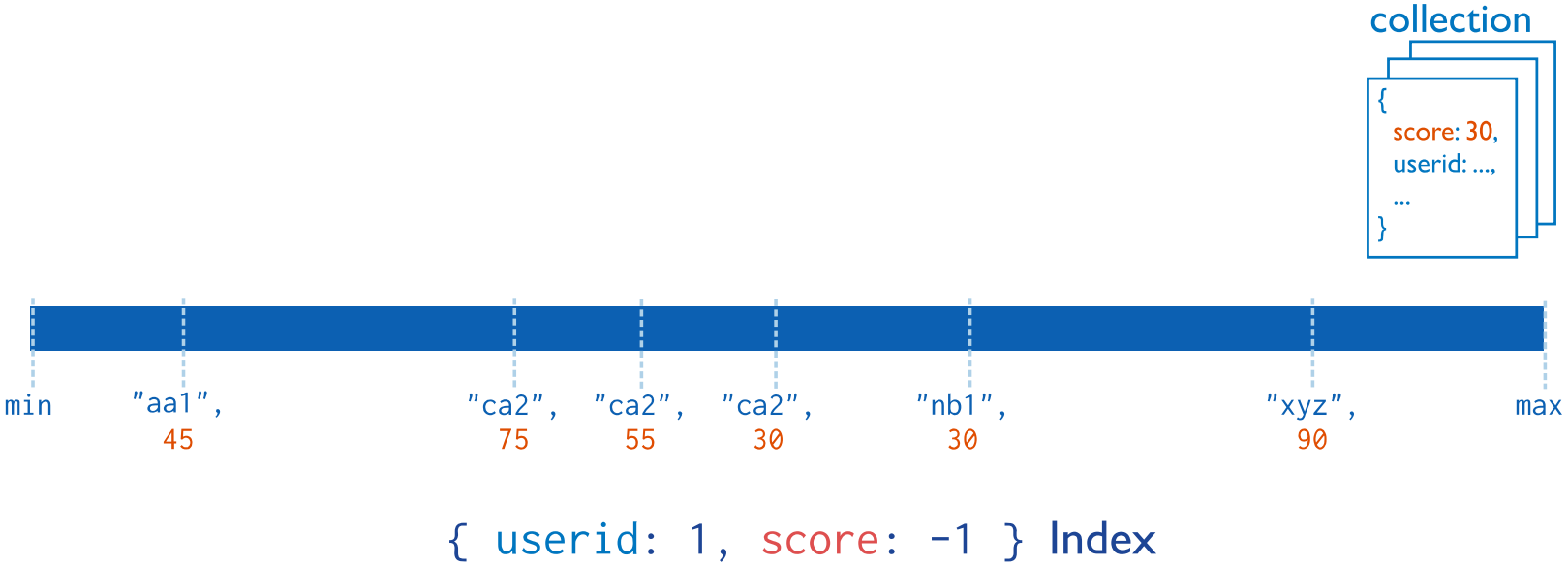
Example: the created index on the *field location of city and state*

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

3. Compound Fields

Compound Indexes

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



Compound Indexes

- **Compound Indexes** does indexing on multiple fields of the document either in ascending or descending order, mean it will sort the data of one field, and then inside that it will sort the data of another field.
- **Syntax**

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

Compound Indexes

- **Example**: collection products

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

Create an **ascending index** on the **item** and **stock** fields

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

The order of the fields listed in a compound index is important. 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

Compound Indexes

- Example: …

Compound indexes can support queries that **match on the prefix of the index** fields.

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

Compound Indexes

Sort Order

- For **compound indexes**, sort order can matter in determining whether the index can support a sort operation.

Example: consider a collection events that contains documents with the fields username and date.

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

the following index can support both these sort operations:

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

However, the above index cannot support sorting by ascending username values and then by ascending date values, such as the following

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

For more information on sort order and compound indexes, see Use Indexes to Sort Query Results.

Compound Indexes

- **Index prefixes**: are the beginning subsets of indexed fields. A compound index includes a set of prefixes. A prefix is a fancy term for the beginning combination of fields in the index.
- **Example**:

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

The index has the following index prefixes

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

- For a compound index, MongoDB can use the index to support queries on the **index prefixes**. As such, MongoDB 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.

Compound Indexes

- The order of the indexed fields has a strong impact on the effectiveness of a particular index for a given query. For most compound indexes, following the ESR (Equality, Sort, Range) rule helps to create efficient indexes.

4. Multikey index

Multikey Index

- MongoDB allows to index a field that holds an array value by creating **an index key** for **each element in the array**, such type of indexing is called **Multikey indexes**.

- **Multikey indexes** supports efficient queries against array fields. It can be constructed over arrays that hold both scalar values (like strings, numbers, etc) and nested documents.

Multikey Index

- **Syntax:** db.collectionName.createIndex({ <field> : <1 or -1> })

*MongoDB automatically creates a **multikey index** if any indexed field is an array; you do not need to explicitly specify the multikey type.*

Multikey Index

Important Points:

- You are not allowed to specify a multikey index as the shard key index.
- In MongoDB, hashed indexes are not multikey index.
- The multikey index cannot support the $expr operator.
- If a filter query specifies the exact match for an array as a whole, then MongoDB scans the multikey index to look for the first element of the array, then, MongoDB retrieves those documents that contain the first element and filter them for the document whose array matches the given query.

Multikey Index

Query on the Array Field as a Whole

- When a query filter specifies an exact match for an array as a whole, MongoDB can use the **multikey index** to look up the first element of the query array *but cannot use the multikey index scan to find* the whole array.
- After using the multikey index to look up the first element of the query array, MongoDB retrieves the associated documents and filters for documents whose array matches the array in the query.

Multikey Index

Query on the Array Field

Example: inventory collection

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

The collection has a multikey index on the ratings field:

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

The following query looks for documents where the ratings field is the array [5, 9]

```
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].

5. Unique Index

Unique Indexes

A unique index ensures that the indexed fields do not store duplicate values; i.e. enforces uniqueness for the indexed fields. By default, MongoDB creates a unique index on the _id field during the creation of a collection.

Create a Unique Index

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

Unique Indexes

Unique Index on a Single Field

Example: create a unique index on the *user_id* field of the members collection.

```
db.members.createIndex({"user_id": 1}, {unique: true})
```

Unique Compound Index: If you use the unique constraint on a compound index, then MongoDB will enforce uniqueness on the combination of the index key values.

Example: create a unique index on *groupNumber, lastname,* and *firstname* fields of the *members* collection

```
db.members.createIndex({
                groupNumber: 1,
                lastname: 1,
                firstname: 1 }
        , {unique: true})
```

Unique Indexes

Unique Compound Index:

Example: consider a collection with the following document. Create a unique compound multikey index on ***a.loc*** and ***a.qty***.

```
{ _id: 1, a: [ { loc: "A", qty: 5 }, { qty: 10 } ] }
```

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

The unique index permits the insertion of the following documents into the collection since the index enforces uniqueness for the combination of ***a.loc*** and ***a.qty*** values:

```
db.collection.insertMany([
        { _id: 2, a: [ { loc: "A" }, { qty: 5 } ] },
        { _id: 3, a: [ { loc: "A", qty: 10 } ] }
])
```

Unique Indexes

- **Unique Constraint Across Separate Documents**
  - The unique constraint applies to separate documents in the collection. That is, the unique index prevents separate documents from having the same value for the indexed key.
  - For a **unique multikey index**, a document may have array elements that result in repeating index key values as long as the index key values for that document do not duplicate those of another document

- **Example**:

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

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

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

The unique index permits the insertion of the following document into the collection if no other document in the collection has an index key value of: `{ "a.loc": "B", "a.qty": null }`.

6. Text Indexes

Text Indexes

To run text search queries on on-premises deployments, you must have a text index on your collection. MongoDB provides text indexes to support text search queries on string content.

Text indexes can include any field whose value is a string or an array of string elements.

A collection can only have **one** text search index

Syntax: `db.collName.createIndex({ <field>: "text" })`

- $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>
   }
}
```

Text Indexes

- **Example**: populate the collection with the following documents:

```
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" })
```

Text Indexes

- $search field specify a string of words that the $text operator parses and uses to query the text index. *(Read more)*
  - Single Word

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

  - Match Any of the Search Terms

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

  - Search for a Phrase

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

  - Exclude Documents That Contain a Term

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

Collations in MongoDB

- **Collation** allows users to specify **language-specific rules** for string comparison, such as rules for lettercase and accent marks.
- You can specify collation for a collection or a view, an index, or specific operations that support collation.
- The **default collation parameter** values vary **depending** on the **language** you specify.

(Read more)

Collations in MongoDB

- **Collation Document**: When specifying **collation**, the locale field is *mandatory*; *all other collation fields are optional*.

```
{
        locale: <string>,
        caseLevel: <boolean>,
        caseFirst: <string>,
        strength: <int>,
        numericOrdering: <boolean>,
        alternate: <string>,
        maxVariable: <string>,
        backwards: <boolean>
}
```

Collations in MongoDB

- **Operations that Support Collation**

Commands	mongosh Methods
create	db.createCollection() db.createView()
createIndexes	db.collection.createIndex()
aggregate	db.collection.aggregate()
distinct	db.collection.distinct()
update	db.collection.updateOne(), db.collection.updateMany(), db.collection.replaceOne()

Collations in MongoDB

- **Operations that Support Collation**

Commands	mongosh Methods
findAndModify	db.collection.findAndModify() db.collection.findOneAndDelete() db.collection.findOneAndReplace() db.collection.findOneAndUpdate()
find	cursor.collation() to specify collation for db.collection.find()
mapReduce	db.collection.mapReduce()

Collations in MongoDB

- **Operations that Support Collation**

Commands	mongosh Methods
delete	db.collection.deleteOne() db.collection.deleteMany() db.collection.remove()
shardCollection	sh.shardCollection()
count	db.collection.count()
	Individual update, replace, and delete operations in db.collection.bulkWrite().

Collations in MongoDB

- **Local Variants**: Some collation locales have variants, which employ special language-specific rules. To specify a locale variant, use the following syntax:

```
{ "locale" : "<locale code>@collation=<variant>" }
```

- Example: use the *unihan* variant of the *Chinese* collation:

```
{ "locale" : "zh@collation=unihan" }
```

Collations in MongoDB

- **Collation and Index Use**: To use an index for **string comparisons**, an **operation** must also **specify the same collation**. That is, an index with a collation cannot support an operation that performs string comparisons on the indexed fields if the operation specifies a different collation.
- **Example**: the collection **myColl** has an index on a string field category with the collation locale "**fr**".

```
db.myColl.createIndex( { category: 1 }, { collation: { locale: "fr" } } )
```

Collations in MongoDB

- **Collation and Index Use**:
- **Example**:The following query operation, which specifies the same collation as the index, can use the index

```
db.myColl.find({category: "cafe"}).collation({locale: "fr" })
```

- However, the following query operation, which by default uses the "simple" binary collator, cannot use the index:

```
db.myColl.find( { category: "cafe" } )
```

Collations in MongoDB

- **Collation and Index Use**: An operation that specifies a different collation can still use the index to support comparisons on the ***index prefix keys***.
- **Example**:
  - The collection **mycoll** has a **compound index** on the numeric fields **score** and **price** and the string field **category**; the index is created with the collation locale "**fr**" for string comparisons:

```
db.myColl.createIndex(
        { score: 1, price: 1, category: 1 },
        { collation: { locale: "fr" } }
)
```

Collations in MongoDB

- **Collation and Index Use**:
- **Example**:
  - The following operations, which use **"simple" binary collation** for string comparisons, can use the index:

```
db.myColl.find( { score: 5 } ).sort( { price: 1 } )
db.myColl.find( { score: 5, price: { $gt: NumberDecimal( "10" ) } } ).sort( { price:1})
```

Collations in MongoDB

- **Collation and Index Use**:
- **Example**:
  - The following operation, which uses ***"simple" binary collation*** for **string comparisons** on the indexed **category** field, can use the index to fulfill only the score: 5 portion of the query

```
db.myColl.find( { score: 5, category: "cafe" } )
```

Use Indexes to Sort Query Results

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

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( { a: 1 } )
```

  - This index can support an **ascending sort** on a:

```
db.records.find().sort( { a: 1 } )
```

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***.
- **Example**:
  - An **index key pattern { a: 1, b: 1 }** can support a **sort on { a: 1, b: 1 }** but not on { b: 1, a: 1 }.

Use Indexes to Sort Query Results

- **Sort on Multiple Fields:**
  - 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** { a: -1, b: -1 } or {a: 1, b: 1}.