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

3

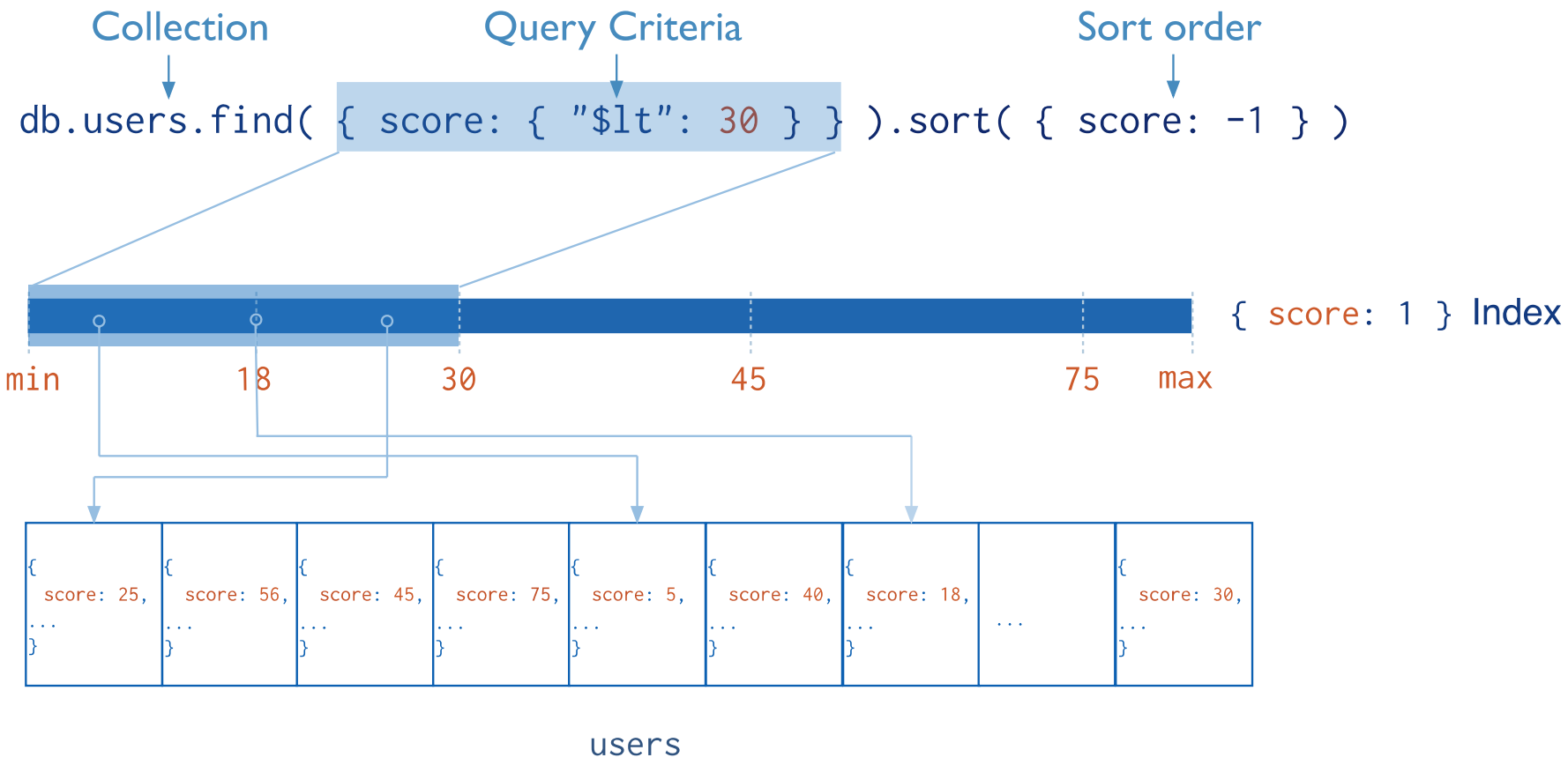
Overview

- **Indexes are special data structures**, that store a small portion of the data set in an easy to-traverse form.

- Without indexes, MongoDB must perform scan every document in a collection, to select those documents that match the query statement.

- By default, MongoDB creates a default index for each collection, which includes the _id field.

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.

- **Example**:

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

*The **default name** an index { item : 1, quantity: -1 } has the name **item_1_quantity_-1***

(Read more)

Index Methods

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

7

Index Methods

db.<collName>.find(<query>).explain("<verbosityMode>");

<verbosityMode>:

- "queryPlanner" (default);
- "executionStats";
- "allPlansExecution".

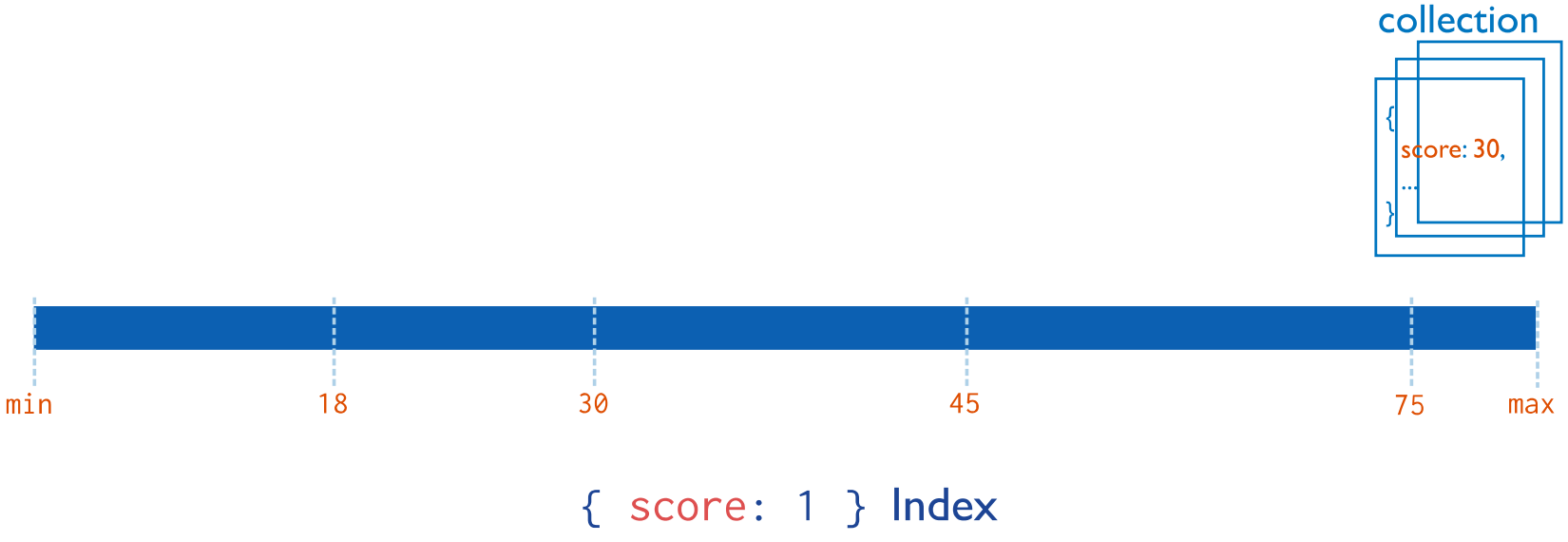
8

2. Single Fields

9

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().sort({ score: 1 })
```


11

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*

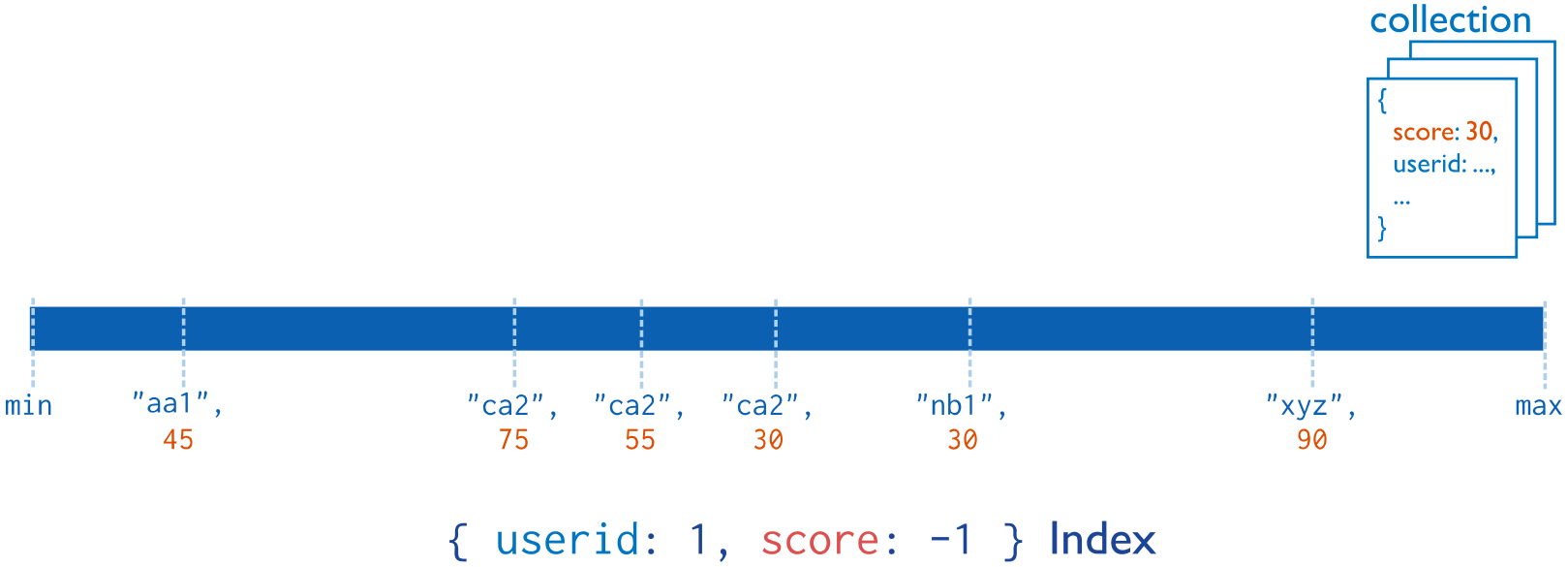
```
db.records.find({
    location: { city: "New York",
    state: "NY" }
})
```

3. Compound Indexes

14

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

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.
  - **Equality** refers to an exact match on a single value;

```
db.cars.find( { model: "Cordoba" } )
db.cars.find( { model: { $eq: "Cordoba" } } )
```

  - **Sort** determines the order for results;

```
db.cars.find( { manufacturer: "GM" } ).sort( { model: 1 } )
db.cars.createIndex( { manufacturer: 1, model: 1 }
```

  - **Range** filters scan fields.

```
db.cars.find( { price: { $gte: 15000} } )
db.cars.find( { age: { $lt: 10 } } )
db.cars.find( { priorAccidents: { $ne: null } } )
```

4. Multikey index

21

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.

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

Important Points:

- MongoDB automatically creates a **multikey index** if any indexed field is an array; you do not need to explicitly specify the multikey type.
- You are not allowed to specify a multikey index as the shard key index.
- The multikey index cannot support the $expr operator.

Multikey Index

Query on the Array Field as a Whole

When a query searches for an exact match of an entire array, MongoDB cannot directly use the multikey index to find the whole array.

Instead, MongoDB:

1. Uses the multikey index to quickly find documents that contain the first element of the query array.
2. Retrieves those documents from storage.
3. Filters them in memory to check if the entire array matches the query.

Key Takeaways:

1. The index only helps find documents containing the first element of the array.
2. MongoDB still needs to filter manually to ensure the entire array matches.
3. This means an exact array match is not fully indexed and may still require additional filtering.

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

25

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 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 Indexes

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 *lopHoc, mssv* fields of the *members* collection

```
db.members.createIndex(
        { lopHoc: 1, mssv: 1 },
        { unique: true }
)
```

Unique Indexes

Example: For example, if your application supports multiple countries, you might want to ensure that each phone number is unique within a specific country, but the same phone number can exist in different countries.You can create a compound unique index as follows:

```
db.users.createIndex({ country: 1, phone: 1 }, { unique: true })
```

```
{ "country": "US", "phone": "1234567890" } ☑
{ "country": "VN", "phone": "1234567890" } ☑ // Valid because the country is different
{ "country": "US", "phone": "1234567890" } ✕ // Error: This phone number already exists in the US
```

Handling Duplicate Errors with Upsert: if you want to update or insert (upsert) a document but there is a possibility of duplication, you can catch the error and handle it:

```
try {
    db.users.insertOne({ name: "Alice", email: "alice@example.com" })
}
catch (e) {
    if (e.code === 11000) {
        print("Error: Email already exists!")
    }
}
```

6. Text Indexes

29

Text Indexes

Text Index in MongoDB enables efficient text searching by indexing string fields. It supports full-text search, allowing you to query data by keywords instead of using regex.

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

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>
   }
}
```


30

Text Indexes

- **Example**: you can create a text index on one or more fields in MongoDB using the command

```
db.articles.createIndex({ content: "text" }) or
db.articles.createIndex({ title: "text", content: "text" })
```

Find all documents that contain the word "**MongoDB**" in indexed fields. You can also search for an exact phrase by enclosing it in double quotes (""):

```
db.articles.find({ $text: { $search: "MongoDB" } })
db.articles.find({ $text: { $search: "\"Full text search\"" } })
```


31

Text Indexes

Some Notes on Text Indexes in MongoDB

- Each collection can have only one text index, but this index can cover multiple fields.

- Accent-insensitive search is not supported, meaning words with accents and their accent-free counterparts are considered different (e.g., "cà phê" and "ca phe" are treated as distinct words).

- Text indexes do not support searching within array fields.

32

Comparison Table of Index Types in MongoDB

Index Type	Usage	Meaning	Similarities	Differences	Advantages	Disadvantages
Single Field Index	db.users.createIndex({ age: 1 })	Creates an index on a single field to speed up queries	All optimize queries on a collection	Only efficient when searching by a single field	Speeds up searches for one field, supports sorting	Not optimal for queries combining multiple fields
Compound Index	db.users.createIndex({ age: 1, name: 1 })	Indexes multiple fields to optimize queries using both	Also helps optimize queries	The order of fields in the index affects performance	Supports multiple fields in one index, making queries more efficient	If queries do not follow the index order, the index may not be fully utilized
Multikey Index	db.products.createIndex({ tags: 1 }) (where tags is an array)	Automatically created when indexing an array field	All improve query performance	Used for array fields, unlike indexes on single-value fields	Enables fast searches on array fields	Cannot be used as a shard key, does not support sorting across multiple fields
Unique Index	db.users.createIndex({ email: 1 }, { unique: true })	Ensures that a field's value is unique in a collection	Also helps optimize queries	Does not allow duplicate values	Maintains data integrity by preventing duplicates	Cannot be applied to fields with pre-existing duplicate values
Text Index	db.articles.createIndex ({ content: "text" })	Used for keyword-based text searches	Speeds up text-based searches	Only one text index is allowed per collection	Enables full-text search, ideal for large content fields	Does not support exact-match searches like standard indexes
Hashed Index	db.users.createIndex({ user_id: "hashed" })	Hashes values to support sharding	Helps optimize searches	Only useful for sharding purposes	Well-suited for sharding, fast lookups based on a hashed field	Does not support range queries or sorting
Wildcard Index	db.logs.createIndex({ "$**": 1 })	Automatically creates indexes for all fields in a document	Improves search speed across entire documents	Unlike other indexes, it does not require predefined fields	Flexible, avoids the need for multiple individual indexes	Consumes more storage, increases write overhea

33

Collations in MongoDB

What is Collation?

Collation in MongoDB allows for customizable string comparison and sorting, including:

- Case sensitivity (e.g., "apple" vs. "Apple")
- Accent sensitivity (e.g., “cà phê" vs. “ca phe")
- Locale-based sorting (e.g., "ä" sorted correctly in German)

By default, MongoDB uses binary comparisons, meaning uppercase letters are sorted before lowercase, and accented characters are treated differently. Collation helps apply natural-language sorting rules instead.

(Read more)

Collations in MongoDB

Basic Syntax: *to define collation in MongoDB, use the collation option in queries, indexes, and aggregations.*

```
db.collection.find().collation({ locale: "en", strength: 2 });
```

"locale": "en" → Uses English collation rules.

"strength": 2" → Case-insensitive comparison (e.g., "apple" = "Apple").

Example: Case-insensitive search

```
db.users.find({ name: "nguyen" }).collation({ locale: "vi", strength: 2 });
```

Example: Accent-insensitive search

```
db.products.find({ name: "cafe" }). collation({ locale: "vi", strength: 1 });
```

Collations in MongoDB

Creating Indexes with Collation

```
db.users.createIndex({ name: 1 }, { collation: { locale: "en", strength: 2 } });
```

Sorting with Collation

- **Default sorting** *(binary order)*:

```
db.countries.find().sort({ name: 1 });
```

```
["Canada", "Germany", "apple", "france"]
```

- **Sorting with collation** *(natural order)*:

```
db.countries.find().sort({ name: 1 }).collation({ locale: "vi" });
```

```
["apple", "Canada", "france", "Germany"]
```

Collations in MongoDB

Collation Strength Levels

Strength	Comparison Behavior	Example Matching
1 *(Primary)*	Ignores case & accents	"cafe" = "café" = "CAFÉ"
2 *(Secondary)*	Ignores case, but considers accents	"cafe" ≠ "café" but "café" = "CAFÉ"
3 *(Tertiary)*	Case-sensitive, accent-sensitive	"cafe" ≠ "café" and "café" ≠ "CAFÉ"
4 *(Quaternary)*	Distinguishes punctuation differences	"hello!" ≠ "hello?"

Collations in MongoDB

Available Locales

MongoDB supports over 100 locales. Some common ones:

- *"en" → English*
- *"fr" → French*
- *"de" → German*
- *"es" → Spanish*
- *"vi" → Vietnamese*
- *…*

Full list of supported locales: MongoDB Collation Locales

Collations in MongoDB

When to Use Collation?

- Case-insensitive search (e.g., "John" = "john")
- Accent-insensitive search (e.g., "café" = "cafe")
- Sorting names correctly based on locale rules
- Ensuring index-based performance for language-based searches