#### Tutorial: MongoDB

If you want to use GUI, download MongoDB Compass to interact with mongodb as follow. Here is a way

<https://www.mongodb.com/try/download/compass>

After downloading MongoDB Compass successfully, boot up. Mongodb do not require you to enter username and password by default

After connecting to MongoDB successfully, you should see the following screen

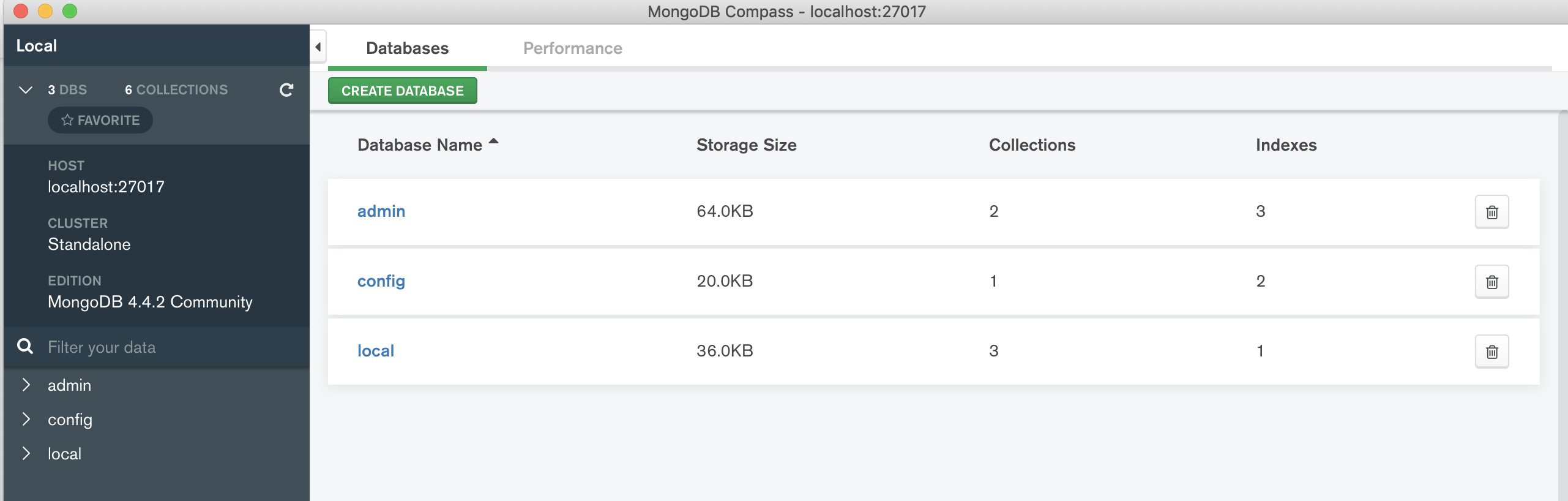


Figure : MongoDB screen after connecting successfully.

Welcome to the Mongo Workbook

This little book accompanies the Mongo course taught over 2 days. Each exercise builds on the last, so it's best to work through these exercises in order.

We'll start off with the helper functions, find, count and distinct, then move into the aggregate pipeline and on to map reduce.

Welcome to MongoDB

Mongo DB is a highly scalable, NoSQL, schema free data store.

Here's why it's great

* It stores data as JSON so you can use the same data client side and server side. This means you write almost no wiring code, everything just works.
* Flexible Schema. If your requirements change, you can adapt.
* Unstructured data - you can store and retrieve unstructured data easily. It's just JSON. Not every document in a collection needs the same fields.
* Denormalized data - Group related content in a single document.
* Clean and simple API - Mongo is nice to talk to.

Here's why you might not like it

* Denormalized data means no joins. If your data is highly relational, Mongo is not your baby. Your data is organized into collections. If you need data from more than collection, you need to hit the database more than once.
* Flexible schema means no built-in data validation. Your data is validated at the application tier. The database is dumb and will store whatever your application gives it, even junk and typos.
* Bugs - Mongo is new and there are still issues in the tracker. Not bad bugs, but occasionally things don't work as you might expect.
* No transactions - A SQL database allows you to bundle multiple writes into a transaction. If one write fails the whole transaction is rolled back. Mongo lacks this feature, writes are small and atomic. If you need a transaction you must build it yourself.
* Theoretical data loss - Mongo scales using a technique called sharding. It creates slaves that mirror data written to the master. If the master goes down before data is mirrored you may lose recent commits depending on your settings.

When you should use it

Mongo represents data as a tree. If your data is tree shaped, or can be made tree shaped, Mongo is great. If your data is a web or a network which can't be flattened out, you likely have relational data, and Mongo is perhaps not for you this time.

If you have unstructured data to store which can be represented as a series of nested lists Mongo will make your life more enjoyable.

Say you have a webpage full of widgets, and each of those widgets can contain arbitrary information. This is a semi-structured tree and Mongo might be a very good choice.

If you have big customer data to store, and each customer record contains lists of communications, subscriptions, etc, the data is tree shaped, and Mongo would again be a good chice.

If you have big data and you want to query it in interesting and complex ways, pulling useful aggregated data out the other side in suprisingly short timeframes, Mongo is perfect.

On the other hand, if your data looks like a web: comments, purchases, kittens, customers, sharks, exploding hats, etc, all linking to each other in a web, then you have relational data, and you may wish to stick with a relational database like Postgres, MySQL or MS SQL Server.

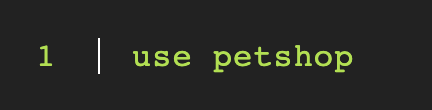
Why is it fast?

Mongo manages to be so fast because it does less. There's no magical difference in the architecture that makes it fast, it just has a simplified streamlined query language that is easier to optimise.

|  |
| --- |
| Exercise - connect to a console  Connect to the console using MongoDB Compass. Try typing some JavaScript expressions.   * Tell me how many seconds there are in a week   Start by writing calculation statement as follow |

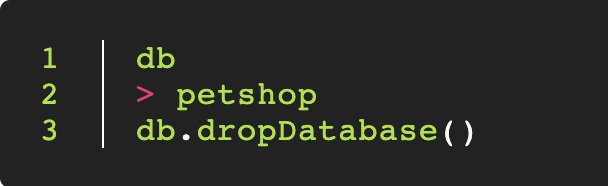
Creating a database

We can switch to a database (even if it’s not created) in Mongo with the use command.



This will switch to writing to the **petshop** database. It doesn't matter if the database doesn't exist yet. It will be brought into existence when you first write a document to it.

You can find which database you are using simply by typing db. You can drop the current database and everything in it using db.dropDatabase.



|  |
| --- |
| Exercise - Create a database   * Use the use command to connect to a new database (If it doesn't exist, Mongo will create it when you write to it).   That was easy wasn't it. Don't worry, it gets a bit harder. |

Collections

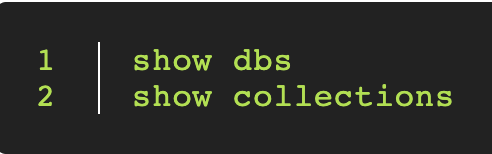
Collections are sets of (usually) related documents. Your database can have as many collections as you like.

Because Mongo has no joins, a Mongo query can pull data from only one collection at a time. You will need to take this into account when deciding how to arrange your data.

You can create a collection using the createCollection command.

Collections will also be created automatically. If you write a document to a collection that doesn't exist that collection will be brought into being for you.

View your databases and collections using the show command, like this:



|  |
| --- |
| Exercise - Create a collection   * Use db.createCollection to create a collection. I'll leave the subject up to you. * Run show dbs and show collections to view your database and collections. |

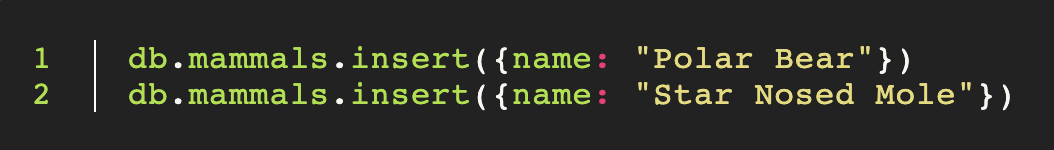
Documents

Documents are JSON objects that live inside a collection. They can be any valid JSON format, with the caveat that they can't contain functions.

The size limit for a document is 16Mb which is more than ample for most use cases.

**Creating a document**

You can create a document by inserting it into a collection

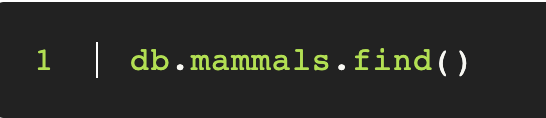


|  |
| --- |
| Exercise - Create some documents   * Insert a couple of documents into your collection. I'll leave the subject matter up to you, perhaps cars or hats. |

**Finding a document**

You can find a document or documents matching a particular pattern using the find function.

If you want to find all the mammals in the mammals collection, you can do this easily.



|  |
| --- |
| Exercise - documents   * Use find() to list documents created above. |

Finding documents

Mongo comes with a set of convenience functions for performing common operations. Find() is one such function. It allows us to find documents by providing a partial match, or an expression.

Uses

You **can** use find to:

* Find a document by id, using find({\_id: ObjectId(‘id’)})
* Find a user by email
* Find a list of all users with the same first name
* Find all cats who are more than 12 years old
* Find all gerbils called 'Herbie' who are bald, have three or more eyes, and who have exactly 3 legs.

Limitations

You **can't** use find to chain complex operators. You can do a few simple things like counting, but if you want to real power you need the *aggregate pipeline*, which is actually not at all scary and is quite easy to use.

The Aggregate pipeline allows us to chain operations together and pipe a set of documents from one operation to the next.

Using find() example

You can use find with no arguments to list documents in a collection.



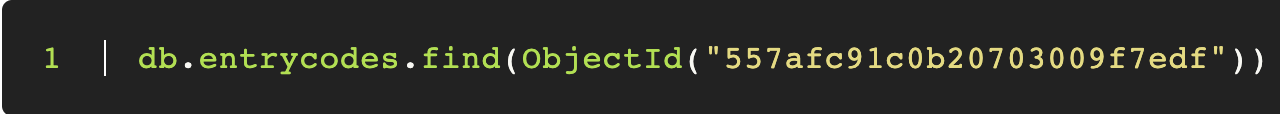
This will list all of the codes, 20 at a time and return a cursor to loop things

You can get the same result by passing an empty object, like so:



Finding by ID

Assuming you know the object ID of a document. You can pull that document by id like so:

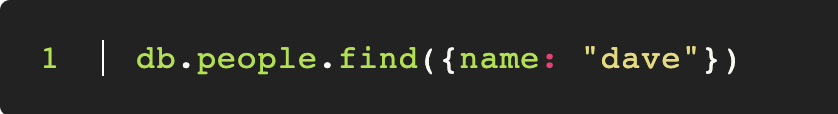


The \_id field of any collection is automatically indexed.

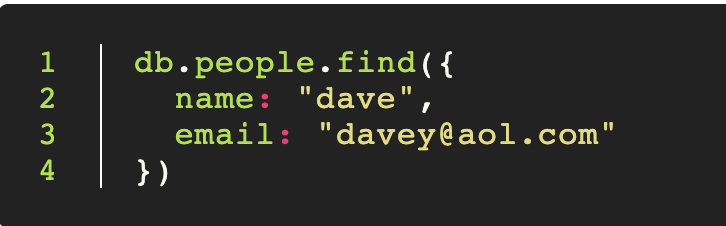
IDs are 12 byte BSON objects, not Strings which is why we need the ObjectId function. If you want to read more on [ObjectId, you can do so here.](http://docs.mongodb.org/manual/reference/object-id/)

Finding by partial match

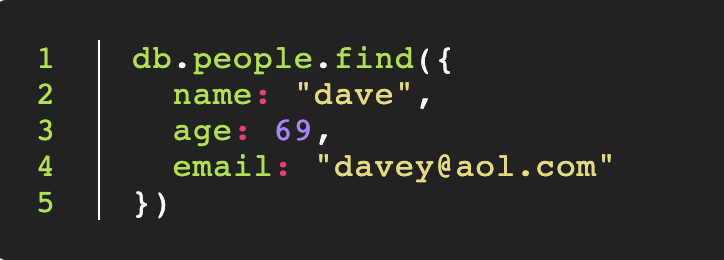
Say you have a list of users and you want to find by name, you might do:



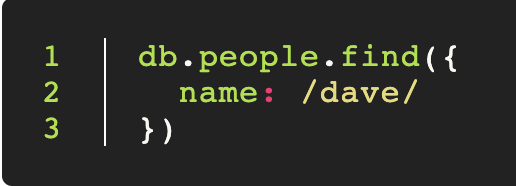
You can match on more than one field:



You can match on numbers:



You also match using a regex (although be aware this is slow on large data sets):

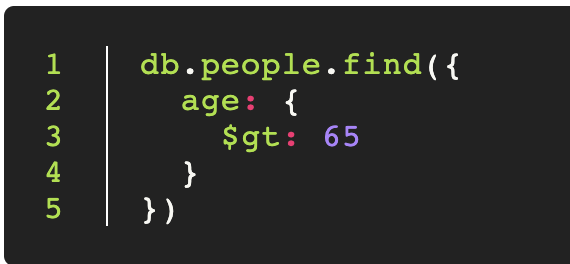


|  |
| --- |
| Exercise  We need to start out by inserting some data which we can work with.   * Type the following into your terminal to create a petshop with some pets in it     Gist: <https://gist.github.com/nguyenhuucam91/d08d481fdc66eb9be60dced75ab94368>   * Add another piranha with your chosen name, and a naked mole rat called Henry. * Use find to list all the pets. Find the ID of Mikey the Gerbil.     Specify {\_id: 1} allows to retrieve only \_id on MongoDB, this is called projection   * Use find() to find Mikey by id. * Use find() to find all the gerbils. * Find all the creatures named Mikey. * Find all the creatures named Mikey who are gerbils. * Find all the creatures with the string ending with "dog" in their species, use Regex to find species ending with “dog” (/dog$/) |

Finding with Expressions and comparison queries

We have seen how we can find elements by passing Mongo a partial match, like so:

We can also find using expressions. We define these using JSON, like so:



We can use operators like this:

* $gt - Greater than
* $lt - Less than
* $gte – Greater than or equal to
* $lte – Less than or equal to
* $exists - The field exists

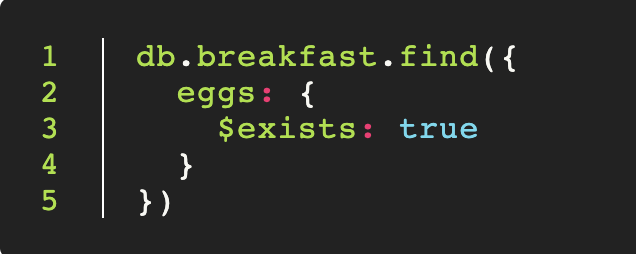
See the full list here:

<http://docs.mongodb.org/manual/reference/operator/query/>

|  |
| --- |
| Exercise  Copy the following code into a Mongo terminal, this code is available on Gist file below. It will create a collection of people, some of whom will have cats.    Gist: <https://gist.github.com/nguyenhuucam91/c36ba6f6b01b22c572c42d4566d8ad41>  Run 2 separate commands use and function(). After that, run db.people.find({}) to check whether your collection has records or not.   * Use find to get all the people who are exactly 99 years old * Find all the people who are eligible for a bus pass (people older than 65) * Find all the teenagers, greater than 12 and less than 20. |

$exists

We can use exists to filter on the existence of non-existence of a field. We might find all the breakfasts with eggs:



|  |
| --- |
| Exercise - $exists   * Find all the people with cats. * Find all the pensioners whose age > 60 with cats * Find all the teenagers whose age<15 with cats |

$where

We can even filter using an arbitrary JavaScript expression using $where. This will allow us to compare two fields in a single document.



Here we find all the sandwiches with jam and peanut butter where the jam quotient outweighs the peanut butter.

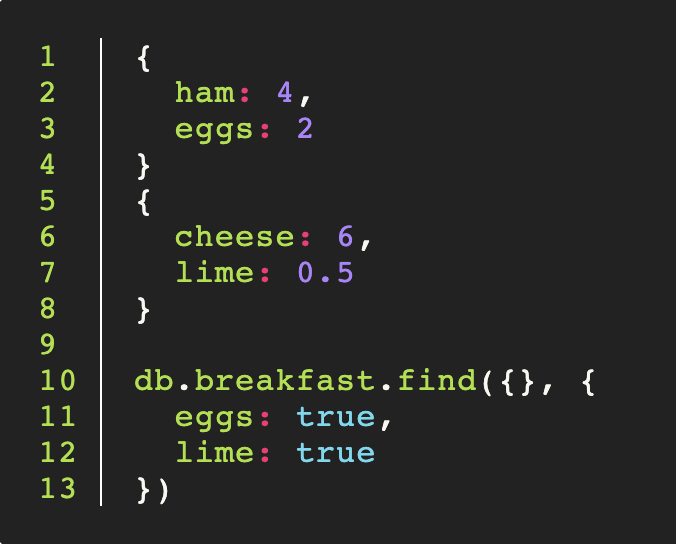
*Warning: It's easy to overuse $where since it appears to do everything with plain old JavaScript. $where is eval-ing a JavaScript expression and as such is slow. Mongo can make no optimizations here, and must execute the JavaScript on every single document in the collection. Prefer the native operators where possible.*

|  |
| --- |
| Exercise - $where   * Use $where to find all the people who have a cat. * Find all the people who are younger than their cats. Remember, not everyone has a cat, so you will need to use a boolean && to filter out the non-cat owners. Note that you can use $expr in Mongo to do this, look up in the doc in MongoDB website * Does anyone have the same name as their cat? Re-run the insertion script to create more records until someone does. |

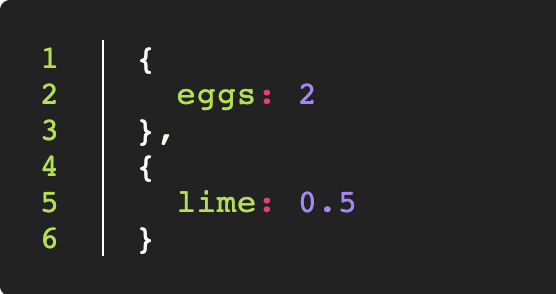
Projection

Find takes a second parameter which allows you to whitelist fields to pass into the output document. We call this projection.

You can choose fields to pass though, like so:



This will yield

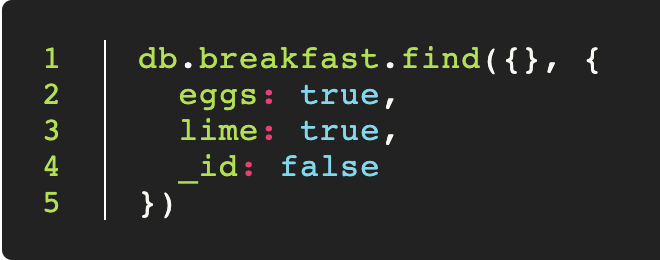


|  |
| --- |
| Exercise - Tidy up your output   * Use projection to format your array of people. We want only the names. * Output just the names of the people who are 99 years old * Output only the cats to create output like this:     *When you output the cats, you will need to find only people who have cats, where cats $exists, or you will have gaps in your data* |

Excluding the id field

You will notice that the ID field is always passed through project by default. This is often desirable, but you may wish to hide it, perhaps to conceal your implementation, or to keep your communication over the wire tight.

You can do this easily by passing \_id: false:



|  |
| --- |
| Exercise - remove the ids   * List the cats. Remove the ids from the output from question above |

Count, Limit, Skip & Sort

We can chain some additional functions onto our find query to modify the output.

Count

Count will convert our result set into a number. We can use it in two ways. We can either chain it:

or we can use it in place of find:



To count the people who have exactly three sharks.

Don't confuse it with length(). Length will convert to an array, then count the length of that array. This is inefficient.

|  |
| --- |
| Exercise - count the people   * Find out how many people there are in total. * Using your collection of people, and $exists, tell me how many people have cats. * Use $where to count how many people have cats which are older than them. |

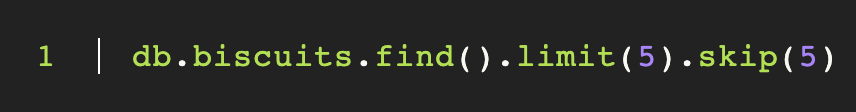
Limit and Skip

Limit will allow us to limit the results in the output set. Skip will allow us to offset the start. Between them they give us pagination.

For example



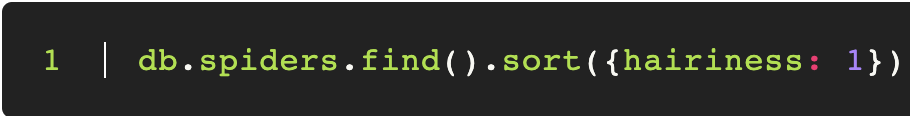
will give us the first 5 biscuits. If we want the next 5 we can skip the first 5.

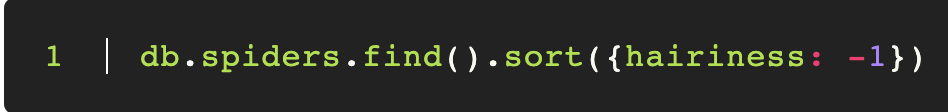


|  |
| --- |
| Exercise - Limit the people   * Give me the first 5 people * Give me the next 5 people * Give me the names and ages of the 5 teenagers with cats, where the cats have the word "Yolanda" in their name. |

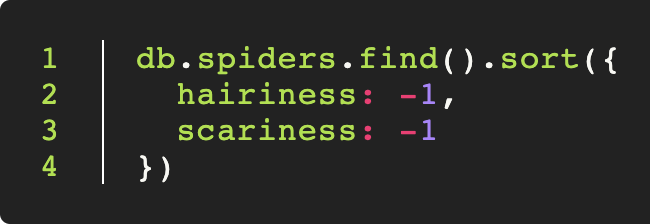
Sort

We can sort the results using the sort operator, like so:

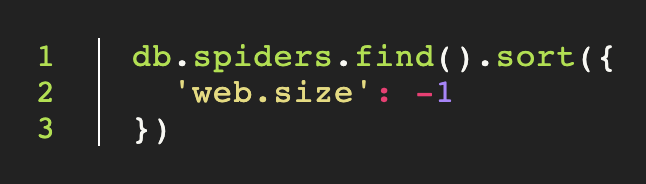
This will sort the spiders in ascending order of hairiness. You can reverse the sort by passing -1.

This will get the most hairy spiders first.

We can sort by more than one field:



We might also sort by nested fields:



will give the spiders with the largest webs.

|  |
| --- |
| Exercise - Order the people   * Give me the 5 oldest cats * Give me the next 5 oldest cats |

Inserting, Updating & Deleting

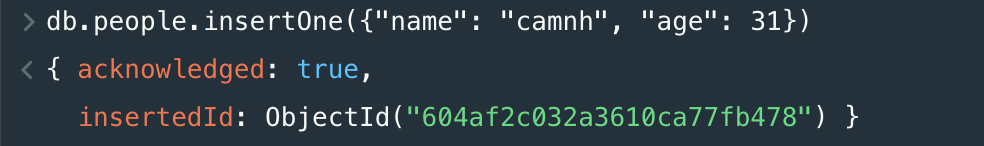
CRUD is a basic function of any database. Crud stands for:

* Create
* Read
* Update
* Delete

The four basic things that any data store needs to give us.

Creating

We create using the insertOne() command, like this:



The JSON object will be created and saved.

|  |
| --- |
| Exercise - Create a document  Refresh your muscle memory. Create a new person now. Ensure that person has a shark. |

Reading

We have many options for finding. We have already seen db.collection.find(). We can also use db.collection.findOne() which will return at most one result.

As we shall see soon, we also have the aggregate framework, and if we need maximum flexibility at the expense of a good deal of speed, we can also use map-reduce.

|  |
| --- |
| Exercise - Find the shark   * Refresh your muscle memory. Find the person who has a shark. * Use findOne() instead of find(). What will you get. |

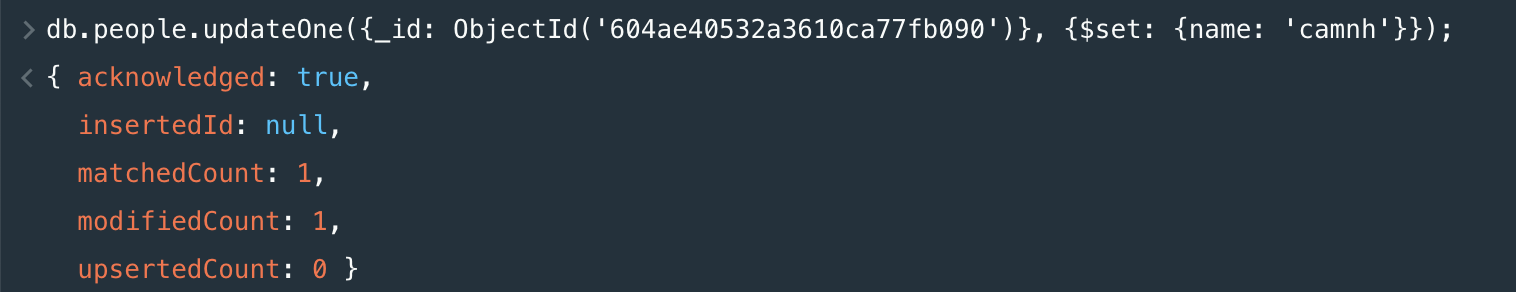
Updating

We save using the db.collection.updateOne function. We pass the function a JSON object that contains the modified object to save, including the \_id. The item will be found and updated.

Formula:

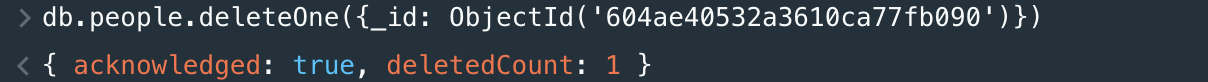
db.<collection\_name>.updateOne({wheresObject}, {$set: {fields\_to\_update}})

For example:



Deleting

We can remove people by using deleteOne() method

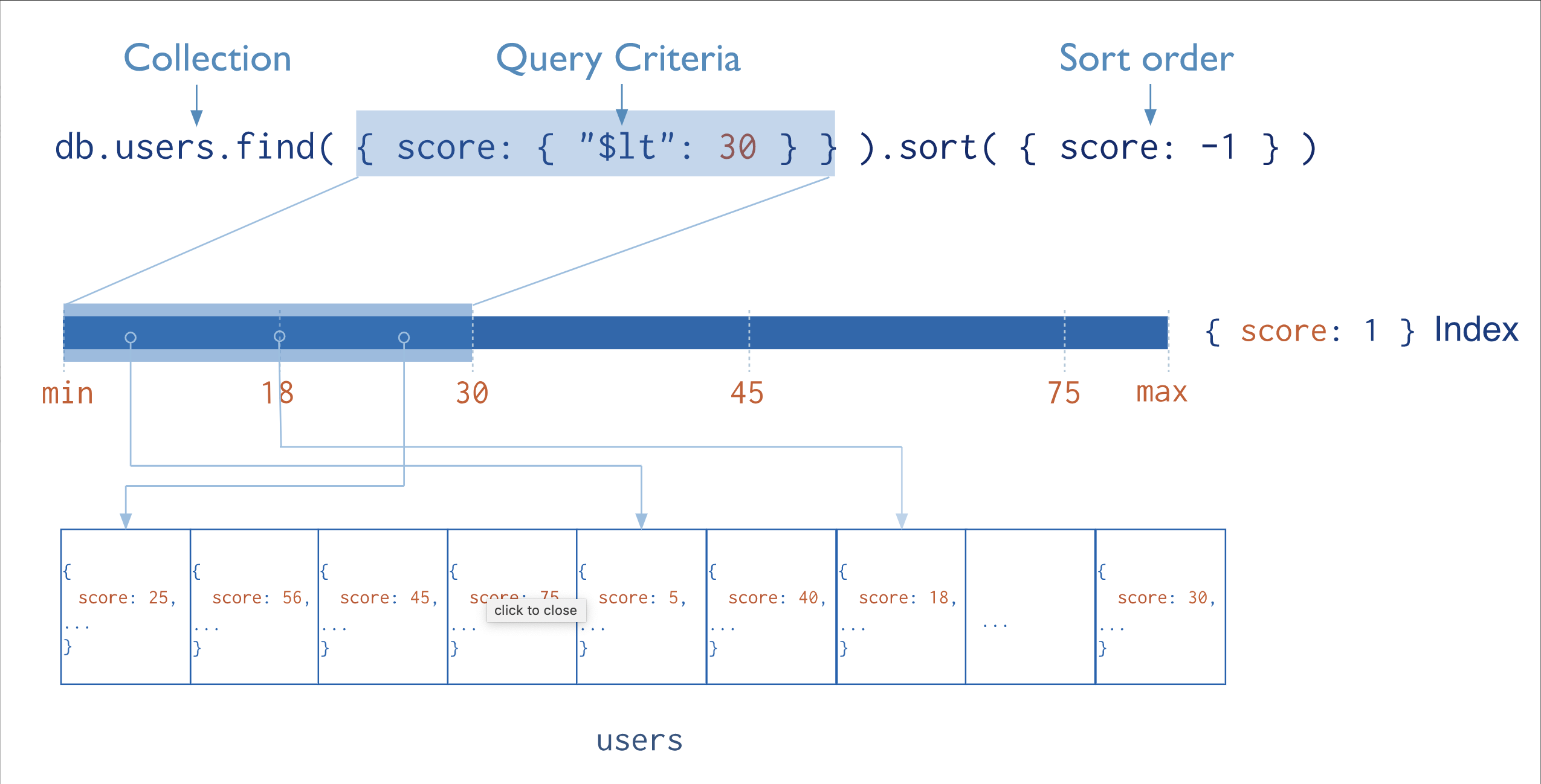


|  |
| --- |
| Exercise - remove all the people.   * It's time for a cull. Delete all the 50 years old people using deleteMany() method. The use of deleteMany() is similar to deleteOne() * We also heard there was some guy running around with a shark. That's a dangerous animal. Take him out, in fact take out anyone with a shark. |

References for reading: Indexing

Indexes support the efficient execution of queries in MongoDB. Without indexes, MongoDB must perform a collection scan, i.e. 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.

The following diagram illustrates a query that selects and orders the matching documents using an index:



Fundamentally, indexes in MongoDB are similar to indexes in other database systems. MongoDB defines indexes at the [collection](https://docs.mongodb.com/manual/reference/glossary/#term-collection) level and supports indexes on any field or sub-field of the documents in a MongoDB collection.

To create an index, use [db.collection.createIndex()](https://docs.mongodb.com/manual/reference/method/db.collection.createIndex/" \l "db.collection.createIndex" \o "db.collection.createIndex()).

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

The following example creates a single key descending index on the name field:

db.collection.createIndex( { name: -1 } )

The [db.collection.createIndex](https://docs.mongodb.com/manual/reference/method/db.collection.createIndex/" \l "db.collection.createIndex" \o "db.collection.createIndex) method only creates an index if an index of the same specification does not already exist.

### Index Names

The default name for an index is the concatenation of the indexed keys and each key’s direction in the index ( i.e. 1 or -1) using underscores as a separator. For example, an index created on { item : 1, quantity: -1 } has the name item\_1\_quantity\_-1.

You can create indexes with a custom name, such as one that is more human-readable than the default. For example, consider an application that frequently queries the products collection to populate data on existing inventory. The following [createIndex()](https://docs.mongodb.com/manual/reference/method/db.collection.createIndex/" \l "db.collection.createIndex" \o "db.collection.createIndex()) method creates an index on item and quantity named query for inventory

db.products.createIndex(

{ item: 1, quantity: -1 } ,

{ name: "query for inventory" }

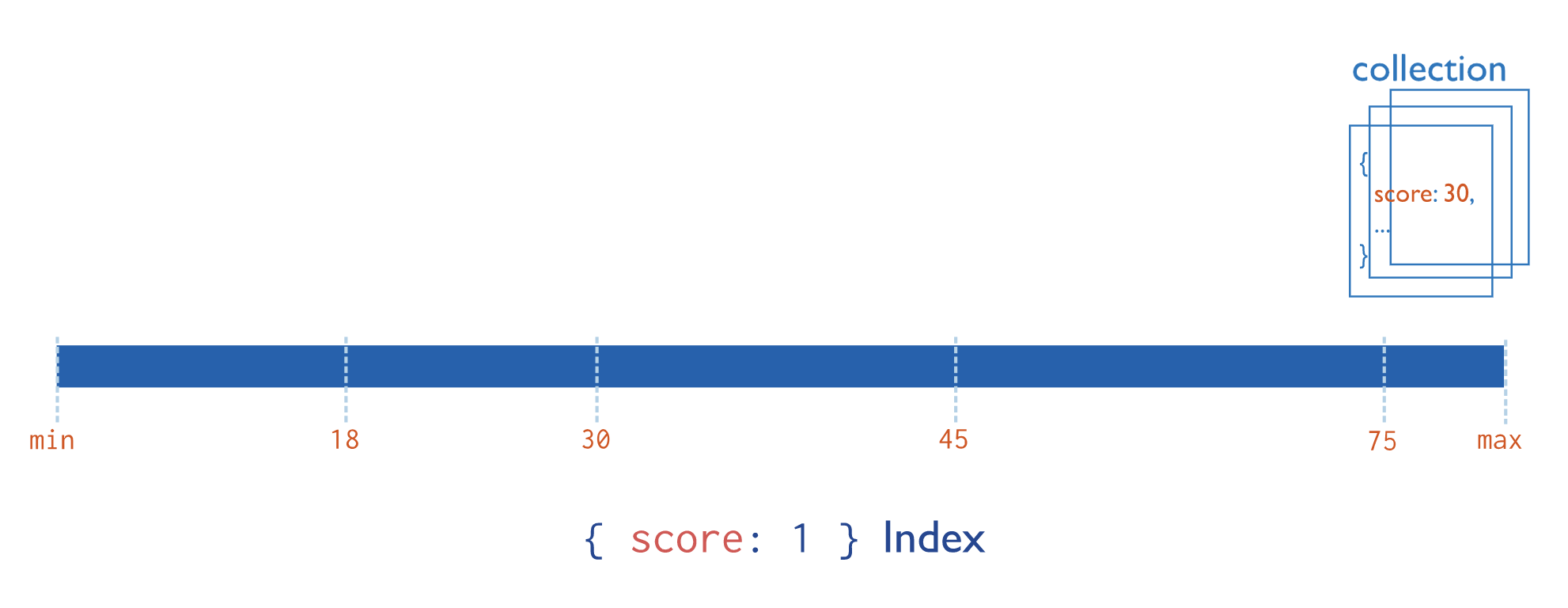
)

You can view index names using the [db.collection.getIndexes()](https://docs.mongodb.com/manual/reference/method/db.collection.getIndexes/" \l "db.collection.getIndexes" \o "db.collection.getIndexes()) method. You cannot rename an index once created. Instead, you must drop and re-create the index with a new name.

**Index types:**

### Single Field

In addition to the MongoDB-defined \_id index, MongoDB supports the creation of user-defined ascending/descending indexes on a [single field of a document](https://docs.mongodb.com/manual/core/index-single/).

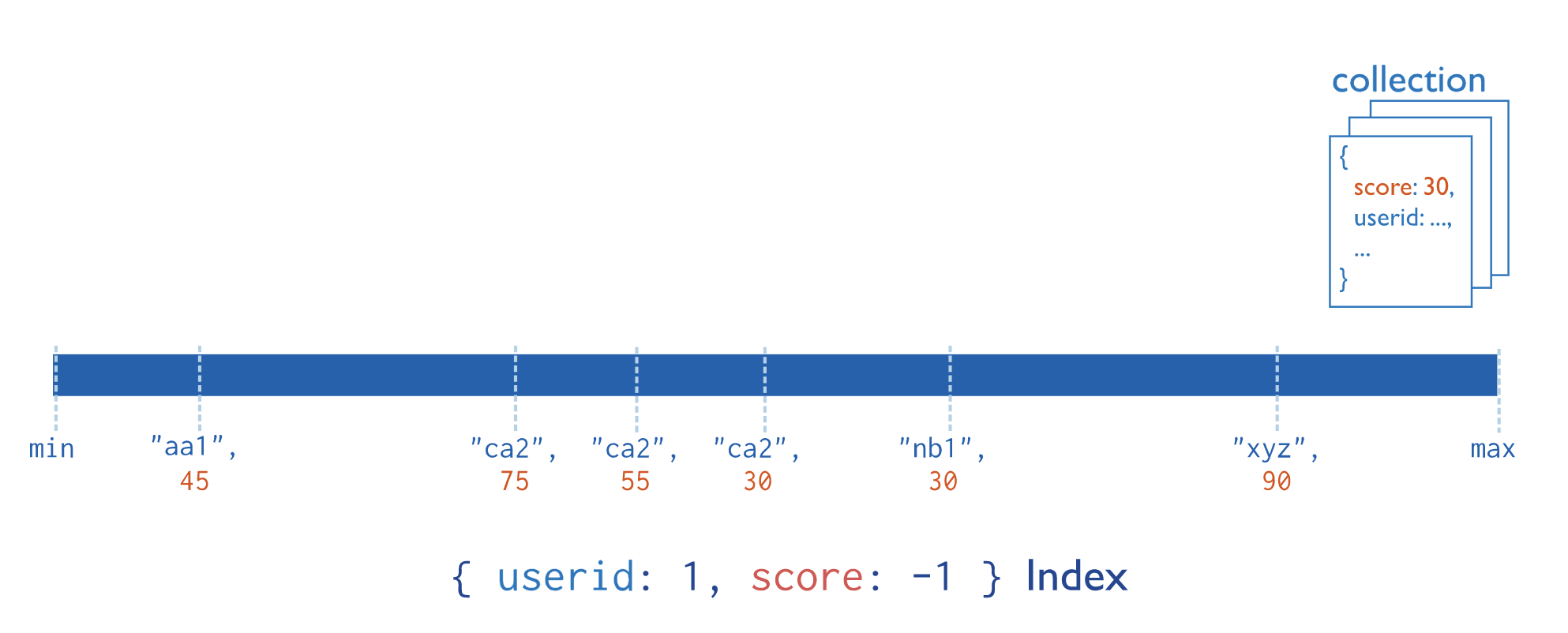


For a single-field index and sort operations, the sort order (i.e. ascending or descending) of the index key does not matter because MongoDB can traverse the index in either direction.

### Compound index

MongoDB also supports user-defined indexes on multiple fields, i.e. [compound indexes](https://docs.mongodb.com/manual/core/index-compound/).

The order of fields listed in a compound index has significance. For instance, if a compound index consists of { userid: 1, score: -1 }, the index sorts first by userid and then, within each userid value, sorts by score.



For compound indexes and sort operations, the sort order (i.e. ascending or descending) of the index keys can determine whether the index can support a sort operation.

**Multikey Index**

MongoDB uses [multikey indexes](https://docs.mongodb.com/manual/core/index-multikey/) to index the content stored in arrays. If you index a field that holds an array value, MongoDB creates separate index entries for every element of the array. These [multikey indexes](https://docs.mongodb.com/manual/core/index-multikey/) allow queries to select documents that contain arrays by matching on element or elements of the arrays. MongoDB automatically determines whether to create a multikey index if the indexed field contains an array value; you do not need to explicitly specify the multikey type.

