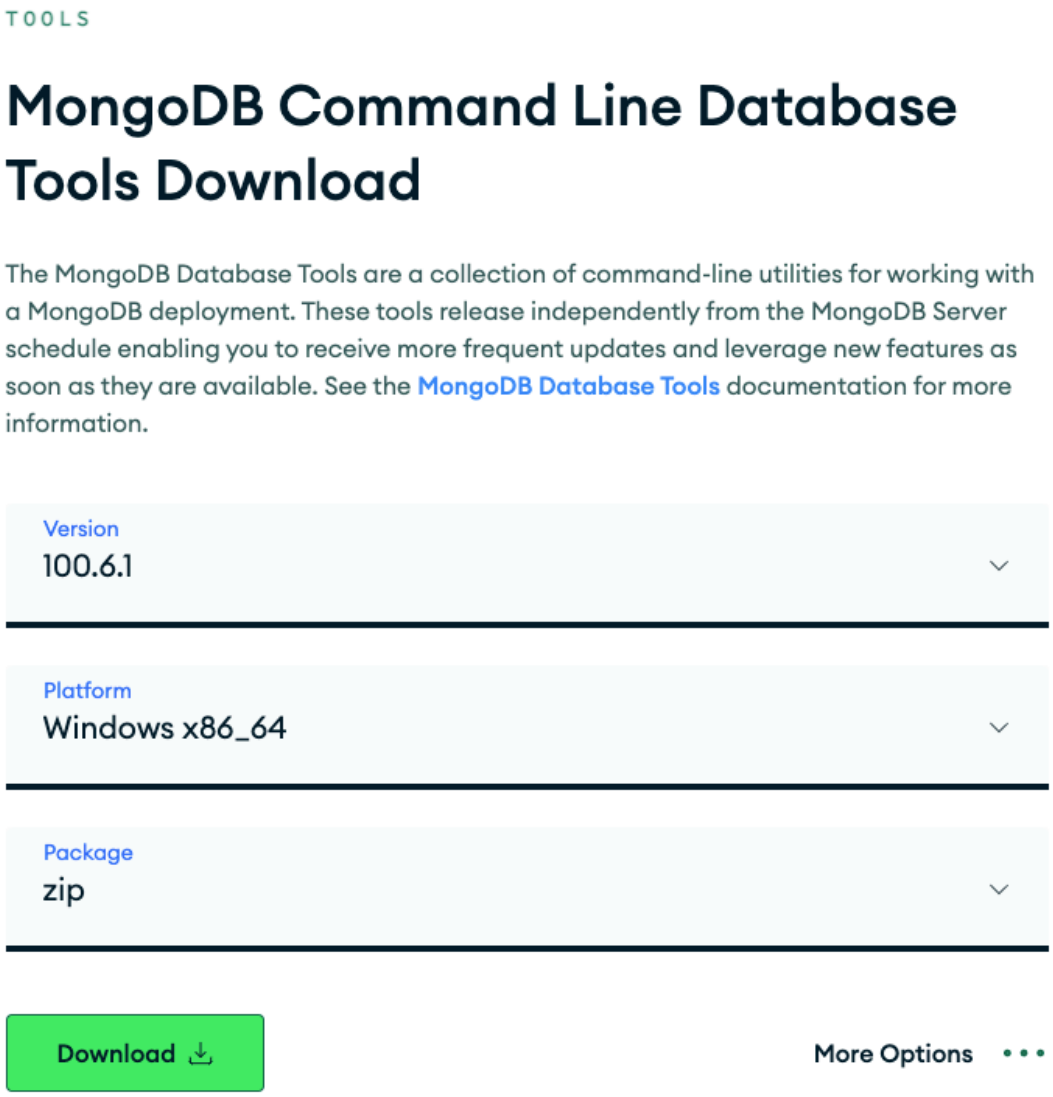
Let’s work on optimization and management tasks in MongoDB

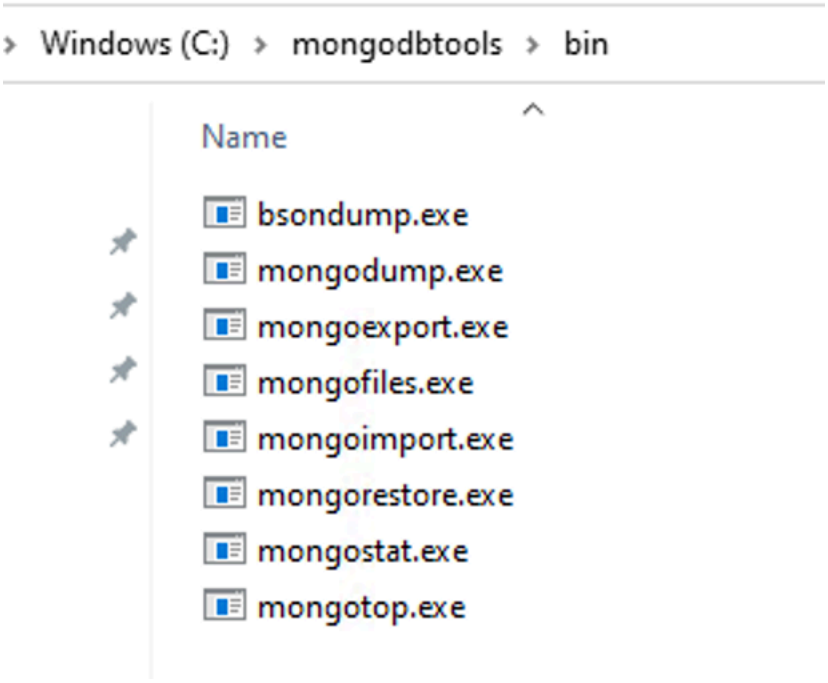
- **Task 1. Download MongoDB Database Tools**
- **Task 2. Importing data into a database**
- **Task 3. Exporting a database**
- **Task 4. Working with indexes**
- **Task 5. Understanding Text Search**
- **Task 6. Implementing a database backup and restore**

Task 1. Download MongoDB Database Tools

The MongoDB Database Tools are a suite of **command-line utilities** for working with MongoDB. You can download them from here: https://www.mongodb.com/try/download/database-tools.



After the download process finishes, extract the folder in a known, specific location.



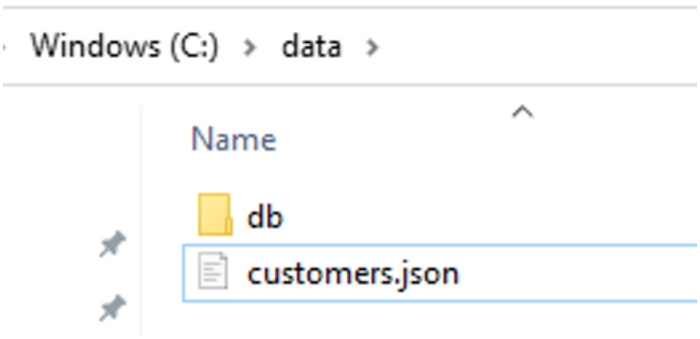
Open a Command Prompt (shell) in the same path

```
C:\Users\luisbeltran>cd C:\mongodbtools\bin

C:\mongodbtools\bin>_
```

Task 2. Importing data into a database

Download the following zip file (https://files.studio3t.com/json/customers-test-data.json.zip). Extract the content and place the json file inside a known path.



The file contains 70, 000 documents (in JSON format) which representes customers data (name, addresses, phone numbers, credit card details, and 'file notes'). These have been generated from random numbers.

Use the **mongoimport** tool (which is available in MongoDBTools) in order to load the JSON file into a new database (it also works with CSV or TSV formats).

Run **mongoimport** from the system command line, not the mongo shell:

mongoimport PATH_TO_customers.json -d company -c customers --jsonArray

```
C:\mongodbtools\bin>mongoimport c:\data\customers.json -d company -c customers --jsonArray
2022-11-20T17:46:23.507-1200    connected to: mongodb://localhost/
2022-11-20T17:46:26.516-1200    [#####...................] company.customers    20.9MB/85.7MB (24.3%)
2022-11-20T17:46:29.507-1200    [############............] company.customers    43.1MB/85.7MB (50.3%)
2022-11-20T17:46:32.515-1200    [#################.......] company.customers    63.8MB/85.7MB (74.4%)
2022-11-20T17:46:35.507-1200    [######################.] company.customers    83.0MB/85.7MB (96.8%)
2022-11-20T17:46:35.927-1200    [########################] company.customers    85.7MB/85.7MB (100.0%)
2022-11-20T17:46:35.928-1200    70000 document(s) imported successfully. 0 document(s) failed to import.

C:\mongodbtools\bin>
```

Arguments:

- -d specifies the name of the database on which run the command
- -c specifies the collection to import
- --jsonArray accepts the import of data expressed with multiple MongoDB documents within a single JSON array
- You can include the --drop flag to remove the collection if it exists before importing the data

Task 3. Exporting a database

mongoexport is a command-line tool that produces a JSON or CSV export of data stored in a MongoDB instance.

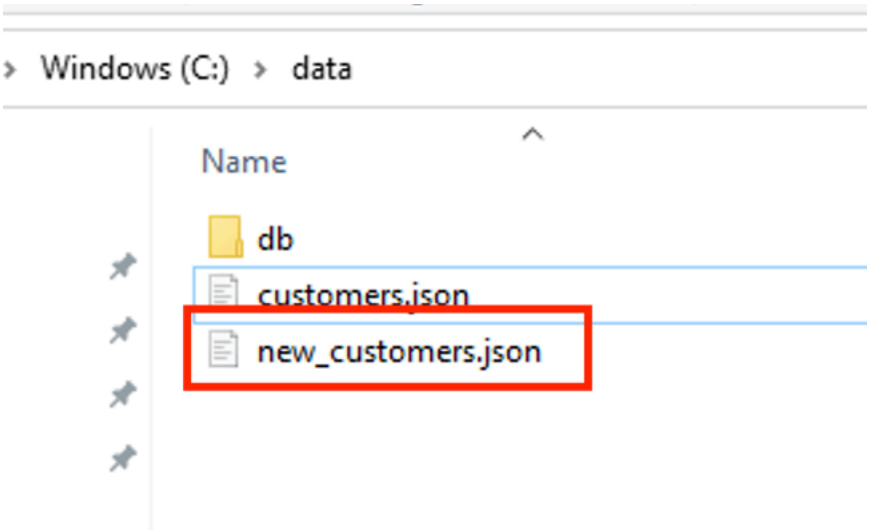
Run mongoexport from the system command line, not the mongo shell:

mongoexport -d company -c customers --jsonArray --out PATH_TO_EXPORT_customers.json

```
C:\mongodbtools\bin>mongoexport -d company -c customers --jsonArray --out c:\data\new_customers.json
2022-11-20T22:59:07.812-1200    connected to: mongodb://localhost/
2022-11-20T22:59:08.817-1200    [##......................]  company.customers  8000/70000  (11.4%)
2022-11-20T22:59:09.818-1200    [#####...................]  company.customers  16000/70000  (22.9%)
2022-11-20T22:59:10.816-1200    [########................]  company.customers  24000/70000  (34.3%)
2022-11-20T22:59:11.825-1200    [#############...........]  company.customers  40000/70000  (57.1%)
2022-11-20T22:59:12.825-1200    [################........]  company.customers  48000/70000  (68.6%)
2022-11-20T22:59:13.817-1200    [###################.....]  company.customers  56000/70000  (80.0%)
2022-11-20T22:59:14.517-1200    [########################]  company.customers  70000/70000  (100.0%)
2022-11-20T22:59:14.518-1200    exported 70000 records
```

Arguments:

- -d specifies the name of the database on which to run the command.
- -c specifies the collection to export
- --jsonArray modifies the output of mongoexport to write the entire contents of the export as a single JSON array.
- --out specifies a file to write the export to.



Task 4. Working with indexes

Indexes are special data structures that store a small portion of the collection's 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. The ordering of the index entries supports efficient equality matches and range-based query operations.

When MongoDB imports your data into a collection, it will create a primary key that is enforced by an **index**. But it can't guess the other indexes you'd need because there is no way that it can predict the sort of searches, sorting and aggregations that you'll want to do on this data. It just provides a unique identifier for each document in your collection, which is retained in all further indexes.

Indexes support the efficient execution of queries in MongoDB. Without them, MongoDB must perform a **collection scan**, which requires scanning every document in a collection to find a match to your query (which is a time- and resource-consuming task). With the right indexes, you can query more efficiently, as the number of documents are limited from the get-go. So you need an effective indexing strategy to quickly and efficiently get the information you need.

Before you create a collection, you need to consider **collation**, the way that searching and sorting is performed.

When you see strings in order, do you want to see lower-case sorted after uppercase, or should your sorting ignore case? Do you consider a value represented by a string to be different according to the characters that are in capitals? How do you deal with accented characters? By default, collections have a **binary** collation. To find out what collation, if any, is used for your collection, you can use this command:

db.getCollectionInfos({name: 'customers'})

You can't change the collation of an existing collection. You need to create the collection before adding the data:

db.createCollection("Customers", {collation:{locale:"en",strength:1}})

```
company> db.createCollection("Customers", {collation:{locale:"en",strength:1}})
{ ok: 1 }
```

The strength of 1 gives you a case-insensitive and diacritic-insensitive search.

```
company> db.getCollectionInfos({name: 'Customers'})
[
  {
    name: 'Customers',
    type: 'collection',
    options: {
      collation: {
        locale: 'en',
        caseLevel: false,
        caseFirst: 'off',
        strength: 1,
        numericOrdering: false,
        alternate: 'non-ignorable',
        maxVariable: 'punct',
        normalization: false,
        backwards: false,
        version: '57.1'
      }
    },
    info: {
      readOnly: false,
      uuid: new UUID("1884eb02-cd20-4be5-9ab4-06dfcb4c4f2b")
    },
    idIndex: {
      v: 2,
```

Exercise #1
Exit the mongo shell and import the data again, but this time into the **Customers** collection.

We will now execute a simple query against our newly-created database to find all customers whose surname is ‘Johnston’.

We wish to perform a projection over, or select, ‘First Name‘ and ‘Last Name’, sorted by ‘Last Name’. We will

```
db.Customers.find(
 { "Name.Last Name" : "Johnston" },
 { "_id" : 0, "Name.First Name" : 1, "Name.Last Name" : 1 }
).sort({ "Name.Last Name" : 1 });
```

```
company> db.Customers.find(
...    { "Name.Last Name" : "Johnston" },
...    { "_id" : 0, "Name.First Name" : 1, "Name.Last Name" : 1 }
... ).sort({ "Name.Last Name" : 1 });
[
  { Name: { 'First Name': 'Carole', 'Last Name': 'Johnston' } },
  { Name: { 'First Name': 'Audrey', 'Last Name': 'Johnston' } },
  { Name: { 'First Name': 'Shayne', 'Last Name': 'Johnston' } },
  { Name: { 'First Name': 'Wilson', 'Last Name': 'Johnston' } },
  { Name: { 'First Name': 'Ty', 'Last Name': 'Johnston' } },
  { Name: { 'First Name': 'Rosalinda', 'Last Name': 'Johnston' } },
  { Name: { 'First Name': 'Darin', 'Last Name': 'Johnston' } },
  { Name: { 'First Name': 'Chastity', 'Last Name': 'Johnston' } },
  { Name: { 'First Name': 'Alison', 'Last Name': 'Johnston' } },
  { Name: { 'First Name': 'Gerardo', 'Last Name': 'Johnston' } },
  { Name: { 'First Name': 'Jonathon', 'Last Name': 'Johnston' } },
  { Name: { 'First Name': 'Renae', 'Last Name': 'Johnston' } },
  { Name: { 'First Name': 'Gus', 'Last Name': 'Johnston' } },
  { Name: { 'First Name': 'Jasen', 'Last Name': 'Johnston' } },
  { Name: { 'First Name': 'Anissa', 'Last Name': 'Johnston' } },
  { Name: { 'First Name': 'Mayra', 'Last Name': 'Johnston' } },
  { Name: { 'First Name': 'Kasey', 'Last Name': 'Johnston' } },
  { Name: { 'First Name': 'Karrie', 'Last Name': 'Johnston' } },
  { Name: { 'First Name': 'Daryl', 'Last Name': 'Johnston' } },
  { Name: { 'First Name': 'Misti', 'Last Name': 'Johnston' } }
]
```

Once we are happy that the query is returning the correct result, we can modify it to return the **execution stats**.

```
db.Customers.find(
  { "Name.Last Name" : "Johnston" },
  { "_id" : 0, "Name.First Name" : 1, "Name.Last Name" : 1 }
).sort({ "Name.Last Name" : 1 }
).explain("executionStats")
```

There are a couple of sections we are interested in:

```
winningPlan: {
  stage: 'SORT',
  sortPattern: { 'Name.Last Name': 1 },
  memLimit: 104857600,
  type: 'simple',
  inputStage: {
    stage: 'PROJECTION_DEFAULT',
    transformBy: { _id: 0, 'Name.First Name': 1, 'Name.Last Name': 1 },
    inputStage: {
      stage: 'COLLSCAN',
      filter: { 'Name.Last Name': { '$eq': 'Johnston' } },
      direction: 'forward'
    }
  }
},
```

The query involves a **COLLSCAN**, meaning that there is no index available, so MongoDB must scan the entire collection.

```
executionStats: {
  executionSuccess: true,
  nReturned: 68,
  executionTimeMillis: 122,
  totalKeysExamined: 0,
  totalDocsExamined: 70000,
  executionStages: {
    stage: 'SORT',
    nReturned: 68,
    executionTimeMillisEstimate: 5,
    works: 70071,
    advanced: 68,
    needTime: 70002,
    needYield: 0,
    saveState: 70,
    restoreState: 70,
    isEOF: 1,
    sortPattern: { 'Name.Last Name': 1 },
    memLimit: 104857600,
```

According to the execution stats, this takes 122 ms on my machine. This isn't necessarily a bad thing with a reasonably small collection, but as the size increases and more users access the data, the collection is less likely to fit in paged memory, and disk activity will increase.

The database won't scale well if it is forced to do a large percentage of COLLSCANs. **It is a good idea to minimize the resources used by frequently-run queries.**

Let’s create our first index over the "Name.Last Name" field in ascending order in order to reduce the time taken by our query.

db.Customers.createIndex({"Name.Last Name" : 1 },{ name: "LastNameIndex"})

```
company> db.Customers.createIndex( {"Name.Last Name" : 1 },{ name: "LastNameIndex"} )
LastNameIndex
company>
```

And run the query again (including the execution stats):

```
company> db.Customers.find( { "Name.Last Name": "Johnston" }, { "_id": 0, "Name.First Name": 1,
"Name.Last Name": 1 }). sort({ "Name.Last Name": 1 }). explain("executionStats")
{
  explainVersion: '1',
  queryPlanner: {
    namespace: 'company.Customers',
    indexFilterSet: false,
    parsedQuery: { 'Name.Last Name': { '$eq': 'Johnston' } },
    collation: {
```

Let’s take a look at the **winningPlan** and **executionStats** sections:

```
  winningPlan: {
    stage: 'PROJECTION_DEFAULT',
    transformBy: { _id: 0, 'Name.First Name': 1, 'Name.Last Name': 1 },
    inputStage: {
      stage: 'FETCH',
      inputStage: {
        stage: 'IXSCAN',
        keyPattern: { 'Name.Last Name': 1 },
        indexName: 'LastNameIndex',
        collation: {
          locale: 'en',
          caseLevel: false,
          caseFirst: 'off',
          strength: 1,
```

```
  executionStats: {
    executionSuccess: true,
    nReturned: 68,
    executionTimeMillis: 26,
    totalKeysExamined: 68,
    totalDocsExamined: 68,
    executionStages: {
      stage: 'PROJECTION_DEFAULT',
      nReturned: 68,
      executionTimeMillisEstimate: 26,
      works: 69,
```

Now that’s an improvement! The query only evaluates **68 documents** because it now involves an **IXSCAN** (an index scan to get keys) followed by a **FETCH** (for retrieving the documents). Moreover, the query takes **only 26 ms** this time!

We can improve on this because the query has to get the first name. If we add the **Name.First Name** into the index, then the database engine can use the value in the index rather than having the extra step of taking it from the database.

Run the following two commands:

db.Customers.dropIndex("LastNameIndex")

db.Customers.createIndex({ "Name.Last Name" : 1,"Name.First Name" : 1 },
{ name: "LastNameCompoundIndex"})

```
company> db.Customers.dropIndex("LastNameIndex")
{ nIndexesWas: 2, ok: 1 }
company> db.Customers.createIndex( { "Name.Last Name" : 1,"Name.First Name" : 1 },
... { name: "LastNameCompoundIndex"} )
LastNameCompoundIndex
```

We have just created a **Compound Index** (it includes two fields)

Run the query with the executionStats:

```
winningPlan: {
  stage: 'PROJECTION_DEFAULT',
  transformBy: { _id: 0, 'Name.First Name': 1, 'Name.Last Name': 1 },
  inputStage: {
    stage: 'FETCH',
    inputStage: {
      stage: 'IXSCAN',
      keyPattern: { 'Name.Last Name': 1, 'Name.First Name': 1 },
      indexName: 'LastNameCompoundIndex',
      collation: {
```

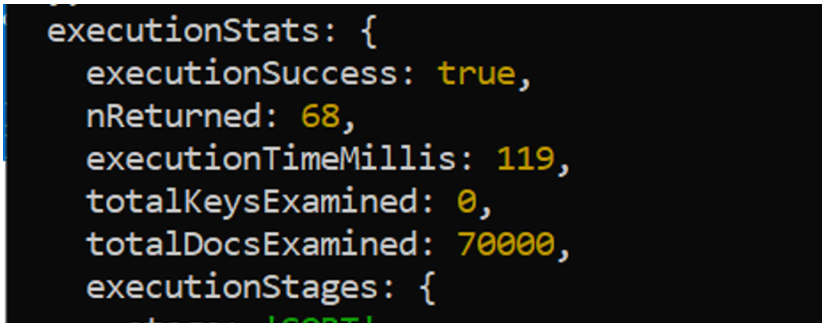
```
executionStats: {
  executionSuccess: true,
  nReturned: 68,
  executionTimeMillis: 11,
  totalKeysExamined: 68,
  totalDocsExamined: 68,
  executionStages: {
    stage: 'PROJECTION_DEFAULT',
```

You can see that the new index is being used and that the query takes less time than before.

But what if you got the index wrong? That can cause problems.

Exercise #2

- Drop the LastNameCompoundIndex index
- Change the order of the two fields in the index so that the Name.First Name comes before Name.Last Name
- Run the query we've been using. You will observe that the execution time shoots up!



The index has actually slowed the execution down so that it takes a longer the time it took with just the default primary index. MongoDB certainly checks the possible execution strategies for a good one, but unless you have provided a suitable index, it is difficult for it to select the right one.

Creating many different indexes, so as to ensure that there will be one that is likely to be suitable, can be tempting, but the downside is that each index uses resources and needs to be maintained by the system whenever the data changes.

If you overdo indexes, they will come to dominate the memory pages and lead to excessive disk I/O. A small number of highly-effective indexes are best.

Exercise #3:

Create an index over the "Name.Last Name" and "Addresses.Full Address" fields to speed up the following query that returns information about a customer called Wiggins who lives in Rutland:

```
db.Customers.find({
   "Name.Last Name" : "Wiggins",
   "Addresses.Full Address" : /.*rutland.*/i
});
```

(Report how much time the query took by including the statistics)

Useful commands: Return the description of all the indexes in the collection with **getIndexes** method:

db.Customers.getIndexes()

```
company> db.Customers.getIndexes()
[
  {
    v: 2,
    key: { _id: 1 },
    name: '_id_',
    collation: {
      locale: 'en',
      caseLevel: false,
      caseFirst: 'off',
      strength: 1,
      numericOrdering: false,
      alternate: 'non-ignorable',
      maxVariable: 'punct',
      normalization: false,
      backwards: false,
      version: '57.1'
    }
  },
  {
    v: 2,
    key: { 'Name.First Name': 1, 'Name.Last Name': 1 },
    name: 'FirstNameCompoundIndex',
    collation: {
      locale: 'en',
      caseLevel: false,
```

Task 5. Understanding Text Search

MongoDB queries that filter data by searching for exact matches, using greater-than or less-than comparisons, or by using regular expressions will work well enough in many situations. However, these methods fall short when it comes to filtering against fields containing rich textual data.

Imagine you typed “coffee recipe” into a web search engine **but it only returned pages that contained that exact phrase**. In this case, you may not find exactly what you were looking for since most popular websites with coffee recipes may not contain the exact phrase “coffee recipe.” If you were to enter that phrase into a real search engine, though, you might find pages with titles like “Great Coffee Drinks (with Recipes!)” or “Coffee Shop Drinks and Treats You Can Make at Home.” In these examples, the word “coffee” is present but the titles contain another form of the word “recipe” or exclude it entirely.

This level of flexibility in matching text to a search query is typical for full-text search engines that specialize in searching textual data.

To start using MongoDB’s **full-text search capabilities**, you must create a **text index** on a collection. A text index is a special type of index used to further facilitate searching fields containing text data. When a user creates a text index, MongoDB will automatically drop any language-specific stop words from searches. This means that MongoDB will ignore the most common words for the given language (in English, words like “a”, “an”, “the”, or “this”).

MongoDB will also implement a form of suffix-stemming in searches. This involves MongoDB identifying the root part of the search term and treating other grammar forms of that root (created by adding common suffixes like “-ing”, “-ed”, or perhaps “-er”) as equivalent to the root for the purposes of the search.

Thanks to these and other features, MongoDB can more flexibly support queries written in natural language and provide better results.

Exercise #4

- Create a recipes database
- Insert the following data:

[
{"name": "Cafecito", "description": "A sweet and rich Cuban hot coffee made by topping an espresso shot with a thick sugar cream foam."},
{"name": "New Orleans Coffee", "description": "Cafe Noir from New Orleans is a spiced, nutty coffee made with chicory."},
{"name": "Affogato", "description": "An Italian sweet dessert coffee made with fresh-brewed espresso and vanilla ice cream."},
{"name": "Maple Latte", "description": "A wintertime classic made with espresso and steamed milk and sweetened with some maple syrup."},
{"name": "Pumpkin Spice Latte", "description": "It wouldn't be autumn without pumpkin spice lattes made with espresso, steamed milk, cinnamon spices, and pumpkin puree."}
]

Now create a text index that includes the name and description fields

db.recipes.createIndex({ "name": "text", "description": "text" });

```
recipes> db.recipes.createIndex({ "name": "text", "description": "text" });
name_text_description_text
```

For each of the two fields, the index type is set to **text**, telling MongoDB to create a **text index** tailored for full-text search based on these fields.

Searching for One or More Individual Words

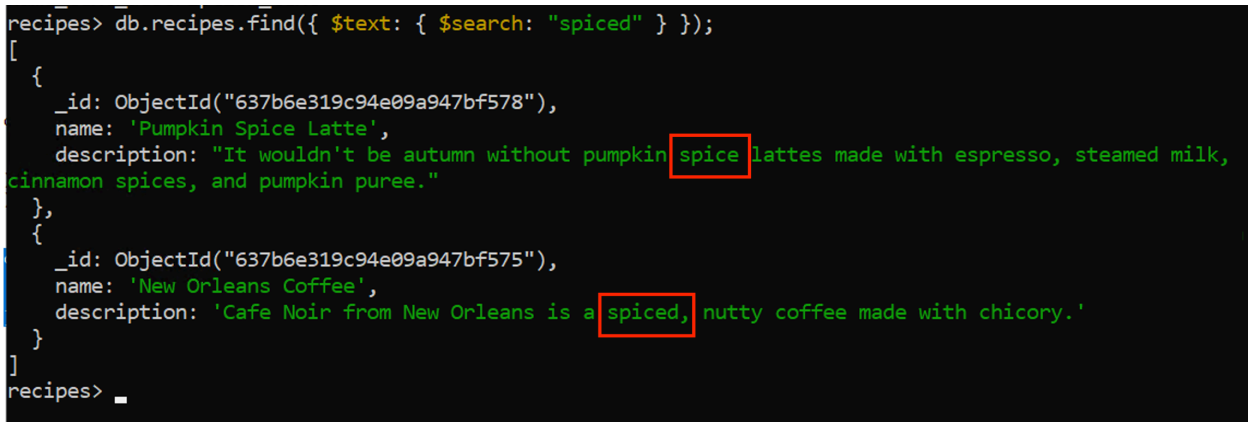
Typically, users expect the search engine to be flexible in determining where the given search terms should appear. As an example, if you were to use any popular web search engine and type in “coffee sweet spicy”, you likely are not expecting results that will contain those three words in that exact order. It’s more likely that you’d expect a list of web pages containing the words “coffee”, “sweet”, and “spicy” but not necessarily immediately near each other.

That’s also how MongoDB approaches typical search queries when using text indexes.

Say you want to search for coffee drinks with spices in their recipe, so you search for the word **spiced** alone using the following command:

db.recipes.find({ $text: { $search: "spiced" } });

The syntax when using full-text search is slightly different from regular queries. **Individual field names** — like name or description — **don't appear in the filter document**. Instead, the query uses the **$text operator**, telling MongoDB that this query intends to use the text index you created previously. Inside the embedded document for this filter is the $search operator taking the search query as its value. In this example, the query is a single word: **spiced**.



There are **two documents** in the result set, both of which contain words resembling the search query. While the New Orleans Coffee document does have the word spiced in the description, **the Pumpkin Spice Late document doesn't**. It was still returned by this query thanks to MongoDB's use of **stemming**. MongoDB stripped the word spiced down to just spice, looked up spice in the index, and also stemmed it. Because of this, the words spice and spices in the Pumpkin Spice Late document matched the search query successfully, even though you didn't search for either of those words specifically.

Exercise #5:
Try looking up documents with a two-word query, **spiced espresso**, to look for a spicy, espresso-based coffee. The list of results this time is longer than before:

```
recipes> db.recipes.find({ $text: { $search: "spiced espresso" } });
[
  {
    _id: ObjectId("637b6e319c94e09a947bf577"),
    name: 'Maple Latte',
    description: 'A wintertime classic made with espresso and steamed milk and sweetened with some maple syrup.'
  },
  {
    _id: ObjectId("637b6e319c94e09a947bf576"),
    name: 'Affogato',
    description: 'An Italian sweet dessert coffee made with fresh-brewed espresso and vanilla ice cream.'
  },
  {
    _id: ObjectId("637b6e319c94e09a947bf578"),
    name: 'Pumpkin Spice Latte',
    description: "It wouldn't be autumn without pumpkin spice lattes made with espresso, steamed milk, cinnamon spices,
and pumpkin puree."
  },
  {
    _id: ObjectId("637b6e319c94e09a947bf574"),
    name: 'Cafecito',
    description: 'A sweet and rich Cuban hot coffee made by topping an espresso shot with a thick sugar cream foam.'
  },
recipes>
    _id: ObjectId("637b6e319c94e09a947bf575"),
```

When using **multiple words** in a search query, MongoDB performs a **logical OR operation**, so a document only has to match one part of the expression to be included in the result set. The results contain documents with:

- Both spiced and espresso
- Or either term alone.

Notice that words do not necessarily need to appear near each other as long as they appear in the document somewhere.

You can also search for exact phrases by wrapping them in double-quotes. If the **$search** string includes a phrase and individual terms, text search will only match documents that include the phrase. Example:

db.recipes.find({ $text: { $search: "\"ice cream\"" } });

```
recipes> db.recipes.find({ $text: { $search: "\"ice cream\"" } });
[
  {
    _id: ObjectId("637b6e319c94e09a947bf576"),
    name: 'Affogato',
    description: 'An Italian sweet dessert coffee made with fresh-brewed espresso and vanilla ice cream.'
  }
]
```

This document matches the search term exactly, and neither cream nor ice **alone** would be enough to count as a match.

Notice the backslashes preceding each of the double quotes surrounding the phrase. The backslashes (\) escape the double quotes so they're not treated as a part of JSON syntax, since these can appear inside the $search operator value.

Another useful full-text search feature is the exclusion modifier. To exclude a word, you can prepend a "-" character. For example, to find all recipes containing "spiced" or "expresso" but not "cream", use the following:

db.recipes.find({ $text: { $search: "spiced espresso -cream" } });

```
recipes> db.recipes.find({ $text: { $search: "spiced espresso -cream" } });
[
  {
    _id: ObjectId("637b6e319c94e09a947bf577"),
    name: 'Maple Latte',
    description: 'A wintertime classic made with espresso and steamed milk and sweetened with some maple syrup.'
  },
  {
    _id: ObjectId("637b6e319c94e09a947bf578"),
    name: 'Pumpkin Spice Latte',
    description: "It wouldn't be autumn without pumpkin spice lattes made with espresso, steamed milk, cinnamon sp
ices, and pumpkin puree."
  },
  {
    _id: ObjectId("637b6e319c94e09a947bf575"),
    name: 'New Orleans Coffee',
    description: 'Cafe Noir from New Orleans is a spiced, nutty coffee made with chicory.'
  }
]
```

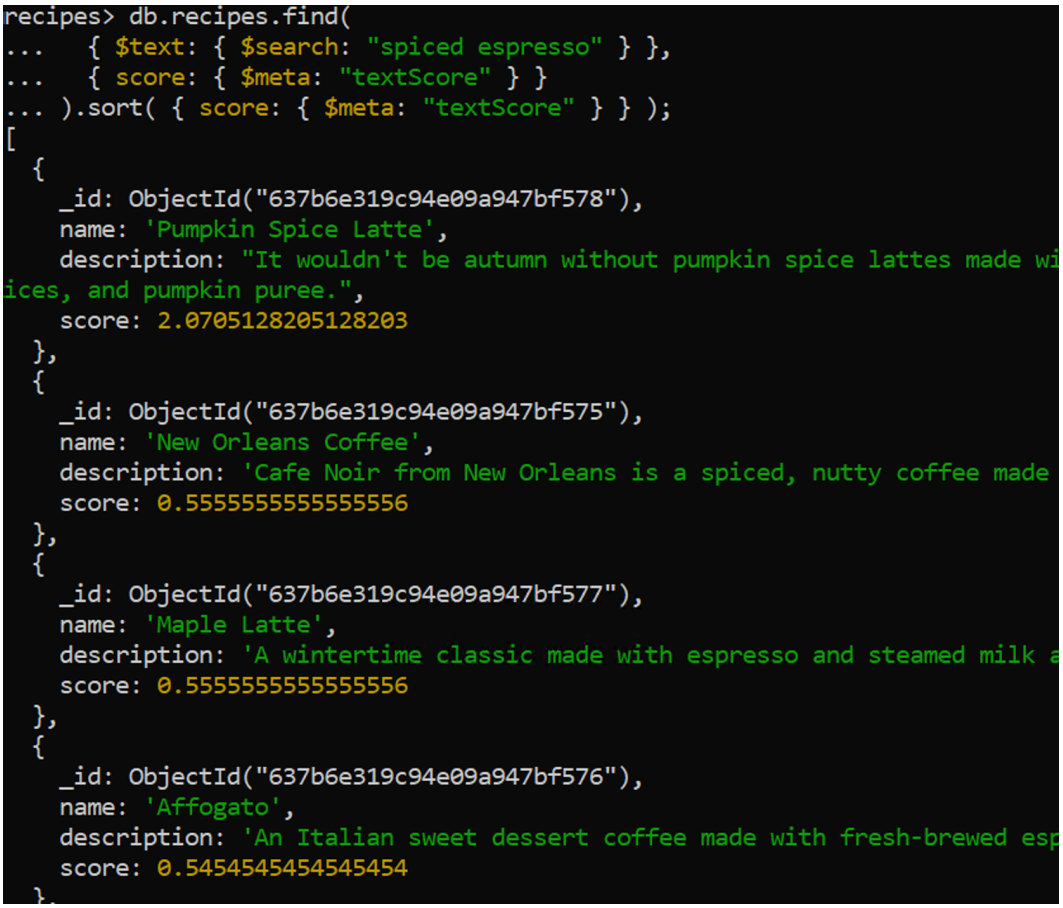
Exercise #6:
You can also exclude full phrases. Try searching for espressos without "ice cream". You need to escape the double quotes with backslashes.

When a query, especially a complex one, returns multiple results, some documents are likely to be a better match than others. For example, when you look for spiced espresso drinks, those that are both spiced and espresso-based are more fitting than those without spices or not using espresso as the base.

Full-text search engines typically assign a relevance score to the search results, indicating how well they match the search query. MongoDB also does this, but the **search relevance is not visible by default**. Moreover, it will return its results in unsorted order by default.

To sort the results in order of relevance score, you must explicitly project the metadata **textScore** field (by using the **$meta** operator) and sort on it:

```
db.recipes.find(
  { $text: { $search: "spiced espresso" } },
  { score: { $meta: "textScore" } }
).sort( { score: { $meta: "textScore" } } );
```



The score for Pumpkin Spice Latte is higher since it is the only coffee drink that contains both the words spiced and espresso.

Task 6. Implementing a database backup and restore

To create backup of databases in MongoDB, you should use the **mongodump** tool (included in **MongoDB Database Tools**). This command will dump the entire data of your server into the dump directory. There are many options available by which you can limit the amount of data or create backup of your remote server.

Syntax	**Description**	**Example**
mongodump	Backups all databases from local server in the **/bin/dump** directory (See example 1)	mongodump
mongodump --host HOST_NAME --port PORT_NUMBER	Backup all databases of specified MongoDB instance (See example 2)	mongodump --host utb.cz --port 27017
mongodump --db=DB_NAME --out=BACKUP_DIRECTORY	Backup only specified database at specified path (See example 3)	mongodump --db=company --out=--/data/backup/
mongodump --collection=COLLECTION --db=DB_NAME	Backup only specified collection of specified database (See example 4)	mongodump --collection=mycol --db=test

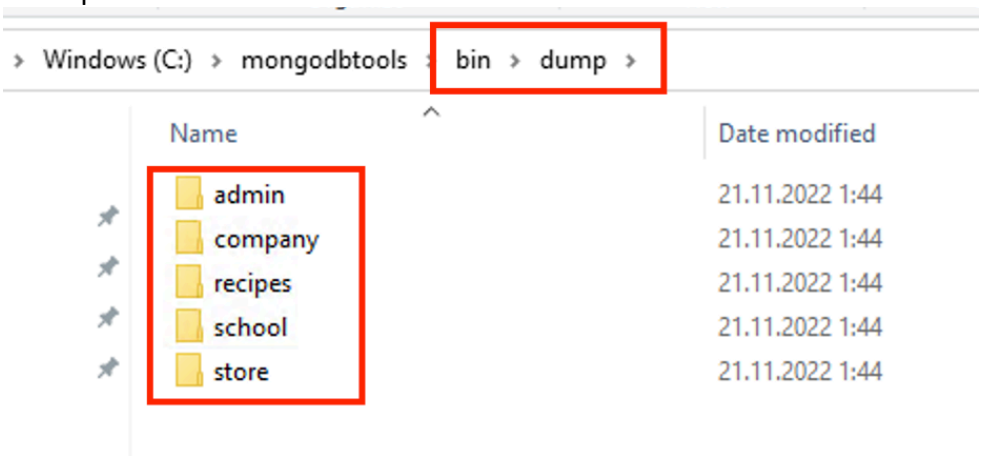
The **mongorestore** tool loads data from either a binary database dump created by mongodump or the standard input into a MongoDB instance.

Syntax	**Description**	**Example**
mongorestore	Restores all databases and collections from the default **/bin/dump** directory (See example 5)	mongorestore
mongorestore --host HOST_NAME --port PORT_NUMBER	Restores all databases to the specified MongoDB instance (See example 6)	mongorestore --host utb.cz --port 27017
mongorestore --nsInclude=DBNAME.COLLECTION --dir=BACKUP_DIRECTORY	Restore only specified collection from specified database using the specified dump directory (See example 7)	mongorestore --nsInclude=recipes.recipes

Example 1:

```
C:\mongodbtools\bin>mongodump
2022-11-21T01:44:01.673-1200    writing admin.system.version to dump\admin\system.version.bson
2022-11-21T01:44:01.677-1200    done dumping admin.system.version (1 document)
2022-11-21T01:44:01.680-1200    writing company.Customers to dump\company\Customers.bson
2022-11-21T01:44:01.681-1200    writing company.customers to dump\company\customers.bson
2022-11-21T01:44:01.681-1200    writing store.store to dump\store\store.bson
2022-11-21T01:44:01.682-1200    writing recipes.recipes to dump\recipes\recipes.bson
2022-11-21T01:44:01.703-1200    done dumping store.store (8 documents)
2022-11-21T01:44:01.711-1200    done dumping recipes.recipes (5 documents)
2022-11-21T01:44:01.911-1200    writing school.students to dump\school\students.bson
2022-11-21T01:44:01.915-1200    writing store.pizzas to dump\store\pizzas.bson
2022-11-21T01:44:02.050-1200    done dumping school.students (3 documents)
2022-11-21T01:44:02.079-1200    writing school.student to dump\school\student.bson
2022-11-21T01:44:02.117-1200    done dumping store.pizzas (3 documents)
2022-11-21T01:44:02.163-1200    writing school.professors to dump\school\professors.bson
2022-11-21T01:44:02.238-1200    done dumping school.student (1 document)
2022-11-21T01:44:02.259-1200    done dumping school.professors (1 document)
2022-11-21T01:44:02.288-1200    writing school.students4 to dump\school\students4.bson
2022-11-21T01:44:02.310-1200    done dumping school.students4 (0 documents)
2022-11-21T01:44:03.105-1200    done dumping company.Customers (70000 documents)
2022-11-21T01:44:03.141-1200    done dumping company.customers (70000 documents)
```

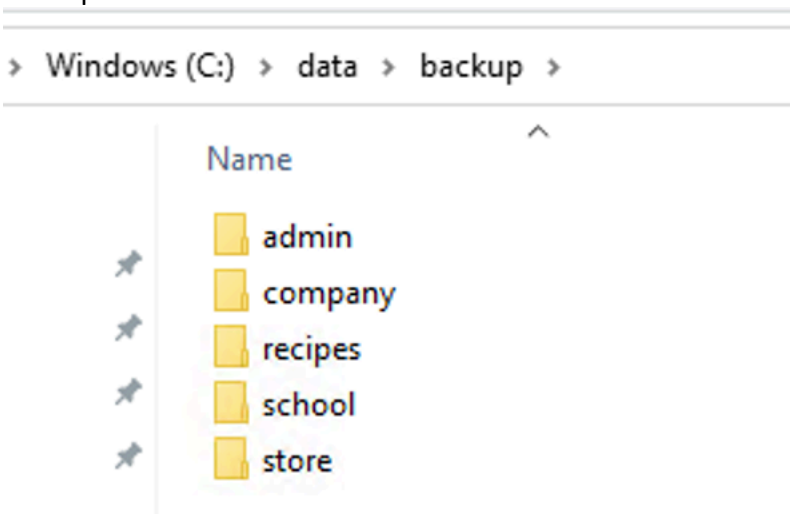
Backup location:



Example 2:

```
C:\mongodbtools\bin>mongodump --host 127.0.0.1 --port 27017 --out=/data/backup
2022-11-21T01:51:25.798-1200    writing admin.system.version to \data\backup\admin\system.version.bson
2022-11-21T01:51:25.806-1200    done dumping admin.system.version (1 document)
2022-11-21T01:51:25.812-1200    writing company.Customers to \data\backup\company\Customers.bson
2022-11-21T01:51:25.818-1200    writing company.customers to \data\backup\company\customers.bson
2022-11-21T01:51:25.820-1200    writing recipes.recipes to \data\backup\recipes\recipes.bson
2022-11-21T01:51:25.823-1200    writing store.store to \data\backup\store\store.bson
2022-11-21T01:51:25.871-1200    done dumping store.store (8 documents)
2022-11-21T01:51:25.901-1200    done dumping recipes.recipes (5 documents)
2022-11-21T01:51:25.943-1200    writing store.pizzas to \data\backup\store\pizzas.bson
2022-11-21T01:51:25.947-1200    writing school.students to \data\backup\school\students.bson
2022-11-21T01:51:25.971-1200    done dumping school.students (3 documents)
2022-11-21T01:51:25.972-1200    writing school.student to \data\backup\school\student.bson
2022-11-21T01:51:25.978-1200    done dumping store.pizzas (3 documents)
2022-11-21T01:51:25.979-1200    done dumping school.student (1 document)
2022-11-21T01:51:25.983-1200    writing school.professors to \data\backup\school\professors.bson
2022-11-21T01:51:26.004-1200    writing school.students4 to \data\backup\school\students4.bson
2022-11-21T01:51:26.012-1200    done dumping school.professors (1 document)
2022-11-21T01:51:26.037-1200    done dumping school.students4 (0 documents)
2022-11-21T01:51:26.997-1200    done dumping company.customers (70000 documents)
2022-11-21T01:51:27.002-1200    done dumping company.Customers (70000 documents)
```

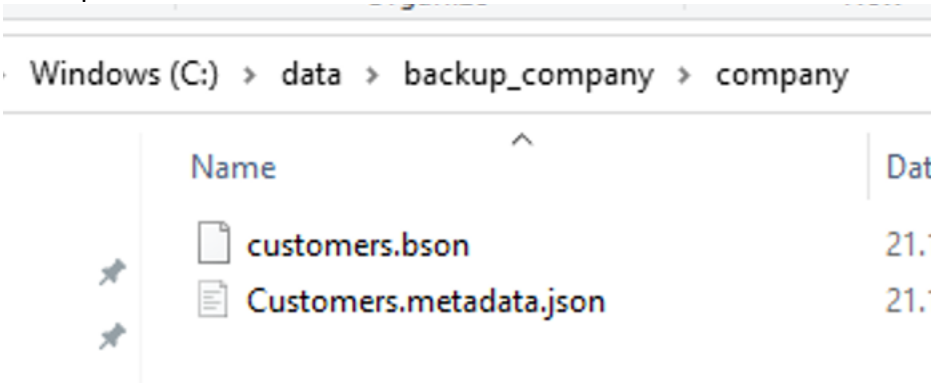
Backup:



Example 3:

```
C:\mongodbtools\bin>mongodump --db=company --out=/data/backup_company/
2022-11-21T01:56:52.044-1200    writing company.customers to \data\backup_company\company\customers.bson
2022-11-21T01:56:52.077-1200    writing company.Customers to \data\backup_company\company\Customers.bson
2022-11-21T01:56:53.158-1200    done dumping company.customers (70000 documents)
2022-11-21T01:56:53.161-1200    done dumping company.Customers (70000 documents)
```

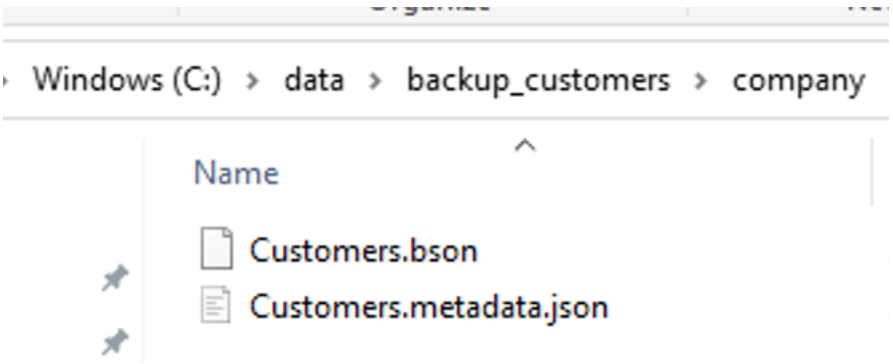
Backup:



Example 4:

```
C:\mongodbtools\bin>mongodump --db=company --collection=Customers --out=/data/backup_customers/
2022-11-21T01:59:05.858-1200    writing company.Customers to \data\backup_customers\company\Customers.bson
2022-11-21T01:59:06.943-1200    done dumping company.Customers (70000 documents)
```

Backup:



Example 5:

```
C:\mongodbtools\bin>mongorestore
2022-11-21T02:35:33.827-1200    using default 'dump' directory
2022-11-21T02:35:33.829-1200    preparing collections to restore from
2022-11-21T02:35:33.833-1200    reading metadata for store.pizzas from dump\store\pizzas.metadata.json
2022-11-21T02:35:33.834-1200    reading metadata for store.store from dump\store\store.metadata.json
2022-11-21T02:35:33.834-1200    reading metadata for company.Customers from dump\company\Customers.metadata.json
2022-11-21T02:35:33.834-1200    reading metadata for recipes.recipes from dump\recipes\recipes.metadata.json
2022-11-21T02:35:33.835-1200    reading metadata for school.professors from dump\school\professors.metadata.json
2022-11-21T02:35:33.835-1200    reading metadata for school.student from dump\school\student.metadata.json
2022-11-21T02:35:33.836-1200    reading metadata for school.students from dump\school\students.metadata.json
2022-11-21T02:35:33.836-1200    reading metadata for school.students4 from dump\school\students4.metadata.json
2022-11-21T02:35:33.837-1200    restoring to existing collection company.Customers without dropping
2022-11-21T02:35:33.839-1200    restoring company.Customers from dump\company\Customers.bson
2022-11-21T02:35:33.842-1200    restoring to existing collection store.store without dropping
2022-11-21T02:35:33.843-1200    restoring store.store from dump\store\store.bson
2022-11-21T02:35:33.847-1200    restoring to existing collection recipes.recipes without dropping
2022-11-21T02:35:33.850-1200    restoring recipes.recipes from dump\recipes\recipes.bson
2022-11-21T02:35:33.851-1200    restoring to existing collection school.students without dropping
2022-11-21T02:35:33.887-1200    finished restoring company.Customers (0 documents, 0 failures)
2022-11-21T02:35:33.904-1200    terminating read on recipes.recipes
2022-11-21T02:35:33.909-1200    restoring school.students from dump\school\students.bson
2022-11-21T02:35:33.909-1200    Failed: company.Customers: error restoring from dump\company\Customers.bson: cann
ot transform type bson.Raw to a BSON Document: not enough bytes available to read type. bytes=901 type=invalid
2022-11-21T02:35:33.912-1200    0 document(s) restored successfully. 0 document(s) failed to restore.
2022-11-21T02:35:33.914-1200    finished restoring recipes.recipes (0 documents, 0 failures)
```

Example 6:

```
C:\mongodbtools\bin>mongorestore --host 127.0.0.1 --port 27017
2022-11-21T02:54:38.877-1200    using default 'dump' directory
2022-11-21T02:54:38.878-1200    preparing collections to restore from
2022-11-21T02:54:38.884-1200    reading metadata for school.professors from dump\school\professors.metadata.json
2022-11-21T02:54:38.886-1200    reading metadata for school.student from dump\school\student.metadata.json
2022-11-21T02:54:38.887-1200    reading metadata for school.students from dump\school\students.metadata.json
2022-11-21T02:54:38.888-1200    reading metadata for school.students4 from dump\school\students4.metadata.json
2022-11-21T02:54:38.889-1200    reading metadata for store.pizzas from dump\store\pizzas.metadata.json
2022-11-21T02:54:38.889-1200    reading metadata for store.store from dump\store\store.metadata.json
2022-11-21T02:54:38.890-1200    reading metadata for company.Customers from dump\company\Customers.metadata.json
2022-11-21T02:54:38.891-1200    reading metadata for recipes.recipes from dump\recipes\recipes.metadata.json
2022-11-21T02:54:38.893-1200    restoring to existing collection company.Customers without dropping
2022-11-21T02:54:38.894-1200    restoring company.Customers from dump\company\Customers.bson
2022-11-21T02:54:38.897-1200    restoring to existing collection store.store without dropping
2022-11-21T02:54:38.906-1200    restoring to existing collection recipes.recipes without dropping
2022-11-21T02:54:38.907-1200    restoring recipes.recipes from dump\recipes\recipes.bson
2022-11-21T02:54:38.907-1200    restoring to existing collection school.students without dropping
2022-11-21T02:54:38.909-1200    restoring store.store from dump\store\store.bson
```

Example 7:
First delete the recipes database:

```
test> use recipes
switched to db recipes
recipes> db.dropDatabase()
{ ok: 1, dropped: 'recipes' }
```

Now let's restore it:

```
C:\mongodbtools\bin>mongorestore --nsInclude=recipes.recipes
2022-11-21T03:04:31.106-1200    using default 'dump' directory
2022-11-21T03:04:31.107-1200    preparing collections to restore from
2022-11-21T03:04:31.114-1200    reading metadata for recipes.recipes from dump\recipes\recipes.metadata.json
2022-11-21T03:04:31.143-1200    restoring recipes.recipes from dump\recipes\recipes.bson
2022-11-21T03:04:31.163-1200    finished restoring recipes.recipes (5 documents, 0 failures)
2022-11-21T03:04:31.163-1200    restoring indexes for collection recipes.recipes from metadata
2022-11-21T03:04:31.165-1200    index: &idx.IndexDocument{Options:primitive.M{"default_language":"english", "lang
uage_override":"language", "name":"name_text_description_text", "textIndexVersion":3, "v":2, "weights":primitive.
M{"description":1, "name":1}}, Key:primitive.D{primitive.E{Key:"_fts", Value:"text"}, primitive.E{Key:"_ftsx", Va
lue:1}}, PartialFilterExpression:primitive.D(nil)}
2022-11-21T03:04:31.199-1200    5 document(s) restored successfully. 0 document(s) failed to restore.
```

See the data

```
recipes> db.recipes.find()
[
  {
    _id: ObjectId("637b6e319c94e09a947bf574"),
    name: 'Cafecito',
    description: 'A sweet and rich Cuban hot coffee made by to
m.'
  },
  {
    _id: ObjectId("637b6e319c94e09a947bf575"),
    name: 'New Orleans Coffee',
    description: 'Cafe Noir from New Orleans is a spiced, nutt
  },
```