mongoDB

{ name: mongo, type: DB }

mongoDB

MongoDB

The Complete Developer's Guide

What is MondoDB?

MongoBD is a database which is created by the company who is also called MongoDB. The name stems from the word “humongous”. This database is built to store a lot of data but also being able to work with the huge data efficiently. Ultimately, this is a database solution.

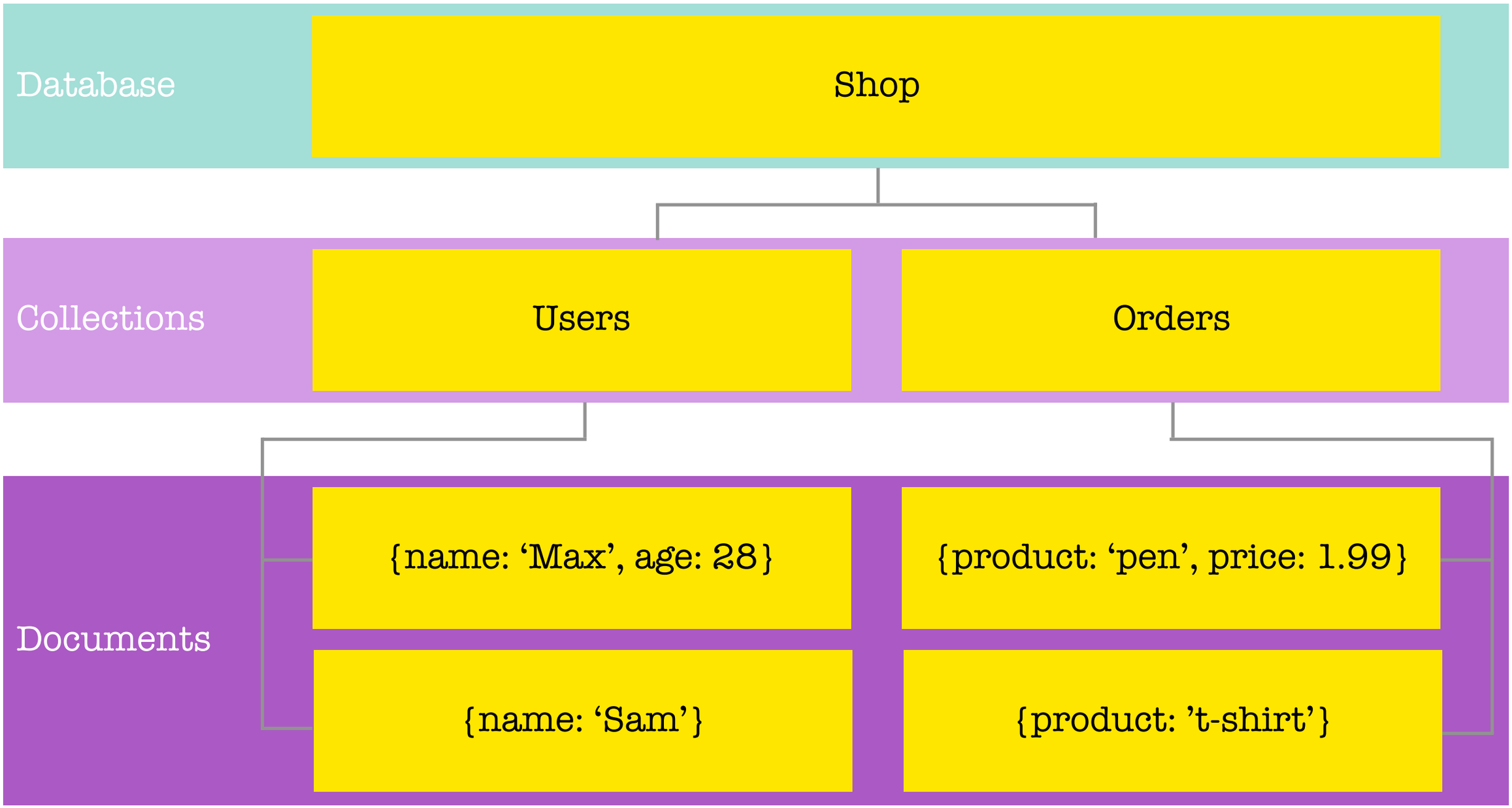
There are many database solutions such as mySQL, PostgresSQL, TSQL etc.

MongoDB is most importantly a database server that allows us to run different databases on it for example a Shop database. Within the database we would have different collections such as a Users collection or a Orders collection. We can have multiple databases and multiple collections per database.

Inside of the collection we have something called documents. Documents look like JavaScript JSON objects. Inside of a collection the documents are schema-less and can contain different data. This is the flexibility that MongoDB provides us with whereas SQL based database are very strict about the data stored within the database tables. Therefore, the MongoDB database can grow with the application needs. MongoDB is a No-SQL database.

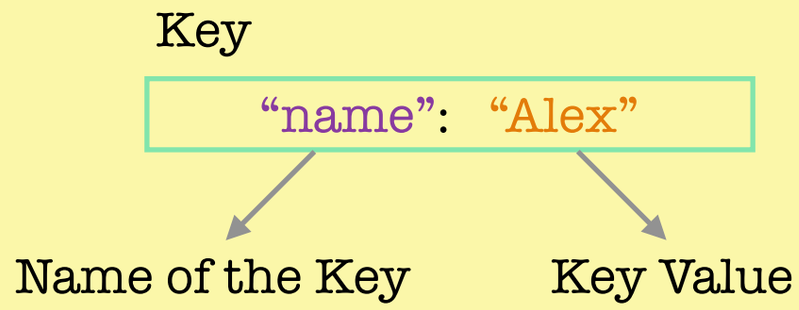
Typically we will need some kind of structure in a collection because applications typically requires some type of structure to work with the data.

Diagram 1.1:



JSON (BSON) Data Format:

```
{
    “name”: “Alex”,
    “age”: 29,
    “address”: {
        “city”: “Munich”
    },
    “hobbies”: [
        { “name”: “Cooking” },
        { “name”: “Football” }
    ]
}
```



The above is an example of the JSON data format. A single document is surrounded by curly brackets. The data is normally structured with a Keys. Keys consist of a Name of the Key and a Key value. The Name of the Key (*which will be referred to as Key from now on*) and the Key Value must be wrapped around quotation marks (unless if the data is a type of number).

There are different types of values we can store such as: string, number, booleans and arrays.

We can also nest documents within documents. This allows us to create complex relations between

data and store them within one document, which makes working with the data and fetching data more efficient because it is contained in one document in a logical way. SQL in contrast requires more complex method of fetching data which require joins to find data in table A and data in table B to retrieve the relevant data.

Behind the scenes on the server, MongoDB converts the JSON data to a binary version of the data which can be stored and queried more efficiently. We do not need to concern ourselves with BSON as we would tend to work with JSON data.

The whole theme of MongoDB is flexibility, optimisation and usability and it is what really sets MongoDB apart from other database solutions because it is so efficient from a performance perspective as we can query data in the format we need it instead of running complex restructuring on the server.

The Key MongoDB Characteristics.

MongoDB is a no SQL solution because it is following an opposite concept/philosophy to SQL based databases. Instead of normalising the data i.e. storing data distributed across multiple tables where every table has a clear schema and then using relations, MongoDB goes for storing data together in a document. It does not force a schema hence schema-less/No-SQL.

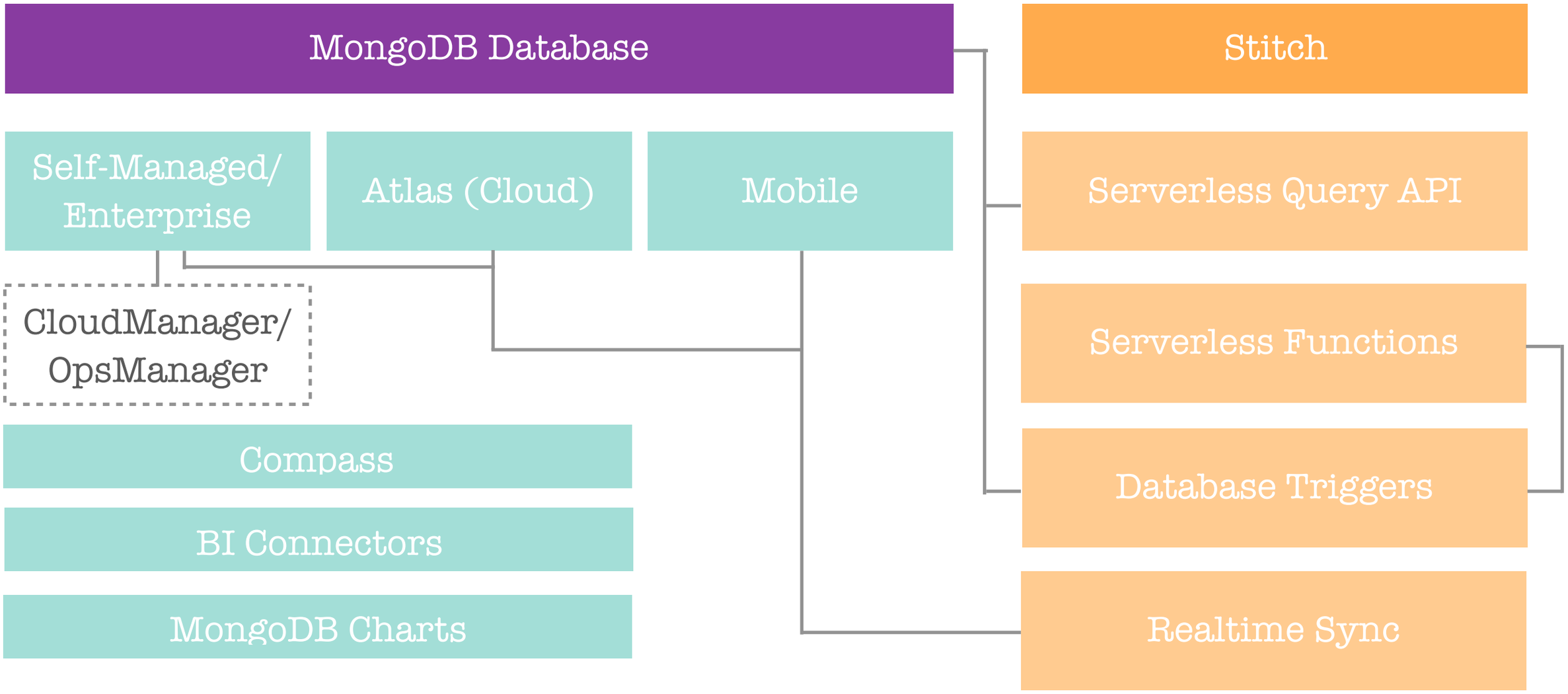
We can have multiple documents in a single collection and they can have different structures as we have seen in Diagram 1.1. This is important, it can lead to messy data but it still our responsibility as developers to work with clean data and to implement a solution that works. On the other hand this provides us with a lot of flexibility. We could use mongoDB for applications that might still evolve, where the exact data requirements are not set yet. MongoDB allows us to started and we could always add data with more information in the same collection at a later point in time.

We also work with less relations. There are some relations, but with these embedded (nested) documents, we have less collections (tables) which we connect but instead we store data together. This is where the efficiencies is derived from, since data is stored together and when we fetch data from our application it does not require to reach out to multiple tables and merge the data because all the data is already within the single collection. This is where the speed, performance and flexibility comes from and can be seen beneficial for when building applications. This is the main reason why No-SQL solutions are so popular for read and write heavy applications.

MongoDB Ecosystem

The below Diagram 1.2 is the current snapshot of the MongoDB companies ecosystem and product offerings. The focus of this guide is on the MongoDB database used locally on our machines and on the cloud using Atlas. We will also dive in Compass and the Stitch world of MongoDB.

Diagram 1.2:



Installing MongoDB

MongoBD runs on all Operating Systems (OS) which include Windows/Mac/Linux. To install MongoDB we can visit their webpage on:

https://www.mongodb.com/

Under products select MongoDB server and download the MongoDB Community Server for our OS platform of choice. Install the MongoDB Server by following the installation steps.

Important Note: On Windows when installing click on the Custom Setup Type. MongoDB will be installed as a service which will be slightly different to how MongoDB runs on Mac & Linux.

On Mac and Linux we simply have a extracted folder which contains files. We would copy all the contents within this folder and paste them into any place within our OS i.e wherever we would want to install MongoDB.

We would then want to create a folder called data and a sub-folder called db anywhere within our OS, preferably in the root of the OS.

On Windows open up the command prompt or on Mac/Linux open up the terminal. This is where

we are going to spend most of out time using special commands and queries. Run the following command:

```
$ mongo
```

This should return command not found.

To fix this problem on a Mac go to the user folder and find a file called .bash_profile file (*if this does not exist we could simply create it*). Edit the file using a text editor. Add the following line:

```
export PATH=/Users/Username/mongobd/bin:$PATH
```

The path should be wherever we placed the mongoDB binary zip files. We need to add :$PATH at the end on Mac/Linux to make sure all our other commands work on our OS. Save the file and close the file.

Important Note: if you run into a problem on not being able to edit the .bash_profile using text editor use the following command to edit it within the terminal:

```
$ sudo nano ~/ .bash_profile
```

This will allow you to edit the file within the terminal and enter the mongo bin file path. Press CRTL + o to save and CTRL + x to exit the nano edit.

To fix this on a Windows OS, we need to create an environment variable. Press the windows key and type environment which should suggest the Edit Environment Variable option. Under the user variables edit Path to add the directory path to where we installed the mongoDB files:

```
C:\Program Files\MongoDB\Server\4.0\bin
```

Restart the terminal/command prompt and now run the command:

```
$ mongo
```

This should now return a error of connect failed on Mac/Linx.

On Windows it will connect because MongoDB is running as a service and has already started as a background service because we would have checked this during the installation. If we open the command prompt as administrator and ran the command 'net stop MongoDB' this will stop the background service running automatically and we can manually start and stop the MongoDB service running on windows. DO NOT RUN THIS COMMAND ON MAC/LINUX.

The mongo command is the client which allows us to connect to the server and then run commands on the databases. To start the service on Mac/Linux we would use the following command:

```
$ mongod
```

When running this command to start the server it may fail if we chose a different default /data/db folder. If we used a different folder and not within the root of our OS we would need to start the mongod command instance followed by the --dbpath flag and the place where the /data/db is located if not within the root directory.

```
$ sudo mongod --dbpath "/data/db"
```

On Mac we would need to run the mongod command every time we wish to run the mongoDB service whereas on Windows this will run automatically even after restarting the system.

Now that we have the mongod server running minimise the terminal on Mac/Linux and open up a new terminal. We cannot close the mongod server terminal because it is running the service and if closed the mongoDB server everything will stop working and we cannot continue to work with the database server. Pressing the CTRL + C keys within the terminal will quit the mongod service, but we would need to re-run the mongod command again should we wish to run the server again.

We are now in the mongo shell which is the environment where we can run commands against our database server. We can create new databases, collections and documents which we will now focus on in the following sections.

Time to get Started

Now that we have the mongod server running and we can now connect to it using the mongo shell we can now enter the following basic commands in the mongo terminal:

Command	Description
$ cls	Clear the terminal.
$ show dbs	Display existing databases (there are three default databases: admin, config and local which store meta data).
$ use databaseName	Connect/Switch to a database. If the database does not exist it will implicitly create a new database using the databaseName. It will not create the database until a collection and document is added.
$ db.collectionName.insertOne({“name of key”: “key value”})	Create a new collection. The db relates to the current connected database. This will implicitly create a new collection if it does not exist. We must pass at least one new data in the collection using the .insert() command passing in a JSON object. This will return the object to confirm the data was inserted into the database.

Important Note: we can omit the quotes around the name of the key within the shell but we must contain the quotes for the key value unless the key value is type of number. This is a feature within the mongo shell which work behind the scenes. MongoDB will also generate a uniqueId for new documents inserted into the collection.

Command	Description
$ db.collectionName.find()	Display the document within the database collection.
$ db.collectionName.find().pretty()	Display the documents within the database collection but prettify the data in a more humanly readable format.

This is a very basic introductory look at the following shell commands we can run in the mongo terminal to create a new database, switch to a database, create a collection and documents and display all the documents within a database collection either in the standard or pretty format.

Tip: to run the mongod server on a different port to the default port 27017 by run the following command. Note you would need to specify the port when running the mongo shell command as well. You would use this in case the default port is being used by something else.

$ sudo mongo --port 27018

$ mongo --port 27018

Shell vs Drivers

The shell is a great neutral ground for working with mongoDB. Drivers are packages we install for different programming languages the application might be written in. There are a whole host of drivers for the various application server languages such as PHP, node, C#, python etc. Drivers are the bridges between the programming language and the mongoDB server.

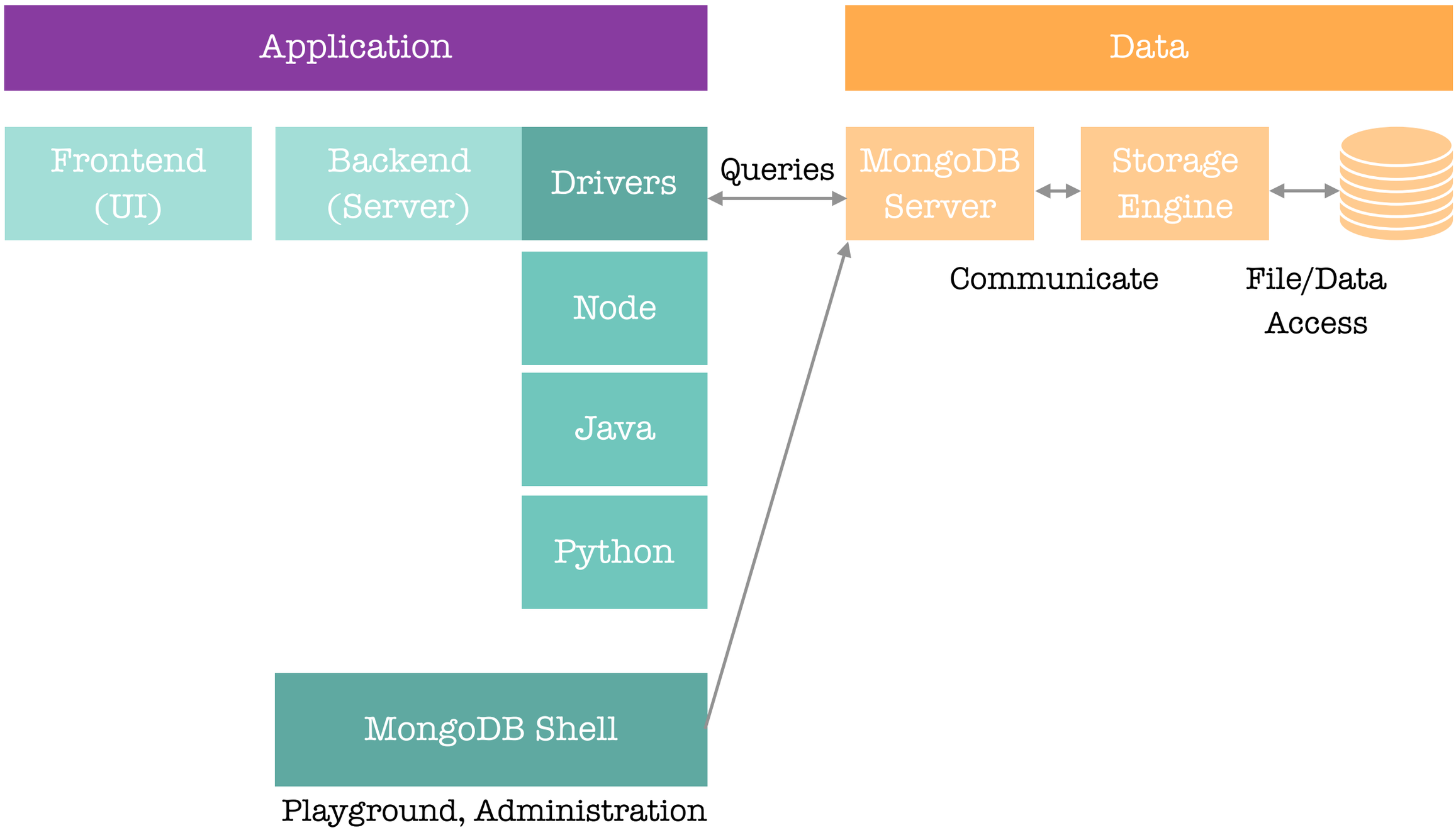
As it turns out, in these drivers, we would use the same command as we use in the shell, they are just slightly adjusted to the syntax of the language we are working with.

The drivers can be found on the mongoDB website:
https://docs.mongodb.com/ecosystem/drivers/

Throughout this document we will continue to use the Shell commands as it is the neutral commands. We can take the knowledge of how to insert, configure inserts, query data, filter data, sort data and many more shell commands. These commands will continue to work when we use the drivers but we would need to make reference to the driver documentation to understand how to use the shell commands but using the programming language syntax to perform the commands using the drivers. This will make us more flexible with the language we use when building applications that uses mongoDB.

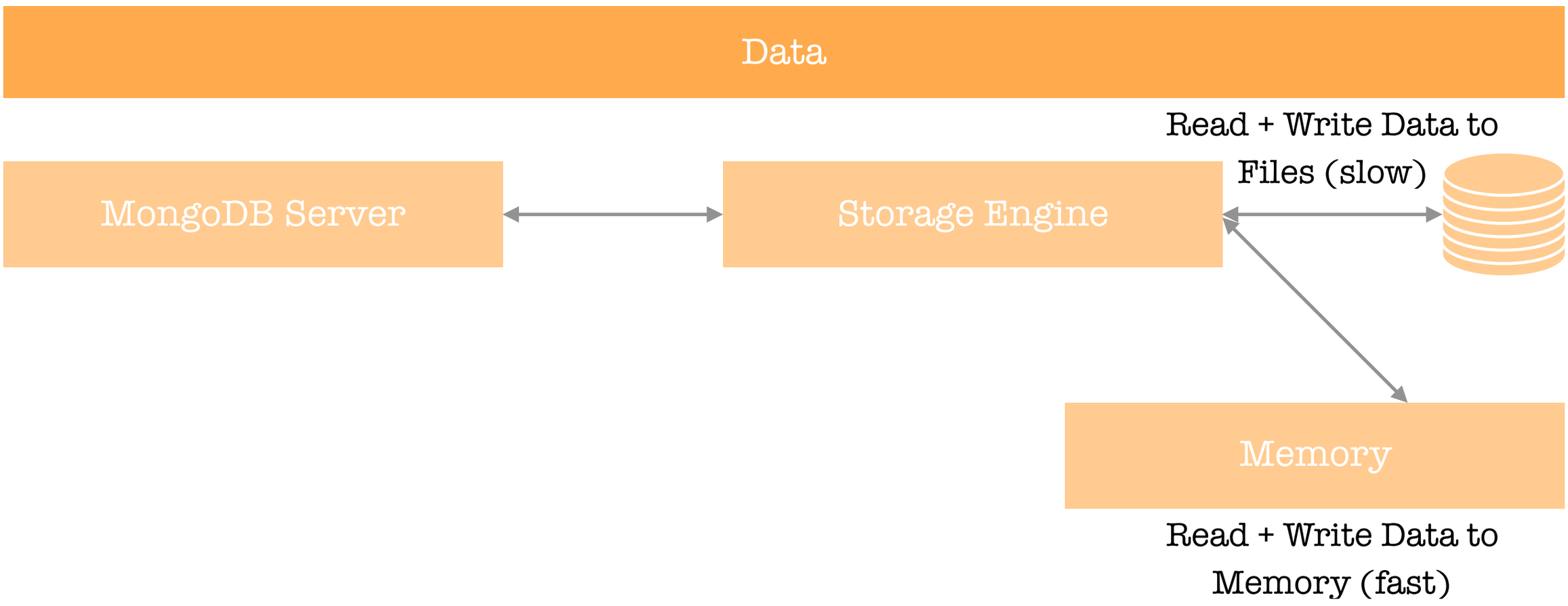
MongoDB & Clients: The Big Picture

Diagram 1.3:



MongoDB & Clients: The Big Picture

Diagram 1.4:



As we can see in Diagram 1.3 the application driver/shell communicates to the mongoDB server. The MongoDB server communicates with the storage engine. It is the Storage Engine which deals with the data passed along by the MongoDB Server, and as Diagram 1.4 depicts it will read/write to database and/or memory.

Create, Read, Update & Delete (CRUD)

We could use MongoDB to create a variety of things such as an application, Analytics/BI Tools or data administration. In an application case, we may have an app where the user interacts with our code (the code can be written in any programming language) and the mongoDB driver will be included in the application. In the case of a Analytics/BI Tools we may use the BI Connector/Shell provided by mongoDB or another import mechanism provided by our BI tool. Finally, in the database administrator case we would interact with the mongoDB shell.

In all the above cases we would want to interact with the mongoDB server. In an application we would typically want to be able to create, read, update or delete elements e.g. a blog post app. With analytics, we would at least want to be able to read the data and as an admins we would probably want to do all the CRUD actions.

CRUD are the only actions we would want to perform with our data i.e. to create it, manage it or read it. We perform all these actions using the mongoDB server.

Diagram 1.5:

CREATE	UPDATE
insertOne(data, options) insertMany(data, options)	updateOne(filter, data, options) updateMany(filter, data, options) replaceOne(filter, data, options)
READ	**DELETE**
findOne(filter, options) find(filter, options)	deleteOne(filter, options) deleteMany(filter, options)

The above are the four CRUD operations and the commands we can run for each action. In later sections we will focus on each CRUD action individually to understand in-depth each of the actions and syntax/command we can use when performing CRUD operation with our mongoDB data collection and documents.

Finding, Inserting, Updating & Deleting Elements

To show all the existing databases within the mongoDB server we use the command “show dbs” while we use the use followed by the database name to switch to a database. The db will then relate to the switched database.

To perform any CRUD operations, these commands must always be performed/executed on a collection where you want to create/update/delete documents. Below are example snippets of CRUD commands on a fictitious flights database (where the collection is called flightData).

```
$ db.fligh Data.insertOne( {distance: 1200} )
```

This will add a single document to the collection as we have seen previously.

```
$ db.flightData.deleteOne( {departureAirport: “LHR”} )
$ db.flightData.deleteMany( {departureAirport: “LHR”} )
```

The delete command takes in a filter. To add a filter we would use the curly brackets passing in

which name of key and key value of the data we wish to filter an delete. In the above example we used the departureAirport key and the value of TXL. The deleteOne command will find the first document in our database collection that meets the criteria and deletes it. The command will return:

{ “acknowledged” : true, “deleteCount” : 1 }

If a document was deleted in the collection this will show the number of deleted documents (the deleteOne command will always return 1). If no documents matched the filter and none were deleted the returned deleteCount value will be 0.

The deleteMany in contrast will delete many documents at once where the documents matches the filter criteria specified.

Note: The easiest way to delete all data in a collection is to delete the collection itself.

```
$ db.flightData.updateOne( {distance: 1200}, { $set: {marker: “delete”} } )
$ db.flightData.updateMany( {distance: 1200}, { $set: {marker: “delete”} } )
```

The update command takes in 3 argument/parameters. The first is the filter which is similar to the delete command. The second is how we would want to update/change the data. We must use the

{ $set: { } } keyword (*anything with a $ dollar sign in front of the keyword is a reserved word in mongoDB*) which lets mongoDB know how we are describing the changes we want to make to a document. If the update key:value does not exist, this will create a new key:value property within the document else it will update the existing key:value with the new value passed in. The third parameter is options which we will analyse in great detail in the latter sections.

Important Note: when passing in a filter we can also pass in empty curly brackets { } which will select all documents within the collection.

If successful with updating many this will return within the terminal an acknowledgement as seen below, where the number of matched the filter criteria and the number of data modified:

{ “acknowledged” : true, “matchedCount” : 2, “modified” : 2 }

If we were to delete all the documents within a collection and use the command to find data in that collection i.e using the db.flightData.find().pretty() command, the terminal will return empty/nothing as there are no existing documents to read/display.

The above demonstrates how we can find, insert, update and delete elements using the update and delete command.

Now we have seen how we can use insertOne() to add a single document into our collection. However, what if we want to add more than one document? We would use the inserMany() command instead.

```
db.flightData.insertMany( [
    {
        “departureAirport”: “LHT”,
        “arrivalAirport”: “TXL”
    } ,
    {
        “departureAirport”: “MUC”,
        “arrivalAirport”: “SFO”
    }
] )
```

We pass in an array of objects in order to add multiple documents into our database collection. The square brackets is used to declare an array. The curly brackets declare a object and we must use comma’s to separate each object. If successful, this will return acknowledged of true and the insertdIds of each object/document added into the collection.

Important Note: mongoDB by default will create a unique id for each new document which is assigned to a name of key called "_id" followed by a random generated key. When inserting a object we could assign our own unique id using the _id key followed by a unique value. If we insert a object and pass in our own _id key value and the value is not unique this will return a duplicate key error collection in the terminal. We must always use a unique id for our documents and if we do not specify a value for _id then mongoDB will generate one for us automatically.

Diving Deeper Into Finding Data

Currently we have seen the .find() function used without passing any arguments for finding data within a collection. This will retrieve all the data within the collection. Just as we would use filter to specify a particular records or documents when deleting or updating a collection, we can also filter when finding data.

We can pass a document into the find function which will be treated as a filter as seen in the example below. This allows us to retrieve a subset of the data rather than the whole data within an application.

```
db.flightData.find( {intercontinental : true } ).pretty()
```

We can also use logical queries to retrieve more than one document within a collection that matches the criteria as demonstrated in the below example. We query using another object and then one of the special operators in mongoDB.

```
db.flightData.find( {distance: {$gt: 1000 } } ).pretty()
```

In the above we are using the $gt: operator which is used for finding documents “greater than” the value specified. If we were to use the findOne() operator this will return the first record within the collection that matches the criteria.

Update vs UpdateManay

Previously we have seen the updateOne() and updateMany() functions. However, we can also use another update function called update() as seen in the example below:

```
db.flightData.update( { _id: ObjectId(“abc123”) }, { $set: { delayed: true } } )
```

The update() function works exactly like the updateMany() function where all matching documents to the filter are updated. The difference between update() and updateMany() is that the $set:

operator is not required for the update() function whereas this will cause an error for either the updateOne() and updateMany() functions. So we can write the above syntax like so and would not get an error:

```
db.flightData.update( { _id: ObjectId("abc123") }, { delayed: true } )
```

The second and main difference is that the update function takes the new update object and replaces the existing object (*this does not affect the unique id*) updating the document. It will only patch the update object instead of replacing the whole existing object (just like the updateOne() and updateMany() functions), if we were to use the $set: operator, otherwise it would override the existing document.

This is something to be aware of when using the update() function. If we intend to replace the whole existing document with a new object then we can omit the $set: operator. In general it is recommended to use updateOne() and updateMany() to avoid this issue.

If, however, we want to replace a document we should use the replaceOne() function. Again, we would place our filter and the object we want to replace with. This is a more explicit and more safer way of replacing the data in a collection.

```
db.flightData.replaceOne( { id: ObjectId("abc123") }, { departureAirport: "LHT", distance: 950 } )
```

Understanding Find() & The Cursor Object

If we have a passengers collection which stores the name and age of passengers and we want to retrieve all the documents within the passenger collection we can use the find() function as we have seen previously.

```
db.passengers.find().pretty()
```

Useful Tip: when writing commands in the shell we can use the tab key to autocomplete for example if we wrote db.passe and tab on our keyboard, this should auto-complete db.passengers.

We will notice where a collection has many data, the find() function will return all the data but display all the data with the shell. If we scroll down to the last record we should see Type “it” for more within the shell. If we type the command it and press enter, this will display more data from the returned find() function. The find() command in general returns back what is called a Cursor Object and not all of the data.

The find() does not give an array of all the documents within a collection. This makes sense as the

collection could be really large and if the find() was to return the whole array, imagine if a collection had 2million documents – this could take a really long time but also send a lot of data over the connection.

The Cursor Object is an object that has many meta data behind it that allows us to cycle through the results, which is what the "it" command did. It used the Cursor Object to fetch the next group (cycle) of data from the collection.

We can use other methods on the find() function such as toArray() which will exhaust the cursor i.e. go through all of the cursors and fetch back all the documents within the array (i.e. not stopping after the first 20 documents – a feature within the mongoDB shell).

```
db.passengers.find().toArray()
```

There is a forEach method that can also be used on the find() function. The forEach allows us to write some code to do something on every element that is in the database. The syntax can be found within the driver documents for whichever language we are using for our application e.g. PHP or JavaScript etc. Below is a JavaScript function which the shell can also use:

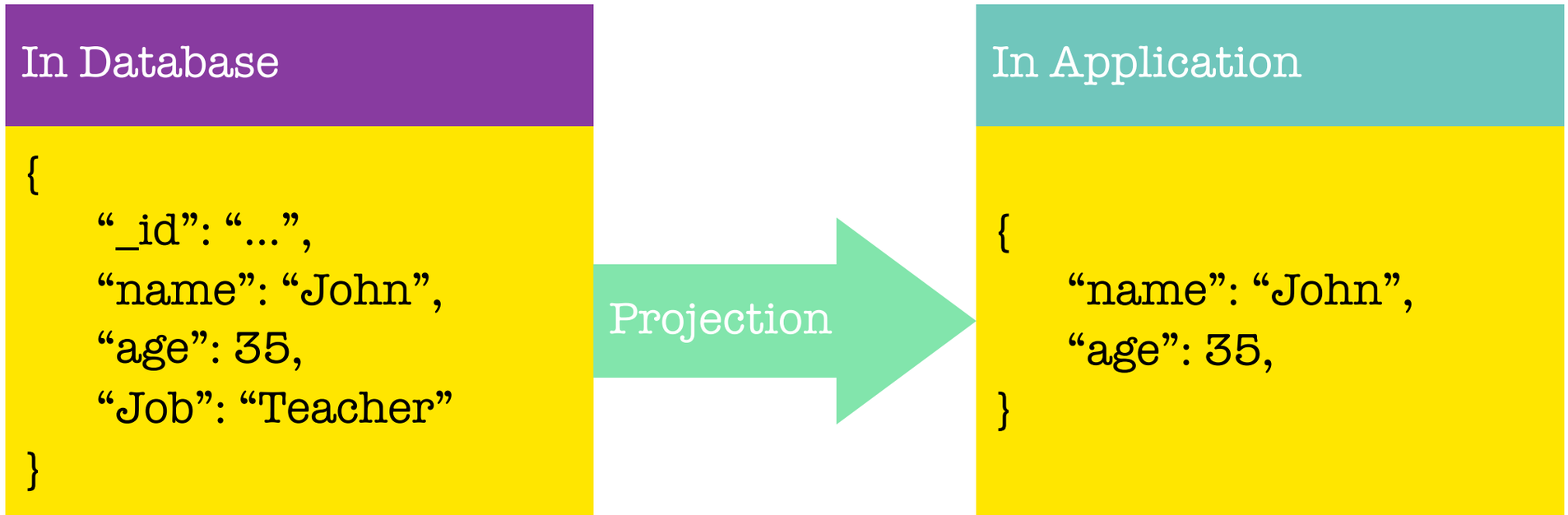
```
db.passengers.find().forEach( (document) => { printjson(document) } )
```

The forEach function in JavaScript gets the document object passed in automatically into the arrow function and we can call this whatever we want i.e. passengersData, data, x, etc. In the above we called this document. We can then use this object and do whatever we want i.e. we used the printjson() command to print/output the document data as JSON. The above will also return all the documents within the collection because the forEach loops on every Cursor Object.

To conclude, the find() function does not provide us with all the documents in a collection even though it may look like it in some circumstances where there are very little data within a collection. Instead it returns a Cursor Object which we can cycle through the return more documents from the collection. It is unto us as the developer to use the cursor to either force it to get all the documents from a collection and place it in an array or better using the forEach or other methods to retrieve more than 20 documents (*the default number of items returned in the shell*) from the collection. Note the forEach is more efficient because it fetches/returns objects on demand through each iteration rather than fetching all the data in advance and loaded into memory which saves both on bandwidth and memory.

The Cursor Object is also the reason why we cannot use the .pretty() command on the findOne() function because the findOne returns one document and not a Cursor Object. For Insert, Update and Delete commands the Cursor Object does not exist because these methods do not fetch data, they simply manipulate the data instead.

Understanding Projections



Imagine in our database we have the data for a person record and in within our application we do not need all the data from the document but only the name and age to display on our web application. We could fetch all the data and filter/manipulate the data within our application in any programming language. However, this approach will still have an impact on the bandwidth by fetching unnecessary data — something we want to prevent. It is better to filter the data out from the mongoDB server and this is exactly what projection allows us to do.

Below are examples of using projections to filter the necessary data to retrieve from our find query.

```
db.passengers.find( {}, {name: 1} ).pretty()
```

We need to pass in a first argument to filter the find search (note: a empty object will retrieve all documents). The second argument allows us to project. A projection is setup by passing another document but specifying which key:value pairs we want to retrieve back. The one means to include it in the data returned to us.

The above will return all the passengers document but only the name and id, omitting the age from the returned search results. The id is a special field in our data and by default it is always included. To exclude the id from the returned results, we must explicitly exclude it. To exclude something explicitly we would specify the name of key and set the value to zero as seen below:

```
db.passengers.find( {}, {name: 1, _id:0} ).pretty()
```

Note: we could do the same for age (e.g. age: 0), however, this is not required because the default is everything but the _id is not included in the projection unless explicitly specified using the one.

The data transformation/filtering is occurring on the mongoDB server before the data is shipped to us and is something that we would want because we do not want to retrieve unnecessary data which will impact on the bandwidth.

Embedded Documents & Arrays

Embedded documents is a core feature of mongoDB. Embedded documents allows us to nest other documents within each other and having one overarching document in the collection.

There are two hard limits to nesting/embedded documents:

1. We can have up to 100 level of nesting (a hard limit) in mongoDB.
2. The overall document size has to be below 16mb

The size limit for documents may seem small but since we are only storing text and not files (we would use file storage for files), 16mb is more than enough.

Along with embedded documents, another documents we can store are arrays and this is not strictly linked to embedded documents, we can have arrays of embedded documents, but arrays can hold any data. This means we have list of data in a document.

Below are examples of embedded documents and arrays.

```
db.flightData.updateMany( {}, {$set: {status: {description: “on-time”, lastUpdated: “1 hour ago”} } } )
```

In the above example we have added a new document property called status which has a embedded/nested document of description and lastUpdated. If we output the document using .find() function, the below document would now look something like the below:

```
{
    “_id”: …
    “departure”: “LHT”,
    “arrivalAirport”: “TXL”,
    “status”: {
        “description”: “on-time”,
        “lastUpdated” “1 hour ago”,
    }
}
```

Note: we could add more nested child documents i.e. description could have a child nested document called details and that child could have further nested child documents and so on.

```
db.passengers.updateOne( {name: "Albert Twostone"}, {$set: {hobbies: ["Cooking", "Reading"] } } )
```

Arrays are marked with square brackets. Inside the array we can have any data, this could be multiple documents (i.e. using the curly brackets {}), numbers, strings, booleans etc.

If we were to output the document using the .find() function, the document would look like something below:

```
{
    "_id": ...,
    "name": "Albert Twostone",
    "age": 63,
    "hobbies": [
        "Cooking",
        "Reading"
    ]
}
```

Albert Twostone will be the only person with hobbies and this will be a list of data. It is important to note that hobbies is not a nested/embedded document but simply a list of data.

Accessing Structured Data

To access structured data within a document we could use the following syntax:

```
db.passengers.findOne( {name: “Albert Twostone”} ).hobbies
```

We can specify the name of a structured data within a document by using the find query and then using the name of key we wish to access from the document, in the above we wanted to access the hobbies data which will return the hobbies array as the output:

```
[“Cooking”, “Reading”]
```

We can also search for all documents that have hobbies of Cooking using the syntax below as we have seen previously. This will return the whole document entry where someone has Cooking as a hobby. MongoDB is clever enough to look in arrays to find documents that match the criteria.

```
db.passengers( {hobbies: “Cooking”} ).pretty()
```

Below is an example of searching for objects (this includes searching within nested documents):

```
db.flightData.find( {“status.description”: “on-time”} ).pretty()
```

We use the dot notation to drill into our embedded documents to query our data. It is important that we wrap the dot notation in quotations (e.g. “status.description”) otherwise the find() function would fail.

This would return all documents (the whole document) where the drilled criteria matches. This allows us to query by nested/embedded documents. We can drill as far as we need to using the dot notation as seen in the example below:

```
db.flightData.find( {“status.details.responsible”: “John Doe”} ).pretty()
```

This dot notation is a very important syntax to understand as we would use this a lot to query our data within our mongoDB database.

Conclusion

We have now covered all the basic and core features of mongoDB to understand how mongoDB works and how we can work with it i.e. store, update, delete and read data within the database as well as how we can structure our data.

Resetting The Database

To purge all the data within our mongoDB database server we would use the following command:

```
use databaseName
db.dropDatabase()
```

We must first switch to the database using the use command followed by the database name. Once we have switched to the desired database we can reference the current database using db and then call on the dropDatabase() command which will purge the specified database and its data.

Similarly, we could get rid of a single collection in a database using the following command:

```
db.myCollection.drop()
```

The myCollection should relate to the collection name.

These commands will allow us to clean our database server by removing the database/collections that we do not want to keep on our mongoDB server.

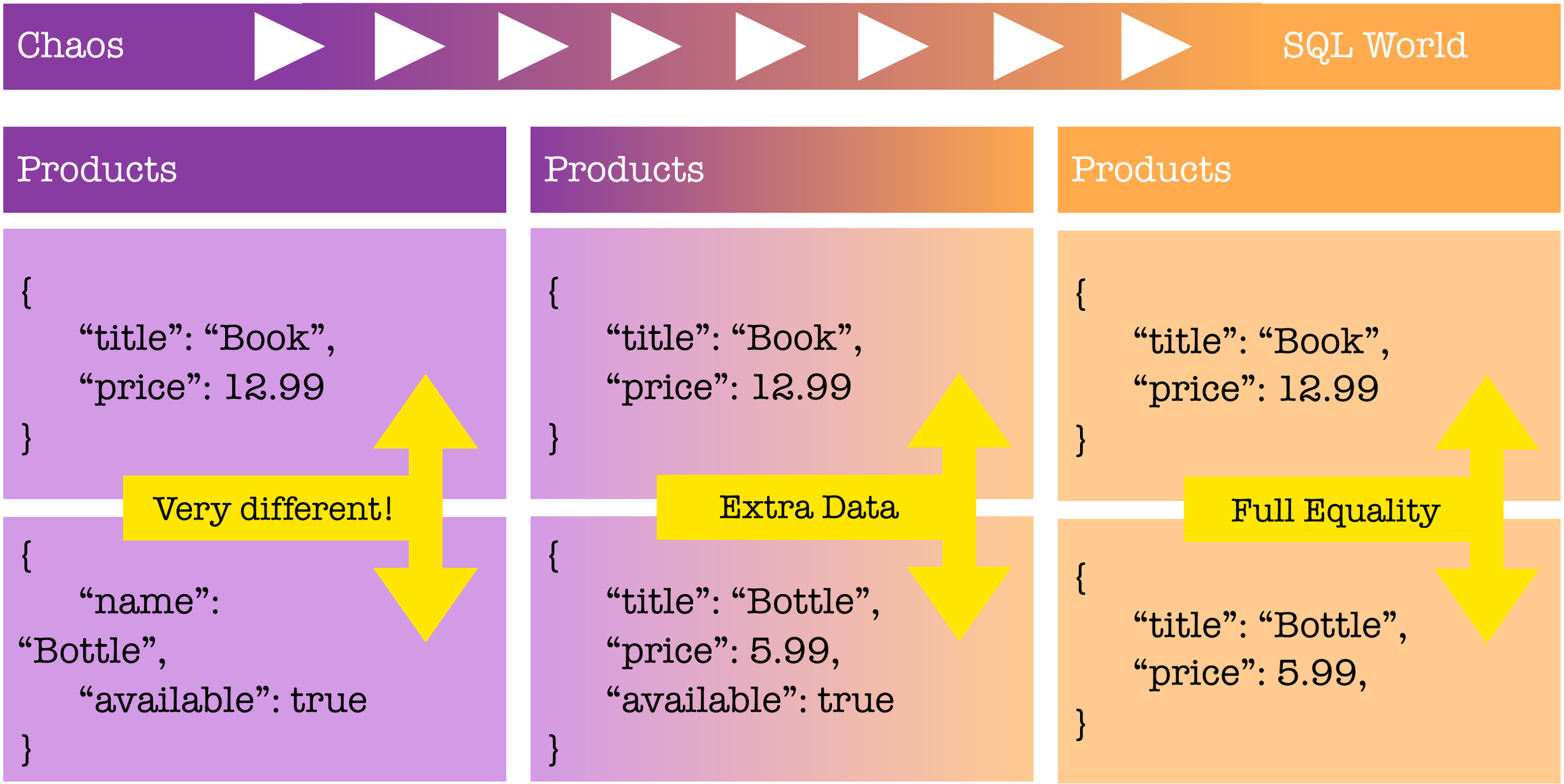
Why Do We Use Schemas?

There is one important question to ask – wasn't mongoDB all about having **no** data Schemas i.e. Schema-less. To answer this question, mongoDB enforces no Schemas. Documents do not have to use the same schema inside of one collection. Our documents can look like whatever we want it to look like and we can have totally different documents in one and the same collection i.e. we can mix different schemas.

Schemas are the structure of one document i.e. how does it look like, which fields does it have and what types of value do these fields have. MongoDB des not enforce schemas; however, that does not mean that we cannot use some kind of schema and in reality we would indeed have some form of schema for our documents. It is in our interest if we were to build a backend database that we have some form of structure to the types of documents we are storing. This would make it easier for us to query our database and get the relevant data and then cycle through this data using a programming language to display the relevant data within our application.

We are most likely to have some form of schemas because we as developers would want it and our applications will need it. Whilst we are not forced to have a schema we would probably end up with some kind of schema structure and this is important to understand.

Structuring Documents



We can use any of the structured approach in the diagram above depending on how we require it in our applications. In reality we would tend to use the approach in the middle or on the right.

The middle approach used the best of both words where there are some structure to the data, however, it also has the flexibility advantage that mongoDB provides us so that we can store extra information.

Note: we can assign the null value to properties in order to have a structured approach although the data may not have any actual values associated with the property. A null value is considered a valid value and therefore we can use a SQL (structured) type approach with all our documents.

There is no single best practice with how to set the structure of our data within our documents and it is up to us as developers to use the best structure that works best for our applications or whichever is to our personal preference.

Data Types

Now that we understand that we are free to define our own schemas/structure for our documents, we are now going to analyse the different data types we can use in mongoDB.

Data Types are the types of data we can save in the fields within our documents. The below table break the different data types for us:

Type	Example Value
String	“John Doe”
Boolean	TRUE
NumberInt (int32)	55, 100, 145
NumberLong (int64)	10000000000
NumberDecimal	12.99
ObjectId	ObjectId(“123abc”)
ISODate	ISODate(“2019–02–09”)
Timestamp	Timestamp(11421532)
Embedded Documents	{“a”: {...}}
Arrays	{“b”: [...]}

Notice how the text type requires quotation marks (*single or double*) around the value. There are no limitation in the size of the text. The only limitation is the 16mb for the whole document. The larger the text the larger the data it takes.

Notice how numbers and booleans do not require a quotation marks around the value.

There are different types of numbers in mongoDB. Integer (int32) are 32bit long numbers and if we try to store a number longer than this they would overflow that range and we will end up with a

different number. For longer integer numbers we would use NumberLong (int64). The integer solution we decide to choose will dictate how much space will be allocated and eaten up by the data. Finally, we can also store NumberDecimal i.e. numbers with decimal values (a.k.a float in other programming languages).

The default within the shell is to store a int64 floating point value but we also have a special type of NumberDecimal provided by mongoDB to store high precision floating point values. Normal floating point values (a.k.a doubles) are rounded and are not precise after their decimal place. However, for many use cases the floating point (double) is enough precision required e.g. shop store. If we are performing scientific calculations or something that requires a high precision calculation, we are able to use the special type that offers this very high decimal place precision (i.e. 34 digits after the decimal place).

The ObjectId is a special value that is automatically generated by mongoDB to provide a unique id but it also provides some temporal component that allows for sorting built into the ObjectId, respecting a timestamp.

The above table provides all the data types within mongoDB that we can use to store data within our database server.

Data Types & Limits

MongoDB has a couple of hard limits. The most important limitation: a single document in a collection (including all embedded documents it might have) must be less than or equal to 16mb. Additionally we may only have 100 levels of embedded documents.

We can read more on all the limitation (in great detail) on the below link:

https://docs.mongodb.com/manual/reference/limits/

For all the data types that mongoDB supports, we can find a detailed overview on the following link:

https://docs.mongodb.com/manual/reference/bson-types/

Important data type limits are:

- Normal Integers (int32) can hold a maximum value of +-2,147,483,674
- Long Integers (int64) can hold a maximum value of +-9,223,372,036,854,775,807
- Text can be as long as we want – the limit is the 16mb restriction for the overall document.

It's also important to understand the difference between int32 (NumberInt), int64 (NumberLong) and a normal number as you can enter it in the shell. The same goes for a normal double and NumberDecimal.

NumberInt creates a int32 value => NumberInt(55)

NumberLong creates a in64 value => NumberLong(7489729384792)

If we just use a number within the shell for example insertOne({a: 1}), this will get added as a normal double into the database. The reason for this is because the shell is based on JavaScript which only knows float/double values and does not differ between integers and floats.

NumberDecimal creates a high precision double value => NumberDecimal("12.99")

This can be helpful for cases where we need (many) exact decimal places for calculations.

When working with mongoDB drivers for our application's programming language (e.g. PHP, .NET, Node.js, Python, etc.), we can use the driver to create these specific numbers. We should always browse the API documents for the driver we are using within our applications to identify the methods for building int32, int64 etc.

Finally we can use the db.stats() command in the mongoDB shell to see stats of our database.

How to Derive Our Data Structure Requirements

Below are some guidelines to keep to mind when we think about how to structure our data:

- What data does our App need to generate? What is the business model?

User Information, Products Information, Orders etc. This will help define the fields we would need (and how they relate).

- Where do I need my data?

For example, if building a website do we need the data on the welcome page, products list page, orders page etc. This help define our required collections and field groupings.

- Which kind of data or information do we want to display?

For example the welcome page displays product names. This will help define which queries we need i.e. do we need a list of products or a single product.

These queries we plan to have also have an impact on our collections and document structure. MongoDB embraces the idea of planning our data structure based on the way we retrieve the data so that we do not have to perform complex joins but we retrieve the data in the format or almost in

the format we need it in our application.

- How often do we fetch the data?

Do we fetch data on every page reload, every second or not that often? This will help define whether we should optimise for easy fetching of data.

- How often do we write or change the data?

Do we change or write data often or rarely will help define whether we should optimise for easy writing of data.

The above are things to keep in mind or to think about when structuring our data structures and schemas.

Schemas & Relations: How to Structure Documents

Understanding Relations

Typically we would have multiple collections for example a users collection, a product collection and a orders collections. If we have multiple collections that are relatable or where the documents in these relations are related, we obviously have to think about how do we store related data.

Do we use embedded documents because this is one way of reflecting a relation or alternatively, do we use references within our documents?

Nested/ Embedded Documents

Customers Collection

```
{
    “userName”: “John”,
    “age”: 28,
    “address”: {
        “street”: “First Street”,
        “City”: “Chicago”
    }
}
```

References

Customers Collection:

```
{
    “userName”: “Alan”
    “favBooks”: [“id1”, “id2”]
}
```

Books Collection:

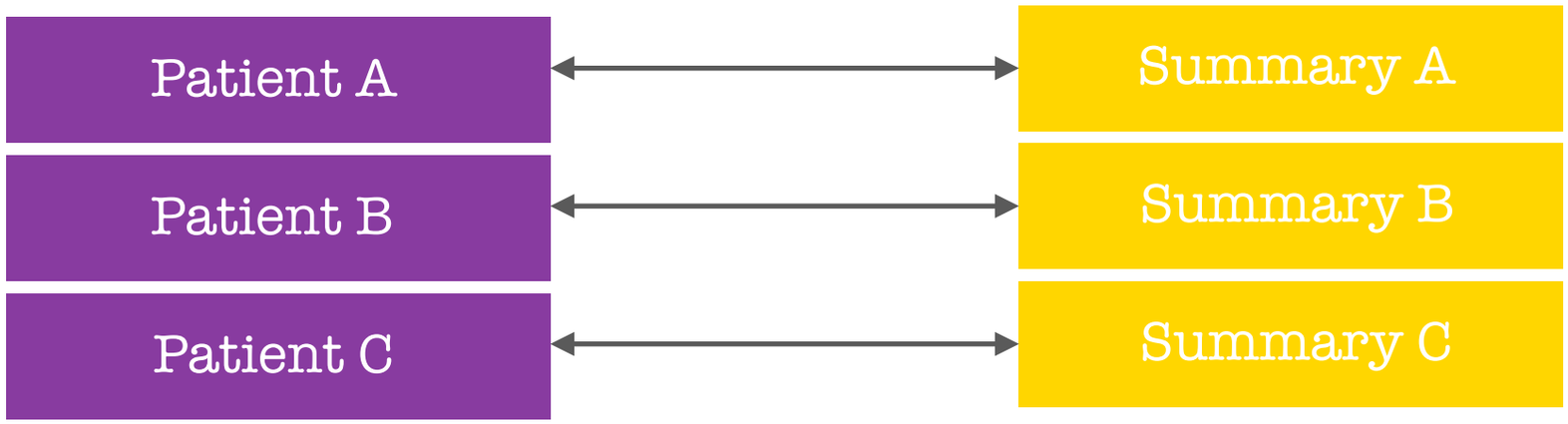
```
{
    “_id”: “id1”,
    “name”: “Lord of the Rings”
}
```

In the reference example above, we would have to run two queries to join the data from the different collections. However, if a book was to change, we would only update it in the books collection as the id would remain the same whereas in a embedded document relation we would have to update multiple customer records affected with the new change.

One to One Embedded Relation Example

Example:

One patient has one disease summary, a disease summary belongs to one patient.



Code snippet:

```
$ use hospital
$ db.patients.insertOne( { name: “John Doe”, age: “25”, diseaseSummary: { diseases: [“cold”, “sickness”] } } )
```

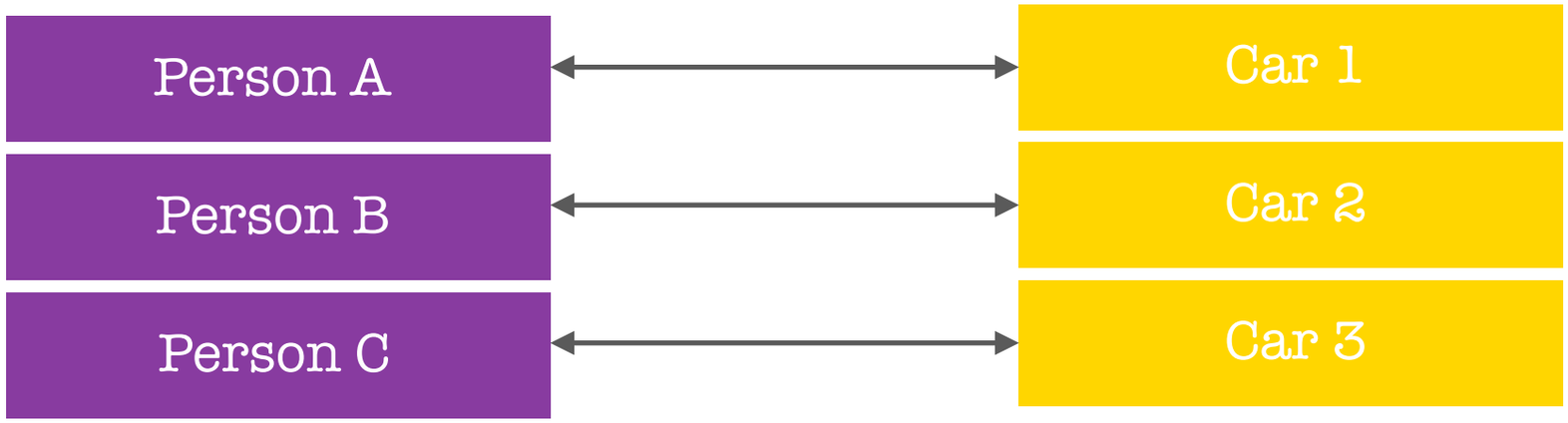
Where there is a strong one to one relation between two data, it is ideal to use a one to one embedded approach as demonstrated in the above example.

The advantage of the embedded nested approach is that within our application we only require a single find query to fetch the necessary data for the patient and disease data from our database collection.

One to One Reference Relation Example

Example:

One person has one car, a car belongs to one person.



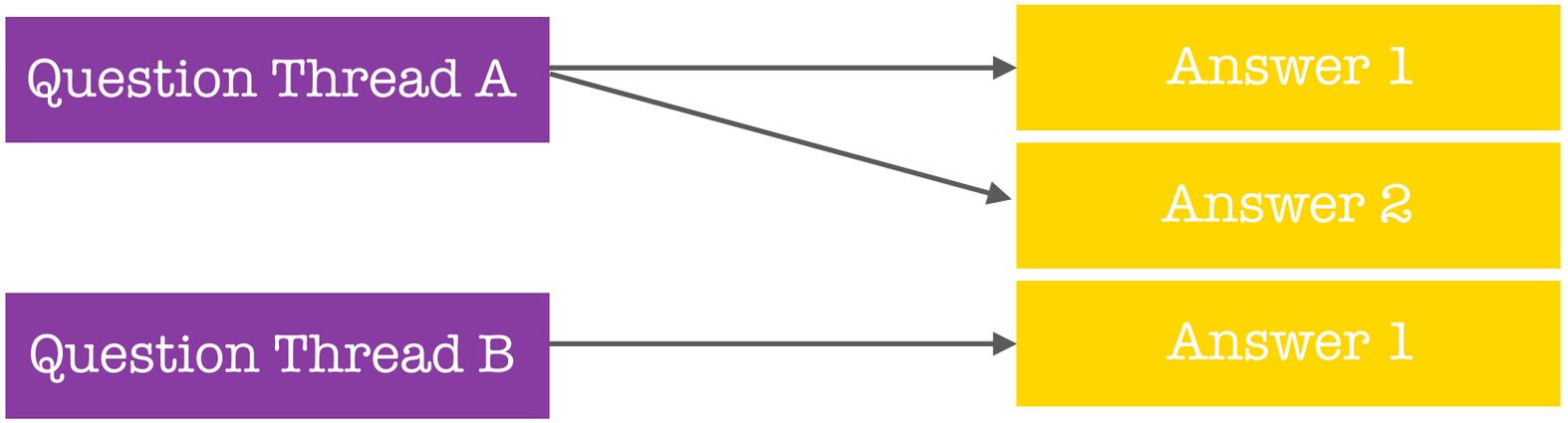
Code snippet:

```
$ use.carData
$ db.persons.insertOne( { name: “John”, age: 30, salary: 30000 } )
$ db.cars.insertOne( { model: “BMW”, price: 25000, owner: ObjectId(“5b98d4654d01c”) } )
```

In most one to one relationships we would generally use the embedded document relations. However, we can opt to use a reference relation approach as we are not forced to use one approach.

For example, we have a more analytics use case rather than a web application and we have a use case where we are interested in analysing the person data and or analysing our car data but not so much in a relation. In this example we have a application driven reason for splitting the data.

One to Many Embedded Relation Example



Example:

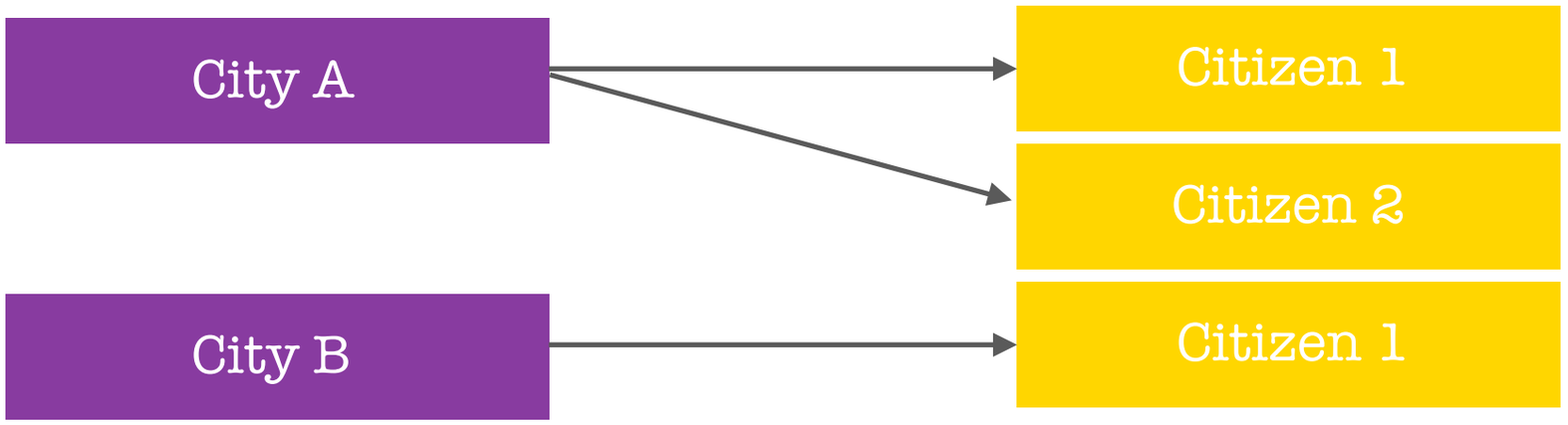
One question thread has many answers, one answer belongs to one question thread.

Code snippet:

```
$ use support
$ db.questionThreads.insertOne( { creator: “John”, question: “How old are you?”, answers: [ { text: “I am 30.” }, { text: “Same here.” } ] } )
```

A scenario where we may use a embedded one to many relation would be post and comments. This is because you would often need to fetch the question along with the answers in an application perspective. Also usually there are not too many answers to worry about the 16mb document limit.

One to Many Reference Relation Example



Example:

One city has many citizens, one citizen belongs to one city.

Code snippet:

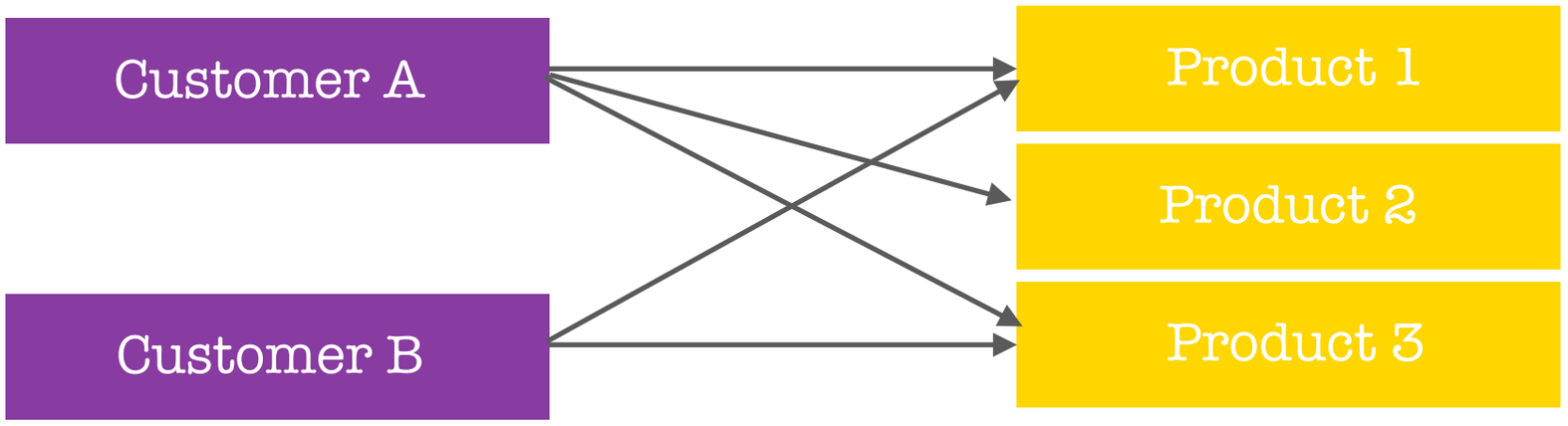
```
$ use cityData
$ db.cities.insertOne( { name: “New York City”, coordinates: { lat: 2121, lng: 5233 } } )
$ db.citizens.insertMany( [ { name: “John Doe”, cityId: ObjectId(“5b98d6b44d”) }, { name: “Bella Lorenz”, cityId: cityId: ObjectId(“5b98d6b44d”) } ] )
```

In the above scenario we may have a database containing a collection of all major cities in the world and a list of every single person living within that city. It would seem to make sense to have a one to many embedded relationship, however, from an application prospective we may wish to only retrieve the city data only. Furthermore, a city like New York may have over 1 million people data and this would make fetching the data slow due to the volume of data passing through the wire. Furthermore, we may end up running into the document size limit of 16mb. In this type of scenario, it would make sense to split the data up and using the reference relation to link the data.

In the above we would only store the city metadata and will not store any citizen reference as this will also end up being a huge list of citizens unique id. Instead, we would create a citizens collection and within the citizens data we would make reference to the city reference. The reference can be anything but must be unique ie. we could use the ObjectId() or the city name etc.

This will ensure that we do not exceed the limitation of the 16mb per document as well as not retrieving unnecessary data if we are only interested in returning just the cities metadata from a collection.

Many to Many Embedded Relation Example



Example:

One customer has many products (via orders), a product belongs to many customers.

Code snippet:

```
$ use shop
$ db.products.insertOne( { title: “A Book”, price: 12.99 } )
$ db.customers.insertOne( { name: “Cathy”, age: 18, orders: [ { title: “A Book”, price: 12.99, quantity: 2 } ] } )
```

We would normally model many to many relationships using references. However, it is possible to

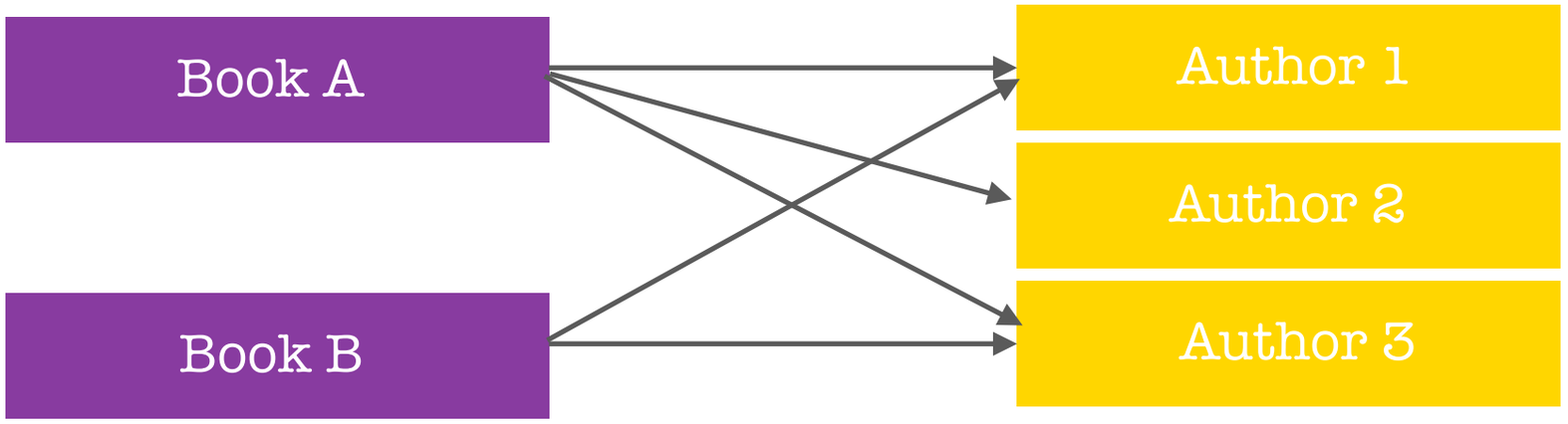
use the embedded approach as seen above. We could store a collection for the products as meta data for an application to retrieve the data in order to help populate the embedded document of the customer collection using a programming language.

A disadvantage to the embedded approach is data duplication because we have the title and price of the product within the orders array as the customer can order the product multiple times as well as other customers which will cause a lot of duplication.

If we decide to change the data for the product, not only do we need to change it within the product collection but we also have to change it on all the orders affected by this change (or do we actually need to change old orders?). If we do not care about the product title changing and the price changing i.e. we have an application that takes a snapshot of the data, we may not worry too much about duplicating that data because we might not need to change it in all the places where we have the duplicated the data if the original data changes — this highly depends on the application we build. Therefore a embedded approach may work.

In other case scenarios where we absolutely need the latest data everywhere, a reference approach may be most appropriate in a many to many relationship. It is important to think about how we would fetch our data and how often do we want to change it and if we need to change it everywhere or are duplicate data fine before deciding which approach to adopt for many to many.

Many to Many Reference Relation Example



Example:

One book has many authors, an author belongs to many books.

Code snippet:

```
$ use bookRegistry
$ db.books.insert( { name: “favourite book”, authors: [ objectId(“5b98d9e4”), objectId(“5b98d9a7”) ] } )
$ db.authors.inserMany( [ { name: “Martin”, age: 42 }, { name: “Robert”, age: 56 } ] )
```

The above is an example of a many to many relation where a reference approach may be suitable for a scenario where the data that changes needs to be reflected everywhere else.

Summarising Relations

We have now explored the different relation options that are available to use. This should provide us enough knowledge to think about relations and when to use the most appropriate approach depending on:

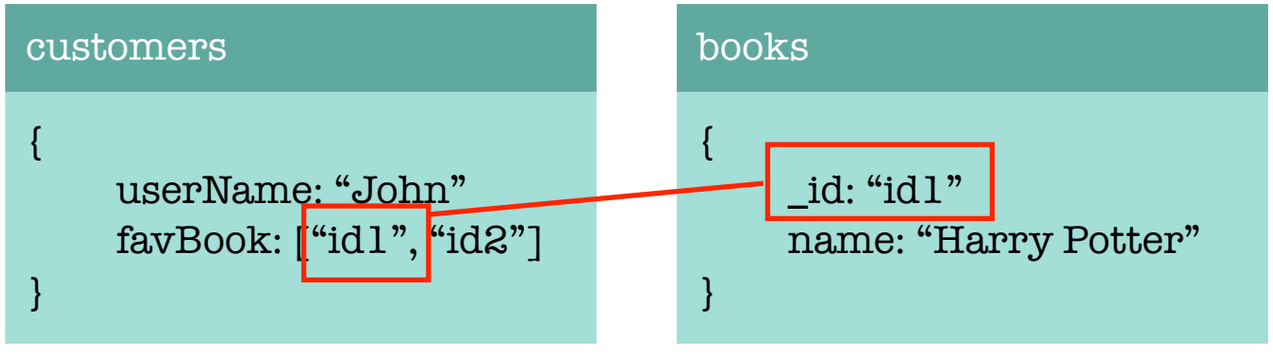
- the application needs
- how often data changes
- if a snapshot data suffice
- how large is the data (how much data do we have).

Nested/Embedded Documents – group data together logically. This makes it easier when fetching the data. This approach is great for data that belong together and is not overlapping with other data. We should always avoid super-deep nesting (100+ levels) or extremely long arrays (16mb size limit per document).

References – split data across collections. This approach is great for related data but also shared data as well as for data which is used in relations and standalone. This allows us to overcome nesting and size limits (by creating new documents).

Using $lookup for Merging Reference Relations

MongoDB has a useful operation called $lookup that allows us to merge related documents that are split up using the reference approach.



The image on the right provides a scenario of a reference approach where the customer and books have been split into two collections. The lookup operator is used as seen below. This uses the aggregate method which we have not currently learned.

```
$ customer.aggregate( [
    { $lookup: { from: “books”, localField: “favBooks”, foreignField: “_id”, as: “favBookData” } }
] )
```

The $lookup operator allows us to fetch two related documents merged together in one document within one step (rather than having to perform two steps). This mitigates some of the disadvantages of splitting our documents across multiple collections because we can merge them in one go.

This uses the aggregate method framework (which we will dive into in later chapters) and within the aggregate we pass in an array because we can define multiple steps on how to aggregate the data. For now we are only interested in one step (a step is a document we pass into an array) where we pass the $lookup step. The lookup passes in a document as a value, where we define 4 attributes:

- **from** – which other collection do we want to relate documents i.e. we would pass in the name of the collection where the other document lives that we wish to merge.
- **localField** – in the collection we are running the aggregate function on, where can the reference to the other (from) collection be found in i.e. the key that stores the reference.
- **foreignField** – which field are we relating to in our target collection (i.e. the from collection)
- **as** – provide an alias for the merged data. This will become the new key which the merged data will sit.

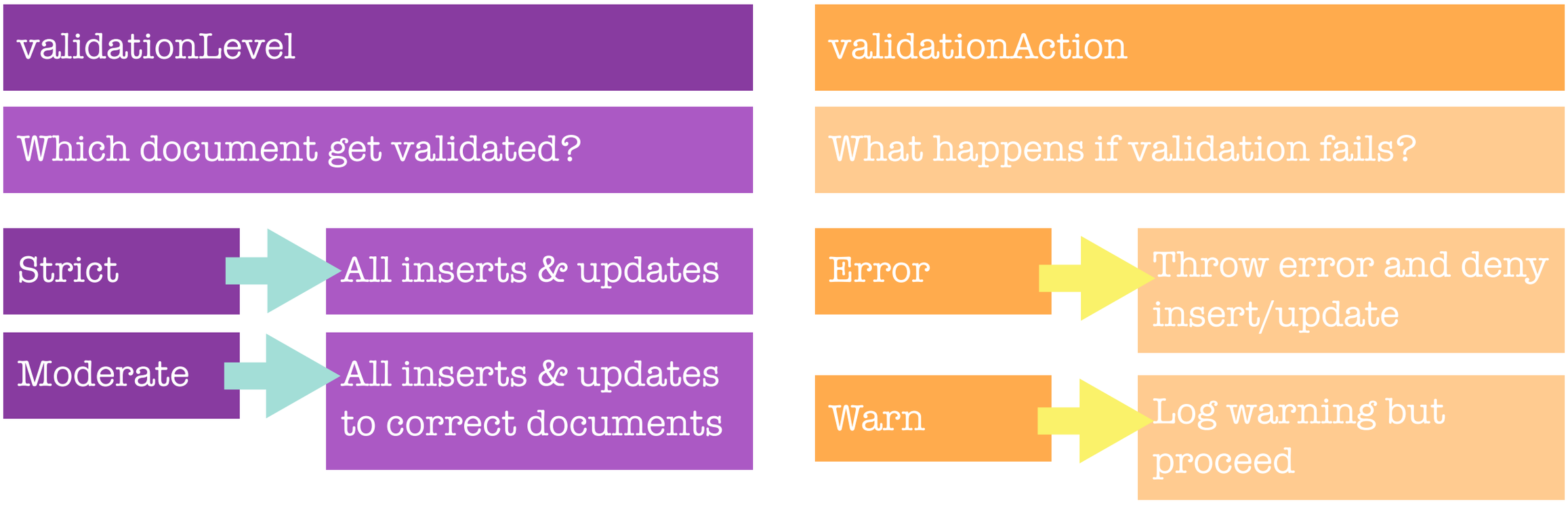
This is not an excuse to always using a reference relation approach because this costs more performance than having an embedded document.

If we have a references or want to use a references, we have the lookup step in the aggregate method that we can use to help get the data we need. This is a first look at aggregate and we will explore what else the aggregate can do for us in later chapters.

Understanding Schema Validation

MongoDB is very flexible i.e. we can have totally different schemas and documents in one and the same collection and that flexibility is a huge benefit. However, there are times where we would want to lock down this flexibility and require a strict schema.

Schema validation allows mongoDB to validate the incoming data based on the schema that we have defined and will either accept the incoming data for the write or update to the database or it will reject the incoming data and the database is not changed by the new data and the user gets an error.



Adding Collection Document Validation

To add schema validation in mongoDB and the easiest method is to add validation when we create a new collection for the very first time explicitly (not implicitly when we add a new data). We can use the createCollection to create and configure a new collection:

```
$ db.createCollection("posts", { validator: { $jsonSchema: { bsonType: "object", required: ["title", "text", "creator", "comments"], properties: { title: { bsonType: "string", description: "must be a string and is required." }, text: { bsonType: "string", description: "must be a string and is required" }, creator: { bsonType: "objectId", description: must be an objectId and is required }, comments: { bsonType: "array", desription: "must be an array and is required", items: { bsonType: "object", required: ["text,"], properties: { text: { bsonType: "string", description: "must be a string and is required" }, author: { bsonType: "objectId", description: "must be an objectId and is required" } } } } } } } } )
```

The first argument to the createCollection method is the name of the collection i.e. we are defining the name of the collection. The second argument is a document where we would configure the new collection. The validator is an important piece of the configuration.

The validator key takes in another sub document where we can now define a schema against incoming data where inserts and updates has to validated. We do this by inserting a $jsonSchema key with another nested sub document which will hold the schema.

We can add a son type with the value of object, so that everything that gets added to the collection should be a valid document or object. We can set a required key which has an array value. In this array we can define names of fields in the document which will be part of the collection that are absolutely required and if we try to add data that does not have these fields, we will get an error or warning depending on our settings.

We can add a properties key which is another nested document where we can define how for every property of every document that gets added to the collection will look like. In the example above we defined the title property, which is a required property, in more detail. We can set the bsonType which is the data type i.e. string, number, boolean, object, array etc. We can also set a description for the data property.

Because an array and has multiple items, we can add an items key and describe how the items should look like. We can nest this and this can have another nested required and properties keys for the items objects that exists within the array.

So the Keys to remember are:

The bsonType key is the data type.

The required key is an array of required properties that must be within an insert/update document.

The properties key defines the properties. This has sub key:value of of bsonType and description.

The Item key defines the array items. This can have sub key:value of all the above.

Important Note: it may be difficult to read in the terminal and may be easier to write in a text editor first and then paste into the terminal to execute the command. We can call the file validation.js to save the collection validation configuration. Visual Studio/Atom/Sublime or any other text editor/ IDE will help with auto-formatting. Visual Studio has a option under code > Preference > Keyboard Shortcuts and then you can search for a shortcut command such as format document (shortcut is Shift + Option + F on a Mac).

We can now validate the incoming data when we explicitly create the new collection. We can copy the command from the text editor and paste it back into the shell and run the command to create the new collection with all our validation setup. This will return { “OK” : 1 } in the shell if the new collection is successfully created.

If a new insert/update document fails the validation rules, the new document will not be added to the collection.

Changing the Validation Action

As a database administrator we can run the following command:

```
$ db.runCommand( { collMod: “post”, validator: {…}, validationAction: “warn” } )
```

This allows us to run administrative commands in the shell. We pass a document with information about the command we wish to run. For example, in the above we run a command called collMod which stands for collection modifier, whereby we pass in the collection name and then we can pass in the validator along with the whole schema.

We can amend the validator as we like i.e. add or remove validations. In the above we added another administrative command after the validator document as a sibling called validationAction.

The validationLevel controls whether all inserts and updates are checked or only updates to elements which were valid before. The validationAction on the other hand will either throw an “error” and stope the insert/update action or “warn” of the error but allow the insert/update to occur. The warn would have written a warning into our log file and the log file is stored on our system. We can update the validation action later using the runCommand() method as seen above.

Conclusion

Things to consider when modelling and structuring our Data.

- **In which format will we fetch your data?**

  How does the application or data scientists need the data? We want to store the data in a way that it is easy to fetch especially in a use case where we would fetch a lot.

- **How often will we fetch and change the data?**

  Do we need to optimise for writes or reads? It is often for reads but it may be different depending on the scenario. If we write a lot then we want to avoid duplicates. If we read a lot then maybe some duplicates are OK, provided these duplicates do not change often.

- **How much data will we save (and how big is it)?**

  If the data is huge, maybe embedding is not the best choice.

- **How is the data related (one to one, one to many, many to many)?**

- **Will duplicate data hurt us (=> many Updates)?**

  Do we update our data a lot in which we have to update a lot of duplicates. Do we have snapshot data where we do not care about updates to the most recent data.

- **Will we hit the MongoDB data/storage limit (embed 100 level deep and 16mb per document)?**

Modelling Schemas

- Schemas should be modelled based on application needs.
- Important factors are: read and write frequencies, relations, amount (and size) of data.

Schema Validation

- We can define rules to validate inserts and updates before writing to the database.
- Choose the validation level and action based on the application requirements.

Modelling Relations

- Two options: embedded documents or references.
- Use embedded documents if we have one-to-one or one-to-many relationships and there are no app or data size reasons to split the data.
- Use reference if data amount/size or app needs require it or for many-to-many relations.
- Exceptions are always possible — keep the app requirements in mind!

Useful Articles & Documents:

https://docs.mongodb.com/manual/reference/limits/

https://docs.mongodb.com/manual/reference/bson-types/

https://docs.mongodb.com/manual/core/schema-validation/

Setting dbpath & logpath

In the terminal we can run the following command to see all the available options for our mongoDB server:

```
$ mongo --help
```

This command will provide a list of all the available options we can use to setup/configure our mongoDB server. For example the --quiet option allows us to change the way things get logged or output by the server.

Note: use the official document on the MongoDB website for more detailed explanation of all the available options.

The --dbpath arg and --logpath arg allows us to configure where the data and log files gets stored to because mongoDB writes our data to real files on our system. The logs allows us to see for example warnings of json schema validation as we seen in the last section.

We can create folders such as db and logs (these can be named as anything we want) and have

these folders located anywhere we want for example we could create it within the mongoDB directory which contains the bin folder and other related files.

If we start using mongod instance without any additional settings, it will use the root folder that has a data/db folder to store all our database records as a default setting. However, we can use the settings above to tell mongod to use another folder directory to store our data, the same is true for our logs.

When we start the instance of our mongoDB server, we can run the following command and passing in the options to declare the path of the dbpath and logpath as seen below:

Mac/Linux:

```
$ sudo mongod --dbpath /Users/userName/mongoDB/db
```

Windows command:

```
$ mongod --dbpath \Users\userName\mongoDB\db
```

Enter our password and this should bring up our mongoDB server as we have seen previously. We should now see in the db folder, mongoDB has created a bunch of files as it is now saving the data

in the specified folder that we passed into our command. This is now using a totally different database storage for writing all our data which is detached from the previous database storage of the default database path. Running the following command will also work for our logs:

Mac/Linux:

```
$ sudo mongod --dbpath /Users/userName/mongoDB/db --logpath /Users/userName/mongoDB/logs/logs.log
```

Windows command:

```
$ mongod --dbpath /Users/userName/mongoDB/db --logpath \Users\userName\mongoDB\logs\logs.log
```

The logs folder path requires a log file which we would define with a .log extension. This will automatically create and add a logs.log file within the directory path if the file does not exist when we run the command. All the output in the terminal will now be logged in the logs.log file compared to previously where it was logged in the terminal shell. This file can be reviewed for persistent and auditing of our server and viewing any warnings/errors.

This is how we set custom paths for our database and log files.

Exploring the mongoDB Options

If we explore the different options in mongoDB using the mongod --help command in the terminal, there are many setup options available to us.

The WiredTiger options is related to our storage engine and we could either use the default settings or change some configurations if we know what we are doing.

We have useful commands such as --repair which we could run if we have any issues connecting or any warnings or issues related to our database files being corrupted. We could use the command --directoryperdb which will store each database in its own separate directory folder.

We could change the storage engine using the —storageEngine arg command, which by default is set to WiredTiger. Theoretically, mongoDB supports a variety of storage engines but WiredTiger is the default high performance storage engine. Unless, we know what we are doing and have a strong reason to change the engine, we should stick to the default.

There are other settings in regards to security which we will touch in the latter chapters.

MongoDB as a Background Service

In the mongoDB options, there is an option called --fork which can only run on Mac and Linux.

```
$ mongod --fork --logpath /Users/userName/mongoDB/logs/logs.log/
```

The above fork command will error if we do not pass in a logpath to the log file. This command will start the mongoDB server as a child process. This does not block the terminal and we can continue to type in other commands in the same terminal with the server running. The server is now running as a background process instead of a foreground process which usually blocks the terminal window. In other words the mongoDB server is now running as a service (a service in the background). Therefore, in the same terminal we could run the mongo command to connect to the background mongoDB server service. This is also the reason why we require to pass in a logpath because the service is running in the background and it cannot log error/warning messages in the terminal, instead it will use/write the warning and errors in the log file.

On Windows, the fork command in unavailable. However, on Windows we can still startup mongoDB server as a service if we checked this option at the installation process. If we right click on command prompt and run as administrator, we can run the following command:

```
$ net start MongoDB
```

This will start up the mongoDB server as a background service. The question then becomes, how do we stop such a service?

On Mac we can stop the service by connecting to the server with the mongo shell and then switching to the admin database and running the shutdown server command to shut down the server we are connected to. Example commands below:

```
$ use admin
$ db.shutdownServer()
```

The exact same approach as the above will work on Windows. On Windows we also have an alternative method by opening the command prompt as administrator and running the following command:

```
$ net stop MongoDB
```

This is how we can use MongoDB server as a background service (instead of a foreground service) on either Mac, Linux or Windows.

Using a Config File

Now that we have seen the various options we can set and use to run our mongoDB server, it is also worth noting that we can save our settings in a configuration file.

https://docs.mongodb.com/manual/reference/configuration-options/

This file could be automatically created for us when we run our mongoDB server, else we could create the config file ourselves and save this anywhere we want. We could create the config file within the Users/userName/MongoDB/bin folder using a text editor such as VS Code to add the configuration code:

```
storage:
    dbPath: "/Users/userName/mongoDB/db/"
systemLog:
    destination: file
    path: "/Users/userName/mongoDB/logs/logs.log/"
```

We can look at the documents or google search for more comprehensive con gif file setup.

Once we have the config file setup, how do we use the config file when we run an instance of the mongoDB server? MongoDB does not automatically pickup this file when we start to run the mongoDB server, instead when starting mongoDB we can use the following command to specify the config file the server should use:

```
$ sudo mongod --config /Users/userName/mongoDB/bin/mongod.cfg
$ sudo mongod -f /Users/userName/mongoDB/bin/mongod.cfg
```

Either above command will prompt mongoDB to use the config file from the path specified. This will start the mongoDB server with the settings setup in the configuration file. This is a useful feature because it allows us to save a snapshot of our settings (reusable blueprint) in a separate file which we can always use when starting up our mongoDB server. This also saves us time on writing a very long command prompt with all our settings when starting up our mongoDB server each time.

Important Note: we could use either .cfg or .conf as the file extension name when creating the mongoDB configuration file.

Shell Options & Help

In this section we will go over the various shell options available to for us to use. Similar to the mongoDB server, there is a help option for the mongoDB shell:

```
$ mongo --help
```

This will provide all the command options for the shell. This has less options compared to the server because the shell is just a connecting client at the end of the day and not a server. We can use the shel without connecting to a database (if we just want to run javascript code) using the --nodb command, or we could use the --quiet command to have less output information in the terminal, we can define the port and host for the server using the --port arg and --host arg commands(by default it uses localhost:27017) and many more other options.

We can also add Authentication Options informations which we will learn more in later chapters.

In the shell we also have another command we can run:

```
$ help
```

This command will output a shortlist of some important help information/commands we can execute in the shell. We can also dive deeper into the help by running the help command followed by the command we want further help on, for example:

```
$ help admin
```

This will show further useful commands that we can execute when using the admin command e.g. admin.hostname() or admin.pwd() etc.

We can also have help displayed for a given database or collection in a database. For example:

```
$ use test
$ db.help()
```

We would now see all the commands that we did not see before that we can use on the new “test” database. We can also get help on the collection level which will provide a list of all th commands we can execute at the collection level.

```
$ db.testCollection.help()
```

Useful Links:

https://docs.mongodb.com/manual/reference/configuration-options/

https://docs.mongodb.com/manual/reference/program/mongo/

https://docs.mongodb.com/manual/reference/program/mongod/

Exploring MondoDB Compass

We can download MongoDB Compass from the below link:

https://www.mongodb.com/products/compass

This is a GUI tool to interact with our MongoDB database. Once downloaded and installed on our machines we are ready to use the GUI tool. It is important to have the mongod server running in the background when we open the MongoDB Compass to connect to the database. We would connect to a Host and this by default will have localhost and port 27017. We can click connect and this will connect the GUI tool to the mongod server. We should be able to see the 3 default databases of admin, config and local.

We can now use the GUI tool to create a new database and collection name. Once a database and collection has been created we can then insert documents to the collection. We can also query our database documents.

We can now start using a GUI tool to interact with our database, collections and documents.

Note: it is best practice to learn how to use the shell first before using GUI tools.

Understanding insert() Methods

We already understand that there are two methods for inserting documents into mongoDB which are insertOne() and insertMany() as an alternative. The most important thing to note is that insertOne() takes in a single document and we can but do not need to specify an id because we will get one automatically. The insertMany() does the same but with an array (list) of documents.

There is also a third alternative method for inserting documents called insert() – below is an example:

```
$ db.collectionName.insert()
```

This command is more flexible because it takes both a single document or an array of documents. Insert was used in the past but insertOne and insertMany was introduced on purpose so that we are more clear about what we will be inserting. Previously, in application code it was difficult to tell with the insert command whether the application was inserting a single or multiple documents and therefore may have been error prone.

There is also an importing data command as seen below:

```
$ mongoimport -d cars -c carsList --drop –jsonArray
```

The insert method can still be used in mongoDB but it is not recommended. The insert() method works with both a single document and multiple documents as seen in the examples below:

$ db.persons.insert({ name: “Annie”, age: 20 })

$ db.persons.insert([{ name: “Barbara”, age: 45 }, { name: “Carl”, age: 65 }])

The output message in the terminal is also slightly different i.e. we would receive a text of:

$ WriteResult({ “nInserted” : 1 })

$ BulkWriteResult({ “writeErrors”: [], “writeConcernErrors”: [], “nInserted”: 2, “nUpserted”: 0, “nMatched”: 0, “nModified”: 0, “nRemoved”: 0, “upserted”: [] })

The above does not mean that the inserted document did not get an autogenerated id. The insert method will automatically create an ObjectId but will not display the ObjectId unlike the insertOne and insertMany commands output messages which does display the ObjectId. We can see the advantages of insertOne and InsertMany as the output message is a little more meaningful/helpful as we can immediately work with the document using the ObjectId provided (i.e. we do not need to query the database to get the new document id).

Working With Ordered Inserts

When inserting documents we can define or specify some additional information. Lets look at an example of a hobbies collection where we keep track of all the hobbies people could possibly have when we insert many hobbies. Each hobby is a document with the name of the hobby:

```
$ db.hobbies.inertMany( [ { _id: “sports”, “name”: “Sports” }, { _id: “cooking”, “name”: “Cooking” }, { _id: “cars”, “name”: “Cars” } ] )
```

The id’s for these hobbies can be auto-generated. However, there may be times when we want to use our own id because the data may have been fetched from some other database where we already have an existing id associated or maybe we need a shorter id. We can use _id and assign a value for the id. In the above the hobby name could act as a good id because each hobby will be unique. We must use _id and not just id if we want to set our own id for our documents. Furthermore, the id must be unique else this would not work. We will no longer see an ObjectId() for these documents as we have used the _id as the unique identifier for the documents inserted.

If we try to insert a document with the same id we would receive an error message in the terminal referencing the index number (mongoDB uses zero indexing) of the document that failed the insert operation along with a description of duplicate key error.

```
$ db.hobbies.inertMany( [ { _id: “yoga”, “name”: “Yoga” }, { _id: “cooking”, “name”: “Cooking” }, { _id: “hiking”, “name”: “Hiking” } ] )
```

The above would fail due to the duplicate key error of cooking which was inserted previously in the above command. However, we would notice on the first item in the insertMany array i.e. Yogo will be inserted into the hobbies collectio, but the cooking and hiking documents will not be inserted into the collection due to the error. This is the default behaviour of mongoDB and this is called an ordered insert.

An ordered insert simply means that every element we insert is processed standalone, but if one fails, it cancels the entire insert operation but does not rollback the elements it has already inserted. This is important to note because it cancels the operation and does not continue to the next document (element i.e. hoking) which we would have known that it would have succeeded insert.

Often we would want this default behaviour, however, sometimes we do not. In these cases, we could override the behaviour. We would pass in a second argument, separated by a comma, to the insertMany command which is a document. This is a document that configures the insertMany operation.

```
$ db.hobbies.inertMany( [ { _id: “yoga”, “name”: “Yoga” }, { _id: “cooking”, “name”: “Cooking” }, { _id: “hiking”, “name”: “Hiking” } ], { ordered: false} )
```

The ordered option allows us to specify whether mongoDB should perform an ordered insert which is the default (*we could set this ordered option to true which is redundant because this is the default option*) or we could set this option to false which will make the insert operation not an ordered insert i.e. an unordered insert.

If we hit enter, we would still get a list of all the error, however, it will continue to the next document to perform the insert operation and this would insert the document that does not have any issues of duplicate keys i.e. hiking will now be inserted into the hobbies collection (yoga and cooking would fail due to the duplicate key issue).

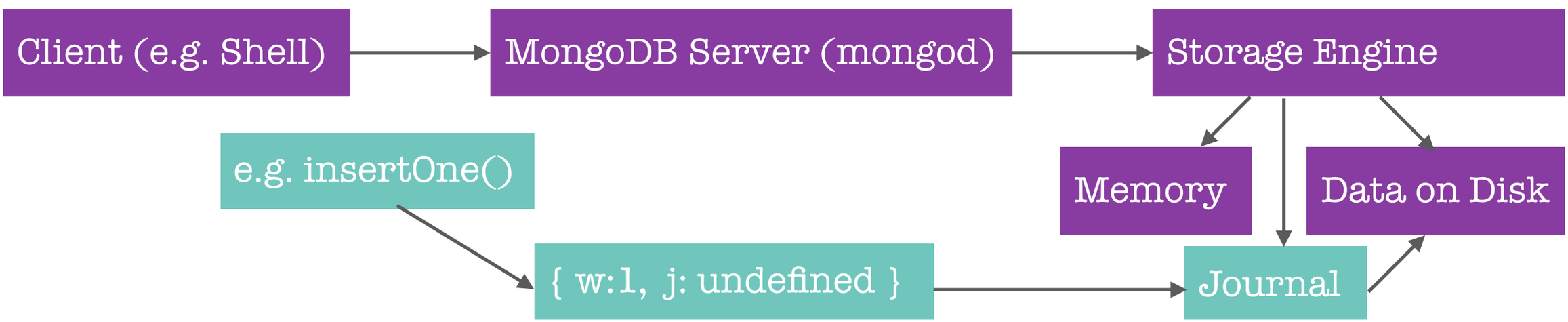
By setting the ordered to false, we have changed the default behaviour and it is up to us to decide what we require or want in our application. It is important to note that this will not rollback the entire insert operation if something failed. This is something we will cover in the Transactions chapter. We can control whether the operation continues with the other documents and tries to insert everything that is perfectly fine.

We may use an unordered insert where we do not have much control with what is inserted into the database but we do not care about any document that fail because they already exist in the database. We could add everything that is not in the database.

Understanding the writeConcern

There is a second option we can specify on insertOne and insertMany which is the writeConcern option. We have a client (either the shell or the application using a mongoDB server) and we have our mongoDB server. If we wanted to insert one document in our mongoDB server, on the mongoDB server we have a so called storage engine which is responsible for really writing our data onto the disk and also for managing it in memory. So our write might first end up in memory and there it manages the data which it needs to access with high frequency because memory is faster than working with the disk. The write is also scheduled to then end up on the disk, so it will eventually store data on the disk. This is true for all write operations i.e. insertMany and update.

We can configure a so-called writeConcern on all the write operations with an additional argument, the writeConcern which is another document where we can set settings.



The w: (default) option tells the mongoDB server of how many instances we want the write to be acknowledged. The j: option stands for journal which is an additional file which the storage engine manages, which is like a To-Do file. The journal can be kept to then for example perform save operations that the storage engine needs to do but have not been completed yet.

The storage engine is aware of the write and that it needs to store the data on disk just by having the write being acknowledged and being in memory. The idea behind a journal file is to make the storage engine aware of this and if the server should go down for some reason or anything else should happen, the journal file is there. If we restart the server or if the server recovers, the server can look to this file and see what it needs to do. This is a nice backup because the memory might have been wiped by then. The journal acts as a backup to-do list for the storage engine.

Writing into the database files is more performance heavy whereas a journal is like a single line which describes the write operations. Writing into the database is of course a more complex task because we need to find the correct position to insert the data and if we have indexes we also need to update these as well and therefore takes longer to perform. Writing in a to-do type list is much quicker.

We can set the j: true as an option which will now report a success for a write operation when it has been acknowledged and has been saved to the journal. This will provide a greater security.

There is a third option to the writeConcern which is the wtimeout: option. This simply sets the timeframe that we give out server to report a success for the write before we cancel it. For example, if we have some issues with the server connection or anything of that nature, we may simply timeout.

If we set the timeout value to a very low number, we may get more fails even though there is no actual problem, just some small latency.

```
{ w:1, j: undefined }
```

```
{ w:1, j: true }
```

```
{ w:1, timeout: 200, j: true }
```

This is the writeConcern option we can add to our write operations and how we can control this using the above document settings. Enabling the journal would mean that our writes will take longer because we do not only tell the server about the write operation but we also need to wait for the server to store the write operation in the journal, however, we get higher security that the write also succeeded. These option will again depend on our application needs.

The writeConcern in Practice

Below is an example of using the writeConcern:

```
$ db.persons.insertOne( { name: “Alan”, age: 44 }, { writeConcern: { w: 1, j: true } } )
```

The w:1 (default) simply means to make sure the server acknowledged the write operation. Note we could set this value to 0 which will return {“acknowledged”: false} in the terminal when we insert the document. This option sends the request and immediately returns without waiting for a response of the request from the server. The storage engine had no chance of storing it in memory and to generate an objectId, hence why we receive {“acknowledged”: false} in the terminal. This makes the write super fast because we do not have to wait for any response but we do not know where the write succeeded or not.

The journal by default is set to undefined or false. We can set this option to j: true. The output in the terminal does not change. The write will be slightly slower (*note if playing around locally we would not notice any change in speed*) because the engine would add the write to the journal and we would have to wait for that to finish before the operation is completed. This will provide a higher security by ensuring the write appears in the to-do list of the storage engine which will eventually

lead to the write operation occurring on the database.

Finally, the wtimeout: option is used to set a timeout operation. This allows us to set a time frame for the write operation so in the case where within a certain period within the year we would have shaky connections, we would rather have the write operation fail and we recognise it in our client application (we would have access to the error) and therefore try again at a later time without having to wait unnecessarily for the write operation where we have shaky connections.

Diving Into Create Operation

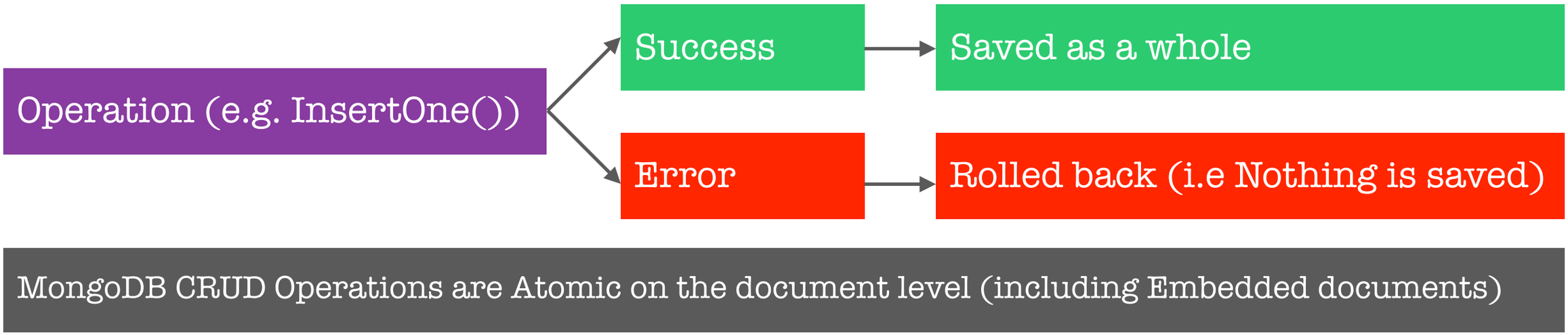
What is Atomicity?

Atomicity is a very important concept to any write operation. Most of the time the write operation e.g. InsertOne() would succeed, however, it can fail (there can be an error). These are errors that occur whilst the document is being inserted/written to memory (i.e. whilst being handled by the storage engine). For example, we were writing a document for a person including name, age and an array of hobbies, the name and age were written but then the server had issues and was not able to write the hobbies to memory. MongoDB protects us against this as it guarantees us an atomic transaction. This means the transaction either succeeds as a whole or it fails as a whole. If it fails during the write, everything is rolled back for the document we inserted.

This is important as it is on a per document level. The document means the top level document, so it includes all embedded documents and all arrays.

If we insertMany() where there are multiple documents being inserted into the database and the server fails during a write, we do not get atomicity because it only works at a document level. If we have multiple documents in one operation like the insertMany() operation, then only each document on its own is guaranteed to either fail or succeed but not on insertMany. Therefore, if we have issues during the insertMany operation, only the documents that failed are not inserted and then the exact behaviour will depend on whether we used ordered or unordered inserts but the document already inserted will not be rolled back.

We are able to control this on a bulk insert or bulk update level using a concept called transactions which we will look at in a later section as it requires some additional knowledge about mongoDB and how the service works.



Importing Data

To import data into our database, we must first exit the shell by pressing the control + c keys on our keyboard.

In the normal terminal, we need to navigate to a folder that contains the JSON file that we would want to import (JSON files can be imported) using the cd command. We can use the ls command to view the list of items within the directory we are currently in.

Once navigated to the folder containing the import file, we can run the following command:

```
$ mongoimport tv-shows.json -d moviesData -c movies --jsonArray --drop
```

The mongoimport command should be globally available since we added the path to our mongo binary to our path variables on our operating systems. If we did not do this, we need to navigate into the folder where our mongoDB binaries are in order to execute the mongoimport command above.

The first argument we pass is the name of the file we want to import (if we are not in the path of the located file we would have to specify the full folder path along with the file name). We then specify the database we want to import the data into using the -d flag. We can also specify the collection by using the -c flag.

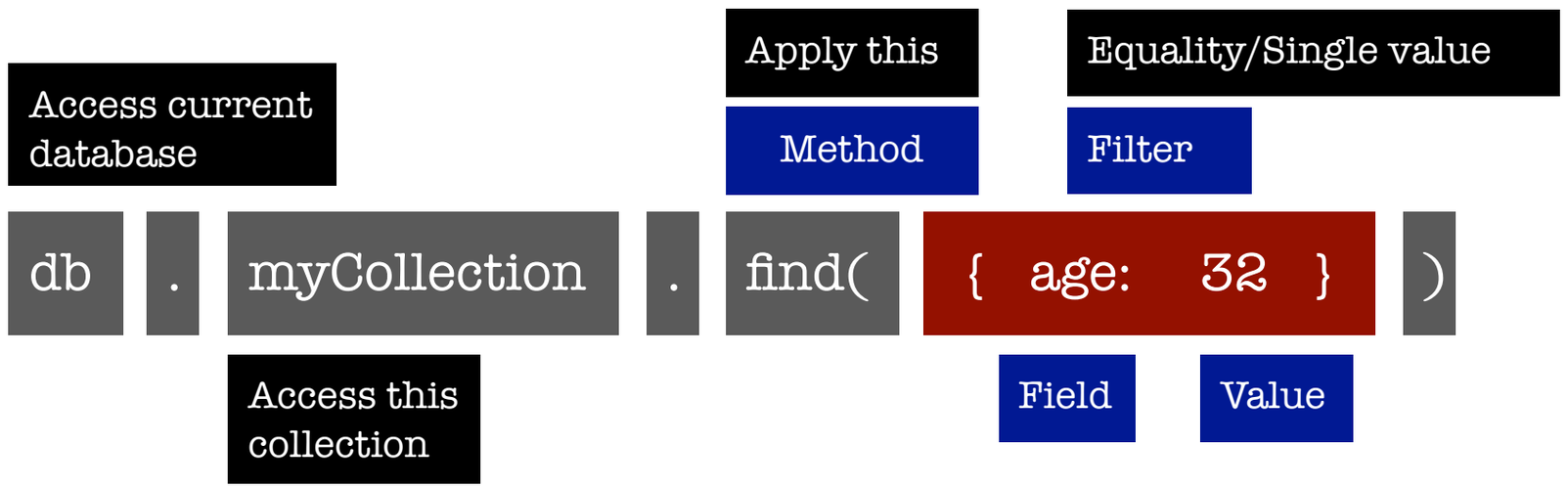
If the JSON file holds array of documents we must also specify the --jsonArray flag to make the mongoimport command aware of this fact about the import data.

The last argument option we can add to the import command is the --drop which will tell the mongoimport that should this collection should already exist, it will be dropped and then re-added, otherwise it will append the data to the existing collection.

Important Note: the mongod server should be running in the background when we use the import command. When we press enter to execute the command, this will return in the terminal the connected to: localhost, dropping: moviesData.movies and imported: # documents in the terminal as a response to inform us which mongoDB server it is connected to, whether a collection was dropped/deleted from the database collection and the total number of data imported into the database collection.

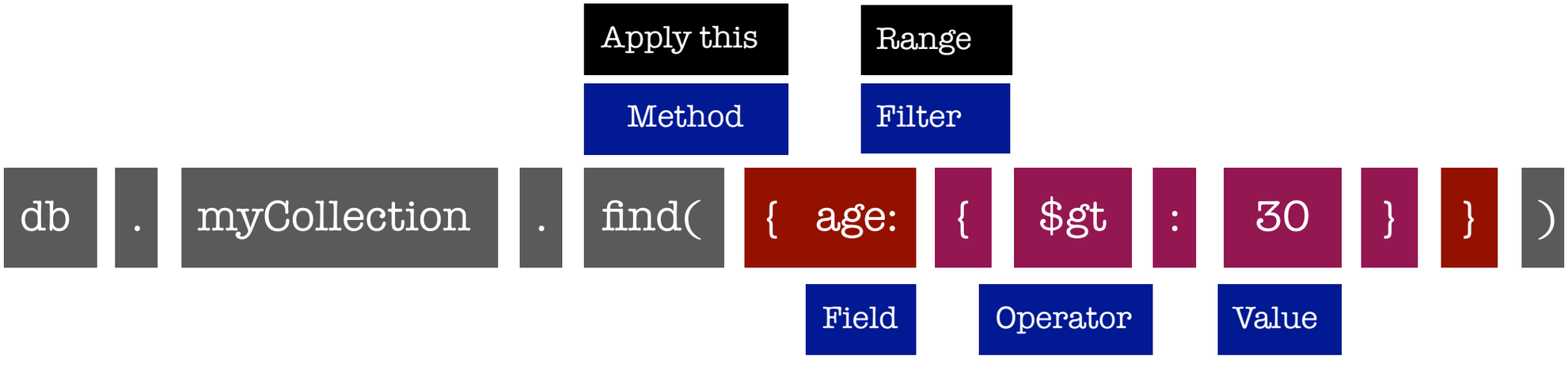
Methods, Filters & Operators

In the shell, we access the database with the db command (this will differ slightly in a mongoDB drive). We would get access to a database and then to a collection in the database. Now we can execute a method like find, insert, update or delete on the collection. We would pass some data into the method as parameters/arguments for the method. These are usually a key:value pair where one is the field and the other is the value for that field name (documents are all about field and values or key and values).



The argument in the above example happens to also be a filter because the find method accepts a filter. It can use a filter to narrow down the set of documents it returns to us. In the above we have a equality or single value filter where the data is exactly the criteria i.e. equality.

We can also use more complex filters as seen in the below example. We have a document which has a field and its value is another document which has an operator as a filed followed by a value.



We can recognise operators by the dollar sign $ at the beginning of the operator. These are all reserved fields which are understood by mongoDB. The operator in the above example is called a range filter because it does not just filter for equality, instead this will look for all documents that have an age that is greater than ($gt) the value i.e. 30.

this is how the Read operator works and we will look at various different operators an the different ways of using them and the different ways of filtering the data that is returned to us. This is the structure we should familiarise ourselves with for all of our Read operations.

Operators and Overview

There are different operators that we can differentiate into two groups:

- Query Selectors
- Projection Operators

Query selectors such as $gt allows us to narrow down the set of documents we retrieve while projection operators allows us to transform/change the data we get back to some extent. Both the Query and Projection operators are Read related operators.

Aggregation allows us to read from a database but also perform more complex transforms. This concept allows us to setup pipeline of stages to funnel our data through and we have a few operators that allow us to shape the data we get back to the form we need in our application.

- Pipeline Stages
- Pipeline Operators

The Update has operators for the fields and arrays. Inserts have no operators and deletes uses the same operators as the Read operators.

How do operators impact our data?

Type	Purpose	Change Data?	Example
Query Operator	Locate Data	No	$eq $gt
Project Operator	Modify data Presentation	No	$
Update Operator	Modify & Add additional data	Yes	$inc

Query Selectors & Projection Operators

There are a couple of categories for Query Selectors:

Comparison, Logical, Element, Evaluation, Array, Comments & Geospatial.

Projections Operators we have:

$, $elemMatc, $meta & $slice

Understanding findOne() and find()

The findOne() method finds exactly one document. We are able to pass in a filter into the method to define which one document to return back. This will find the first matching document.

```
$ db.movies.findOne( )
$ db.movies.findOne( { } )
```

Both of the above syntax will return the first document within the database collection. Note, this does not return a cursor as it only returns one document.

The alternative to findOne() is the find() method. The find() method will return back a cursor. This method theoretically returns one document, but since it provides us a cursor, it does not give us all the document but the first 20 documents within the shell.

```
$ db.movies.find( )
```

To narrow the find search we would need to provide a filter. To provide a filter we would pass in a document as the first argument (this is true for both find and findOne methods). The difference would be that findOne will return the first document that meets the criteria while the find method

will return all documents that meet the criteria.

```
$ db.movies.findOne( { name: “The Last Ship” } )
$ db.movies.findOne( { runtime: 60 } )
$ db.movies.find( { name: “The Last Ship” } )
$ db.movies.find( { runtime: 60 } )
```

To filter the data, we would specify the name of the field/key followed by the value we are expecting to filter the field by. In the above example we are filtering the name of the movie to be “The Last Ship”. By default mongoDB will try to find the filter by equality.

This is the difference between find and findOne and how we would pass in a filter to narrow down the return read results. It is important to note that there are way more operators and ways to filtering our queries to narrow down our Read results when using either of the find commands.

Diving Into Read Operation

Working with Comparison Operators

In the official documentation we can view all the various operations available to us:

https://docs.mongodb.com/manual/reference/operator/query/

We will explore some of the comparative operators in the below examples:

```
$ db.movies.find( { runtime: 60 } ).pretty( )
$ db.movies.find( { runtime: { $eq: 60 } } ).pretty( )
```

The $eq operator is the exact same as the default equality query which will find the document that matches equally to the query value which in the above case is runtime = 60.

```
$ db.movies.find( { runtime: { $ne: 60 } } ).pretty( )
```

This will return all documents that have a runtime not equal to 60.

```
$ db.movies.find( { runtime: { $gt: 40 } } ).pretty( )
$ db.movies.find( { runtime: { $gte: 40 } } ).pretty( )
```

The $gt return all documents that have a runtime greater than to 40 while the $gte returns greater than or equal to 40.

```
$ db.movies.find( { runtime: { $lt: 40 } } ).pretty( )
$ db.movies.find( { runtime: { $lte: 40 } } ).pretty( )
```

The $lt return all documents that have a runtime less than to 40 while the $lte returns less than or equal to 40.

Querying Embedded Fields & Arrays

We are not limited to querying top level fields and are also able to query embedded fields and arrays. To query embedded fields and arrays is quite simple as demonstrated below:

```
$ db.movies.find( { “rating.average”: { $gt: 7 } } ).pretty( )
```

We specify the path to the field that we are interested in querying the data. We must put the path within quotations marks because we use the dot (which will invalidate the syntax) to detail each embedded field within the path that leads to the field we are interested in. The above example is a single level embedded document, if we wrote e.g. rating.total.average, this is a 2 level embedded document. We can make the path as deep as we need it to be and we are not limited to one level.

We can also query arrays as seen below:

```
$ db.movies.find( { genres: “Drama” } ).pretty( )
$ db.movies.find( { genres: [“Drama”] } ).pretty( )
```

The casing is important in the query. This will return all genres thats has Drama included in it. Equality in an array does not mean that Drama is the only item within the array; it means that Drama exists within the array. If we wanted an exactly only Drama within the array we would use square brackets. We can also use dot to go down embed level paths that has an array.

Understanding $in and $nin

If there are two discrete values that we wish to check/query our data, for example runtime that is either 30 and/or 42, we can use the $in operator. The $in operator takes in an array which holds all the values that will be accepted to be values within the key/field.

The below example return data that have a runtime equal to 30 or 42.

```
$ db.movies.find( { runtime: { $in: [ 30, 42 ] } } ).pretty( )
```

The $nin on the other hand is the opposite to $in operator. It finds everything where the value is not within the set of values defined in the square brackets. The below example returns all entries where the runtime is not equal to 30 or 42.

```
$ db.movies.find( { runtime: { $nin: [ 30, 42 ] } } ).pretty( )
```

We have now explore all the Comparison operators within mongoDB and will continue to now look at logical query operators such as $and, $not, $nor and $or in the next section.

Understanding $or and $nor

There are four different logical operators and these are $and, $not, $nor and $or operators. We would probably use the $or logical operator more compared to the other logical operators. Below is an example of the $or and $nor operator in action.

```
$ db.movies.find( { $or: [ {“rating.average”: { $lt: 5 } }, {“rating.average”: { $gt: 9.3 } } ] } ).pretty( )
$ db.movies.find( { $or: [ {“rating.average”: { $lt: 5 } }, {“rating.average”: { $gt: 9.3 } } ] } ).count( )
```

We start the filter with the $or to tell mongoDB that we have multiple conditions and then add an array which will hold all the conditions that mongoDB will check. The or logical condition means that it will return results that match any of these conditions. We would specify our filters as we would normally would do within our find but now held within the $or array. We can have many expressions as we want within the $or array, in the above we have two conditions. Note: if we change pretty( ) for count( ), this will return the total number of documents that meet the criteria rather than the document itself.

```
$ db.movies.find( { $nor: [ {“rating.average”: { $lt: 5 } }, {“rating.average”: { $gt: 9.3 } } ] } ).pretty( )
```

The $nor operator is the opposite/inverse of the $or operator. It returns where neither of the conditions are met i.e neither conditions are true the complete opposite.

Understanding $and Operator

The syntax for the $and operator is similar to the $or and $nor operator. The array of documents acts as the logical conditions and will return all documents where all conditions are met. We can have as many conditions as we want. Below is an example of the $and logical operator:

```
$ db.movies.find( { $and: [ { “rating.average”: {$gt: 9} }, { genres: “Drama” } ] } ).pretty( )
```

In this example, we are trying to find all documents that are Drama with a high rating that is greater than 9. This is the old syntax and there is now a shorter syntax as seen below:

```
$ db.movies.find( { { “rating.average”: {$gt: 9}, genres: “Drama” } ).pretty( )
```

The new shorter syntax does not require the $and operator, instead we use a single document and write our conditions separating each condition with a comma. By default, mongoDB ands all key fields that we add to the filtered document. The $and is the default concatenation for mongoDB.

The reason why we have the $and operator is because not all drivers accepts the above syntax. Furthermore, the above shorthand syntax would return a different result when we filter on the same key elements.

If we examine the two syntax below, we would notice that they both return a different result.

```
$ db.movies.find( { $and: [ { genre:"Drama" }, { genres: "Horror" } ] } ).count( )
$ db.movies.find( { { genre: "Drama", genres: "Horror" } ).count( )
```

When we use the second syntax, mongoDB replaces the first condition with the second and therefore it is the same as filtering for Horror genre only.

```
$ db.movies.find( { { genre: "Drama", genres: "Horror" } ).count( )
$ db.movies.find( { genres: "Horror" } ).count( )
```

Therefore, in the scenario where we are looking for both the Drama and Horror values from the genre key element, it is recommended to use the $and operator for mongoDB to look for both conditions to return true.

Understanding $not Operator

The $not operator inverts the effect of a query operator. For example if we query to find movies that do not have a runtime of 60minutes as seen in the below syntax.

```
$ db.movies.find( { runtime: { $not: { $eq: 60 } } } ).count( )
```

The $not operator is less likely to be use as it can be achieved using much simpler alternatives for example we can use the not equal operator or $nor operator:

```
$ db.movies.find( { runtime: { $ne: 60 } } ).count( )
$ db.movies.find( {$nor [ { runtime: { $eq: 60 } } ] } ).count( )
```

There are a lot of ways for querying the inverse, however, where we cannot just simply inverse the query in another way, we have the $not which we can use to look for the opposite.

We have now examined all four of the logical operators available within mongoDB that we can use as filters for our Read operations.

Element Operators

There are two types of element operators which are $exists and $type. This allows us to query by elements within our database collection.

```
$ db.users.find( { age: { $exists: true } } ).pretty( )
```

This will check within our database and return all results where the document contains an age element/field. Alternatively, we could have queried $exists to be false in order to retrieve all documents that do not have age as an element/field.

We can query the $exists operator with other operators. In the below example we are filtering by the age element to exist and age is greater than or equal to 30:

```
$ db.users.find( { age: { $exists: true, $gte: 30 } } ).pretty( )
```

To search for a field that exists but also has a value in the field, we would query as seen below:

```
$ db.users.find( { age: { $exists: true, $ne: null } } ).pretty( )
```

The $type operator on the other hand, as the name would suggests, returns the document that have

The specified field element of the specified data type.

```
$ db.users.find( { phone: {$type: “number”} } ).pretty( )
```

The example above returns documents where the phone field element has values of the data type number. Number is an alias that basically sums up floats and integers. If we searched for the type of double this would also return back a document even if there are no decimal places. Since the shell is based on JavaScript, by default, a number inserted into the database will be stored as a floating point number/double because JavaScript which drives the shell does not know the difference between integers and doubles as it only knows doubles. The shell takes the number and stores it as a double even though if we have no decimal places. This is the reason why we could also search by the type of double and retrieve the documents as well.

We can also specify multiple types by passing an array. The below will look for both data types of a double and a string and return documents that match the filter condition:

```
$ db.users.find( { phone: {$type: [ “double”, “string” ] } } ).pretty( )
```

We can use the type operator to ensure that we only work with the right type of data when returning some documents.

Understanding Evaluation Operators - $regex

The $regex operator allows us to search for text. This type of query is not super performant. Regex stands for regular expression and it allows us to search for certain text based on certain patterns. Regular expressions is a huge complex topic on its own and is something not covered deeply within this mongoDB guide. Below is an example of using a simple $regex operator.

```
$ db.movies.find( { summary: { $regex: /musical/ } } ).pretty( )
```

In this example the query will look at all the summary key field values to find the word musical contained in the value and return all matching results.

Regex is very useful for searching for text based on a pattern, however, it is not the most efficient/ performant way of searching/retrieving data (the text index may be a better option and we will explore this in later chapters).

Understanding Evaluation Operators - $expr

The $expr operator is useful if we want to compare two fields inside of one document and then find all documents where this comparison returns a certain result. Below is an example code:

```
$ use financialData
$ db.sales.insertMany( [ { volume: 100, target: 120 }, { volume: 89, target: 80 }, { volume: 200, target: 177 } ] )
$ db.sales.find( { $expr: { $gt: [ "$volume", "$target" ] } } ).pretty( )
```

In the above $expr (expression) we are retrieving all documents where the volume is above the target. This is the most typical use case where we would use the expression operator to query the data in such a manner.

The $expr operator takes in a document describing the expression. We can use comparison operators like gt, lt and so on – more valid expressions and which operators we can use can be found in the official documentation. We reference fields in the array rather than the number, and these must be wrapped in quotation marks along with a dollar sign at the beginning. This will tell mongoDB to look in the field and use the value in the expression. This should return two documents that meet the expression criteria.

Below is another more complex expression example:

```
$ db.sales.find( { $expr: { $gt: [ { $cond: { if: { $gte: [“$volume”, 190 ] }, then: { $subtract: [“$volume”, 10 ] }, else: “$volume” } }, “$target” ] } } ).pretty( )
```

Not only are we comparing whether volume is greater than target but also where volume is above 190, the difference between volume and target must be at least 10. To achieve this we have to change the expression inside our $gt operator.

The first value will be a document where we use a special $cond operator for condition. The $cond works in tandem with the $expr operator. We are using an if: and then: to calculate the value dynamically. The if is another comparative operator. We are $subtracting 10 from the volume value for all the items that are greater than or equal to 190. We use an else: case to define cases that do not match the above criteria, and in this case we would just use the volume value.

We would finally compare the value with the target to check whether the value is still greater than or equal to the target. This should return 2 documents.

As we can see from the example above, this is a very powerful command within our tool belt when querying data from a mongoDB database.

Diving Deeper into Querying Arrays

There are multiple things we can perform when querying arrays and there are special operators that help us with queuing arrays. If we want to search for example all documents that have an embedded sports document, we cannot use the normal queries that we have previously used for example, if we had embedded documents for hobbies that had title and frequency:

```
$ db.users.find( { hobbies: “Sports" } ).pretty( )
```

```
$ db.users.find( { hobbies: { title: “Sports” } } ).pretty( )
```

Both of these will not return any results if there are multiple fields within an embedded document.

```
$ db.users.find( { hobbies: { title: “Sports”, frequency: 2 } } ).pretty( )
```

This will find any documents that meet both the criteria, however, what if we only want to retrieve all documents that have an embedded Sports document in an array, regardless of the frequency?

```
$ db.users.find( { “hobbies.title”: “Sports” } ).pretty( )
```

We search for a path using dot notation. This must be wrapped in quotation. MongoDB will go through all of the hobbies elements and within each element it will dig into the document and compare title to our query value of Sports. Therefore, this will retrieve the relevant documents even if within an embedded array and there are multiple array documents. This is how we would query array data using the dot notation.

Using Array Query Selector - $size, $all & $elemMatch

There are three dedicated query selectors for Arrays which are $size, $all and $elemMatch operators. We will examine each of these selectors and their applications.

The $size selector operator allows us to select or retrieve documents where the array is of a certain size, for example we wanted to return all documents that had an embedded array size of 3 documents. For example:

```
$ db.users.find( { hobbies:  { $size: 3 } } ).pretty( )
```

This will return all documents within the users collection where the hobbies array size is 3 documents. Note: the $size operator takes an exact match of a number value and cannot be something like greater or less than 3. This is something mongoDB does not support yet and we will have to retrieve using a different method.

The $all selector operator allows us to retrieve documents from an array based on the exact values without worrying about the order of the items within the array. For example if we had a movie collection where we wanted to retrieve those with a genre of thriller and action but without caring

for the order of the values, this is where the $all array selector will help us.

$ db.movies.find({ genre: [“action”, “thriller”] }).pretty( )

$ db.movies.find({ genre: { $all: [“action”, “thriller”] } }).pretty( )

The second syntax will ensure both array elements of action and thriller exists within the genre field and ignores the ordering of these elements (i.e. ordering does not matter) whereas, the first syntax would take the order of the elements into consideration (i.e. the ordering matters).

Finally, the $elemMatch array selector allows us to retrieve documents where one and the same element should match our conditions.

In the syntax below, this will find all documents where the hobbies has at least one document with the title of Sports and a document with a frequency greater than or equal to 3 and it does not have to be the same document/element. This would mean that a user who has the title of Sports but a frequency below 3 and a title of a different hobby but a frequency greater then 3, will match the criteria as it has at least one of each document criteria.

$ db.users({ $and: [{ “hobbies.title”: “Sports” }, { “hobbies.frequency”: { $gte: 3 } }] }).pretty( )

To ensure we retrieve all documents where the hobbies is Sports and its frequency is greater than or equal to 3 is returned, the $elemMatch operator is useful:

$ db.movies.find({ hobbies: {$elemMatch: { title: “Sports”, frequency: { $gte: 3 } } } }).pretty( )

Understanding Cursors

The find() method unlike the findOne() method yields us with a cursor object. Why is this cursor object important? If our client communicates with the mongoDB server and we potentially retrieve thousands if not millions of documents with our find() query depending on the scale of the application. Retrieving all these results is very insufficient because all these documents have to be fetched from the database, they have to be sent over the wire and then loaded into memory in the client application. These are the three things that are not optimal. In most cases we would not need all the data at the same time; therefore, find gives us a cursor.

A cursor is basically a pointer which has the query we wrote stored and can go back to the database as request the next batch. We therefore work with batches of data and we fetch documents over the wire one document at a time. The shell by default takes the cursor and retrieves the first 20 documents, it then fetches in batches of 20 documents.

If we write our own application with a mongoDB driver, we have to control that cursor manually to make sure that we get back our results. The cursor approach is beneficial because it saves on resources and we only load a batch of documents rather than all the documents from a query.

Applying Cursors

When using the find command in the shell this will display the first 20 documents. We can use the "it" command in shell to retrieve the next 20 batches of documents and keep using this command until we have exhausted the cursor i.e. there are no more documents to load. The "it" command will not work with the mongoDB drivers. Instead most drivers will have a next() method we can call instead. If we use the next() method within the shell, this will retrieve only one document and there will be no "it" command to retrieve the next document. If we run the command again this will restart the find query retrieving the first document again.

```
$ db.movies.find( ).next( )
```

Since the shell uses JavaScript, we can use JavaScript syntax to store the cursor value in a variable. We can then use the next() method the to cycle through the next document on the cursor which will retrieve the next document continuing on from the last cursor value.

```
$ const dataCursor = db.movies.find( )
$ dataCursor.next( )
```

There are other cursor methods available to us in mongoDB that we can use on our find() query.

```
$ dataCursor
$ it
```

This will return the first 20 batch of documents. Using the it command will retrieve the next 20 documents i.e. the default shell behaviour for cursors.

```
$ dataCursor.forEach( doc => { printjson(doc) } )
```

The forEach() method will vary on the driver we are using, but in JavaScript the forEach() method takes in a function that will be executed for every element that can be loaded through the cursor. In javascript we get a document which is provided by the forEach method which is passed in as the input and then our arrow function provides the body what we want to do. In the above example we are using the printjson() method which is provided by the shell to output the document.

This will cycle through all the remaining documents inside of the cursor (this will exclude any documents we have already searched for i.e. using the next() method or find() method). The forEach ill retrieve all the remaining documents and there will be no “it” command to fetch the next documents as we would have exhausted all the documents in the cursor.

```
$ dataCursor.hasNext( )
```

The hasNext() method will return true or false to indicate if we have/have not exhausted the cursor. We can create a new variable to reset the cursor for const variables or if we used let or var variables we can re-instantiate the original variable again to reset the cursor to the beginning.

We can learn more on cursors and the shell or the mongoDB drivers on the official mongoDB documentation:

https://docs.mongodb.com/manual/tutorial/iterate-a-cursor/

Diving Into Read Operation

Sorting Cursor Results

A common operation is to sort the data that we retrieve. We are able to sort by anything whether it is a string sorted alphabetically or a number sorted by numeric value. To sort the data in mongoDB we would use the sort() method on our find function. Below is an example:

```
$ db.movies.find( ).sort( { “rating.average”: 1 } ).petty( )
```

The sort takes in a document to describe how to sort the retrieved data. We can sort by a top level document field or an embedded document field. The values we use to sort describe the direction to sort the data i.e. 1 means ascending (lowest value first) and -1 means descending (highest value first). We are not limited to 1 sorting criteria for example we want to sort by the average ratings first but then we want to sort by the runtime:

```
$ db.movies.find( ).sort( { “rating.average”: 1, runtime: -1 } ).pretty( )
```

Skipping & Limiting Cursor Results

We are able to skip a certain amount of elements. Why would we want to skip elements? If on our application or web app we implement pagination where users can view results distributed across multiple pages (e.g. 10 elements per page) because we do not want to show all results on one page. If the user switches to a page e.g. page 3, we would want to skip the first 20 results to display the results for page 3.

The skip method allows us to skip cursor results. Below is an example of skipping by 20 results. Skipping allow us to move through our dataset.

```
$ db.movies.find( ).sort( { “rating.average”: 1, runtime: -1 } ).skip(20).pretty( )
```

The limit method allows us to limit the amount of elements the cursor should retrieve at a time and then also means the amount of documents we can then move through a cursor. Limit allows us to retrieve a certain amount of documents but only the amount of documents we specify.

```
$ db.movies.find( ).sort( { “rating.average”: 1, runtime: -1 } ).skip(100).limit(10).pretty( )
```

We can have the sort, skip and limit methods in any order we want and this will not affect the sorting as mongoDB will automatically do the actions in the correct order.

Using Projections to Shape Results

How can we shape the data which we get back from our find into the form we need it? When we use the find function this retrieves all the data from the retrieved document. This is not only too much redundant data transferred over the wire but also makes it hard to work with the data if we have to manually parse all the data. Projection allows us to control which data is returned from our Read Operation.

Below is an example code for projecting only the name, genre, runtime and rating from the returned results and all other data on the document does not matter to us.

```
$ db.movies.find( { }, { name: 1, genre: 1, runtime: 1, rating: 1 } ).pretty( )
```

In order to perform a projection we need to pass a second argument to the find function. If we do not want to specify any filter criteria for the first criteria of find, we would simply add an empty document as seen above. The second argument allows us to configure how values are projected i.e how we extract the data fields. We name the field and pass the value of 1. All fields that we do not mention with a 1 or we explicitly add with a 0 will be excluded by default. This will retrieve a reduced version of the document, but the id will always be included in the results even if we do not specify it within the projection. If we want to exclude the id we must explicitly exclude it as seen

in the below example:

```
$ db.movies.find( { }, { name: 1, genre: 1, runtime: 1, rating: 1, _id: 0 } ).pretty( )
```

This will retrieve the data in the projection and explicitly exclude the _id value from the results. This is only required for the _id filed and all other fields are implicitly set to 0 to exclude by default on projections.

We are also able to project on embedded documents for example we are only interested in the time from the schedule embedded document and not the day field. We would use the path notation to select the embedded document to project as seen below:

```
$ db.movies.find( { }, { name: 1, genre: 1, runtime: 1, “schedule.time”: 1 } ).pretty( )
```

Projections can also work with arrays and help with array data that we include in our returned find query results. For example, if we want to project Drama only from the genres, we would first filter the data by the criteria of all documents with an array containing Drama in the Genres and then use the projection argument to display only the Drama from the array:

```
$ db.movies.find( { genres: “Drama” }, { “genres.$”: 1 } ).pretty( )
```

The special syntax of $ after the genres will tell mongoDB to project on the one genre it found.

Now the document may have had more than Drama within the array in the retrieved document, but we have told mongoDB to only fetch the Drama and only output that result because that is the only data we are interested in retrieving. The items behind the scenes may have more data, just as they have other fields too. However, with the $ syntax will find the first matching document from our criteria query which in the above scenario was Drama.

If we had a more complex criteria whereby we are searching for all documents with both Drama and Horror, the $ projection syntax will return Horror because the Horror is the first matching criteria in the below example. The $all requires a match when Drama is present but the matching is when Horror is also present and therefore the projection will display Horror as it is technically the first matching criteria because Drama didn't match anything.

```
$ db.movies.find( { genres: { $all: [ "Drama", "Horror" ] } }, { "genres.$": 1 } ).pretty( )
```

There may be cases where we want to project items from an array in our document that are not items we queried for. In the below example we query to retrieve all documents with the value of Drama, but we want to project Horror only. Using the $elemMatch operator allows us to do this:

```
$ db.movies.find( { genres: "Drama" }, { genere: {$elemMatch: { $eq: "Horror" } } } ).pretty( )
```

The filter criteria and projection can be totally unrelated as seen in the below example:

```
$ db.movies.find( { "rating.average": { $gt: 9 } }, { genere: {$elemMatch: { $eq: "Horror" } } } ).pretty( )
```

Understanding $slice

The final special projection relating to arrays is called the $slice operator. This operator returns the first x amount of items from the array. In the below syntax example we are projecting the first 2 array items within the generes field.

```
$ db.movies.find( { “rating.average”: { $gt: 9 } }, { genres: { $slice: 2 }, name: 1 } ).pretty( )
```

The documents may have more genres assigned to them but we only see the first two items in the array because we used the $slice: 2 to return only 2 items. The number can be any value to return any number of items from the array. The alternative method of slice is to use an array form.

```
$ db.movies.find( { “rating.average”: { $gt: 9 } }, { genres: { $slice: [ 1, 2 ] }, name: 1 } ).pretty( )
```

The first element in the slice array is the amount of elements in the array which we skip e.g. we skip one item. The second element in the slice array is the amount of data we want to limit it to e.g. we want to limit it to two items. This will return item 2 and item 3 from the array and skip item 1 when it projects the results (i.e. we projected the next two items in the array).

We have many ways of controlling what we see using the 1 and 0 for normal fields and using the $, $elemMatch and $slice to control which element in the array are projected in our results.

We have now completely explored how to fully control what we fetch with our filtering criteria and then control which fields of the found document to include in our displayed result sets using projections.

Useful Links:

https://docs.mongodb.com/manual/reference/method/db.collection.find/

https://docs.mongodb.com/manual/tutorial/iterate-a-cursor/

https://docs.mongodb.com/manual/reference/operator/query/

Updating Fields with updateOne(), updateMany() and $set

An update operation requires two pieces of information (two properties):

1. Identify the document that should be updated/changed (i.e. the filter)
2. Describe how should it be updated/changed

For identifying the document we can use any of the filters we could use for finding documents and do not necessarily need to use the _id value. Using the _id is a common use for updating documents as it will guarantee the correct document is being updated.

The updateOne() method simply takes the first document that matches the filter criteria and updates it even if multiple documents may match the criteria. The updateMany() method on the other hand will take the criteria/filter and update all documents that match.

```
$ db.users.updateOne( { _id: ObjectId("5b9f707ead7") }, { $set: { hobbies: [ { title: "Sports", frequency: 5 }, { "Cooking", frequency: 3 }, { title: "Hiking", frequency: 1 } ] } } )

$ db.users.updateMany( { "hobbies.title": "Sports" }, { $set: { isSporty: true } } )
```

The second argument/parameter is how to update the document and the various update operators can be found on the official mongoDB documentation:

https://docs.mongodb.com/manual/reference/operator/update/

The $set takes in a document where we describe some fields that should be changed or added to the existing document. The updateOne example overwrites the existing hobbies array elements with the new hobbies array elements. The console will provide a feedback when it updates the document:

$ { “acknowledged” : true, “matchedCount” : 1, “modifiedCount” : 1 }

If we ran the exact updateOne command again the console will still show a matchedCount of 1 but the modifiedCount will be 0 as no document data would have been modified because it is exactly the same as the existing value.

$ { “acknowledged” : true, “matchedCount” : 1, “modifiedCount” : 0 }

If we were to find all the documents using the find() command, we would notice the user would still have the same _id and other document fields. This is because the $set operator does not override the existing values, instead it simply defines some changes that mongoDB evaluates and then if they make sense it changes the existing document by adding or overwriting the second argument fields. All the existing fields are left untouched. The $set operator by default simply adds or edits the fields specified in the update command.

Updating Multiple Fields with $set

In the previous section we demonstrated the $set operator used to update one field at a time in a document. It is important to note that the $set is not limited to updating one field in a document but can update multiple fields within a document as seen in the below example whereby we add a field of age and another field of phone:

```
$ db.users.updateOne( { _id: ObjectId("5b9f707ead7") }, { $set: { age: 40, phone: 07933225211 } } )
```

The console again will output when a document has matched the filter and have been modified whether the update modification was a overwrite or adding new fields to the matched document.

```
$ { "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

The $set operator can add fields, arrays and embedded documents inside and outside of an array.

Incrementing & Decrementing Values

The update operator allows us not only allows us to simply setting some values, but it can also

increment or decrement a number for us. For example, if we wanted to update a users age without using the $set operator as we would not necessarily know ever users age, rather we would want mongoDB to perform a simple common transformation of taking the current age and then recalculate the new age and then issue the update request.

MongoDB has a built in operator to allow us to perform common increment and decrement operations using the $inc operators.

```
$ db.users.updateOne( { name: “Emanuel” }, { $inc: { age: 1 } } )
```

This will increment the existing age field value by one. Note we could choose a different number such as 5 and increment the number by 5. To decrement a filed value we would continue to use the $inc operator but use a negative number instead.

```
$ db.users.updateOne( { name: “Emanuel” }, { $inc: { age: –1 } } )
```

Note we can perform multiple different things in the same update such as increment certain fields while setting new fields/editing existing fields, all within the second update parameter.

```
$ db.users.updateOne( { name: “Emanuel” }, { $inc: { age: 1 }, $set: { gender: “M” } } )
```

If we tried to increment/decrement a value as well set the same field value, this will give us an error:

```
$ db.users.updateOne( { name: “Emanuel” }, { $inc: { age: 1 }, $set: { age: 30 } } )
```

This would error in the console because updating the path will cause a conflict because we have two update operations working on the same field and this is not allowed in mongoDB and will fail providing a message in the shell something like the below:

```
$ WriteError: Updating the path ‘age’ would create a conflict at ‘age’: …
```

Diving Into Update Operation

Using $min, $max and $mull Operators

The $min, $max and $mull are other useful operators available to us. Below are scenarios and the syntax to overcome various update problems.

Scenario 1: We want to set the age to 35 only if the existing age is higher than 35. We would use the $min operator because this will only change the current value if the new value is lower than existing value. Note: this will not throw any errors if the existing value is higher than the new value, it will simply not update the document.

```
$ db.users.updateOne( { name: “Chris”}, { $min: { age: 35 } } )
```

Scenario 2: We want to set the age to 38 only if the existing age is lower than 38. We would use the $max operator which is the opposite of the $min operator. Again this will not throw any error for existing values is lower than the update value as it will simply ignore the update.

```
$ db.users.updateOne( { name: “Chris”}, { $max: { age: 38 } } )
```

Important Note: the modifiedCount in the terminal will show as 0 to show if no update occurred.

Scenario 3: We want to multiply the existing value with a multiplier. We would use the $mul operator which stands for multiply to perform this type of update operation.

```
$ db.users.updateOne( { name: “Chris”}, { $mull: { score: 1.1 } } )
```

This will multiply the existing score value by the multiplier of 1.1 to update the score document with the new value.

Diving Into Update Operation

Getting Rid of Fields

If we want to update records to drop a filed based on a certain criteria(s). There are two solutions to this problem and below is an example syntax to drop all phone numbers for users that are isSporty.

```
$ db.users.updateMany( { isSporty: true }, { $set: { phone: null } } )
```

We could use the $set operator to set the phone to null. This will not drop the field but it would mean that the field has no value which we can use in our application to not display the phone data.

The alternative solution is to use the special operator of $unset to truly get rid of a field from a document.

```
$ db.users.updateMany( { isSporty: true }, { $unset: { phone: "" } } )
```

The value we use with phone (or key field) is totally up to us but typically set to "" which represents empty. The key would be ignored in the update as the important part of the $unset operator document is the field name we wish to drop.

Renaming Fields

Just as we have the $unset operator to drop a field from a document, we are also able to rename fields using the $rename operator. We can leave the first update argument empty to target all documents and update the field name. The $rename document takes in the field name we want to rename and the key as a string of the new field name value. This will only update all documents that has a field called age to the new updated field name.

```
$ db.users.updateMany( { }, { $rename: { age: "totalAge" } } )
```

Understanding The Upsert Option.

If we wanted to update some documents but we were uncertain if the document existed or not. For example we have an application but we did not know if the data was saved in the database yet and if it wasn't saved yet, we now want to create a new document and if it did exist we want to overwrite/ update the existing document.

In this scenario if Maria did not exist as a document, nothing will happen other than a message in the terminal to tell us no document was updated. Instead, we would want mongoDB to create the document for us instead of having us manually doing this.

```
$ db.users.updateOne( { name: "Maria" }, { $set: { age: 29, hobbies: [ { title: "Cooking", frequency: 3 } ], isSporty: true } } )
```

To allow mongoDB to update or create documents for us, we would pass a third argument option called upsert and set this to true (by default this is set to false). Users is a combination of update and insert and will work with both updateOne and updateManu methods. If Maria does not exist it will create a new document and it will also include the name: "Maria" field automatically for us.

```
$ db.users.updateOne( { name: "Maria" }, { $set: { age: 29, hobbies: [ { title: "Cooking", frequency: 3 } ], isSporty: true } }, { upsert: true } )
```

Understanding Matched Array Elements.

Example scenario: we want to update all users document where the person has a hobby of sports and the frequency is greater or equal to three.The hobby field has an array of embedded documents.

```
$ db.users.find( { $and: [ { “hobbies.title”: “Sports”, { “hobbies.frequency”: { $gte: 3 } } } ] } ).pretty( )
```

This syntax will find all users which has hobbies title of Sports and hobbies that as a frequency that is greater or equal to three. This will find documents even where the Sports frequency is below three so long as there is another embedded hobbies document which has a frequency greater or equal to three.

The correct query is to use the $elemMatch operator which will look at the same embedded document for both criteria:

```
$ db.users.find( { hobbies: {$elemMatch: { title: “Sports”, frequency: { $gte: 3 } } } } ).pretty( )
```

Now that we know the correct query to find the documents we wish to update, the question now becomes how do we update that embedded array document only. So essentially we know which

overarching document we want to change but we want to change something exactly within that document found in the array.

The $set operator, by default will apply the changes to the overall document and not the document in the array we found. We would use the $set operator and then select the array field and use .$ as the syntax. This will automatically refer to the element matched in our filter criteria (first argument to the update command). We can define the new value after the colons.

```
$ db.users.updateMany( { hobbies: {$elemMatch: { title: “Sports”, frequency: { $gte: 3 } } } }, { $set: { “hobbies.$”: { title: “Sports”, frequency: 3 } } } )
```

Note this will update all matching documents to the updated changes. However, if we only want to add a new field to all matching documents the syntax would be to add a dot after the .$ as seen below:

```
$ db.users.updateMany( { hobbies: {$elemMatch: { title: “Sports”, frequency: { $gte: 3 } } } }, { $set: { “hobbies.$.highFrequency”: true } } )
```

The above syntax will find all documents which match the embedded document criteria and update that embedded document to add a new field/value of highFrequency: true if the highFrequency field did not exist (if it did, it would simply update the existing field value). The $set works exactly as it did before the only difference is adding the special placeholder within the array path to quickly get access to the matched array element.

Updating All Array Elements.

Example Scenario: The below find method returns the overall person document where the filter matched and does not return the document we filtered on only, the filter is just a key factor for returning to us the overall document.

```
$ db.users.find( { “hobbies.frequency”: { $gt: 2 } } ) .pretty( )
```

This will also retrieve a document with a frequency lower than two provided at least one embedded document has a frequency above two to match the filter. Now that we have found all documents meeting the criteria but not all array documents fulfilled our filter criteria. However, we want to change all embedded documents in the hobbies array that did fulfil the filter criteria only.

```
$ db.users.updateMany( { “hobbies.frequency”: { $gt: 2 } }, { $set: { “hobbies.$.goodFrequency: true } } )
```

Again we can use the $set operator, but we want to change all the matched hobbies with a frequency above two. The “hobbies.$” syntax we saw in the previous section only edits hobby for each person and if there is multiple matching hobbies per person, it will not update them all but only the first element within the array.

Now in order to update all elements in an array, we would use a special placeholder in mongoDB that is the .$[] which simply means update all elements. We can use the dot notation after the .$[] to select a specific field in the array document.

```
$ db.users.updateMany( { totalAge: { $gt: 30 } }, { $inc: { “hobbies.$[].frequency: -1 } } )
```

This will update all users that has a totalAge greater than 30 and decrement the hobbies frequency by 1. The .$[] syntax is used to update all arrays elements.

Diving Into Update Operation

Finding and Updating Specific Fields

Continuing on from the last section, we were able to use the .$[] notation to update all elements within the array per person. We can now build on this notation to update specific fields and below is the solution to the previous problem.

```
$ db.users.updateMany( “hobbies.frequency”: { $gt: 2 } }, { $set: { “hobbies.$[el].goodFrequency”: true } }, { arrayFilters: [ { “el.frequency”: { $gt: 2 } } ] } )
```

Note: Within the the el within the square bracket is an identifier which we could have named as anything. For the third argument in our update method we would use the arrayFilters option to define the identifier/filter elements. The identifier does not need to be the same as the filter criteria for example we could filter by age but use the $set to identifier to update based on a

complete different filter such as the frequency greater than two. The arrayFilter can have multiple documents for each identifier.

This would now update specific array elements that meet the identifier arrayFilter criteria. Note that the third argument of updateOne and updateMany is an options argument whereby we previously used upsert as an option to update/insert documents. We can also use arrayFilters to provide a filter criteria for our identifiers.

Note: if an identifier is used we must use arrayFilter option to define the identifier filter criteria otherwise the update method will fail as mongoDB will not know what to do with the identifier.

Adding Elements to Arrays

If we want to add elements onto an array for a document instead if using $set operator (which we can still use to update fields), we can use $push to push a new element onto the array. The $push operator takes in a document where we describe firstly the array we wish to push to and then the element we want to push/add to the existing array documents.

```
$ db.users.updateOne( { name: “Maria” }, { $push: { hobbies: { title: “Sports”, frequency: 2 } } } })
```

The $push operator can be used with more than one document to be added. We use the $each operator which takes in an array of multiple documents that should be added/pushed.

```
$ db.users.updateOne( { name: “Maria” }, { $set: { $push: { hobbies: { $each: [ { title: “Running”, frequency: 1 }, { title: “Hiking”, frequency: 2 } ] } } } } )
```

There are two options sibling options we can add to the above $each syntax. The first is the $sort operator. Technically, we are still in the same object where we have the $each operator i.e. it is a sibling to $each. The sort describes how the elements in the array should be sorted before they are pushed into hobbies.

```
$ db.users.updateOne( { name: “Maria” }, { $set: { $push: { hobbies: { $each: [ { title: “Running”, frequency: 1 }, { title: “Hiking”, frequency: 2 } ], $sort: { frequency: -1 } } } } } )
```

This will sort the hobbies array by frequency in a descending order i.e. having the highest frequency first.

The second sibling is the $slice operator which allows us to add only a certain number of element. We can use this in conjunction with the $sort operator as seen below.

```
$ db.users.updateOne( { name: “Maria” }, { $set: { $push: { hobbies: { $each: [ { title: “Running”, frequency: 1 }, { title: “Hiking”, frequency: 2 } ], $sort: { frequency: -1 }, $slice: 1 } } } } )
```

In this example the slice is taking only the first element after the sort to push to the hobbies array. The sort is on the overall array i.e. new and existing elements and not just the elements we add.

Removing Elements from Arrays

Not only are we able to push elements to an array but we are also able to pull elements from an array using the $pull operator, as demonstrated below.

```
$ db.users.updateOne( { name: “Maria” }, { $pull: { hobbies: { title: “Hiking” } } } )
```

The $pull operator takes in a document where we describe what we want to pull from the array. In the above we are pulling from the hobbies array based on a condition i.e. pull every element where the title is equal to Hiking. We do not need to only use equality conditions but we can also use all the normal filter operators that we have seen before such as the greater than or less than operator.

Sometimes we may wish to remove the last element from an array and have no specific filter criteria. We would use the $pop operator and use a document to define the name of the field of which we want to pop. The -1 defines to pop the first element and the 1 defines to pop the last element from the array.

```
$ db.users.updateOne( { name: “Chris” }, { $pop: hobbies: 1 } )
```

Understanding the $addToSet Operator

The final update command we will explore is the $addToSet operator.

```
$ db.users.updateOne( { name: “Maria” }, { $addToSet: { hobbies: { title: “Hiking”, frequency: 2 } } } })
```

The difference between $addToSet and $push operator, the $push operator allows us to push duplicate values whereas the $addToSet does not allow for this. It is important to note the console will not error but simply show that no document was updated with the change. Always remember that $addToSet operator adds unique values only.

This concludes the Update Operations available to us in mongoDB. We now understand the three arguments we can pass to both the updateOne and updateMany commands which are:

1. Specify a filter (query selector) using the same operators we know from find() command.
2. Describe the updates via $set or other update operators.
3. Additional Options e.g. $upsert or $arrayFilters to the update operation.

In the official documentation we can view all the various operations available to us:

https://docs.mongodb.com/manual/tutorial/update-documents/

Understanding deleteOne() and deleteMany()

To delete a single document from a collection we would use the deleteOne() command. We need to specify a query selector/filter. The filter we specify here is exactly the same as we would use for the the finding and updating documents. We simply need to narrow down the document we want to delete. DeleteOne will delete the first document that matches the criteria.

```
$ db.users.deleteOne( { name: “Chris” } )
```

We can use the deleteMany() command to delete all documents where the query selector/filter criteria has been met. Below are two examples.

```
$ db.users.deleteMany( { totalAge: {$gt: 30}, isSporty: true } )
$ db.users.deleteMany( { totalAge: {$exists: false}, isSporty: true } )
```

Note: we can add as many query selectors as we want to narrow down the document(s) we wish to delete from the database.

Deleting All Entries in a Collection

There are two approaches to deleting all entries in a collection. The first method is to reach out to the collection and execute the .deleteMany( ) command and pass and empty document as the argument. This argument is a filter that matches every document in the collection and therefore will delete all entries within the collection.

```
$ db.users.deleteMany( { } )
```

The alternative approach is to delete the entire collection using the .drop( ) command on the specified collection. This will return true if successfully dropped a collection.

```
$ db.users.drop( )
```

When creating application it is very unlikely we would drop collections. Adding and dropping collections is more of a system admin task. We can also drop an entire database using dropDatabase command. We would then use the use command followed by the database to navigate to the desired database collection.

```
$ db.dropDatabase( )
```

https://docs.mongodb.com/manual/tutorial/remove-documents/

What are Indexes and why do we use them?

An index can speed up our find, update or delete queries i.e. all the queries where we are looking for certain documents that should match some criteria.

```
$ db.products.find( { seller: “Marlene” } )
```

If we take a look at this find query, we have a collection of documents called products and we are searching for a seller called Abbey. Now by default if we don’t have an index on the seller set, mongoDB will go ahead and do a so-called collection scan. This simply means that mongoDB to fulfil this query will go through the entire collection, look at every single document and see if the seller equals “Marlene” (equality). As we can imagine, for a very large collection with thousands or millions of document, this can take a while to complete. This is the default or only approach mongoDB can take when there are no indexes setup in order to retrieve maybe two documents out of the thousands of documents in the collection.

We can create an index and an index is not a replacement for a collection but rather an addition. We would create an index for the seller key of the product collection and that index then exists additionally to the collection and the index is essentially an ordered list of all the values that are stored in the seller key for all the documents.

It is not an ordered list of the documents, it is just the values for the field for which we created that index. Also it is not just an ordered list of the values, every value/item in the index has a pointer to the full document it belongs to.

This allows mongoDB to perform a so-called index scan to fulfil this query. This means mongoDB will see that for seller, such an index exists and it therefore simply goes to that seller index and can quickly jump to the right values because unlike for the collection, it knows the values are sorted by that key. This means if we are searching for a seller starting with M, it does not need to search through the first few records. This allows mongoDB very efficiently go through that index and find the matching product because of the ordering and the pointer that every items within the index has. So mongoDB finds the value for the query and then finds the related document to return.

This is how an index works in mongoDB and also answers why we would use indexes because creating indexes drastically speeds up our queries. However, we also should not overdo it with indexes. Lets take the example of a Products collection which has a _id, name, age and hobbies fields. We could create a index for all four fields and we would have the best performance because no matter what we look for, we have an index and can query for every field efficiently which will speed our find queries. Having said this, index does not come without cost. We would have to pay some performance cost on inserts because that extra index that has to be maintained would need to be updated with every inserts. This is because we have an ordered list of elements with pointers

to the documents. So if we add a new document, we also have to add a new element to the index. This may sound simple and it would not take super long, but if we have 10 indexes for our document in our collection, we would have to update all 10 indexes for every insert. We may then quickly run into some issues because we will have to do a lot of work for all these fields for every insert and for every update too. Therefore, indexes do not come for free and we have to figure out which indexes makes sense to have and which indexes don't.

We are now going to explore indexes in more detail and look at all the type of indexes that exist in mongoDB and how to measure whether an index makes sense or does not make sense to have.

Adding a Single Field Index?

To determine whether an index can help us in our find query, mongoDB provides us with a nice tool that we can use to analyse how it executed the query. We can chain the explain method to our normal query. This method works with find, update and delete commands but not for inserts (i.e. it works for methods that narrow down documents.

```
$ db.contacts.explain( ).find( { "dob.age": { $gt: 60 } } )
```

This will provide a detailed description of what mongoDB did to derive to the results.

MongoDB thinks in the so-called plans and plans are alternatives it considers for executing the query and in the end it will find the winning plan. The winning plan is simply what mongoDB did to get to our results. Without indexes, a full scan is always the only thing mongoDB can do. However, if there were alternatives and they were rejected then this will appear in the rejectedPlans array.

To get a more detailed report we can run the same command but passing in an argument:

```
$ db.contacts.explain( “executionStats” ).find( { “dob.age”: { $gt: 60 } } )
```

The executionStats provides a detailed output for our query and how the results were returned. This will show things such as executionTimeMillis which is the time it took to execute the query in milliseconds and totalDocsExamined which shows the number of documents that needed to be scanned in order to return our query results. The larger the gap between the totalDocsExamined and nReturned values, this shows how inefficient the query is.

To add an index on a collection, we would use the createIndex method passing in a document. The first value (key) in the document is the name of the field we want to create a index on (this can be on top level fields as well as embedded field names). The value is whether mongoDB should create a list of values in the field in an ascending (1) or descending (-1) order.

```
$ db.contacts.createIndex( { “dob.age”: 1 } )
```

Once we run the command we should see in the terminal that the index has been created.

```
{
    “createdCollectionAutomatically” : false
    “numIndexesBefore” : 1
    “numIndexesAfter” : 2
    “ok” : 1
}
```

If we were to run the above explain command again on our collection, we should notice the executionTimeMillis for the same query has been sped up. We should also see two execution stages the first being an index scan (IXSCAN). The index scan does not return the documents but the keys/pointers to the document. The second stage is the fetch (FETCH) which will take the pointers returned from the index scan and reach out to the actual collection and then fetch the real documents. We would notice that mongoDB would only have to look at a reduced number of documents to return the documents from our query.

This is how an index can help us to speed up our queried searches and how to use the explain method to determine whether an index should be used in our collection to speed up the query.

Indexes Behind the Scenes

What does createIndex( ) method do in detail?

Whilst we can't really see the index, we can think of the index as a simple list of values and pointers to the original document.

Something like this (for the "age" field):

(29, "address in memory/ collection a1")

(30, "address in memory/ collection a2")

(33, "address in memory/ collection a3”)

The documents in the collection would be at the "addresses" a1, a2 and a3. The order does not have to match the order in the index (and most likely, it indeed won’t).

The important thing is that the index items are ordered (ascending or descending - depending on how we created the index). The syntax of createIndex({age: 1}) creates an index with ascending sorting while the syntax of createIndex({age: -1}) creates an index with descending sorting.

MongoDB is now able to quickly find a fitting document when we filter for its age as it has a sorted list. Sorted lists are way quicker to search because we can skip entire ranges (and don't have to look at every single document).

Additionally, sorting (via sort(...)) will also be sped up because you already have a sorted list. Of course this is only true when sorting for the age.

Understanding Index Restriction

In the previous section we created an index which sped up our query when looking for people with an age greater than 60. However, if we run the same query but find people older than 20, we will notice that the execution time is higher than it was for people above the age of 60.

To drop an index from our collection we would use the dropIndex method and pass in the document that we created to create the index.

```
$ db.contacts.dropIndex( { “dob.age”: 1 } )
```

If we were to run a full scan against our collection, we will notice that the query is much faster than

having an index. The reason for why the query is much faster is because we have saved a step from going through the index. If we have a query that will return a large portion or the majority of our documents, an index can actually be slower because we have an extra step to go through almost the entire index list and we then have to go to the collection and get all these documents. If we do a full scan, we do not have this extra step of going through the collection to get the documents because with a full collection scan we already have all the documents in memory and an index doesn't offer us any more because it will only be an extra step.

Important note: if we have queries that regularly return the majority or all of our documents, an index will not help us and it might even slow down the execution. This is the first important note to keep in mind (a first restriction) when planning our queries and whether or not to use indexes.

If we have a dataset where our queries typically return a fraction like 20% or lower than that of the documents, then indexes will certainly always speed up our queries. If we have a lot of queries that give us back all the documents or close to all the documents, then indexes can not do much for us. The whole point of indexes is to quickly get to a narrow subset of our document list and return the documents from that index.

Creating Compound Indexes

Not only can be have indexes on fields that have number values but we can also have indexes on fields that have text values (*both can be sorted*). We cannot create indexes for booleans as we only have two kind of values i.e. true and false and the change of index on booleans will not speed up our queries. Below is an example of creating a index on a text field.

```
$ db.contacts.createIndex( { gender: 1 } )
```

Now the above index would not make too much sense for an index because gender has two values of Male and Female and would probably return more than half the results. However, if we want to to find as an example all people who are older than 30 and are male, we can create a so called compound index.

```
$ db.contacts.createIndex( { “dob.age”: 1, gender: 1 } )
```

The order of the two fields in our createIndex method do matter because a compound index simply is an index with more than one field. This will store one index where each entry in the index is now not on a single value but on two combined values. This does not create two indexes and this is really important to note with compound indexes, it creates one index where every element is a connected value (it creates a pair value for example in the above this is a pair of the age and gender values).

The order of the fields defines which kind of pairs mongoDB will create in our compound index (for example does mongoDB create a 31 male index or a male 31 index – this will be important for our queries).

There are two queries we can now run which will take advantage of the compound index. The first is to find based on age and gender:

```
$ db.contacts.explain( ).find( { “dob.age”: 35, gender: “male” } )
```

This will perform an index scan with our index name (*the index name is auto-generated e.g. “indexName” : “dob.age_1_gender_1”*)

The second query that can utilise the compound index is a query on the age only:

```
$ db.contacts.explain( ).find( { “dob.age”: 35 } )
```

This will also use the same compound index we created for the index scan even though we never specified to search for the gender. Compound indexes can be used from left to right, but the left must always be used in the search i.e. a find query on the gender alone will not work.

```
$ db.contacts.explain( ).find( { gender: “male” } )
```

The above query would use a full collection scan and not the index scan using the compound index.

The compound indexes are grouped together, the first field (left) will be ordered whereas the other fields (right) will not be ordered. We can have a compound index with more than 2 fields but up to a maximum of 31 field. However, we cannot utilise the compound index without the first field.

These are the restrictions we have on compound indexes but compound indexes allows us to speed up queries that uses multiple values.

Using Indexes for Sorting

Now that we have had a look at the basics of indexes, it is important to know that indexes are not only used for narrowing our find queries but they can also help with sorting. Now that we have a sorted list of elements of the index, mongoDB can utilise this in case we want to sort in the same way that the index list is sorted.

```
$ db.contacts.explain( ).find( { “dob.age”: 35 } ).sort( { gender: 1 } )
```

In the above we can find people with the age of 35 but sort them by gender in an ascending order. We will notice that this will use an index scan for both the gender and age even though we filtered by age only. It uses the gender information for the sorting. Since we already have an ordered list of values, mongoDB can utilise this to quickly give back the order of documents we need.

It is important to understand that if we are not using indexes and we do a sort on a large amount of documents, we can actually timeout because mongoDB has a threshold of 32mb in memory for sorting. If we have no index, mongoDB will essentially fetch all our documents into memory and do the sort there and for very large collections and large amount of fetched documents, this can be too much to then sort.

Sometimes we would need indexes not only to speed up the query but also to be able to sort at all. This is not a problem for small dataset but where we fetch so many documents that an in-memory sort which is the default is just not possible and we then need an index which is already sorted so that mongoDB does not have to sort in memory but can take the order we have in the index.

Important Note: mongoDB has a threshold of 32mb which it reserves in memory for the fetched document and sorting them. This is the second important note to keep in mind as to whether or not to create an index.

Understanding the Default Index

When creating an index it would seem like that there is an index already existing in our collection. To be able to see all indexes that exists for a collection we can use the getIndexes command.

```
$ db.contacts.getIndexes( )
```

This command will print all the indexes we have on that collection within the shell. We will notice, if we have created new indexes on our collection that there are two indexes. The first index on the _id field is a default index mongoDB maintains for us. The second index are the indexes that we have created.

The default index for _id is created and painted on every collections by mongoDB automatically. This means if we are filtering for _id or sorting by _id which is then the default sort order or order by which the document are fetched, mongoDB is utilising the index for that at least.

Configuring Indexes

The _id index that we get out of the box for this field is actually unique by default. This is a setting mongoDB gives us to ensure that we cannot add another document with the same value in the same collection. There are use cases where we also need that behaviour for a different field and therefore we can add our own unique indexes.

For example, if we wanted email to be a unique index. We would create an index on the email field and then pass in a second argument to the createIndex command. The second argument allows us to configure the index — this is where we can set the unique option to true.

```
$ db.contacts.createIndex( { email: 1 }, { unique: true } )
```

If we execute this command, we may receive a duplicate key error collection if we already have have duplicate values within our collection. This will also show the document(s) where the duplicate key field exists. This is an advantage of the unique index because we would get such a warning if we try to add it or we already have it in place and tried to add a document with a value that already existed. Unique indexes can help us as developers to ensure data consistency and avoid duplicate data for fields that we need to have as unique. This index is not only useful to speed up our find queries but also to guarantee that we have unique values for that given field in that collection.

Understanding Partial Filters

Another interesting kind of configuration for a filter is setting up a so-called partial filter.

For example, if we were creating an application for calculating what someone will get once they retire – we would typically only look for a person older than 60. Having an index on the dob.age field might make sense. The problem of course is that we have a lot of values in our index that we never actually query for. Now the index will still be efficient but it will be unnecessarily big and an index eats up size on our disk. Additionally, the bigger the index is, the more performance certain queries will take nonetheless.

Is we know certain values will not be looked at or only very rarely and we would be fine using a full collection scan, we can actually create a partial index where we only add the values we are regularly going to look at.

```
$ db.contacts.createIndex( { “dob.age”: 1 }, { partialFilterExpression: { “dob.age”: { $gt: 60 } } } )
$ db.contacts.createIndex( { “dob.age”: 1 }, { partialFilterExpression: { gender: “male” } } )
```

We can add this option to compound indexes as well. In the partialFilterExpression, we define which field will narrow down the set of values we want to add (*we can use a totally different field e.g. gender)*. We can use all the equality expression we have previously seen e.g. $gt, $lt, $exist etc.

The second expression would create a index on the age but only for elements where the underlying document is for a male while the first will only create a index on age but for elements where the underlying document is for person older than 60.

If we only created a partial index using the second example and performed the below query, we would notice that mongoDB will perform a full collection scan and will ignore the partial index. This is because mongoDB determined that yes we are looking for a field that is part of the index (age) but since we did not search for gender in our query, it considered the partial index too risky use and mongoDB as a top priority ensures that we do not lose any data. therefore, the results we receive back will also include documents that has a gender of female and not male only because it performed a full collection scan and not a partial index scan.

```
$ db.contacts.explain( ).find( { “dob.age”: { $gt: 60 } } )
```

In order for mongoDB to use the partial index we must also filter by gender:

```
$ db.contacts.explain( ).find( { “dob.age”: { $gt: 60 }, gender: “male” } )
```

The difference between a partial index and a compound index, for partial indexes the overall index is smaller. In the above example only the ages of males are stored and female keys are not sorted in the index and therefore the index size is smaller leading to a lower impact on our hard drive. Also our right queries are sped up because if we insert a female, that will never have to be added to the

index. This still make a lot of sense if we often filter for this type of combination i.e. for the age and then only males – a partial index can make a lot of sense if we rarely look for the other result i.e. we rarely look for women.

Whenever mongodb has the impression that our find request would yield more than what's in our index, it will not use that index but if we typically run queries where we are within our index (filtered or partial index) then mongodb will take advantage of it. We would then benefit from having a smaller index and having less impact with writes.

So again it depends on the application we're writing and whether we often just need a subset or whether we typically need to be able to query everything, in which case a partial index won't make much sense.

Applying the Partial Index

An interesting variation or use case of the partial index can be seen in conjunction with a unique index. Below is a example demo of this use case.

```
$ db.users.insertMany( [ { name: “Abel”, email: “abel@email.com” }, { name: “Diane” } ] )
```

We have collection of users but not all documents has an email field (only Abel has an email). We can try to create an index on our email field within the users collection.

```
$ db.users.createIndex( { email: 1 } )
```

The above will successfully create an index in an ascending order on our email field within our users collection. If we drop this index and then create a new index using the unique option.

```
$ db.users.dropIndex( { email: 1 } )
$ db.users.createIndex( { email: 1 }, { unique: true } )
```

The above will successfully create a unique index in an ascending order on our email field within our users collection. If we now try to insert some new document without an email:

```
$ db.users.insertOne( { name: “Anna” } )
```

We would now see a duplicate key error because the non-existing email for which we have an index is treated as a duplicate key because now we have a no email value stored twice. This is an interesting behaviour we need to be aware of. MongoDB treats non-existing values still as values in our index i.e. it stores as a null value and therefore if we have two documents with null values for an indexed field and that index is unique, we will get this error.

Now if we have a use case where we want to create a unique index on a field and it is ok for that field to have a null value, then we would have to create the index slightly different.

```
$ db.users.dropIndex( { email: 1 }, { unique: true } )
$ db.users.dropIndex( { email: 1 }, { unique: true, partialFilterExpression: { email: { $exists: true } } } )
```

In the above, we now use the partialFilterExpression as a second option along with the unique option. The partialFilterExpression of $exists: true on email lets mongoDB know that we only want to add elements into our index where the email field exists. This will avoid the case of having a clash with our unique option. Therefore, if we now try to run the below insert command (the same as before) this will now work and we will not see any errors.

```
$ db.users.insertOne( { name: “Anna” } )
```

We use the combination of unique and partialFilterExpression to not index fields where no value or where the entire field does not exist and this allows us to continue to use the unique option on that field.

Understanding Time-To-Live (TTL) Index

The last interesting index option is the Time-To-Live (TTL) index. This type of index can be very helpful for a lot of applications where we have self-destroying data for example a session of a user where we want to clear their data after some duration or anything similar to that nature. Below is an example of a TTL index:

```
$ db.sessions.insertOne( { data: “randomText”, createdAt: new Date( ) } )
$ db.sessions.fid( ).pretty( )
```

The seasons data will receive a random string data and the createdAt will be a data stamp of the current date. The new Date will provide a ISODate for us, for example:

```
ISODate(“2019-03-31T19:52:24.272Z”)
```

To create a TTL index for our sessions collection we would use the expireAfterSeconds option. The TTL option below is created on the createdAt field.

```
$ db.sessions.createIndex( { createdAt: 1 }, { expireAfterSeconds: 10 } )
```

This is a special feature mongoDB offers and will only work on date fields/indexes. We could add

this to other field types (i.e. numbers, texts, booleans etc) but this would simply be ignored. In the above we have set the expireAfterSeconds at 10 seconds. It is important to note that the index does not delete elements in hindsight i.e. elements already existing before the TTL index was created. If we were to now insert a new element within this collection using the below, we would notice after 10 seconds both element will now be deleted.

```
$ db.sessions.find( ).pretty( )
```

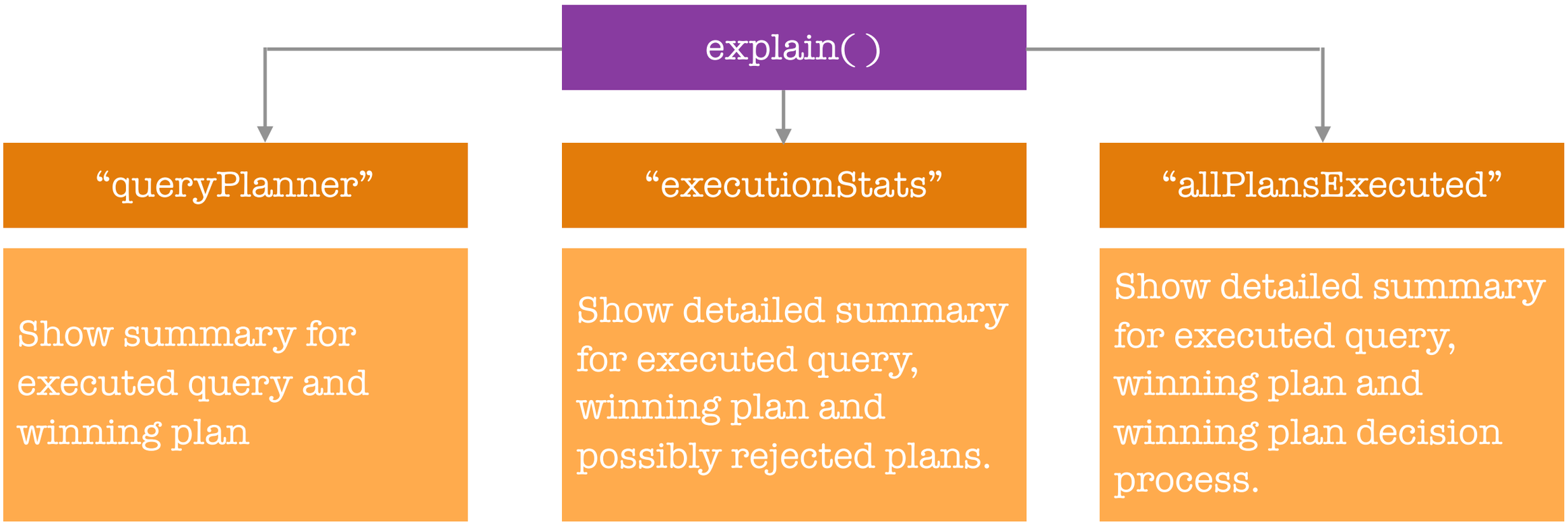
Adding a new element to the collection will trigger mongoDB to re-evaluate the entire collection after 10 seconds which will include the existing elements, to see whether the createdAt field which is indexed has fulfilled the expireAfterSeconds criteria (i.e. only being valid for 10 seconds).

This can be very useful because it allows us to maintain a collection of documents which destroy themselves after a certain time span. This can be very helpful for many applications for example session data for users on our web app or maybe an online shop where we want to clear a cart after one day etc. Whenever we have a use case where data should clean up itself, we do not need to write a complex script for that as we can use TTL index expiryAfterSeconds option.

It is important to note that we can only use this option on single field indexes that are date objects and it does not work on compound indexes.

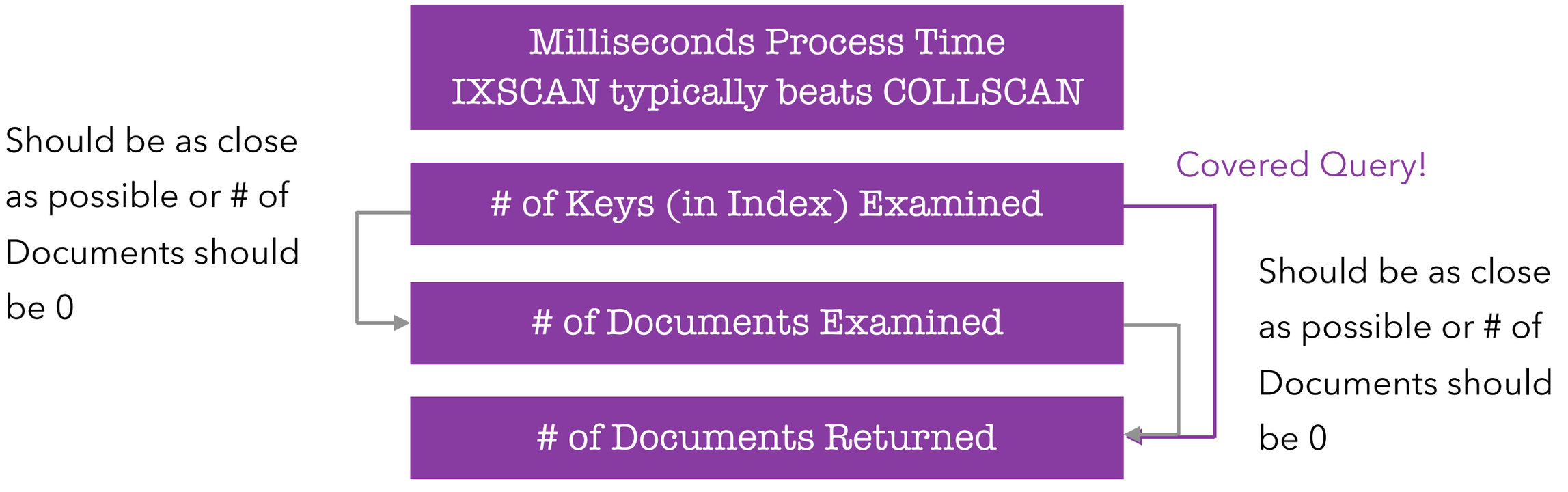
Query Diagnosis & Query Planning

Now that we have looked at what indexes do and how we can create our own indexes, it is important to keep playing around with it to get a better understanding of the different options and how indexes work. In order to play around and understand if an index is worth the effort, we need to know how to diagnose our queries. We have already seen the explain( ) method for this.



It is important to note we can execute the explain as is or pass in queryPlanner as an argument to get the default minimal output where it tells us the winning plan and nothing much else. We can also use executionStats as an argument to see a detailed summary output and see information about the winning plan and possibly the rejected plan as well as how long it took to execute the query. Finally, there is also the allPlansExecuted argument which shows a detailed summary and information on how the winning plan was chosen.

To determine whether a query is efficient, it is obvious to look at the milliseconds process time to compare the solution with/without an index i.e. does index scan beat the collection scan. Another important measure is to compare the number of keys in the index and how many documents examined and how many documents are returned.



Understanding Covered Queries

We can reach a so-called covered query if we only return fields which are also the indexed fields in which case the query does not examine any documents because it can do this entirely from inside the index. We will not always be able to reach this state but if we can optimise our index to reach the covered query state (as the name suggests the query is fully covered by the index) then we have of course have a very efficient query because we have skipped the stage of reaching out to the collection to get the documents which will obviously speed up our query and have a very fast solution.

If we have an opportunity and have a query that we typically run an store fields, it might be worth storing them in a single field or if it's two fields, to store them in a compound index so that we can fully cover the query from inside of our index.

Below is an example of a Covered Query - using a projection to only return the name in our query:

```
$ db.customers.insertMany( [ { name: “Abbey”, age: 29, salary: 30000 }, { name: “Bill”, age: 20, salary: 18000 } ] )
$ db.customers.createIndex( { name: 1 } )
$ db.customers.explain(“executionStats”).find( { name: “Bill” }, { _id: 0, name: 1 } )
```

How MongoDB Rejects a Plan

To understand how mongoDB rejects a plan we will use the customers collection example from the section. In the customer collection we have two indexes, the standard _id index and our own name index. We will now add a compound index on the customers collection which will create an index on age in ascending order and the name as seen below:

```
$ db.customers.createIndex( { age: 1, name: 1 } )
```

We now have three indexes on our customer collection. We can now query our collection and use the explain method as see how mongoDB rejects a plan.

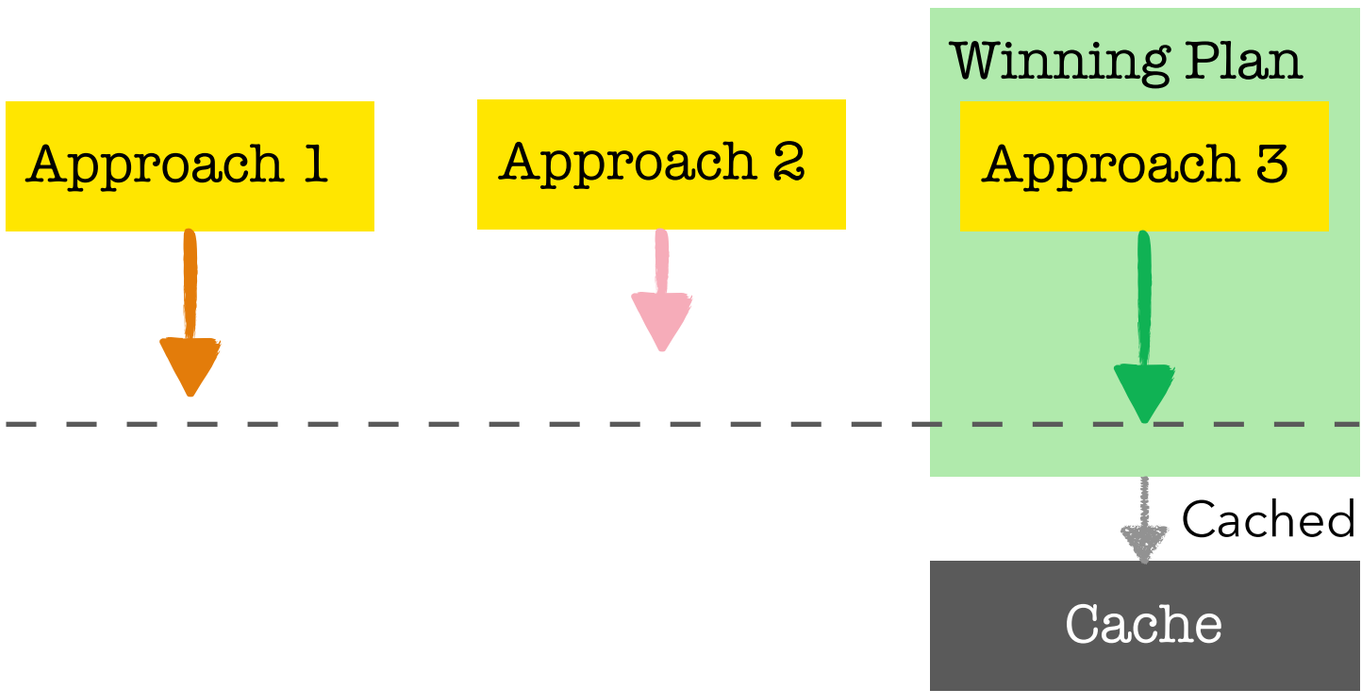
```
$ db.customers.explain( ).find( { name: “Abbey”, age: 29 } )
```

We will notice the winningPlan will be a IXSCAN using the compound age_1name_1 index. We should also now see a rejectedPlan which was the IXSCAN on the single field name_1 index. MongoDB considered both indexes because the query on the name field fits both indexes. This is interesting to know which indexes was rejected and which one was considered the winningPlan. The question now is how exactly does mongoDB figure out which plan is better?

MongoDB uses an approach where it simply, first of all, looks for indexes that could help with the

query at hand. Since our find query includes a look for the field name, mongoDB automatically derived that both the single field index and compound index could help. In this scenario we only have two approaches but for other scenarios we may have even more approaches. Hypothetically, lets say we had three approaches to our find query, mongoDB then simply lets those approaches race against each other but not for the full dataset. It sets a certain winning condition e.g. against 100 documents. So it looks at which approach is the first to find 100 documents, and whichever approach is first, mongoDB will then use that approach for the real query.

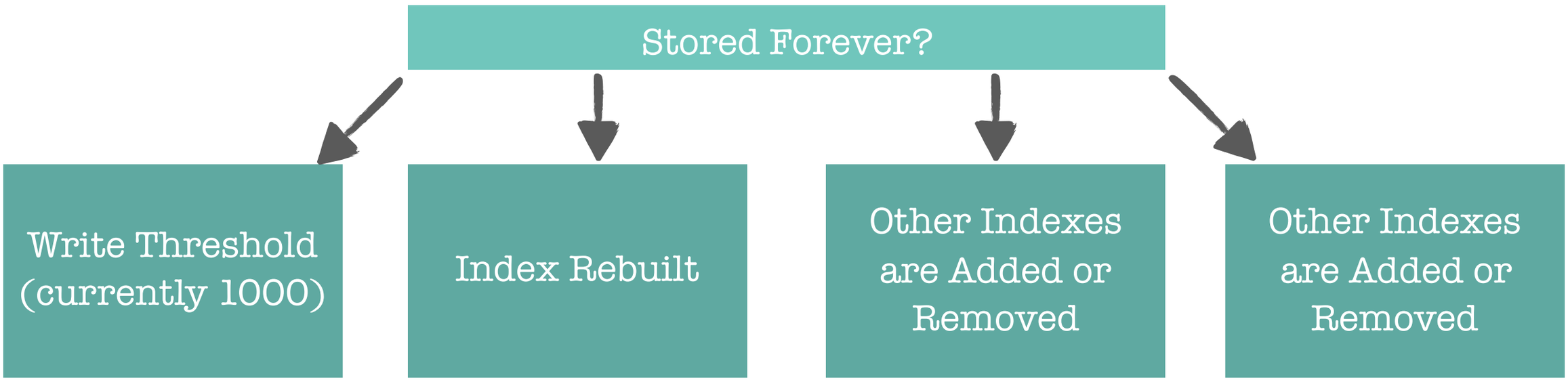
This would be cumbersome if mongoDB would have to do this for every find query we send to the database because it would obviously cost a little bit of performance. Therefore, mongoDB caches this winningPlan for this type of query. For future queries that are looking exactly equal, it uses this winningPlan and for future queries that look different i.e. uses different values or different keys, mongoDB will race the approaches again and find a winning plan for that type of query.



This cache is not there forever and is cleared after a certain amount of inserts or a database restart. To be precise, instead of being stored forever, the winningPlan is removed from cache after we :

a. We wrote a certain amount of documents to that collection because mongoDB will say it does not know if the current winningPlan will still win because the collection has changed a lot and it should then reconsider.
b. If we rebuilt the index i.e. we dropped and recreated the index.
c. If we add other indexes because the new index could be better.
d. If we restart the mongoDB server.

This is how mongoDB derives the winningPlan and how it stores it in cache memory.



This is interesting for us as a developer to regularly check our queries (our find, update or delete queries) and see what mongodb actually does, if it uses indexes efficiently, if maybe a new index should be added (something we can do on your own if we own the database instead we can pass that information to your db administrator) or if we maybe need to adjust the query.

Maybe we're always fetching data that we do not really need and we could use a covered query if we just would project the data we need which happens to be the data stored in the index.

This is why, as a developer, we need to know how indexes work because either we need to create them on our own in our next project on which we work alone or because we can optimise our queries or tell the db administrator to optimise the indexes.

The last level of verbosity that the explain method offers to us is the allPlansExecution:

```
$ db.customers.explain("allPlansExecution").find( { name: "Abbey", age: 29 } )
```

What this will do, it will provide a bunch of output with detailed statistics for all plans including the rejected plans. We can therefore see in detail how an index scan on our compound index perform as well as how it would perform on any other indexes. With this option, we can get detailed analytics on different indexes & queries and the possible ways of running our query. We should now have all the tools we need to optimise our queries and our indexes.

Using Multi-Key Indexes

We are now going to explore two new type of indexes and the first one is called a multi-key index.

```
$ db.contacts.drop( )
$ db.contacts.insertOne( { name: “Max”, hobbies: [ “Cooking”, “Football” ], addresses: [ { street: “First Street” }, { street: “Second Street” } ] } )
```

In mongoDB it is also possible to index arrays as seen below:

```
$ db.contacts.createIndex( { hobbies: 1 } )
$ db.contacts.find( { hobbies: “Football” } ).pretty( )
```

If we explain the above findQuery to see how mongoDB arrived at the winningPlan using the executionStats command, we will notice that mongoDB used the index scan and the isMultiKey set to true for the hobbies index.

```
$ db.contacts.explain(“executionStats”).find( { hobbies: “Football” } ).pretty( )
```

MongoDB treats index on arrays as a multi-key index because it is an index on an array of values. Multi-key indexes technically work like regular indexes but are stored slightly differently.

MongoDB pulls out all the values in our index key i.e. hobbies from the above case and stores them as separate elements in an index. This will mean that multi-key indexes for a lot of documents are larger than single field indexes. For example, if every document has an array with four values on average and we have a thousand documents and we indexed that array field, we would store four thousand elements (4 x 1,000 = 4,000). This is something to keep in mind, multi-key are possible but are also bigger, this does not mean we shouldn't use them.

```
$ db.contacts.createIndex( { addresses: 1 } )
$ db.contacts.explain("executionStats").find( { "addresses.street": "First Street" } )
```

We will notice with the above, we can create an index on the addresses array, however, when we explore the fir query we will notice mongoDB would use a collection scan and not the index. The reason for this is because our index holds the whole document and not the fields of the documents. MongoDB does not go so far to pull out the elements of an array and then pull out all field values of a nested document that array might hold. If we were looking for the addresses where the street is First Street, then we would see mongoDB using the index scan because it is the whole document which is in our index.

```
$ db.contacts.explain("executionStats").find( { addresses: { street: "First Street" } } )
```

MongoDB pulls out elements of the array for addresses as single elements which happens to be a document, so that document is what mongoDB pulled out and then stored in the index registry. This is something to be aware of with multi-key indexes.

Note that what we can do is to create an index on address.street as seen below. This will also be a multi-key index and if we try the earlier find query now on the address.street, we would notice that mongoDB would now use an index scan on the multi-key index.

```
$ db.contacts.createIndex( { “addresses.street”: 1 } )
$ db.contacts.explain(“executionStats”).find( { “addresses.street”: “First Street” } )
```

We can therefore use an index on a field in an embedded document which is part of an array with the multi-key feature. We must be aware though that using the multi-key index feature on a single collection will quickly lead to some performance issue with writes because for every new document we add, all these multi-key indexes have to be updated. If we add a new document with 10 values in that array which we happen to store in a multi-key index, then these 10 new entries need to be added to the index registry. If we then have four or five of these multi-key indexes per document we would then quickly end up in a low performance world.

Multi-key indexes are helpful if we have queries that regularly target array values or even nested values or values in an embedded document in arrays.

We are able to create an index whereby we have a multi-key index that we add as part of a compound index which is possible as seen below:

```
$ db.contacts.createIndex( { name: 1,  hobbies: 1 } )
```

However, there is one important restriction to be aware of and that is a compound index made up of two or more multi-key indexes will not work, for example the below:

```
$ db.contacts.createIndex( { addresses: 1,  hobbies: 1 } )
```

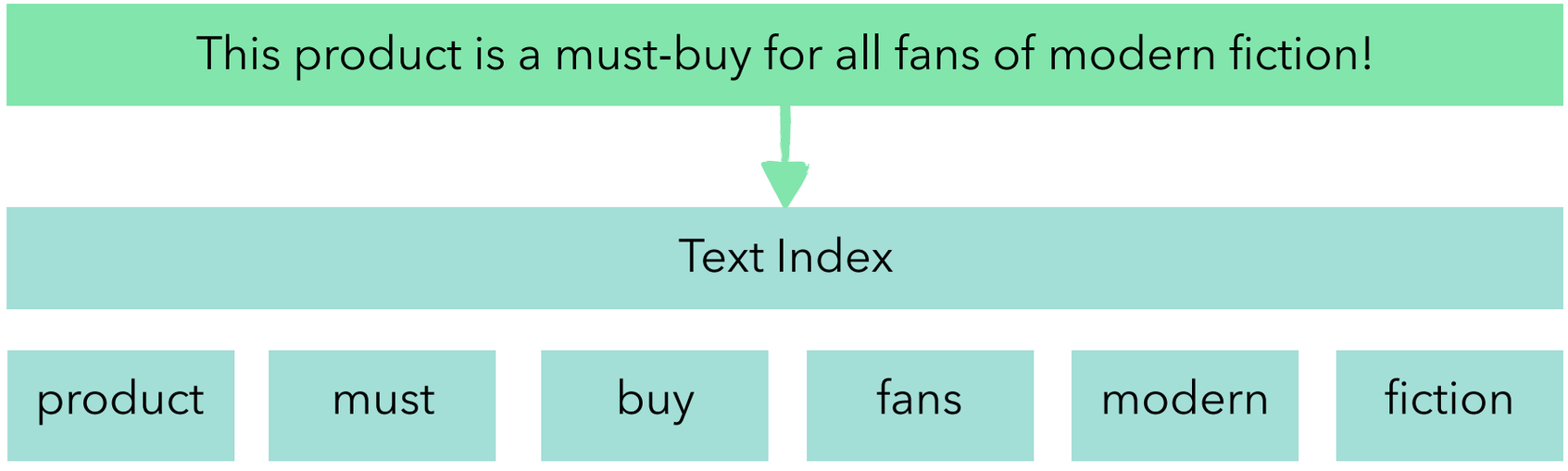
We cannot index parallel arrays because mongoDB would have to store the cartesian product of the values of both indexes, of both arrays, so it would have to pull out all the addresses and for every address it would have to store all the hobbies. So if we have two addresses and five hobbies, we would have to store ten values and this would become worse the more values we have addresses, which is why this is not possible.

Compound indexes with multi-key indexes are possible but only with one multi-key index i.e with one array and not multiple arrays. We can however have multiple multi-key indexes in separate indexes but in one and the same index only one array can be included.

Working with Indexes

Understanding Text Indexes

There is a special kind of multi-key indexes which is a text index. Lets take the below text as an example which could be stored in a field in our document as some kind of product description.



If we want to search for the above text, we have previously seen that we could use regex operator. However, regex is not a really great way of searching text as it offers very low performance. A better method is to use a text index which is a special kind of index that is supported by mongoDB which will essentially turn the text into an array of single words and will store it as such. A extra thing mongoDB does for us is that it removes all the stop words and it stems all words so that we have an array of keywords and words such as “is” or “a” etc. are not stored because they are not typically something we would search on as they would appear all over the place. The keywords are what matters for text searches.

Using the below example of a products collection, we will explore the syntax to setup a text index:

$ db.products.insertMany([{ title: “A Book”, description: “This is an amazing book about a young explorer!” }, { title: ““Red T-Shirt”, description: “This T-Shirt is red and it’s pretty amazing.” }])

```
$ db.products.createIndex( { description: 1 } )
$ db.products.createIndex( { description: “text” } )
```

We would create the index the same as we would do for any other indexes, however, the important distinction is that we do not add the 1 or -1 for ascending/descending. We could add this but then the index will be a single field index and we can search exactly for the whole text to utilise this index but we cannot search for the individual key words. Instead, we would add the special “text” keyword which will let mongoDB know to create the text index by removing all the stop words and store the keywords in an array.

When performing the find command, we can now use the $text and $search keys to search for the keyword. The casing is not important as every keyword is stored as lowercase.

```
$ db.products.find( { $text: { $search: “amazing” } } )
```

We do not specify the field in which we we want to search on because we are only allowed to have one text index per collection because text indexes are very expensive especially if we had a lot of long text that has to be split up, we do not want to do this for example ten times per collection. Therefore, we only have one text index where the $search can look into.

We can actually merge multiple fields into one text index and we will then look through them automatically which we will see in the later section.

Note if we look for the keyword “red book” this will find both documents as it treats each word as individual keywords and will search all documents which has red and all documents which has book. If we want to specifically want to find the word red book which is treated as one keyword then we would have to wrap the text in double quotes like so:

```
$ db.products.find( { $text: { $search: “\”red book”\” } } )
$ db.products.find( { $text: { $search: “\”amazing book”\” } } )
```

Because we are already in double quotes, we would need to add a backward slash at the beginning and end of the phrase to escape them. This will not find anything in the collection because we do not have a red book phrase anywhere in our text (“*amazing book” would work though*).

Text indexes are very powerful and much faster than regular expressions and this is definitely the way to go if we need to look for keywords in text.

Working with Indexes

Text Indexes and Sorting

If we want to find texts from a text index, however, we would want to order the returned documents where the closest matches are at the top, this is possible in mongoDB.

For example, if we want to search for “amazing t-shirt”, this will return both documents because the amazing keyword exists in both document. However, we would rather have the t-shirt product appear before the book because it is the better match as it has both keywords in the description.

```
$ db.products.find( { $text: { $search: “amazing t-shirt” } } ).pretty( )
```

MongoDB does something special when managing/searching text indexes – we can find out how it scores its results. If we use projection as the second argument to our find method in order to project the score, we can use the $meta operator to add the textScore. The textScore is a meta field added/managed by mongoDB for text searches i.e. $text operator on a text index.

```
$ db.products.find( { $text: { $search: “amazing t-shirt” } }, { score: { $meta: “textScore” } } ).pretty( )
```

We would see the score mongoDB has assigned to a result and it automatically sorts all returned documents by the score. To make sure that the returned documents are sorted we could add the sort command as seen below – however, this is a longer syntax and the above already sorts by the score:

```
$ db.products.find( { $text: { $search: “amazing t-shirt” } }, { score: { $meta: “textScore” } } ).sort( { score: { $meta: “textScore” } } ).pretty( )
```

We can therefore use the textScore meta managed by mongoDB to sort the returned results for us.

Creating Combined Text Indexes

As previously mentioned we can only have one text index per collection. If we look at the indexes using the below syntax we would notice that the default_language for the text index is English which we are able to change which we will see later.

```
$ db.products.getIndexes( )
```

If we try to add another text index to the same collection but now on the title like so:

```
$ db.products.createIndex( { title: “text” } )
```

Notice that we would now receive an IndexOptionsConflict error in the shell and this is because we can only have one text index per collection. However, what we can do is merge the text of multiple fields together into one text index. First, we would need to drop the existing text index – dropping text indexes is a little harder as we cannot drop by the field name (i.e. { title: “text” } will not work), rather we need to use the text index name.

```
$ db.products.dropIndex( { title: “text” } )
$ db.products.dropIndex(“description_text”)
```

Now that we have dropped the existing text index from the collection, we can now create a new text index combining/merging multiple fields.

```
$ db.products.createIndex( { title: “text”, description: “text” } )
```

Ultimately, we will still only have one text index in our collection; however, it will contain the keywords from both the title and description fields. We can now search for keywords that we have in the title for example we can search for the keyword book which appears in both the title and description or a keyword that only appears in the title and not the description and vice versa.

Using Text Indexes to Exclude Words

With text indexes not only can we search for keywords but we can also exclude/rule out keywords.

```
$ db.products.find( { $text: { $search: “amazing –t-shirt” } } ).pretty( )
```

In the example above, by adding the minus in front of the keyword, this will tell mongoDB to exclude any results that has the keyword t-shirt. This is really helpful to narrow down text search queries like the above where we find amazing products that are not T-Shirts or which at least don’t have T-Shirt in the title or in the description (*the above result will only return one document and not both as previously seen with the keyword of amazing t-shirt*).

Setting the Default Language & Using Weights

To drop an existing text index we would first need to search for the index name and then use the dropIndex command as seen below:

```
$ db.products.getIndexes( )
$ db.products.dropIndex("title_text_description_text")
```

If we now create a new index but now pass in a second options argument, we have two interesting options available to configure about our text indexes. The first option is the default language - we can assign the default language to a new value. The default language is English, but we can set this to a different language such as German — mongoDB has a list of supported languages we can use. This will determine how words are stemmed i.e. how prefixes are removed and what stop words are removed for example words like "is" or "a" are removed in English while words like "iste" and "deya" are removed in German. It is important to note that English is the default language but we can explicitly specify the option.

```
$ db.products.createIndex( { title: "text", description: "text" }, { default_language: "german" } )
```

The second option we have available to us is the ability to set different weightings for the different

fields we merge together. So in the below example we are merging the text and description fields together, however, we would want to specify that the description should be a higher weight. The weights are important for when mongoDB calculates the score of the results. To set up such weights, we can add the weights key in our config object and this key holds a document as a value where we reference the field name and assign a weights that are relative to each other.

```
$ db.products.createIndex( { title: “text”, description: “text” }, { default_language: “english”, weights: { title: 1, description: 10 } } )
```

The description will be worth/weigh in ten times as much as the title. If we search for our keyword in our products collection index we can not only search for a keyword but also set the language as seen below:

```
$ db.products.find( { $text: { $search: “red”, $language: “german” } } )
```

This is an interesting search option if we use a different way of storing the language for different documents. We can also turn on case sensitivity using the caseSensitive set to true. The default for caseSensitive is false, demonstrated below:

```
$ db.products.find( { $text: { $search: “red”, $caseSensitive: true } } )
```

If we print the score we would notice that the scoring will be weighted differently if we set weight options.

```
$ db.products.find( { $text: { $search: “red” } }, { score: { $meta: “textScore” } } ).pretty( )
```

Building Indexes

There are two ways in which we can build indexes; a foreground and a background.

So far we have always added indexes in the foreground with the createIndex just as we executed it. Something we did not notice because it always occurred instantly; but during the creation of the index the collection will be locked and we cannot edit the collection. On the other hand we can also add indexes in the background and the collection will still be accessible.

The advantage of the foreground mode is that it is faster and the background mode is slower. However, if we have a collection that is used in production, we probably do not want to lock it just because we are adding an index.

Foreground	Background
Collection is locked during index creation.	Collection is accessible during index creation.
Faster	Slower

We will observe how we can add an index in the background and see what difference it would make. To see the difference we can use the credit-rating.js file, and mongoDB shell can execute this file by simply typing mongoDB followed by the JavaScript file name.

```
$ mongo credit-rating.js
```

MongoDB will still connect to the server but it will then execute the file and basically execute the command in the .js file against the server. In this file we have a for loop that will add one million documents to a collection with random numbers. Executing this will take quite a while depending on our system and we can always quit the command using control + c on our keyboard or alternatively reduce the number of document created in the .js file for loop. Once completed we would have a new database and collection with one million documents in the collection.

```
$ show dbs
$ use credit
$ show collections
$ db.ratings.cound( )
```

We can use this collection to demonstrate the difference between both the foreground and background modes. If we were to create an index now on this collection, we would notice that the indexing does not occur instantly because we have a million documents although this can still be quick depending on our system.

To demonstrate the point where the foreground mode takes time to create but also blocks us from doing anything with the collection while it is creating the index, we would open a second mongoDB shell instance and prepare a query in that new shell.

```
$ db.ratings.findOne( )
```

In the first shell instance we would need to create the index, but then quickly change to the second shell instance to run the findOne query (as it does not take too long to create the new index).

```
$ db.ratings.createIndex( { age: 1 } )
```

We will notice the findOne( ) query does not finish instantly as it takes a while as it waits for the foreground mode index creating to complete before it can execute the query. There are no errors but the commands are being deferred until the index has been created.

For more complex indexes such as a text index or for even more documents etc. which would make the index creation take much longer. This will become a problem because the database or the collection might be locked for a couple of minutes or longer and therefore this is not an alternative for a production database. This is because we cannot suddenly lock down the entire database and the app can’t interact with the database anymore, which is why we can create indexes in the background.

To create a background mode index, we would pass in a second option argument to our createIndex command setting the background to true (the default is set to false, meaning indexes are created in the foreground).

```
$ db.ratings.createIndex( { age: 1 }, { background: true } )
```

If we now create the new index and in the second shell instance run a command, we can demonstrate that the database/collection is no longer locked during the index creation.

```
$ db.ratings.insertOne( { person_id: “dfjve9f348u6iew”, score: 44.2531, age: 60 } )
```

We should notice that the insertOne command should continue to work and a new document inserted immediately while the indexes is being created in the background. This is a very useful feature for a production databases as we do not want to add an index in the foreground in production especially not if the index creation will take quite a while.

Useful Links:

https://docs.mongodb.com/manual/core/index-partial/

https://docs.mongodb.com/manual/reference/text-search-languages/#text-search-languages

https://docs.mongodb.com/manual/tutorial/specify-language-for-text-index/#create-a-text-index-for-a-collection-in-multiple-languages

Adding GeoJSON Data

In the official documents we will find an article about GeoJSON and how it is structured and which kind of GeoJSON objects mongoDB supports. MongoDB supports all major important objects such as points, lines or polygons as well as special advanced objects.

https://docs.mongodb.com/manual/reference/geojson/

The most important thing is to understand how GeoJSON objects are created and creating it is very simple. To work with some data we can open up Google maps and work with different locations. On Google maps if we click on a location, we can easily access the coordinates of the place from inside the URL. The first coordinate is the latitude while the second coordinate after the comma is the longitude. We will need to remember this in order to store it correctly in mongoDB. The longitude describes a position on a vertical axis and the latitude describes a horizontal axis on the earth globe. With this coordinate system we can map any point onto our earth.

Below is an example of adding a GeoJSON data in mongoDB.

```
$ use awesomeplaces
$ db.places.insertOne( { name: “California Academy of Sciences”, location: { type: “Point”, coordinates: [ –122.46636, 37.77014 ] } } )
```

There is nothing special about GeoJSON as we can add any key name we want i.e. location, loc or something completely different. What matters with GeoJSON data is the structure of the value. The value should be an embedded document and in that embedded document we need two pieces of information, the type and the coordinates. The coordinates is an array where the first value has to be longitude and the second value is the latitude. The type must be one of the supported types by mongoDB such as point.

We have now created a GeoJSON object and mongoDB will treat the document as a GeoJSON object because it has fulfilled the requirements of having a type which is one of the supported objects and having coordinates which is an array where the first value is treated as a longitude and the second value is treated as a latitude.

Running Geo Queries

We may have a web application where users can locate themselves, we can do this through some web API or a mobile app where the use can locate themselves. Location APIs will always return coordinates in the form of latitude and longitude which is the standard format. Our application will give us some latitude and longitude data for whatever the user did, for example locating themselves.

We can simulate this by taking another location from google maps to query whether the location we created in the previous section is located near the new location coordinates.

```
$ db.places.find( { location: { $near: { $geometry: { type: "Point", coordinates: [ -122.471114, 37.771104 ] } } } } )
```

The location will relate to what we named the key and is not a special reserved key (i.e. if we called this loc then we would need to use loc). The $near operator provided by mongoDB is a operator for working with geospatial data. The $near operator requires another document as a value and in there we can now define a $geometry for which we want to check if it is near to. The $geometry takes in a document which describes a GeoJSON object. We could check here if a point we add her is close to our point.

The above query requires a geospatial index in order to run this query without running into any errors (*the above query will not run and will error*), but not all geospatial queries require index but they all, just as with other indexes, will most likely benefit from having such an index.

Important note: The $near operator requires a geospatial index whereas the other operators such as $geoIntersect and $geoWithin operators (which we will look at in later sections) do not require an index although an index can help speed the search query.

Adding Geospatial Index to Track the Distance

In the previous section query example, this would have failed because we had no geospatial index. A geospatial index is required for the $near query operator. So the question now is how do we add such an index? This is quite straight forward and is similar to how we create other indexes.

```
$ db.places.createIndex( { location: “2dsphere” } )
```

In the above example we add the index to the location field but note that if we called this field something different such as loc then we would need to use that field name. The difference is the type of index, we do not use/sort by ascending or descending (1 or -1) or the text index but rather a special 2dsphere index. This will create a geospatial index on the location field.

If we now repeat the geospatial query we had from the previous section, it should now succeed without giving any error message in the console.

```
$ db.places.find( { location: { $near: { $geometry: { type: “Point”, coordinates: [ –122.471114, 37.771104 ] } } } } ).pretty( )
```

Now this will find the one point but it does not really tell us too much i.e. how is near defined? The $near operator does not make much sense unless we restrict it. Typically, we would not just pass in a

geometry $near as we have done above, but we would also pass in another argument and define a max and maybe also a min distance.

```
$ db.places.find( { location: { $near: { $geometry: { type: “Point”, coordinates: [ –122.471114, 37.771104 ] }, $maxDistance:  30, $minDistance: 10 } } } ).pretty( )
```

The $maxDistance and $minDistance is simply a value in meters. The above would look for a location that is a minimum distance away of 10 meters and a maximum distance of 30 meters. If we run this query, this will not find any geolocation because the California Academy of Sciences is further away from the location point we added in our query (the point in the query represents our current location). We can look on Google maps and measure the distance by right clicking on the map and selecting measure distance and we would notice the distance is around 435.36 meters. If we run the below query but update the maximum distance to a larger number, we should have one geospatial location point returned from our find query.

```
$ db.places.find( { location: { $near: { $geometry: { type: “Point”, coordinates: [ –122.471114, 37.771104 ] }, $maxDistance:  500, $minDistance: 10 } } } ).pretty( )
```

This query allows us to find the nearest places/locations within a certain radius of our current location which is an often question we would want to answer in an application that uses geolocations.

Adding Additional Locations

In the last section we answered the question of which points are near to our current location point. The next common typical question we want to answer: we have a sphere or polygon area and want to know what points are inside of that area?

In order to answer the above question, we need to learn how to add more points to our database before we can dive into the above question. Below is a sample syntax of adding more new locations to our database collection:

```
$ db.places.insertOne( { name: “Conservatory of Flowers”, location: { type: “Point”, coordinates: [ –122.4615748, 37.7701756 ] } } )
$ db.places.insertOne( { name: “Golden Gate Park Tennis Courts”, location: { type: “Point”, coordinates: [ –122.4593702, 37.7705046 ] } } )
$ db.places.insertOne( { name: “Nopa”, location: { type: “Point”, coordinates: [ –122.4389058, 37.7747415 ] } } )
```

We should now have a total of four locations added to our database collection to answer the above question in the next section. Note: when taking locations from google maps the coordinates are based on our screen and if not centred properly the coordinates may be slightly off.

Finding Places Inside a Certain Area

In the previous section we asked wanted to answer the question: what locations are within a certain sphere/polygon area?

Using Google Maps to make it easier to run this query, we can go into the Menu and select Your Places tab and within the Maps tab create a new Map (note: we may need to be logged into our Google account to use this feature). This will make it easier to navigate around and find coordinates. To draw a polygon around the area we wish to capture for our query we can use the add marker to mark locations to get the exact coordinates (we could delete these once we are done with them). We can use this information within our query:

Note: we can create variables within the shell to store these variables so that we can use this within our query as the shell uses JavaScript.

```
$ const p1 = [ –122.46636, 37.77014 ]
$ const p2 = [ –122.45303, 37.76641 ]
$ const p3 = [ –122.51026, 37.76411 ]
$ const p4 = [ –122.51088, 37.77131 ]
```

These four coordinates will act as our area where we want to query what locations exists within this area and the returned query should bring back only 3 out of the 4 locations within our collection.

Below is the syntax to query what locations are within our polygon area:

```
$ db.places.find( { location: { $geoWithin: { $geometry: { type: “Polygon”, coordinates: [ [ [ –122.4547, 37.77473 ], [ –122.45303, 37.76641], [ –122.51026, 37.76411 ], [ –122.51088, 37.77131 ], [ –122.4547, 37.77473 ] ] ] } } } } ).pretty( )
```

Or using the variables, the syntax would look something like the below:

```
$ db.places.find( { location: { $geoWithin: { $geometry: { type: “Polygon”, coordinates: [ [ p1, p2, p3, p4, p1 ] ] } } } } ).pretty( )
```

The $geoWithin operator provided by mongoDO allows us to find all elements within a certain shape, typically a polygon, within a certain object. The $geoWithin takes a document as a value and inside here we can add a geometry object which is simply a GeoJSON object.

Note that the type is now not a Point but a Polygon which has an array of coordinates of a polygon. A Point used a pair of coordinates whereas a Polygon uses more than a single pair of coordinates and therefore we must use a nested array i.e. an array within an array. Within that array we then again add more arrays, where each array now describes one longitude and latitude pair for each corner of the polygon. We can use the const variables which store our array coordinates of each point. We require the p1 variable again at the end of the array because the polygon must end with the starting point to close the polygon. This should return back the results of locations/points within the polygon area.

Finding Out if a User is Inside a Specific Area

In another typical use case scenario would be to check whether a user is in a certain area. For example, we do not want to find all places in an area but we want to store a couple of different areas potentially in the database e.g. neighbourhoods of a city, and the user sends some coordinates because they have located themselves and we want to know which neighbourhood the user is located. Essentially the opposite of the the last section query.

We can store the multiple polygons within our database within a new collection called areas (we could name this whatever we want) as seen below:

```
$ db.areas.insertOne( { name: “Golden Gate Park”, area: { type: “Polygon”, coordinates: [ [ p1, p2, p3, p4, p1 ] ] } } )
```

To check whether the user is within the area we would first need to create an index for our areas collection.

```
$ db.areas.createIndex( { area: “2dsphere” } )
```

Now that we have our 2dsphere index created on our areas collection on the area field we can now run our find query to see if the user is inside of a specific area.

```
$ db.areas.find( { area: { $geoIntersects: {$geometry: { “Point”, coordinates: [ -122.49089, 37.76992 ]
} } } } ).pretty( )
```

The $geoIntersects operator gets used on the area field to return the area where the user point intersects the geometry area. The area field is used because this is where we stored the indexed areas that we are looking for in our query.

The $geoIntersects operator returns all areas that have a common point or common area. The reason we cannot use $geoWithin because we cannot find the areas within the point where the user is and the area will never be within that point. The other way round is to check whether the user point intersect with the area and if the answer is yes then the user is within the area and we can return the area because this is what we are trying to find out i.e. which area is the user in.

We pass in a $geometry document which is now just a Point for the users coordinates which is an array of a single longitude and latitude pair.

If the point is within multiple areas then we would be returned all the areas that point intersects. Not only can we intersect points within an area but we could also intersect areas within areas to see if the two areas touch one another. If no area intersect then we would retrieve no results.

Finding Places Within a Certain Radius

To conclude the geospatial queries we can run, the final query is to return all places in a certain radius around the user. We kind of looked at this before using the $near operator, however, there is an alternative method. The $near operator also sorted the results which is an important difference to this alternative approach.

We want to find all elements in an unsorted order that are within a certain radius:

```
$ db.places.find( { location: { $geoWithin: { $centerSphere: [ [ –122.46203, 37.77286 ], 1 /
6378.1 ] } } } ).pretty( )
```

We want to use the $geoWithin and not the $geoIntersect operator because we want to find all places within a place or area which we will define in the query. Previously we had the opposite, where we had a point in the query and wanted to find an area that surrounds the point.

The $geoWithin operator takes in a $geometry operator which describes the GeoJSON object, however, there is a special operator we can use in mongoDB and that is the $centerSphere operator. This operator allows us to quickly get a circle around a point. Essentially it uses a radius and a centre to give us a whole circle.

The $centerSphere takes in an array as the value and that array has two elements. The first element is another array that holds the coordinates of the centre of the circle we want to draw while the second element is the radius itself.

The radius needs to be translated manually from meters or miles to radians. This conversion is relatively easy to do and we can find an article in the mongoDB documentations. This article explains how to translate miles to radians or kilometres. In the above example we worked with kilometres (i.e. we wanted a 1km radius = 1 / 6378.1).

https://docs.mongodb.com/manual/tutorial/calculate-distances-using-spherical-geometry-with-2d-geospatial-indexes/

The important difference between the $near and $geoWithin & $centerSphere operator is that the former operator returns an ordered list while the latter returns an unordered list which keeps the order of the elements in the database (we can manually sort this using the sort method). The $near operator gave us elements in a certain radius and sorted them by proximity. This really depends on our requirements and what we need i.e. do we need a sorted results with the nearest results first or do we want an inserted results and we just want to get a list of elements in general?

Useful Links:

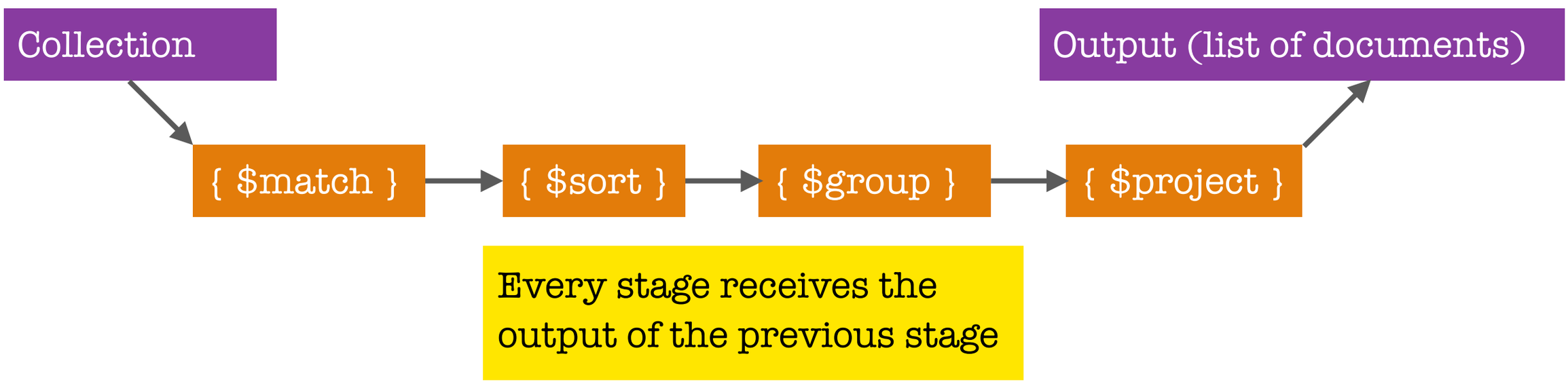
https://docs.mongodb.com/manual/geospatial-queries/

https://docs.mongodb.com/manual/reference/operator/query-geospatial/

What is the Aggregation Framework?

The aggregation framework in its core is just an alternative to the find method. We have our collection and the aggregation framework is all about building a pipeline of steps thats runs on the data that is retrieved from our collection and then gives us the output in the form we need it.

These steps are sometimes related to what we already know from the find for example the match is equivalent to filtering in the find method. There are a lot of different steps we can combine as we want and we can reuse certain stages and therefore we have a very powerful way of modelling our data transformation.



We can have a very structured way of having some input data and then slowly modify it the way we need it in the end.

Getting Started with the Aggregation Pipeline

To get started with the aggregation pipeline, we require some data. We will use the persons data by importing the file using the mongoimport command ensuring the command is run in the directory where we have stored the person.json file.

```
$ mongimport persons.json –d analytics –c persons ––jsonArray
```

The above will store the import file in a database called analytics within the persons collection. We use the jsonArray flag because the data contains an array of documents and therefore the flag is required in the import query. This should import 5000 documents into the database.

We can now connect to the database within the shell and we should see the persons collection in the analytics database using the commands below. We can use the findOne to see how a single document looks. We will use this dataset to learn how to use the aggregation framework on this dataset.

```
$ use analytics
$ show collections
$ db.persons.findOne( ).pretty( )
```

Using the Aggregation Framework

To use the aggregation framework with the persons collection we created in the last section; instead of running the find, findOne or findMany command on the persons collection we now run the aggregate command.

```
$ db.persons.aggregate( [ {…}, {…}, {…} ] )
```

The aggregate method takes in an array whereby we define a series of steps that should be run on our data. The first step will receive the entire dataset right from the collection and then the next step can do something with the data returned by the first step and so on.

It is important to note that the aggregate does not go ahead and fetch all the data from the database and then give it to us and then do something on it. The first step runs on the database and can take advantages of indexes, so if we filter or sort in the first step then we can take advantage of the index and therefore we do not have to fetch all the documents just because we are using the aggregate. Aggregate (the same as find) execute on the mongoDB server and therefore can take advantages of things like the indexes.

Every step is a document. Below is an example and has been separated onto multiple lines to make

it easier to read the syntax of each step within the aggregate.

```
$ db.persons.aggregate( [
… { $match: { gender: “female” } }
] ).pretty( )
```

The match is simply a filtering step. We define some criteria on which we want to filter our data in the persons collection. We can filter here in the same way we can filter in the find command. We could at this stage finish our pipeline by closing the square brackets around our stages. Just like the find method, the aggregate method returns a cursor.

Understanding the Group Stage

Following on from the last section, we can now add a new $group stage onto our aggregate pipeline.

```
$ db.persons.aggregate( [
… { $match: { gender: “female” } },
… { $group: { _id: { state: “$location.state”, totalPersons: { $sum: 1 } } } }
] ).pretty( )
```

The group stage allows us to group our data by certain field or by multiple fields. There are a couple of parameters that we need to define, the first being the _id field. Thus far we have always used an objectId, string or maybe a number but we have never used a document. Just as with any other field we can assign a document to the _id field it is just not that common. We often see the document used in the group stage syntax because it will be interpreted in a special way an it will basically allow us to define multiple fields by which we want to group.

In the above we have grouped by location state, and we do this by assigning a key (which we can name as anything we want) and then the value of $location.state – the dollar sign is important because it tells mongoDB that we are referring to a field of our document which is passed into the group stage. This will group our results by the state. We can now add a new key to each document and we can name this as whatever we want, in the above we used totalPersons. We would pass in a document where we now describe the kind of aggregation function we want to execute and these functions (accumulation operators) can be found in the official mongoDB documentation.

In the above we used the $sum accumulation operator and passed in a value we want to add for every document that is grouped together. For example, if we have three people from the same location state, the sum would be incremented by 1, three times i.e. 1x3 = 3. MongoDB will keep the aggregated sum in memory until it is done with a group and then writes the total sum into the totalPersons field.

It is important to understand that group accumulates data which simply means that we may have multiple documents with the same state but the group will only output one i.e. the three documents with the same state will be merged into one because of the aggregating (we are building a sum in this case).

We now have a totally different data output. We no longer have any person data because we changed it. We used group to merge our documents into new documents with totally different data with the total persons and that ID. The ID is an object we defined with the state in which its grouped.

We can verify that our aggregation is working correctly by manually reaching out to our persons collection and finding all persons where the location is equal to a state e.g. Sinop and then sum those that are female from the results.

```
$ db.persons.find( { “location.state”: “sinop” } ).pretty( )
```

This is the group stage in action within the Aggregation framework.

Diving Deeper Into the Group Stage

When we group our data using the aggregation pipeline we saw that we lost all the existing data but we are typically fine with losing the data because we are grouping them together.

When we ran our pipeline method we got a bunch of outputs in a totally unsorted order. We are of course able to sort our data and one advantage of the aggregation pipeline is that we are able to sort at any place in the pipeline. If we want to sort on the amount of persons in a state (i.e. the totalPersons field) we can only sort after we have grouped the data.

```
$ db.persons.aggregate( [
… { $match: { gender: “female” } },
… { $group: { _id: { state: “$location.state”, totalPersons: { $sum: 1 } } } },
… { $sort: { totalPersons: -1 } }
] ).pretty( )
```

We use the $sort stage which takes in a document as an input to define how the sorting should happen. We can sort as we have seen previously, however, we are now able to use the totalPersons field from the previous group pipeline stage.

Each pipeline stage passes some output data to the next stage and that output data is the only data

that the next stage has. So in the above, the sort stage does not have access to the original data as we fetched it from the collection, it only has access to the output data of our group stage. So in there we only have a totalPersons field that we can now sort by, in the above we sorted in descending order i.e. highest value first.

As we can see, we already have a lot of power with the first three tools that we have seen in the aggregation framework. This is a kind of operation we could not perform with the normal find method because we could not group and then sort on the result or group. We would have had to do that in the client side code using the find method. Using the aggregation framework, we can run the aggregate on the mongoDB server and simply get back the data in the client that we need in our client to work with.

Working with $project

We are now going to look at a different pipeline stage called $project that allows us to transform every document instead of grouping multiple together. We already know projection from the find method, however, the project becomes more powerful as an aggregate stage.

If we start simple and we want to just transform every document using only the $project stage.

```
$ db.persons.aggregate( [
… { $project: { _id: 0, gender: 1, fullName: { $concat: [ “$name.first”, “ “, “name.last” ] } } }
] ).pretty( )
```

As with all stages, the $project takes in a document as its value to configure the stage. In its most simplest form, the project works in the same was as the projection work in the find method. In the above we do not want the id, but we want the gender, name, location and email field. Notice that we are able to add new fields here and also reformat some of the fields; for example the name field is now called fullName ad treats the name data as one field rather than the embedded document with separated the title, first and last name.

Using the $concat operator allows us to concatenate multiple strings together. We would need to pass in an object as the value to fullName because we are performing an operation. We pass in an array of strings we wish to join together. We can either hard code these string values or use the fields from our collection. In order to use our data it is important to wrap the field in double quotations and then use the dollar $ to name the field name – using dot notation for embedded documents. MongoDB will know from the $ that the value is not hardcoded and is coming from field value instead. Concatenate does not add white spaces and we must add these ourselves.

If we run the aggregate method we get the same amount of documents s before because unlike group the project does not group multiple documents togehter, it simply transforms every single

document and therefore we get the same amount of documents but with a totally different data. Interestingly, not only can we include and exclude data but we can also add new fields with hardcoded values or derived values from the data in the documents if we wanted to.

If we wanted to make sure the first and last name start with uppercase characters, we are also able to do this in the projection stage. Below is an example broken in a much more readable syntax:

```
$ db.persons.aggregate( [
…   { $project: {
…         _id: 0, gender: 1,
…         fullName: { $concat: [
…                 { $toUpper: { $substrCP: [ “$name.first”, 0, 1 ] } },
…                 { $substrCP: [ “$name.first”, 1, { $subtract: [ { $strLenCP: “$name.first” }, 1 ] } ] },
…                 “ “,
…                 { $toUpper: { $substrCP: [ “$name.last”, 0, 1 ] } },
…                 { $substrCP: [ “$name.last”, 1, { $subtract: [ { $strLenCP: “$name.last” }, 1 ] } ] },
…         ] }
…   } }
] ).pretty( )
```

We can use the $toUpper operator (note: all operators are wrapped in an object) to upper case the whole string.

The $substrCP operator returns the substring of a string. The $substrCP operator uses an array with the first argument of the string, second argument the starting character point from the string and the last argument of how many characters should be included in the substring (this uses zero indexing).

The $subtract operator allows us to return the difference of two numbers, while the $strLenCP operator calculates the length of a string.

As you can see we can nest operators to transform our documents in the project stage and this in not uncommon. The more we get use to the different operators the more powerful transformations we can perform on our data within the project stage and is something we would get better at the more we practice.

Important Note: it may be easier to use a text editor such as Visual Studio Code to write our complex $project transformation code rather than the terminal as it will allow us to easily separate our code into multiple lines without executing the code. This would also allow us to read our code much more easily and ensure there are no syntax errors in the code.

Turning the Location Into a GeoJSON Object

We can prepare the location data into a GeoJSON object so that we can later work with it.

Important Note: We can have have multiple $project stage and this is not uncommon for example we could do some matching, sorting, grouping and then we project and then maybe we sort again (so often we have some in-between stages).

```
$ db.persons.aggregate( [
{ $project: { _id: 0, name: 1, email: 1,  location: { type: “point”, coordinates: [
      { $convert: { input: “$location.coordinates.longitude”, to: “double”, onError: 0.0, onNull: 0.0 } },
      { $convert: { input: “$location.coordinates.latitude”, to: “double”, onError: 0.0, onNull: 0.0 } }
] } } },
{ $project: { gender: 1, email: 1, location: 1, … } }
] ).pretty( )
```

We want the coordinates from the location field however if the field values are of the wrong format type we can transform this into the correct type by hardcoding the type as seen above.

The original coordinates are in a nested document and of the type string, however we can easily change this into an array with the type of number for the longitude and latitude coordinates. Converting data is something we would have to do often. In the above we converted the string data for the longitude and latitude into a number using mongoDB $convert operator.

The $convert operator takes in a couple of fields. The first is the input and the second is the to field which defines the type we want to convert the input field value to. We can finally define the onError and onNull values i.e. the default values to be returned in case the transformation fails. This will transform the original data type to the new required data type we want returned from our aggregate pipeline.

Using Shortcuts for Transformations

In the previous section we looked at the $convert operator and how we can use it to transform data from one type to another. Below is another example of transforming the dob.date from a string to a date type. We can omit the onError and onNull should we wish to:

```
$ db.persons.aggregate( [
    { $project: { _id: 0, name: 1, email: 1, birthdate: { $convert: { input: "$dob.date", to: "date" } } } }
] ).pretty( )
```

We should now have a special ISODate( ) date object/type for out birthdate rather than a string. Also note that If we have multiple $project stages whereby the first projection has new fields added, we must also include these new fields in the second projection stage in order output the results.

This demonstrates how we are able to transform our data considerably from the original data using the aggregate framework.

If we require a simple conversion where we do not specify onError and onNull values, as seen above, we can therefore use a shortcut. MongoDB has special operators starting with to such as $toDate, $toDecimal, $toBool, $toString, etc. which are shortcuts if we need to do a specific transformation. So in the above case rather than using the $convert operator we could use the $toDate operator passing in the field we wish to convert which will transform the original data into a ISODate( ) date object.

```
$ db.persons.aggregate( [
      { $project: { _id: 0, name: 1, email: 1, birthdate: { $toDate: “$dob.date”, to: “date” } } }
] ).pretty( )
```

These shortcuts will help us reduce the amount of syntax we would have to write and we should be weary of these simple transformations and use these shortcuts. If we want to specify the onError and onNull fallback values because we have incomplete data in our dataset, then we would have to use the $convert operator.

Understanding the $isoWeekYear Operator

Having looked at the transformation in the last section, we can now look at a scenario where we want to group the result data by the new birthdate field but not by the date but by the birth year instead.

```
$ db.persons.aggregate( [
    { $project: { _id: 0, name: 1, email: 1, birthdate: { $toDate: "$dob.date", to: "date" } } },
    { $group: { _id: { birthYear: { $isoWeekYear: "$birthdate" } }, numPersons: { $sum: 1 } } },
    { $sort: { numPersons: -1 } }
] ).pretty( )
```

The $isoWeekYear operator returns the year of the date. The above will therefore group the by the year returning the number of people in our dataset born in a particular year. We can also sort the returned grouped data using the $sort stage either ascending or descending order.

The $group vs $project Stage

It is really important for us to understand the difference between the $group and $project Operators. The $group operator is for grouping multiple documents into one document whereas the $project operator is a one to one relation i.e. we would get one document and return one document but that one document would have changed.

So for group we have multiple documents and we return one grouped by one or more categories of our choice and also with any new fields with some summary/statistic calculations. In projections we have the one for one relation.

Therefore in grouping we do things such as summing, counting, averaging and so on while in the projection phase we transform a single document, add new fields and so on. This is a very important difference to understand and to get right.

Pushing Elements into Newly Created Arrays

We are now going to look at operators and stages that will help us with working with arrays. To help us demonstrate this, we can import a new array dataset to work that contains arrays. We are able to do quite a lot of things with arrays in the aggregation framework.

One thing we often want to do with arrays is to merge or combine arrays into a grouping stage.

```
$ db.friends.aggregate( [
    { $group: { _id: { age: "$age" }, allHobbies: { $push: "$hobbies" } } }
] ).pretty( )
```

There are two operators that help with combining array values, the first is the $push operator. The $push operator allows us to push a new element into the allHobbies array for every incoming document.

What we will see is that we have two groups of age 30 and age 29 and allHobbies is an array of arrays (nested) because we pushed the hobbies into our array and hobbies happens to be an array itself. This is what we meant that we can push any values. What if we wanted to push existing array

values but not as an array but we went to pull these values out of the hobbies array and then add them to the allHobbies array?

Understanding the $unwind Stage

So in the previous section we saw how we can push elements into newly created arrays, however, the values we push do not have to be arrays themselves. We left off with the question of how can we have the hobbies array values in the allHobbies array but not in their nested array. To do this we have a new pipeline stage we can use called the $unwind stage.

The $unwind stage is always a great stage when we have an array of which we want to pull out the elements. The $unwind has two different syntaxes but in its most common usage, we just pass the name of a field that holds an array.

```
$ db.friends.aggregate( [
      { $unwind: “$hobbies” }
] ).pretty( )
```

The $unwind flattens the array by repeating the document that held the array as often as needed to

merge it with the array elements. So the original array of Max simply has sports and cooking and therefore Max was repeated twice i.e. two new documents – Max with hobbies sport and Max with hobbies of cooking and the same is true for all other elements.

So where the $group stage merges multiple documents into one, the $unwind stage takes on document and spits out multiple documents. Now with that, we can group again but take hobbies which is no longer an array but a single value – as seen below:

```
$ db.friends.aggregate( [
    { $unwind: “$hobbies” }
    { $group: { _id: { age: “$age” }, allHobbies: { $push: “$hobbies” } } }
] ).pretty( )
```

This will now solves the original question where we still have two groups for the age (29 and 30) but the allHobbies now holds just an array of values and not an embedded array. However, we would notice that this can have duplicate values. So how can we solve this problem?

Eliminating Duplicate Values

We may not want to have duplicate values in our aggregate returned results. To avoid duplicate values, we can use an alternative to the $push operator called the $addToSet operator.

```
$ db.friends.aggregate( [
    { $unwind: “$hobbies” }
    { $group: { _id: { age: “$age” }, allHobbies: { $addToSet: “$hobbies” } } }
] ).pretty( )
```

The $addToSet does almost the same as the $push operator; however, we no longer see any duplicate values because $addToSet essentially pushes but avoids duplicate values. If it finds that an entry already exists it does not push the new value.

So with the $unwind stage, the $push and $addToSet operators (in the $group stage), we have some powerful features that should help us manage our array data efficiently and transform them into whichever format we require.

Using Projections with Arrays

We are now going to look at projections with arrays. Looking at the friends data and the examScores array value, what if we only wanted to output the first value of that array instead of all the examScores values from the array?

```
$ db.friends.aggregate( [
    { $project: { _id: 0, examScore: { $slice: [ "$examScores", 1 ] } } }
] ).pretty( )
```

The $slice operator allows us to get back the slice of an array. The $slice operator takes in an array itself and the first value is the array itself we wish to slice. This first value can be hardcoded as well as pointing to a collection field. The second argument is the number of elements we want to slice from the array seen from the start. In the above we sliced the first array value. If we use a negative value as the second value for example -2, mongoDB will slice the last two values from the array.

If we want to retrieve one element but starting at the second element then the syntax will be slightly different where the second argument number is the starting position and the third argument is the number of elements we wish to slice. Indexing is zero based.

```
{ $project: { _id: 0, examScore: { $slice: [ "$examScores", -2 ] } } }
{ $project: { _id: 0, examScore: { $slice: [ "$examScores", 2, 1 ] } } }
```

Getting the Length of an Array

To get the length of an array for example if we wanted to know how many exams a friend took? This can be achieved in the projection stage using the $size operator which calculates the length of an array.

```
$ db.friends.aggregate( [
      { $project: { _id: 0, numScore: { $size: “$examScores” } } }
] ).pretty( )
```

We can either hard code the array or point the operator to the field which holds the array for which we want to calculate the length and this will get stored in the new numScore (or whatever we wish to call this) projection field.

Using the $Filter Operator

What if we want to transform examScores to be an array where we only see scores higher than 60? Now this can also be done in the projection stage because we want to transform every single record but we do not want to group by anything. We just want to transform the array in there.

```
$ db.friends.aggregate( [
    { $project: { _id: 0, scores: { $filter: { input: "$examScores", as: "sc", cond: { $gt: [ "$$sc.score",
    60 ] } } } } }
] ).pretty( )
```

The $filter operator which allows us to filter out certain elements in an array and only return elements that fulfil a certain condition. So in our scenario we want to filter for the score being greater than 60. The first argument is the input i.e. the array we wish to filter. The second argument we assign a temporary name which is known as the local variable and we can name this whatever we want. The last argument is the condition. This last argument can take a bunch of expression operators and in our example we used the greater than operator.

Note that the $gt operator works a bit differently than when we use it in match or find. Here the $gt takes an array of values it should compare which makes sense as we are now in the context of another operator. We want to compare the temporary variable (i.e. sc local variable) which will refer to the different values in our examScores and see if it is greater than 60. The filter operator will execute over an over again on all the elements in the array and compares to condition to filter the list or returned array. One $ sign will tell mongoDB to look for a field name while $$ sign will tell mongoDB to refer to the temporary variable.

Applying Multiple Operations to Arrays

What if we wanted to transform our friend array so that we only output the highest exam score for every person so that we no longer have the examScores array but we still only get three person results, however, we have no examScores but only the highest exam score?

There may be multiple ways of achieving the above but the below is one solution to solving the above problem.

```
$ db.friends.aggregate( [
    { $unwind: “$examScores” },
    { $project: { _id: 1, name: 1, age: 1, score: “$examScores.score” } },
    { $sort: { score: -1 } }
    { $group: { _id: “$_id”, name: { $first: “$name” } , maxScore: { $max: “$score” } } },
    { $sort: { maxScore: -1 } }
] ).pretty( )
```

We would firstly need to unwind the examScores array to get multiple documents per person with the score being the top level element in order to then sort the documents by the score and then group them together by person and take the first score for that person. The $first operator tells mongoDB to return the first value it encounters and the $max operator allows us to get a maximum.

Understanding $bucket Stage

Sometimes we want to get a feeling for the distribution of the data we have and there is a useful pipeline that can help us with that called the $bucket stage. The $bucket stage allows us to output our data in buckets for which we can calculate certain summary statistics.

```
$ db.persons.aggregate( [
    { $bucket: { groupBy: "$dob.age", boundaries: [ 0, 18, 30, 50, 80, 120 ], output: { numPersons:
    { $sum: 1 }, averageAge: { $avg: "$dob.age" } } } }
] ).pretty( )
```

The $bucket stage takes a group by parameter (i.e a groupBy field) where we define by which field do we want to put our data into buckets. This groupBy argument tells the $bucket command where the input data is that we want to put into buckets. We then define some boundaries which are essentially our categories for our data. Finally, we want to define what we want to output into these buckets. This is where we define the structure of what we get back from our results. This allows us to get an idea of distribution, average age, etc. in each bucket

There is an alternative to the above which is called the bucketAuto which the name suggests, it does the bucketing algorithm for us. We define the groupBy, the number of buckets and the output.

```
$ db.persons.aggregate( [
    { $bucketAuto: { groupBy: “$dob.age”, buckets: 5, output: { numPersons: { $sum: 1 },
    averageAge: { $avg: “$dob.age” } } } }
] ).pretty( )
```

Now each bucket holds almost the same amount of values because mongoDB tries to derive equal distribution and bucketAuto can also be an even quicker way of getting a feeling for our data.

Understanding the Aggregate Framework

Diving into Additional Stages

We are going to look at some stages which we have to get the syntax right.

We want to find the 10 persons with the oldest birth date and thereafter we want to find the next 10.

```
$ db.persons.aggregate( [
    { $project: { _id: 0, name: { $concat: [ “$name.first”, “ “, “$name.last” ] }, birthdate: { $toDate:
    “$dob.date” } } },
    { $sort: { birthdate: 1 } },
    { $limit: 10 }
] ).pretty( )
```

The $limit stage allows us to limit the entries we want to see, in the above we set this to 10. Notice that there is no cursor as we have exhausted our cursor here because we only see 10 entries and therefore this is what we get back.

If we want to see the next 10 eldest persons, we have another command called $skip stage and it is important that this stage comes prior to the $limit stage as seen below.

```
$ db.persons.aggregate( [
    { $match: { gender: “male” } }
    { $project: { _id: 0, name: { $concat: [ “$name.first”, “ “, “$name.last” ] }, birthdate: { $toDate:
    “$dob.date” } } },
    { $sort: { birthdate: 1 } },
    { $skip: 10 }
    { $limit: 10 }
] ).pretty( )
```

This will skip the first 10 records and then show the next 10 records. The order of $sort, $skip and $limit is important. If we have $skip come after $limit we will have no results returned because $limit will return 10 documents and when we skip by 10 then this leads to zero. The order does not matter on the find method but it does in the aggregation pipeline because our pipeline is processed step

by step. The same is true for sorting whereby if we sort after $skip and $limit, we would get a totally different set of results. We will notice that we will return persons that are not that old and this is because we would have skipped the first ten persons in our dataset as its stored in the collection and then we take the next ten persons and then only sort on those ten persons we limited our output to. This is also true for the matching stage where the order is important to what is returned back from our aggregation pipeline.

This is very important concept to understand when ordering the aggregation pipeline stages. We should also note that mongoDB does some optimisation for us to optimise our pipelines, so it might very well fix the issue, but we shouldn't rely too much on this and we should always try to build correct pipelines with the correct order that optimises for performance and builds the kind of performance we want to have.

MongoDB actually tries its best to optimise our Aggregation Pipelines without interfering with our logic. More can be read about the default optimisations mongoDB performs:

https://docs.mongodb.com/manual/core/aggregation-pipeline-optimization/

Writing Pipeline Results Into a New Collection

We can take the result of a pipeline and write it into a new collection. To do this we need to specify another pipeline stage called $out stage for output. The $out stage will take the result of our operation and write it into a collection, either a new one or an existing one.

```
$ db.persons.aggregate( [ …
      { $out: “transformedPersons” }
] )
```

If we run the show collections command in the terminal we should see the new collection appear should we have outputted the results to a new collection that is created on the fly. The $out operator is great if we have a pipeline where we want to funnel our results right into a new collection.

Working with the $geoNear Stage

We are going to explore how to work with the $geoNear stage within mongoDB aggregation framework using the transformedPersons collection as an example.

First we would create a geospatial index on the locations field:

```
$ db.transformedPersons.createIndex( { location: “2dsphere” } )
```

Now with the index created, we can now use the transformedPersons collection for geolocation queries as well as the geolocation/geospatial aggregation pipeline stage. The $geoNear stage takes in a bunch of arguments to configure it.

```
$ db.transformedPersons.aggregate( {
    { $geoNear: {
        near:  { type: “Point”, coordinates: [ -18.4, -42.8 ] },
        maxDistance: 100000,
        num: 10,
        query: { age: { $gt: 30 } },
        distanceField: “distance”
    } }
} ).pretty( )
```

Firstly, we need to define the point where we are for which we want to find close points. This is because $geoNear allows us to simply find elements in our collection which are close to our current position. The near argument takes in a GeoJSON object.

The second argument defines the maxDistance in meters. We can limit the amount of results we want to retrieve in the num argument.

The third argument allows us to add a query where we can filter for other things. This is available because $geoNear has to be the first element in a pipeline because it needs to use the geospatial index. The first pipeline stage is the only stage with direct access to the collection while other pipeline stages just get the output of the previous pipeline stage. Therefore if we have any filters which we want to run directly on the collection we can add it here and mongoDB will execute a very efficient query against the collection and not force us to use a match stage, which will mean that we have to fetch all the data in order to be able to match in the next step.

Finally, we can use the distanceField argument. The $geoNear will give back the distance that is calculated between our point and the document it found and we can tell mongoDB in which new field it should store that value. We named this newfield distance but could be named as anything we want. We will notice, when executing the aggregation pipeline, in the output there is a new distance field added.

This is how we can use the $geoNear as a pipeline stage. The most important think to remember with $geoNear is that it must be the first pipeline stage and thereafter we can add all other stages that we have looked at before.

Useful Links:

Official Aggregation Framework:

https://docs.mongodb.com/manual/core/aggregation-pipeline/

Learn more about $project:

https://docs.mongodb.com/manual/reference/operator/aggregation/project/

Learn more about $cond:

https://docs.mongodb.com/manual/reference/operator/aggregation/cond/

Number Types an Overview

Which important number types do we have to differentiate? These are mostly integers, long integers (a.k.a long) and doubles which also has different types. To be precise we can work with the four number types shown below. The below table provides details on each number type as follow:

Integers (int32)	Longs (int64)	Doubles (64bit)	High Precision Doubles (128bit)
Only full numbers		Numbers with decimal places	
-2,147,483,648 to 2,147,483,647	-9,223,372,036, 854,775,808 to 9,223,372,036, 854,775,807	Decimal values are approximated	Decimal values are stored with high precision (up to 34 decimal digits)
Used for normal integers	Used for large integers	Used for floats where high precision is not required	Used for floats where high precision is required

MongoDB by default stores numbers as a 64bit double when passing in the number through the shell no matter if the number is theoretically an integer and has no decimal places. It is important to note that the decimals are approximated and not guaranteed/stored with high precision.

If we know a number is within an integer range, we should consider using an integer because it will simply take up less space than if we just enter it as a normal value and therefore automatically stored as a 64bit double. We should use a long if we are working with full numbers above the integer threshold range.

We can use doubles for basically all values where we do not need high precision i.e. the quick and lazy approach to storing numbers, but it is also a valid approach for storing numbers that have decimal places where we do not need high precision.

Finally, we have high precision doubles if we need high precision for calculations with monetary/ scientific data calculations.

Understanding Programming Language Defaults

It is important to note that the mongoDB Shell is based on JavaScript and runs JavaScript. This is also the reason why we can use JavaScript syntax in the shell. The default data types are the default JavaScript data types. This matters especially for the numbers.

JavaScript does not differentiate between integers and floating point numbers and every number is a 64bit float instead. So 12 and 12.0 are seen as exactly the same number in JavaScript and therefore also in the Shell and stored as such. Behind the scenes the number is stored with some kind of imprecision because it is a 64bit float (i.e. 12.0 could be stored behind the scenes as 12.0000003).

This is inherent to the mongoDB shell because it is based on JavaScript and we would face the exact same behaviour when working with nodeJS mongo driver. However, for other languages and their drivers e.g. Python this would differ. In Python 12 would be stored as an integer and a value of 12.0 would be stored as a float because Python does differentiate the numeric data types. We would always need to know the language we are working in and know the defaults for the language i.e. does it differentiate between integers and doubles? If yes, what is the default integer - is it an int 32? We would then know if we need to convert the number.

Working with Int32

Why would we use Int32? Let's say we have a collection in our database and we insertOne a person record. This person has a name and an age. We would normally store the data as seen below:

```
$ db.persons.insertOne( { name: "Andy", age:  29 } )
```

Now there is nothing wrong with the above and if we retrieve our data using the findOne( ) method we would see that the age seems to be stored as 29. This is because it's a 64bit float/double and therefore the integer path is actually stored with full precision while the decimal part is not. Even though we do not see the decimal part of the number, it is stored behind the scenes and there will be some imprecision at some point. In our app we Ould use this number as an integer so we do not care about some imprecision but it is worth noting that there is some imprecision.

```
$ db.persons.stats( )
```

If we look at the stats for the collection we would notice the size of the document e.g. a size of 49, and this is due to a single entry with a name and an age. If we delete all entries and insertOne again but this time looking at the age, we should notice the difference in size when using a double or int32.

```
$ db.persons.deleteMany( { } )
```

```
$ db.persons.insertOne( { age: 29 } )
$ db.persons.stats( )
```

Notice the size is now 35 for the one entry in our database. If we now contrast this with an int32 for the age we would see a difference in size.

```
$ db.persons.deleteMany( { } )
$ db.persons.insertOne( { age: NumberInt(29) } )
$ db.persons.stats( )
```

NumberInt (a command within the shell) allows us to store a number as an int32 number rather than the default float/double. It is important to note that we can pas in the number as seen above or in quotation marks. We will now notice the size of the single age entry is now 31 which is slightly smaller. This may be one reason why it might be worth considering using int32.

If we are using the drivers, we would need to look t the driver documentation and the language to see how we would implement the conversion to Int32 as the NumberInt is a method only for the shell which is based on JavaScript while the driver will be specific to the programming language used and what is the default number for that language in order to convert into an int32.

Working with Int64

In this example we will look at storing a very huge int64 value for example storing a company's net value. Int32 can only store a number above 2.1 billion, but if we have a company valued higher than this value and therefore need to store it as a int64 value.

Below is an example of storing a company worth 5 billion but trying to store this using an int32. This will store successfully without any errors, however, if we try to find this record we will notice that what is actually stored is a totally different value. The reason for this is because we exceeded the range limit available and monoDB ends up storing it as a different value to that of the original.

```
$ db.companies.insertOne( { valuation: NumberInt("5000000000") } )
$ db.companies.findOne( )
```

If we store a 32bit integer using the default 64bit double we would notice that there will be no error in the number because the 64bit double is larger than the 32 bit int. A 64bit double will not have the same range as a 64bit integer because the 64 bit double doesn't just store integers as it also handles decimal places. So it's not like the 64bits are fully available for integer values.

```
$ db.companies.insertOne( { valuation: 2147483648 } )
```

If we have really large numbers, the best way to store it and guarantee that we can store the biggest possible number that are supported by the int64 range is that we use the NumberLong wrapper (if using driver, we should refer to the driver guide and check the default value for that language).

```
$ db.companies.insertOne( { valuation: NumberLong(2147483648) } )
```

Lets say we want to store the largest possible number in an int64 as seen below. We will notice that we now get an error, even though we are in the range of accepted values. The problem is that this value, which is still a number, is simply too big because it's a double64 which gets wrapped by a NumberLong.

```
$ db.companies.insertOne( { valuation: NumberLong(92233720368547758O7) } )
```

To fix the above we should wrap the number in quotation marks instead and this should now work without any errors.

```
$ db.companies.insertOne( { valuation: NumberLong("92233720368547758O7") } )
```

This is really important to understand that both NumberInt and NumberLong can both be used with a number passed as a value as well as a number in quotation marks passed as a value. We should always use quotation marks i.e. basically pass in a string representation of the number because mongoDB internally will convert the string and store it appropriately as a number. If we pass a number it still faces the JavaScript limitations in the shell whereas a string does not.

Performing Maths with Float int32s & int64s

Previously we saw that we could store numbers as text and this allows us to store really huge numbers without losing the accuracy of the number. However, the problem we face with storing number as a string is that we are not able to perform mathematical calculations using string values. The below demonstrates the problem when calculating using strings:

```
$ db.accounts.insertOne( { amount: “10” } )
$ db.accounts.updateOne( { }, { $inc: { amount: 10 } } )
```

The increment function will not work and this will cause an error in the terminal of “Cannot apply $inc to a value of a non-numeric type”. We cannot use strings to calculate; however, calculations will work with NumberInt and NumberLong though.

```
$ db.accounts.insertOne( { amount: NumberInt(“10”) } )
$ db.accounts.updateOne( { }, { $inc: { amount: 10 } } )
```

MongoDB will convert the string into a int32 number when we insert, however, when we increment in the update command, mongoDB will automatically convert the number into a flat 64bit number. To keep the number as a int32 we would need to write the update command like so:

```
$ db.accounts.updateOne( { }, { $inc: { amount: NumberInt("10") } } )
```

This will succeed as we are no longer working with strings but these special number types provided by mongoDB. We must remember that we must work with either NumberInt or NumberLong within our insert and update commands should we wish for the number to be stored behind the scenes as either an int32 or int64, else mongoDB will store the number as a float 64bit behind the scenes (although in the terminal it is displayed as a integer).

Note: if we have a int64 number and we update the value without specifying NumberLong, mongoDB will convert the number into a float 64bit, however, due to the limit of float 64bit it will round the number. In the below example the float 64bit updated the number as 12345678912345678O:

```
$ db.companies.insertOne( { }, { valuation: NumberLong("12345678912345789") } )
$ db.companies.updateOne( { }, { $inc: { valuation: 1 } } )
```

If we update the number using the NumberLong constructor provided by mongoDB then this will correctly increment the int64 number by 1 i.e. 12345678912345678O.

```
$ db.companies.updateOne( { }, { $inc: { valuation: NumberLong("1") } } )
$ db.companies.findOne( )
```

Whats Wrong with Normal Doubles?

We have looked at both int32 and int64 numbers and it is important to note that we can also sort and query for them as well using the special NumberInt and Number Long constructors mongoDB provides us. We are now going to look at doubles number types.

Lets say we have some database which we use for scientific calculations as a hypothetical example. The default type in the shell is the 64bit floating point number and so if we insertOne document, as seen below, this will be stored as such and not the high precision decimals.

```
$ db.science.insertOne( { a: 0.3, b: 0.1 } )
$ db.science.findOne( )
```

When we find the document, the number will look good as well i.e. they look the same as we stored them. However, behind the scenes they will actually be stored slightly different for example 0.3 will be stored something like 0.3000000001 (there will be more decimal places than what is displayed within the shell and inherently some imprecision).

We will notice the difference between the default 64bit floating point and the high precision decimal when we perform some form of calculation.

```
$ db.science.aggregate( [ { $project: { result: { $subtract: [ "$a", "$b" ] } } } ] )
```

If we perform the above calculation, using the aggregation projection (or any regular calculation method), we will notice that the number for the calculation of a – b (i.e. 0.3 – 0.1) is not equal to 0.2 but rather mongoDB shows the result of 0.1999999999999998 which is not what we expected. This demonstrates the imprecision we are talking about when it comes to the default 64bit floating point number type.

These values are not stored exactly as we insert them. In some use cases, this might not even matter even if we are building an online shop with products where we store the price. This might be because we are only displaying the price on the web page and the approximation is alright because we only display 2 decimal places and would therefore be correctly displayed. Even if we are charging we might be fine because we would send the data returned from the database to some kind of third party service and we rely on the provider charging exactly the amount and not some incrementally lower amount.

If we work with the number and perform some calculation on the server (as aggregate performs on the mongoDB server) then we might have a problem because the result may not be acceptable for the application; this is where the high precision double, the 128bit double, can help us.

Working with the 128bit Decimal

Continuing on from the previous example, we will look at the syntax (i.e. the constructor) in order to use a high precision decimal.

```
$ db.science.insertOne( { a: NumberDecimal("0.3"), b: NumberDecimal("0.1") } )
$ db.science.findOne( )
```

The NumberDecimal is the mongoDB constructor for converting into a 128bit decimal value. Again, if we were to use drivers for Python, Node, C++, Java, etc. we would need to look at the documents for the driver to find the constructor we would have to use to create a 128bit decimal. Again we would use the quotation marks to pass in the number although we do not necessarily have to use them but will face the issue of imprecision if we do not use the quotation marks.

```
$ db.science.aggregate( [ { $project: { result: { $subtract: [ "$a", "$b" ] } } } ] )
```

If we now run the aggregation command we saw before, we should now see the result we were expecting i.e. 0.2 as the result *(i.e. an exact number decimal*). Just like the other number types we can calculate, sort filter, update etc. using this high precision decimal constructor. The same rule we saw for int32 and int64 also applies for the 128bit floating decimal.

It is worth noting that the high precision does come at a price to the size of the document.

```
$ db.nums.insertOne( { a: 0.1 } )
$ db.nums.stats( )
$ db.nums.deleteMany( { } )
$ db.nums.insertOne( { a: NumberDecimal("0.1") } )
$ db.nums.stats( )
```

We will notice the size of the default 64bit floating decimal for this document is 33 whereas the size for the 128bit floating decimal is 41. This is the cost we must pay when using the high precision decimal because more space is reserved for this number type. Therefore using this for all decimal values may not be optimal, but cases where we do need the high precision then this is the solution for performing mathematical calculations without losing the precision.

Useful Links:

Float vs Double vs Decimal - A Discussion on Precision:

https://stackoverflow.com/questions/618535/difference-between-decimal-float-and-double-in-net

Modelling Number/ Monetary Data in MongoDB:

https://docs.mongodb.com/manual/tutorial/model-monetary-data/

Number Ranges:

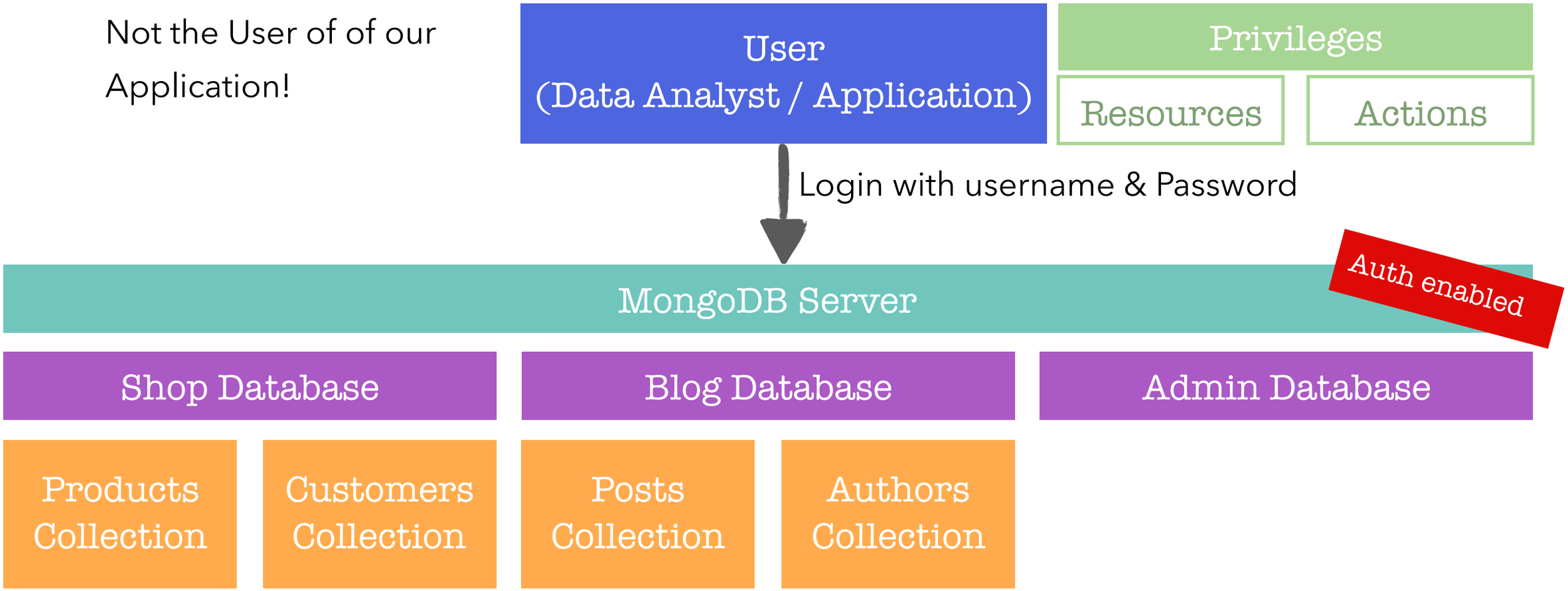
https://social.msdn.microsoft.com/Forums/vstudio/en-US/d2f723c7-f00a-4600-945a-72da23cbc53d/can-anyone-explain-clearly-about-float-vs-decimal-vs-double-?forum=csharpgeneral

MongoDB & Security

Understanding Role Based Access Control

Authentication and Authorisation are actually two concepts which are closely related. Authentication is all about identifying users in our database, while authorisation on the other hand is all about identifying what these users may then actually do in the database. MongoDB uses these two concepts to control who is able to connect to the database server and then what these people who are able to connect can do to the database. It is important to understand when we say people or users, we mean apps/users directly interacting with the mongoDB server and not the application we are building.

It is important to fully understand the authentication system mongoDB uses. MongoDB employs a role based access control system. Below is a example diagram to demonstrate how the authentication system works.

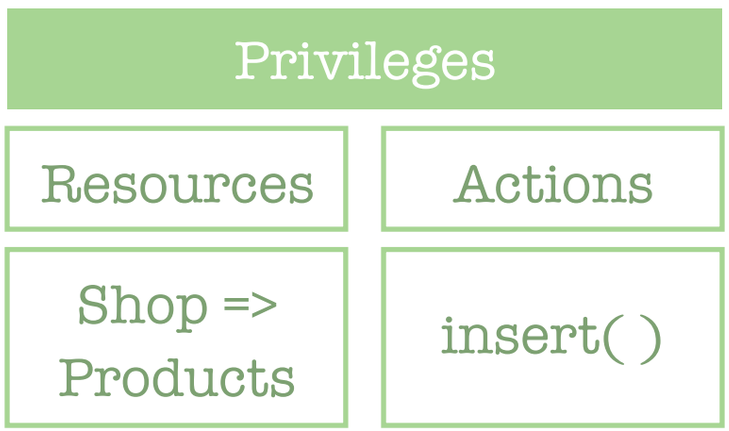


We have a mongoDB server with three databases, the admin collection being one of them (the admin collection being a special database collection that exists out of the box). We enable authentication and suddenly our mongoDB server only allows authenticated users to interact with the collections, documents and the overall server.

Lets say we have a user, it is important to note the user is either a real person like a data analyst who directly connects to the database through the shell or an application which uses the mongoDB driver to connect to our database. They need to login with a username and a password. MongoDB automatically has a process in place where we have to enter a username and password.

The user must exist on the mongoDB server, otherwise logging in will be impossible. If the user exists and is logged with that information only, then we are not able to do anything. Users in mongoDB are not just entities that are made up of a username and password but they are also assigned some roles and these roles are essentially groups of privileges.

A privilege is a combination of a resource and an action. A resource would be something like the products collection in our shop database and an action would be something like an insert command. So actions are essentially the kind of tasks/commands we can execute i.e. we can do in our mongoDB database and the resource is the collection we can execute the command.



So in the privilege example, we have access to insert documents into the Products collection in the Shop database. This will allow us to do something as a logged in user. This is what is known as a privilege and typically we do not just have one privilege but instead multiple privileges are grouped into a so-called roles and therefore a user has a role and that role includes all the privileges.

This is the model mongoDB uses and this is a very flexible model because it allows us to create multiple users where we as the database owner or administrator can give every user exactly the rights the user needs which is very important.

Creating a User

Users are created by a user with sufficient permissions with the createUser( ) command. We then create a user with a username and a password and as mentioned this user will have one or more roles and then each role will typically contain a bunch of privileges. Now a user is created on a database, so a user is attached to a database.

This does not actually limit the access of the user to that database only as we might think. This is the database against which the user will have to authenticate, the exact rights that the user has will depend on the roles we have assigned to that user. So even if the user authenticates against the shop database, the user might still have the role to work with all databases that exists in our mongoDB environment. Whilst this sounds strange, it will become more apparent as to why we can assign users to different databases.

Not only can we create users but we can also update users if we ever need to. This means that the administrator update user for example change the password but we should tell the user too.

To update the user we would use the updateUser( ) command.

So to see this in practice we would use the following commands. First we must bring up the mongoDB server but now adding the --auth flag. This flag simply means that we now need to authenticate to work with that database.

$ sudo mongod --auth

Previously we did not have the flag setting which meant that everyone who connects is allowed to do everything. This now ends with this setting. When we start the server with that setting, we now require users to authenticate. If we open up a second terminal and connect using mongo, we would have expected it to fail but actually we are still able to connect. However we will face authorisation issues whereby nothing will display is we use the show dbs or show collection commands.

$ mongo

There are are two different ways of signing in. We can either sign in with db.auth( ) command which takes in a username and password or during the connection process using the -u and -p flags for user and password. Both demonstrated below:

$ db.auth(“Abel”, “Password”)

$ mongo –u Abel –p Password

The issue we have is that we have no users created; however, mongoDB has a special solution called the localhost exception. When connecting to our database in this state, where we have not added any users, we are allowed to add one user and that one user should of course be a user who is then allowed to create more users. To do this we first of all should switch to the admin database and then use db.createUser command.

```
$ use admin
$ db.createUser( { user: “Abel”, pwd: “Password”, roles: [ “userAdminAnyDatabase" ] } )
```

A user is created by passing in a document into the createUser command and the first argument should be the user and the value to be the username. The second argument should be the password. The third argument should be the roles key and this hold an array of roles the user should have. We can add roles in different ways, but the one role we should add is the userAdminAnyDatabase role. This is a special inbuilt role which will grant the user the right to administrate any database in this mongoDB environment. Once the user has been created, we then need to authenticate using the .auth( ) command:

```
$ db.auth( “Abel”, “Password” )
```

When executing the command we must make sure we are within the admin database to authenticate. Once we have authenticated we should be able to run the show dbs or show collection commands without any error.

Built-in Roles Overview

MongoDB ships with a bunch of built-in roles that basically cover all the typical use cases we have. We can also create our own roles which is beyond the scope of this guide because this will be a pure admin task – there will be a link to the official document where we can learn how to create such roles and it's actually pretty simple.

So what built-in roles does mongoDB provide? We have typical roles as seen in the below table:

Role Type	Roles
Database User	read, readWrite
Database Admin	dbAdmin, userAdmin, dbOwner
All Database Roles	readAnyDatabase, readWriteAnyDatabase, userAdminAnyDatabase, dbAdminAnyDatabase
Cluster Admin	clusterManager, clusterMonitor, hostManager, clusterAdmin

Sometimes we will have cases where we want to give a user rights to do something in all databases and also in all future databases and we do not want to manually add this, therefore we have these all database roles.

Besides these roles we also have cluster administration roles. Clusters are essentially constructs where we have multiple mongoDB servers working together. This is used for scaling, so that we can have multiple machines running mongoDB servers and store our data, which then works together. Managing these clusters of servers in a meaningful task and performed by people who know what they are doing and they have their own rights and we can assign their own rights to let them do their job.

Role Type	Roles
Backup / Restore	Backup, restore
Super User	dbOwner (admin), userAdmin (admin), userAdminAnyDatabase, root

We also have special backup and restore roles in case we have some users who are only responsible for this type of job.

Finally, we have the super user roles. If we assign the dbOwner or a userAdmin role to the admin database, this will be a special case because the admin database is a special database and these users are then able to create new users and also change their own role and this is why its a super user and are very powerful roles. The root role is basically the most powerful role as this user can do everything (i.e. the same rights we had before locking down the database with –auth flag).

Assigning Roles to Users & Databases

An alternative method of signing into the mongoDB database is to use the flags, but it is important to add the --authenticationDatabase flag if we have not switched to the database the user was created.

```
$ mongo -u Abel -p Password --authenticationDatabase admin
```

This will successfully log us into the database without any errors and we do not need to run db.auth again. Below is another example of creating a new user on the shop database and assigning a role.

```
$ use shop
$ db.createUser( { user: “appdev”, pwd: “dev”, roles: [ “readWrite” ] } )
```

We now have a new user and we can log in as the user using the db.auth command passing in the username and password as arguments.

```
$ db.auth( “appdev”, “dev” )
```

The 1 in the shell will signify the login was successful. If we now try to insert a new document into the shop collection we would notice that we get an error of “too many users are authenticated”.

To mitigate this we should have ran the logout command before switching to the new user.

```
$ db.logout( )
```

The quickest way to fix this error is to quit the mongo shell and then restart the mongo shell.

```
$ ^C
$ mongo -u appdev -p dev --authenticationDatabase shop
```

Notice the appdev authenticationDatabase is not admin but rather shop. This is because the user was created in the shop database and must be authenticated against the shop database. If we try against the wrong database the login will fail.

If we now try to insert a new product into the shop database logged in as the appdev user account we would still get an error and this is because we must use the shop database (if we quit mongo shell and logged back in).

The readWrite role was assigned to this user, but since the user was created on the shop database, the readWrite role by default also only gave us readWrite access to that shop database. This is the one thing that assigning a user to a database does for us out of the box. Therefore we can really create the users who are assigned to the database they should work with and they only have the necessary roles, so that the user would have to login against the database they should work with.

Updating & Extending Roles to other Databases

If we want to change our users roles so that they can access another database we can use the updateUser command. We must perform this command with a user account that has access to create/update users, otherwise we would receive an error. Furthermore, we must execute the command connected to the shop database in order to find the user account that requires updating:

```
$ db.updateUser( “appdev”, { roles: [ “readWrite”, { role: “readWrite”, db: “blog” } ] } )
```

The first argument to the updateUser command is the username of the user that we want to update. The second argument is a document describing of the changes/update we want to make to that user. We have a couple of key value pairs we can use which can be found in the official documentations (e.g. pwd is a key we can use to change the password of a user).

It is important to note that these roles we specified above are not added to the existing roles but rather will replace the existing roles with the new roles we specify as part of the updateUser command. The first argument provides the readWrite role to the database the user was registered against. The second argument allows us to pass in a document where we give the user access to a new database and their role. We can check the user account for the update:

```
$ db.getUser( “appdev” )
```

Important Note: When logging into an account using the auth command we must either be in the database the user is registered to or use the --authenticationDatabase flag to authenticate against the database the user was created. When logging out using the logout command we must be within the database the user account is registered to. Therefore using the use command to switch to the correct database before running commands in crucial, else this could be the cause for errors in the shell when trying to run certain commands such as too many users authenticated. Ultimately we can quit the shell and re-log into the shell which will log us out from all accounts completely.

We can create a user and add multiple roles to different database, without giving any roles to the database the account was registered to as seen below:

```
$ db.createUser( “appdev”, { roles: [ { role: “readWrite”, db: “customers” }, { role: “readWrite”, db: “sales” } ] } )
```

MongoDB & Security

Adding SSL Transport Encryption

Now that we know how to lock down our database using users, we are now going to look at making sure data that is transferred from our app to the database is also secure. This could be a node, PHP, Python, C++ or whatever language the application uses the mongoDB driver to then communicate to the mongoDB server and store data. It is important that the data is encrypted whilst it is in

transport, so that someone who is spoofing our connection cannot read our data and mongoDB has everything we need for this built in. We are going to now explore how we can easily secure our data whilst it is on its way from client to server.

MongoDB uses SSL (actually TLS which is the successor to SSL). SSL in the end is just a way of encrypting our data whilst it is on its way. We will use a public private key pair to decrypt the information on the server and to prove that we as the client are who we make the server think we are. So essentially it is a secure way of encrypting our data and decrypting it on the server and whilst it is on its way, it is consistently encrypted.

We will need a command to execute which will create us the files we need to enable SSL encryption. On mac/linux we can simply run the command in the terminal but on windows this will not work. For Windows OS we need to install openSSL in order to run the command.

```
$ openssl req -newkey rsa:2048 -new -x509 - days 365 -nodes -out mongodb-cert.crt -keyout
mongodb-cert.key
```

We will be asked a couple of questions – note we can skip some of the first few question by pressing dot and then enter:

```
$ Country Name: UK
$ State or Province Name (full Name) [Some-State]: .
```

```
$ Locality Name (eg, city) []: London
$ Organization Name (eg, company) [Internet Widgits Pty Ltd]: .
$ Organization Unit Name (eg, section) []: .
$ Common Name (e.g. server FQDN or your name): localhost
$ Email Address []: email@email.com
```

The important part to the setup is the common name here. We must fill in localhost during development whilst running this on local host. If we were to deploy our mongo database onto a server in the web, we need to fill in the address of that web server and this will be important, otherwise the connection will fail because this will later be validated that if we are connecting to a server, the server we are connecting to is the server that is mentioned in our certificate.

Once we add the email address we are now done and should have two files, the mongodb-cert.key and the mongodb-cert-crt files. We now need to concatenate both these files into one file by using a command on mac/linux:

```
$ cat mongodb-cert.key mongodb-cert.crt > mongodb.pem
```

On Windows we would run the command instead:

```
$ type mongodb-cert.key mongodb-cert.crt > mongodb.pem
```

We will now have a mongodb.pem file and this is the file we will need to enable SSL encryption.

How does this now work? In the folder where we stored the .pem file, we can now copy and move this file around in our file system, but in the folder where the file lives we can start our mongod server with SSL enabled.

Running the --help command in the mongod shell will show us all the SSL commands available to us. One of the option we need to set is the mode. The --sslMode flag defines whether SSL is disabled, allowSSL (allows to connect with/without SSL), preferSSL (only important if we are using replica sets) or requireSSL (must have SSL connection else denied).

We would need to point to our .pem file using the --sslPEMKeyFile flag. We can also have a --sslCAFile flag argument passed for a certificate authority file which we can get our SSL certificate through an official authority online paide/unpaid. This will be passed in addition to our .pem file which the CA file will be an extra layer of security that basically ensures that man in the middle attacks can be prevented.

If we deploy our mongoDB database in production, we would get our SSL certificate from a certified authority and they would give us a .pem file and a .ca file so that we can basically add both arguments and point at the respective files when launching our server.

Mac/Linux:

```
$ sudo mongod --sslMode requireSSL --sslPEMKeyFile mongodb.pem
```

Windows:

```
$ mongod --sslMode requireSSL --sslPEMKeyFile mongodb.pem
```

This will start the server and we will see a warning that we have no SSL certificate validation in place because we are missing that sslCAFile, but besides this we now have our server which is now waiting for connection on port 27017 ssl.

To connect to the server, we should navigate in a new terminal window, to the same folder where we have the .pem file and launch our mongo client. Running the mongo command should now fail, if it succeeds then we have connected to another mongoDB instance which we can shut down using the db.shutdownServer( ) command. This should fail because we have no mongod running which would allow connections from non-SSL clients.

We need to set two things, first we need to enable SSL using --ssl flag and we will need to pass our .pem file as a –sslCAFile. We may also need to add the host (we specified during certification creation) for this command to work:

```
$ mongo --ssl --sslCAFile: mongodb.pem --host: localhost
```

If we do not specify the host this will try to connect to 127.0.0.1 which is the localhost but technically is a different word and therefore not considered equal to localhost and therefore by specifying the --host: localhost we have made it really clear that this is the host we are expected to see named on the backend. This is also the host on the certificate and therefore this works.

Now obviously we can have more elaborate setup, but this will do for now and demonstrates how we can generally connect with SSL turned on and now all data we send from the client i.e. from the mongo shell to the server (mongod) will be encrypted.

MongoDB & Security

Encryption at REST

Encryption at REST simply means that the data we stored on our mongoDB server in a file, this might also be encrypted. We can encrypt two different things.

We can encrypt the overall storage so the files themselves – this is a feature built into mongoDB enterprise. We as a developer should always encrypt at least certain values in our code such as user passwords are hashed and not stored as plain text. We could go as far as hashing/encrypting all data.

We can encrypt and hash both our data as well as the overall file to have the most security possible.

The hashing of password will be something we will see in the From Shell to Driver Chapter.

Useful Links:

Official MongoDB Users& Auth Doc:

https://docs.mongodb.com/manual/core/authentication/

Official Security Checklist:

https://docs.mongodb.com/manual/administration/security-checklist/

What is SSL/TLS?:

https://www.acunetix.com/blog/articles/tls-security-what-is-tls-ssl-part-1/

Official MongoDB SSL Setup Doc:

https://docs.mongodb.com/manual/tutorial/configure-ssl/

Official “Encryption at Rest” Doc:

https://docs.mongodb.com/manual/core/security-encryption-at-rest/

What Influences Performance?

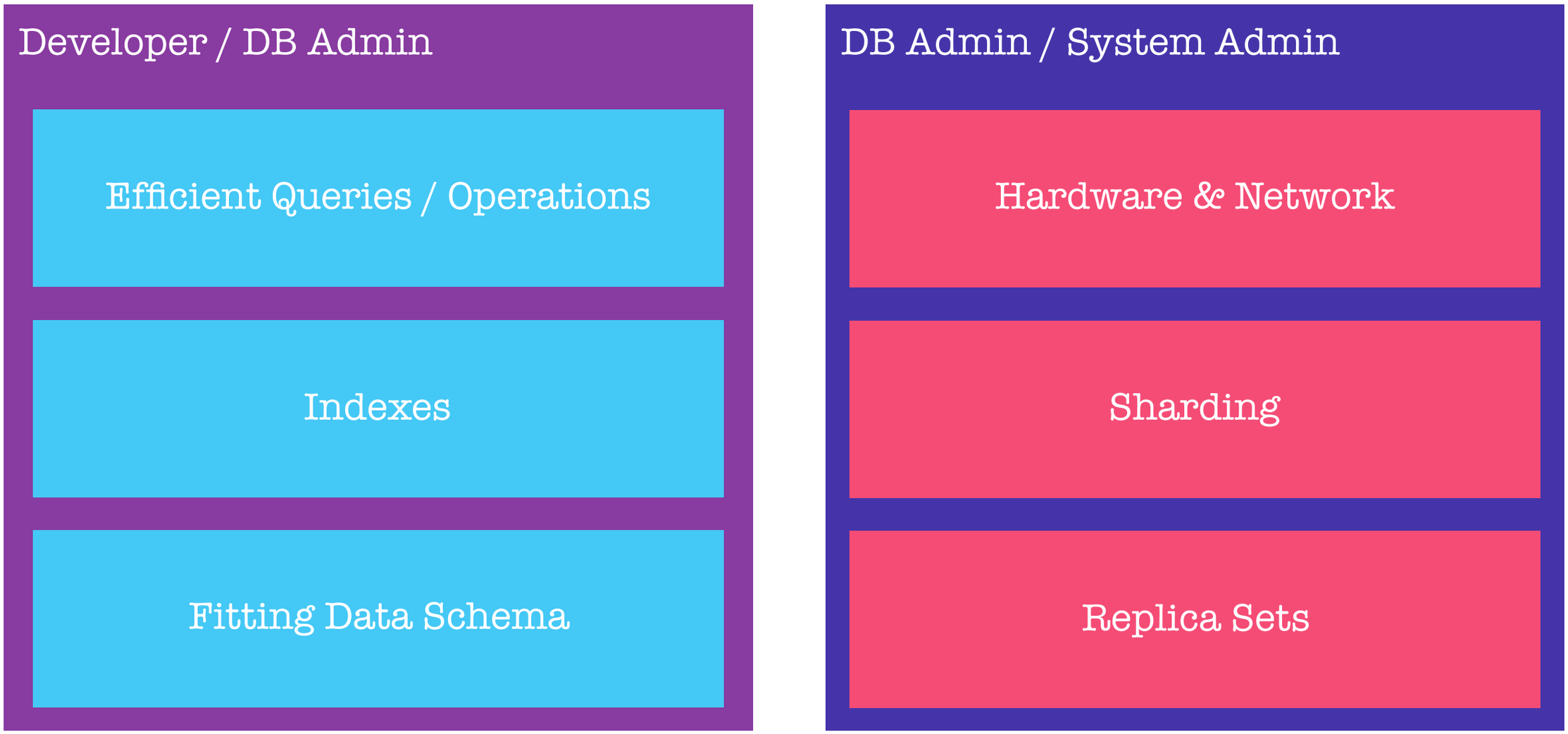
What influences performance?

On one hand, there are things which we directly or indirectly control as a developer.

1. We should write efficient queries and operations in general i.e. inserting, finding, etc. and all this should be done in a careful way that we only retrieve data we need, we insert data in the right format with the right write concerns and so on.
2. We should use indexes, either we have access to the database and we can create them on our own or we need to communicate with our database admin, so that we can ensure that for the queries our application does, we got the right indexes to support these queries to run efficiently.
3. A fitting data schema is also important. If we always need to transform our data, either on the client side or when fetching it through the aggregation framework (if we need to do a lot of transformation for common queries), then our data format as it is stored in our collection might not be optimal. We should try to reach a data schema in our database that fits the application or our use case needs.

These are block of factors which we can influence as a developer and as a database administrator.

On the other hand the hardware and network on which we deploy our mongoDB server and database matters. Sharding is another important concept and so are replica sets. These factors are not really tasks a developer needs to be involved in too much as these are typical DB/System Admin tasks. We will not dive too deeply into these because these are very complex matters – but will be something we have to understand to understand the big picture of mongoDB and what it is all about.



Understanding Capped Collections

Capped Collections are a special type of collection which we have to create explicitly where we limit the amount of data or documents that can be stored in the collection. The old documents will simply be deleted when the size is exceeded. It is basically a store where the oldest data is automatically deleted when new data comes in.

This can be efficient for high throughput application logs where we only need the most recent logs or as a caching service where we cache some data and if the data then was deleted because it has not been used in a while, then we are fine with this and we can just re-add it.

To create such a collection, we use the createCollection command:

```
$ db.createCollection( { “collectionName”, { capped: true, size: 10000, max: 10 } } )
```

Firstly, we define the name of the collection and then pass in the document specifying the options for the collection. We want to set capped to true to enable the capped collection. By default the size limit will be 4 bytes but we can increase/set the size to any value which will then automatically be turned into a multiple of 256 bytes. The max option allows us to define the maximum number of documents that can be stored in the collection i.e. capping by number of documents.

It is important to note that for capping collections, the order in which we retrieve the document is always the order in which they were inserted. For a normal collection that may be the case but it is not guaranteed. If we want to change the order and sort in reverse there is a special key we can use called $natural which allows us to sort by the natural order. Positive 1 will sort in the normal order while negative 1 will sort in the reverse order. We can also create indexes in a capped collection and we have an index on the _id by default.

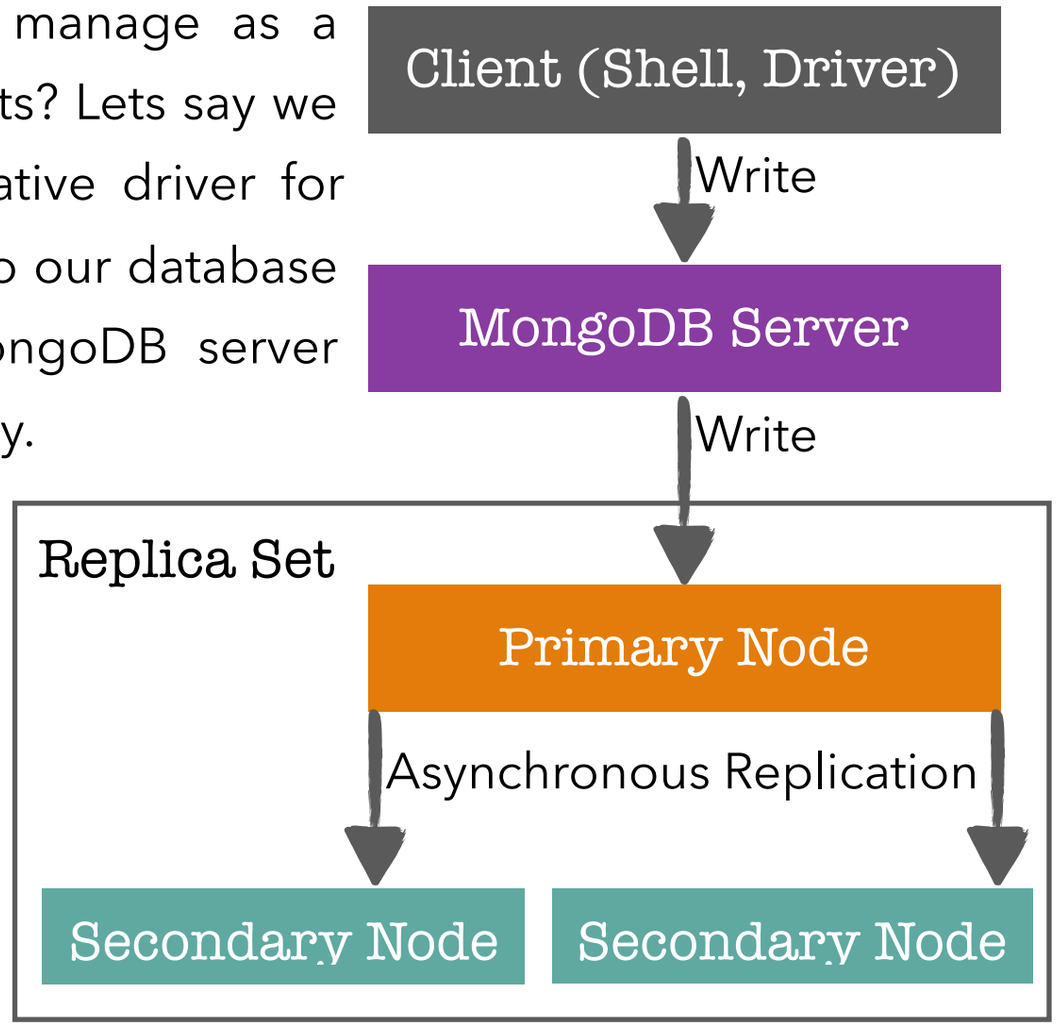
```
$ db.collectionName.find( ).sort( $natural: -1 ).pretty( )
```

The idea of a capped collection is not to give an error when we insert too many documents but rather it clears the oldest document i.e. if we already have 100 documents, the next insert should delete the very first inserted document in the collection but the total documents would remain 100 documents using the above example.

When would we use a capped collection instead of a normal collection? Well because of the automatic clear up. We can keep this collection fairly small and therefore the collection will be more efficient to work with and we do not have to worry about manually deleting old data. So for use cases where we need to get rid of old data anyways with the new data coming in or where we need a high throughput and it is ok to lose old data at some point, like for caching, then the capped collection is something we should keep in mind as a tool to improve performance.

What are Replica Sets?

Replica sets are something we would create and manage as a database or system administrator. What are replica sets? Lets say we have our client, either the mongo shell or some native driver for Node, PHP, Python, etc. We want to write some data to our database and we send our insert/write operation to the mongoDB server which in the end talks to the primary node we could say.

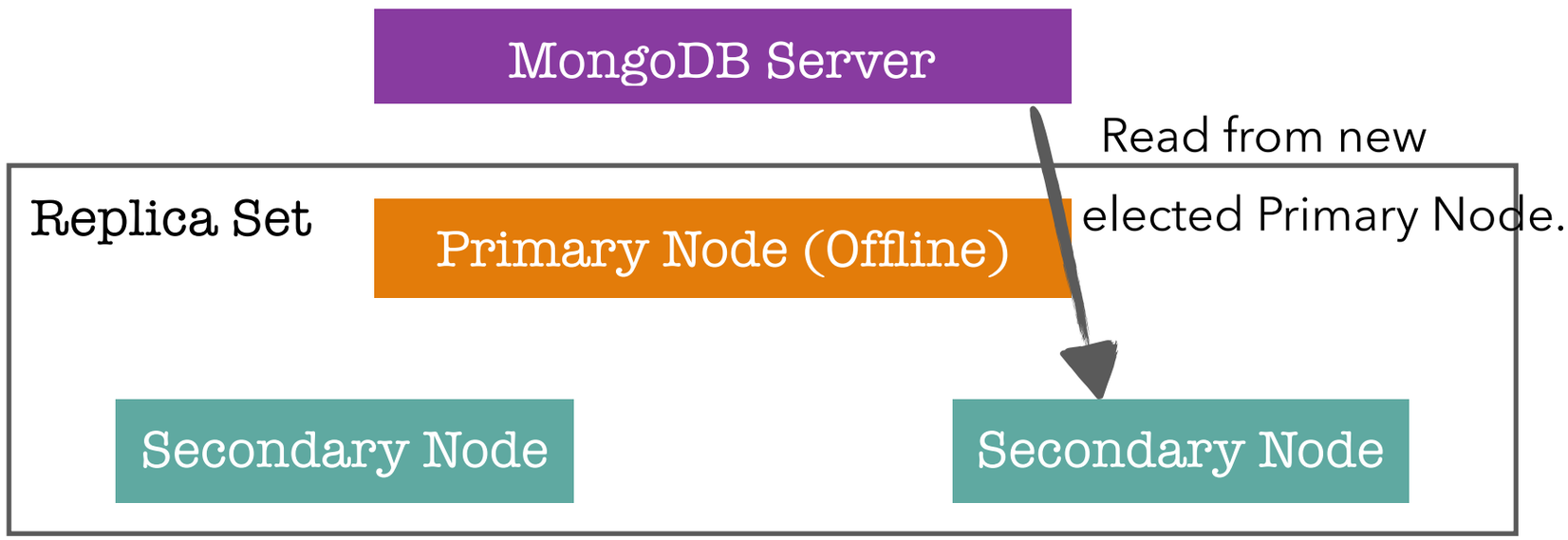


Note that a Node here is simply a mongoDB server. So when we used thus far with the mongod command gave us a node and this was the only node we had. So the mongoDB server is technically attached to that node but it is easier to understand it like this.

We can add more nodes which are called secondary nodes, which are essentially additional database servers that we start up but are all tied together into what is known as a replica set. The idea here is that we always communicate with the primary node automatically and we do not need to do this manually. If we send a insert command to our connected mongo server, it will

automatically talk to the primary node, but behind the scenes the primary node will asynchronously replicate the data on the secondary nodes. Asynchronously means if we insert data, it is not immediately written to the secondary nodes but relatively soon.

So we have this replication of data. Why do we replicate data? If we have the setup seen above and we read data and for some reason, our primary node should be offline, we can reach out to a secondary nodes in a replica set which will then be the elected new primary node. The secondary nodes in a replica set hold a so-called election when the primary node goes down to elect and select a new primary node and then we talk to that new primary node until our entire replica set is restored. We therefore get some fault tolerance here because even if one of our servers was to go down we can talk to another node instance in that server network (in the cluster) to still read data. As a new primary we can then also not just read but also write data.

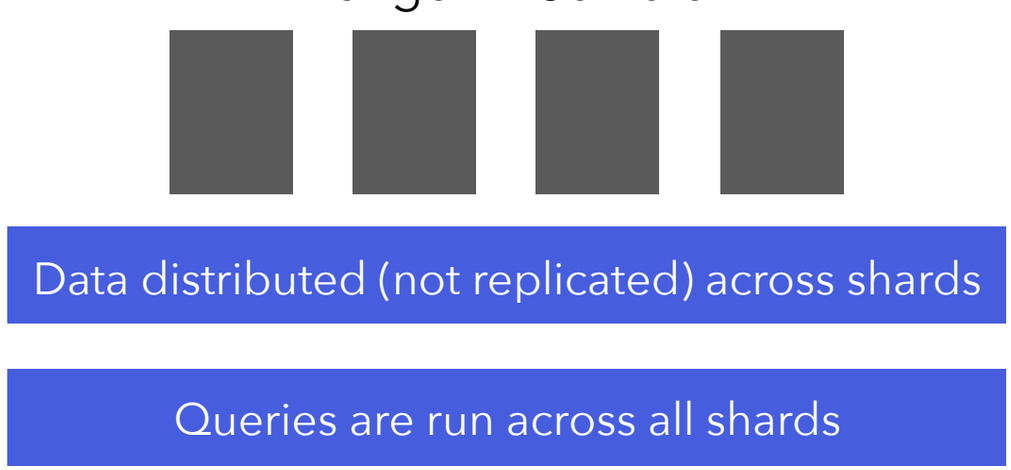


This is the reason we use replica sets. We have the backup and fault tolerance and we also get better read performance as well. We can configure everything such that our backend will automatically distribute incoming read requests across all nodes. This is only for read request only as the write request will always go to the primary node. The configuration of the reads is a task for the system/database administrator and we want to ensure we can read our data as fast as possible. Therefore, with replica sets we get the backup, fault tolerance and improved read performance.

Understanding Sharding

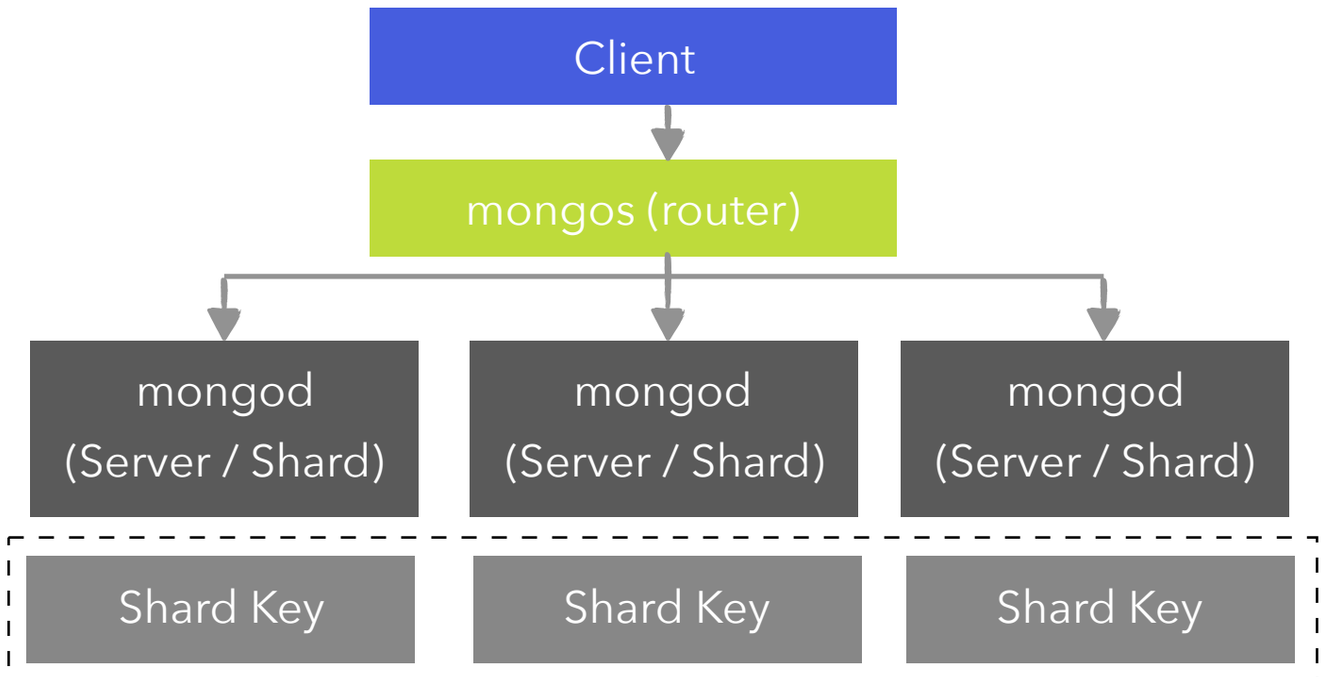
Sharding and Replica sets are sometimes confused with each other but they are actually different things. Sharding is all about horizontal scaling. When we talk about scaling, if we have a mongoDB server (and we really mean a computer which runs our mongoDB server and where our database files are stored on) and we need more power because we are now getting more requests coming in because we have more users and more read and write operations. What can we do? We can upgrade the server i.e. buy more CPU, memory etc. and put that into the server (using a cloud provider we can simply upgrade on a click of a button). This is a solution but only gets us so far because at some point we cannot squeeze any more CPU or memory into a single machine. At that point we would need horizontal scaling, which means we need more servers.

The issue here of course is that the servers now do not duplicate the data and they are not backups but instead split the data. So server A stores data for the same application as the other servers but a different chunk of the data. With sharding, we have multiple computers who all run mongoDB servers but these servers do not work standalone but work together and split up the available data, so the data is distributed across our shards and not replicated.



Therefore, queries for read, find, insert, update and delete operation have to be run against all the servers or the correct server because each chunk manages its data. Its range of data if we were to split and store alphabetical data i.e. the first server would store A to F, the next server stores F to L and so on, the operations would have to be forwarded to the correct servers. How does this work?

If we have our different mongod instances (our different servers/shards) and each shard can and will typically be a replica set (i.e. we have multiple replica sets because each shard is also a replica set) – so if we have these servers and we have our client, we then have a new middleman which we have not used in this guide called mongos. Mongos is the executable and is our router mongod offers.



The mongos router is responsible for forwarding our operations such as inserts, reads, update and so on to the correct shards. So the router has to find out which shard is responsible for the data we are inserting i.e. where should this data be stored and which shard will hold the data we want to retrieve.

For this splitting we would use a so-called shared key. Shard key is essentially just a field thats added to every document which is important for the server to understand where this document belongs to. This shard key configuration is actually something which is not trivial because we want to ensure we have a shard key that is evenly distributed. We can assign our own values but we should ensure that the data is roughly evenly distributed so that the data is not stored all on one server.

Example of Sharding – We issue a find query on our client to look for a user named Andrea, this reaches mongos and now there are two options:

The first option is that our find query did not specify a value for the shard key e.g. we are looking for Andrea but our shard key is some other value and not the name. In our find filter there is no

information regarding the shard key and therefore mongos does not know which shard is responsible for handling this request. In such a case mongos has to broadcast our request to all shards and then each shard has to check if it is responsible i.e. has the data and then each shard returns its response which is either the data or not the data. Mongos will then have to merge all that data together and return it.

The second option is that our find query does contain a shard key which is the username and we are searching for the username in our find filter. Mongos can directly forward this to the correct shard and fetch the data from where it is stored and therefore of course is more efficient.

Choosing the shard key wisely is important and usually a job of the admin. This is an important part of mongos, the router, finding out where an operation should go i.e. which server is responsible for the operation.

As a developer if we know that we are using sharding, we should sit together with the administrator and choose a wise shard key based on our application needs that is evenly distributed so that we can write queries that uses the shard key as often as possible.

Sharding is all about distributing data across servers and setting up everything so that data can be queried and used efficiently. This is an advanced topic just as with replica sets and is something a developer will not have to worry about but is something we should be aware of.

Deploying a MongoDB Server

We now want to get our localhost and mongod instance onto the web i.e. we want to get it on a server that we can reach from anywhere and not only from inside of our local computer. Deploying a mongoDB server is a complex task and is definitely more complex than just deploying a website because we need to do a lot of configuration. We need to manage shards if we have sharding, we have to manage replica sets, setup authentication to the database, protect the web server and the network (totally unrelated to mongoDB), ensure software stays up to date (e.g. general software, mongoDB related software and security patches etc), setup regular backups (including backup to a disk) and encryption both during transportation and at rest.

There are a lot of complex tasks that we have to manage when deploying and is beyond the scope of most developers as this is very much a system administrator task. This is something outside the scope of this guide but we can look at a managed solution provided by mongoDB called mongoDB Atlas, which gives us scalable and best practice mongoDB server running in the cloud which we can configure through a convenient interface. We can scale up and scale down the servers and there is also a free tier which provides us all the things we have mentioned above automatically.

Using MongoDB Atlas

If we visit mongoDB.com there is a chance that we see the mongoDB Atlas on the home page where we can click on the get started free button. We are then required to sign up to the service and there is no credit card required so that we can start absolutely free without any danger of getting charged.

Once signed in we will either be greeted to the dashboard page with the clusters option selected or have an option to create a project first before shown the dashboard.

A cluster simply describes the mongoDB environment. A cluster contains all our replica sets and shards. It is basically what we would deploy i.e. ur deployed mongod server. On the cluster option page we can click on build a new cluster and this will take us to a page where we can now configure our mongoDB environment.

Generally, we have a global cluster configuration which is not available for free but we can choose different parts of the world where we want to deploy different clusters or shards that then talk to each other with a couple of clicks. The idea is so that our data is distributed across the world so that users have the shortest way to the data possible. This is something we can enable or disable.

The next step is to choose the underlying cloud provider, we do not need to sign up to these cloud provider but mongoDB the company does not host its own data centres, instead the mongoDB solution we configure here will be deployed in one of the free data centre providers. We can choose from Amazon AWS, Google Cloud Platform or Microsoft Azure providers and then the region where we want to deploy in the case we are not using the global cluster configuration. We need to make sure we select a region where the free tier is available.

The next option is to select the Cluster Tier which defines the power we have and what we can do. In the free tier we have to select M0 cluster instance else we can choose a different tier such as M10. We can then customise the storage i.e. how much can be stored overall in gigabytes (GB). We can also enable the auto-expand storage option to expand storage before we exceed it.

In the Additional Settings we can choose the storage engine version we want to use (this is only available in the paid tier).

We can then configure backups of which we have two types of backup. Continuous means that all data is backed up all the time i.e. a continuous backup history, while the alternative is a snapshot approach where we have a 24 hour period backup. The snapshot approach poses danger of losing data for the last 23hours or so if the last backup is that long ago. Again this is only available if we are on a paid tier.

The next option is sharding (we require M30 tier or above for this option) where we can choose the number of shards we want which adds more power besides the tier we chose. We can also add options such as Encryption at Rest and enabling Business Intelligence Connects but again only available for the paid tiers only.

There are then more configuration options which we can either ignore or setup some more configurations. Finally, we can then assign a cluster name and then click on the create cluster button. MongoDB will deploy this solution onto the web i.e. onto a couple of servers and will take a couple of minutes and thereafter we will have a fully functional mongoDB environment running in the cloud, automatically secured and configured according to the best practices. It will also automatically be a replica set i.e. we get a free nodes replica set. We can add sharding and also reconfigure the cluster after its running. We can see more on this within the mongoDB Atlas documentations.

We can then go to the security tab and here setup secure access to our mongoDB databases. We should definitely add a user here and we can add as many users as we want. Another important setting is the IP Whitelist, we would need to add the IP address of the server that is running our application or during development. We can automatically fetch our current local IP address using the button. We can also turn on some Enterprise Security Features but is again beyond the scope of this guide.

Backups & Setting Alerts in MongoDB Atlas

Atlas is a powerful tool for getting our mongoDB environment up and running. If we have backups turned on, we can restore them here once they are available. We can also configure alerts which allows us to see what happened and also allow us to create new alerts with the add new alert button option where we can get an email alert when something happens e.g. when a user logs in or when the average execution time for reads is above a certain millisecond value. We can setup a bunch of alerts for all kinds of things to always keep track of what is happening in our cluster which is of course really useful.

In general we should look at the Atlas documentation to learn everything about Atlas when planning to use it in production. It is a strong recommendation to use Atlas as a managed solution for getting our mongoDB environment up and running unless we are a system admin and absolutely know what we are doing when we want to configure everything on our own which we can do but is not covered in this guide which is for developers.

Connecting to Our Clusters

Now with the cluster up and running, we can now work with the cluster. On the cluster we have a couple of options such as migrate data into the cluster, pause the cluster, terminate the cluster, check our general metrics about the cluster and view all other informations regarding our cluster. Note some of the options may be outside the free tier. The interesting part is how can we connect to the cluster?

On the overview page we can click on the connect button and a modal should appear. We will see all the IP addresses that will be able to connect and we can also add a whitelist entry from here. We can choose the way of connecting to the cluster. We can connect through the shell but of course are also able to connect from an application. If connecting via shell, we can follow the instruction for our OS to download the necessary software and run the shell command. This will require our password to connect. We no longer need to run the mongod on our local machine as we now have it on the web. Below is an example of the shell command:

```
$ mongo “mongodb+srv://cluster0-ntrwp.mongodb.net/test” –username John
```

The path is important as it tells mongo not to connect locally but to connect to the server at the address specified in the command. The /test tells which default database to connect to.

We can change this to another database if we want. We then need to add our username using the flag and we can then hit enter. This will prompt for a password which we should enter. This password will be the password we used when creating the user we are trying to login as. This should then connect to our mongoDB cluster server running in the cloud. We can now use the terminal to run our mongoDB commands on our database as normal. We do not need a separate terminal when running the commands because the mongod server is already running in the cloud. We have connected to the server from our shell to run our normal mongoDB commands on our databases. We can now use the mongoDB server from anywhere and not just from our local computer.

Useful Links:

https://docs.mongodb.com/manual/replication/

https://docs.mongodb.com/manual/sharding/

https://docs.atlas.mongodb.com/getting-started/

What are Transactions?

MongoDB v4.0 or higher is required for transactions. Atlas does not provide MongoDB v4.0 for the free tier.

What are Transactions? If we take a use case of a Users collection and a Post collection. Lets suggest most users have a couple of posts. The posts are related to the users because the user is the person who created the post. So we have a stored relation either via a reference or a key stored in the user or in the posts document, it doesn't really matter. We now delete the user account. Therefore, we want to delete the documents in both the users and posts together i.e. delete documents in two collections.

Now this can be done without transactions, we can simply delete a user and right before we do that, we can save the id of the user and then reach out to the posts collection and find all posts linked to that user id and delete those posts. This is perfectly possible without transactions.

However, what if we have a use case where the deletion of the user succeeds but during the post deletion something goes wrong i.e. temporary server outage or network issue, etc. We now end up in a state where the user was deleted but the posts are still there but the user they are pointing at

doesn't exist anymore. This is the exact use case where transactions help us with.

With a transaction we can basically tell mongoDB that these operations (as many as we want) either succeed together or they fail together and we roll back i.e. restore the database in the state it was before the transaction regarding the documents that were affected in the transaction. This is the idea behind transactions. In order to try this out we need MongoDB v4.0 and a replica set.

Transactions

A Typical Use Case

So if we had a Atlas Server setup that has access to MongoDB v4.0 or above we can connect to it and setup a database collection to trail this out using the following commands below:

```
$ use blog
$ db.users.insertOne( { name: "Beth" } )
$ db.posts.insertMany( [ { title: "First Post", userID: ObjectId("5ba0adfacfd31f") }, { title: "Second Post", userId: ObjectId("5ba0adfacfd31f") }  ] )
```

We now have a user called Beth created and two posts that has a reference to Beth's userID created. Now if we want to delete the user, we would obviously try to find the userID which is something we

would know in the application as the user would be in their account and click some button to delete their account. So in the end we would do something like the below:

```
$ db.users.deleteOne( { _id: ObjectId("5ba0adfacfd31f") } )
$ db.products.deleteMany( { userId: ObjectId("5ba0adfacfd31f") } )
```

This is how we could clear the data and this would work 99.9% of cases. The problem with the above is where it doesn't work because something goes wrong and we end up in a state where it suddenly deleted the user but not the posts or vice versa. Now we can use Transactions for this use case and will explore how transactions work in the next section.

How Does a Transaction Work?

Now for a transaction, we need a so-called session. A session basically means that all our requests are grouped together logically. We create a session and store it in a constant as seen below:

```
$ const session = db.getMongo( ).startSession( )
```

db.getMongo gives us access to the mongo server while .startSession creates a session. We now have a session stored in a const variable. This session is basically an object that now allows us to

group all requests that we send based on that session together. We can now use that session to start a transaction:

```
$ session.startTransaction( )
```

The session is important because technically every command we issue is sent to the server and then normally the server would forget us and therefore we need a session so that when we send something based on that session, the server remembers that session because behind the scenes the sessions are managed through a session key and so on. The server will know that the command we just sent it should be considered in the context of the other commands it was sent based on that session.

Below is more commands using the session:

```
$ const usersCol = session.getDatabase( “blog” ).users
$ const postsCol = session.getDatabase( “blog” ).posts
```

The above creates a new const variable which we want to get the database blog and the users collection. We repeat the same now for the post collection. We have now started the transaction and got access to our collections. We can now write all the commands we want to run against these collections that we got.

$ db.users.find( ).pretty( )

If we run the above command to get the user id, note that this will not be included in the transaction because the command was not included in the session which we started the transaction. We just needed to get the id right now. We can now go back to the session transaction using the userId we retrieved.

$ usersCol.deleteOne(_id: ObjectId("5ba0adfacfd31f"))

The usersCol variable is a pointer to our user collection but also mapped to the session. We are now using the collection in the context of our session through the variable and we can now use our normal operations such as insert, update, delete etc. If we hit enter to run the command, this session transaction command will be acknowledged and deleted, but if we repeat the db.users.find( ) command, we will see the user still exists. So the user has not been deleted yet, it was just saved as a to-do. We can then run a command on the postsCol variable.

$ postsCol.deleteMany(userId: ObjectId("5ba0adfacfd31f"))

This will also be acknowledged and in the terminal said to be deleted too, however, it has not yet write this to the database as we can prove by looking into the posts using the db.posts.find( ).pretty command to view all posts in the collection.

In order to actually commit the session transaction changes to the database we have to run the following command:

```
$ session.commitTransaction( )
```

The commitTransaction( ) command will execute the session commands. To abort a session command there is a command we can use should we not wish to continue with the session:

```
$ session.abortTransaction( )
```

This allows us to cleanly close the session command. Therefore, the commands will not be committed to the database.

If we execute the steps: create a session const, getting access to the collection for the session, starting the tranactions and then specifying the two commands that belong to the transaction and then committing it, this should work with no errors – i.e. we should not put any db commands in-between our session commands. If we then look at the database collections we should no longer see the user or the posts created by that use i.e. they have been deleted by the delete commands within the transaction.

Therefore this is how transactions works with MongoDB v4.0 and above. To summarise what we have learnt about transaction and how the work: We get access to a session, based on that session, we use the session to store a reference to our collection in some variable(s)/constant(s) and we do

this from our native drivers e.g. Node, Python, PHP etc. We then start a transaction on the session and then we would execute our manipulating queries. Finding does not make much sense in transactions as transactions are all about safely changing data like in our use case of deleting the user and their posts in two different collections. Finally we commit to the transaction. We can abort the transaction if we no longer wish to continue with the transaction.

It is also important to understand that for this transaction if something was to go wrong, if it somehow would fail, but the users was deleted but the post was not or vice versa – it will roll back these operations in the commit transaction and thereafter our database will be in the same state as before. Therefore the actions in the transaction either succeed together or they fail together. This is the idea behind transactions. This provides atomicity across multiple operations, so across multiple steps or even something like deleteMany [*atomicity is on a document level and is either written entirely or nor written at all*]. We can wrap our InsertMany or deleteMany commands in a session transaction to guarantee that either all documents are inserted/deleted or none at all. Therefore we can get atomicity on a operational level and not just on a document level. We should definitely not overuse it though as this takes a bit more performance than a normal delete or insert, so we should only use it if we need that cross operation consistency.

Useful Links:

https://docs.mongodb.com/manual/core/transactions/

Splitting Work Between the Driver & the Shell

We have to understand how to work between management of the database probably through the shell and interacting with the database through the driver. We have to understand how to split the work between the driver and the shell as seen below:

Shell	Driver
Configure Database	CRUD Operations.
Create Collections	Aggregation Pipelines
Create Indexes	

The task in the shells are typically things we do up front i.e. setting up the backend for our application. The driver is tightly coupled with our application i.e. app logic. If we build an application for a shop as an example, our code will basically be responsible for handling products, users, order etc. The initial setup is not something the application deals with as it assumes that the database is there to then communicate with. The driver typically handles CRUD operations to handle data within the database i.e. Create, Read, Update and Delete. This is how we would roughly split the responsibilities but technically we can create collections and indexes from inside the driver which is still a possibility.

Preparing Application Project

We will create an application to demonstrate the Shell to Driver module and how we would use the driver code to perform CRUD operations within our application which will communicate with the database. For this project we will use MongoDB Atlas as our database host. All following sections will demonstrate the driver commands to communicate with the database via the drivers and how easy it is to transfer from the shell to driver.

To prepare for the project we will setup a new free cluster on MongoDB Atlas using one of the cloud hosting platforms and the free hosting region. We can call this cluster something like sandbox or FromShellToDriverTutorial. Once the cluster has been created, we would need to go into security and make sure we have at least one user with readWriteAnyDatabase access in our database which is really important. The setup of the server will be done upfront and we do not need atlasAdmin role. If we do need to do something we will do it by connecting to the server in the shell – so anything related to setting up connections, collection validation or indexes, we would do that through the shell. For the cluster we should also make sure that our local IP is whitelisted. Once setup we should wait for MongoDB Atlas to finish setting up and in the mean time we could do something else such as setting up NodeJS and our application project files as we will be using NodeJS as our server language to help build our react node.js application.

We should head over to the NodeJS webpage to download and install NodeJS on our local machine. This will also install Node Package Module (NPM) which will allow us to install all the dependencies for our project file reading from the package.json file.

In the terminal, we should navigate to our project directory that contains the package.json file and run the following terminal command to install all the dependencies.

```
$ npm install
```

This will take a while as it installs all the dependency files. Once all files have been installed we can run the following command within the terminal to start the project in our browser on our localhost server.

```
$ npm start
```

This will run the react script and open the single page application within the browser. We should noticed an error modal message appear on the webpage and this is because it fails to fetch data from the backend of the application as this has not been setup. We need to start the Node REST API by simply opening another terminal navigated to the same project directory but run the following command:

```
$ npm run start:server
```

We need to run both processes at the same time and keep them both running. With both running we can reload the app and notice the products have been loaded from a local stored dummy data and not from a database. We are now ready to play with the application and learn driver code and see the slight difference between the shell and driver and how to connect to our database.

Installing the Node.JS Driver

Within the MongoDB official documentation we can go into the MongoDB drivers section and we can find instructions on how to install and use the driver for the language our application uses.

https://docs.mongodb.com/ecosystem/drivers/

For our demo application we would use the node.js drivers. This will forward us to the link/page containing the instructions on how to install and use the node.js driver. We should find similar documentation for all drivers so that we can learn how to use the driver with our application language. We can also look at the Driver API which shows a list of all methods/objects that are part of the driver.

To install the node.js driver in our project we must run the terminal command within our project directory:

```
$ npm install mongodb --save
```

We now have the mongodb driver installed in our project file. Our mongoDB driver code (credentials) will sit in the backend server and will never be exposed to the front end client.

Connecting Node.js and the MongoDB Cluster

On the MongoDB Atlas clusters page click on the connect button. This is where we can whitelist our localHost IP address as well as find the various methods of connecting to our cluster. We can now select the Connect Your Application as the method. We can then select the driver and version we are using and then copy the connection string.

In the project application directory we will need to add some code to our app.js file.

/backend/app.js file:

We need to import our mongoDB client. We do this by creating a const variable and importing the mongodb package to use the MongoClient in order to establish a connection. We can then use the const variable and call the connect method on it, passing in the url connection string we got from the MongoDB Atas page. Replace the <user> and <password> with the actual users created on the database.

```
const mongodb = require('mongodb').MongoClient;
mongodb.connect('mongodb+srv://<user>:<password>@fromshelltodrivertutorial-gtajb.mongodb.net/test?retryWrites=true')
    .then( client => {
```

```
        console.log( 'Connected!' ) ;
        client.close( );
    } )
    .catch( err => {
        console.log(err);
    } );
```

This will establish a connection to our database. We can pass a function within our promises which will be executed when the connection completed and this will output successful connections or errors for unsuccessful connections. In our .then clause we can pass in the client as an argument which will allow us access to the client which will then allow us to execute a database method which will then allow us for example to work with collections and so on. In the example above we close the client immediately but will later look at how to interact with the database using the client.

After saving the changes to our server side code, we have to go to the terminal where we ran the npm run start:server command and quit that process using ctrl + c on our keyboard and simply re-run the command again for the changes to the server to take effect. After restarting we should see Connected! In the terminal if the connection was successful to our mongoDB database server.

Now that we are connected to our database server, in the following sections we will look at driver code on how to perform database actions to interact with our database server.

Storing Products in the Database

Previously we saw how to connect to the mongoDB Atlas Database Server using a dummy connection. We now want to use that connection and do something useful with it.

We can cut the previous code from the app.js file and head over to the products.js file. This file contains a function that allows us to add a new product but we want to store the new product data to our database. We want to also store the price as a 128bit decimal value and therefore we can also practice transformation of our data.

backend/routes/products.js File:

```
const mongodb = require('mongodb');
Const MongoClient = mongodb.MongoClient;

Router.post( '', (req, res, next) => {
    const newProduct = {
        name: req.body.name,
        description: req.body.description,
```

```
        price: parseFloat(req.body.price), //Store as 128bit decimal in MongoDB
        image: req.body.image
};
~~console.log(newProduct);~~
MongoClient.connect('mongodb+srv://<user>:<password>@fromshelltodrivertutorial-
gtajb.mongodb.net/test?retryWrites=true')
        .then( client => {
                client.db( ).collection( 'products' ).insertOne(newProduct);
                client.close( );
        } )
                .catch( err => {
                        console.log(err);
                } );
        res.status(201).json( { message: 'Product Added', productId: 'DUMMY' } );
} );
```

We can paste in the mongoDB connection function code from our app.js file below the newProduct function. We always have to connect from scratch every time we want to do something with the database.

In the .then clause of our promise we can use client.db( ) to get access to our database and then call collection as a method which allows us to access a collection. The products collection does not currently exist. If we wanted to create the collection ahead of time because we wanted to add schema validation or anything of the sort, we would create it within the shell.

```
$ mongo "mongodb+srv://fromshelltodrivertutorial-gtajb.mongodb.net/test" --username <username>
```

We would replace <username> with the username we wish to connect to and the shell will then ask you to enter the password when we run the command to login to that user.

Inside the Shell if we want to interact with our test database and create the collection we would use the commands seen below:

```
$ use shop
$ db.createCollection( )
```

Now we do not need any special settings on our collections and therefore we can use the on the fly approach and simply use the CRUD operation to insert something into the collection and the collection and the database will then be created.

Notice how the insert command for the Node.JS driver is exactly the same as it is in the shell i.e. we have the insertOne and insertMany commands as we have seen before. This is true for all the other

languages whereby the operations we can do are the same but the language may have a slight difference in the syntax. In our above example we used the insertOne method and passed in our newProduct variable.

From Shell to Driver

Storing the Price as a 128bit Decimal

It is always a great idea to dive into the official documentation for the driver we are using to understand the driver syntax. The Driver API page will have a list of all available driver methods and we could look for the method we want – in this example we are looking for something related to the 128bit decimal (there is a Decimal128 link which will provide sample snippet of the object method).

We can use the new Decimal128( ) constructor or better the Decimal128.fromString( ) operators.

We can import this into our code using our mongodb variable:

```
const mongodb = require('mongodb');
const MongoClient = mongodb.MongoClient;
const Decimal128 = mongodb.Decimal128;
```

This is the reason we have mongodb imported separately as a const variable as we can use it to import various objects from the mongodb package. We can now use this decimal const variable

as a reference to convert our price into a 128bit decimal. We would replace the native JavaScript parseFloat( ) object method with the mongoDB driver Decimal128.fromString( ) method:

```
Router.post( '', (req, res, next) => {
    const newProduct = {
        name: req.body.name,
        description: req.body.description,
        price: ~~parseFloat~~ Decimal128.fromString(req.body.price.toString()), //Store as 128bit
        decimal in MongoDB
        image: req.body.image
    };
```

When we create a new product this should convert the price and store it as a 128bit decimal value. For the Decimal128.fromString( ) to work we need to ensure the value we pass through is a string. We can use the JavaScript toString( ) method to convert a number into a string value.

Finally we can use the .then and .catch on our insertOne( ) because it also returns a promise so that we can either react to any errors or console.log our results. We want to make sure to close the client within the promises as we do not want to close the client before the operation has completed.

```
MongoClient.connect('mongodb+srv://<user>:<password>@fromshelltodrivertutorial-
gtajb.mongodb.net/test?retryWrites=true')
    .then( client => {
        client.db( ).collection( 'products' ).insertOne(newProduct)
            .then( result => {
                console.log(result);
                client.close( );
                res.status(201).json( { message: 'Product Added', productId: result.insertedId } );
            } )
            .catch( err => {
                console.log(err);
                client.close( );
                res.status(500).json( { message: 'An error occurred.' } );
            } )
    } )
    .catch( err => {
        console.log(err);
    } );
```

We have now restructured the code so that we are now connected to our database inside the post

route and we are sending data to the database to the products collection and we should have a working code. We can restart the server running the npm run start:server again as we changed some code and now in our application (npm start to run application dev server) we should be able to create a new product and have it saved into our database.

In the terminal running our backend server we will notice something that does not look like an error. We receive a commandResult which has a lot of information about the operation that was executed. The host sends the data to us and at the bottom we would see the insertedCount to indicate one document was inserted/added to our database along with the ID. This is not an objectId( ) because in JavaScript the objectId( ) would not exist but the id is basically the string which is wrapped by the mongoDB objectId.

In the other terminal which is the shell connected to our mongoDB Atlas database server we can run the command:

```
$ show collection
$ db.products.find( ).pretty( )
```

We will notice that our products collection has been created and if we look into the collection we should also have the new product document added to the collection. We now have a working insert through the client application.

Fetching Data From the Database

Below is an example JavaScript code for fetching data from a mongoDB database. Note we have copied the same code from the insert but changed it to a fetch command using the driver syntax.

backend/routes/products.js File:

```
router.get( '/', (res, req, next) => {
    MongoClient.connect('mongodb+srv://<user>:<password>@fromshelltodrivertutorial-
    gtajb.mongodb.net/test?retryWrites=true')
    .then( client => {
        const products = []
        client.db( ).collection( 'products' ).find( ).forEach( productDoc => {
            productDoc.price = productDoc.price.toString( );
            products.push(productDoc);
        } );
            .then( result => {
                // console.log(result);
                client.close( );
```

```
                res.status(200).json(products);
            } )
            .catch( err => {
                console.log(err);
                client.close( );
                res.status(500).json( { message: 'An error occurred.' } );
            } )
} )
.catch( err => {
    console.log(err);
        res.status(500).json( { message: 'An error occurred.' } );
    } );
} );
```

We use the same connection code as we did for the insert command however we would change insertOne( ) to the find( ) command. Note that the find command will return us a cursor and we must traverse the cursor through our data. We can learn more about the cursor for our driver in the official driver documentation (in the Driver API list we should see the cursor object and we can see all the methods we could execute on the cursor and how they work and how we can interact with the cursor).

The forEach method simply goes through all the documents inside our cursor (i.e. it fetches all the data in the forEach) and we can have access to each product document one at a time and do something with each document i.e. push the documents into our products array. This method also returns a promise as this is a asynchronous task as it will fetch each document from the database server.

We also need to convert the 128bit decimal into a number that JavaScript can handle else we would receive an error. The mongodb module provides us with a toString( ) method which converts the 128bit decimal back into a string.

If we restart the server after making changes to the server code and rerun our application we should notice the terminal showing that some data is being fetched and we should also see in our application that the single document that we added to our database is now appearing within the list of products.

Note that images should always be stored in a file storage and never on the database. If we had an application that allows upload of images, we would save this uploaded image in a file storage and then store the file path to the image in the database and not the image itself. This is because images will bloat our database, is insufficient (a lot of data transfer) and is not what a database is built for.

Creating a More Realistic Setup

Now that we understand the basic interactions setup with the driver, in a more realistic setup the code will be more optimised. In programming we use the concept of DRY which stands for Don't Repeat Yourself. Below is the optimised setup to demonstrate this concept.

The db.js File (new file):

```
const mongodb = require('mongodb');
const MongoClient =mongodb.MongoClient;
const mongodbUrl = 'mongodb+srv://<user>:<password>@fromshelltodrivertutorial-gtajb.mongodb.net/test?retryWrites=true'

let _db;
const initDb = initDb = callback => {
    if (_db) {
        return callback(null, _db);
    }
    MongoClient.connect(mongodbUrl)
```

```
        .then(cleint => {
            _db = client;
            callback(null, _db);
        })
        .catch(err => {callback(err) });
}
```

We create a const called initDb which will act as a utility script to setup connection to the database and then get access to the established connection. The initDb simply points to a function which receives a callback so that another function can execute as an argument.

We use an if statement to check if the _db variable is uninitialised and if it is not uninitialised i.e. if the statement returns true (which only happens if it is initialised) then we will return the callback function. The callback function takes in a null as the first argument because the callback will receive an error as the first argument (we use null as we do not want to throw any errors) and the second argument returns the existing database.

We create a const mongodbUrl variable that stores our mongoDB connection URL we got from mongoDB Atlas as a string. Inside of our initDb function we will call the MongoClient.connect method passing in the mongodbUrl value.

Note we can add a second argument where we can configure the connection. We should look at the driver documents to see what we can configure but the default settings should be fine.

```
MongoClient.connect(mongodbUrl, {<optionalConfigurationsArgument>});
```

The catch promise will return an error if there is any and we can use the callback function to log the error by passing the error as the first argument. The then promise will return the client if successful. The promise logic is pretty much the same. We can therefore store the database in the _db variable.

We now have a method that we can execute to establish a connection with our database. We would create a second const variable in the same file called getDb to get the access to this established database connection. This variable will be an arrow function.

```
const getDb( ) {
    if (!_db) {
        throw Error('Database not initialised');
    }
    return _db;
}
```

We will check if the database is not initialised and if true then throw the error else we would return the initialised database value.

Finally we will export these functions so that we can utilise these two functions in other files within ur application.

```
module.exports = { initDb, getDb };
```

The app.js File:

```
const db = require('./db')
```

We would import the db.js file into our app.js file as a utility file. Before we call the app.listen(3100) we would call on our functions that establishes a connection to the database.

```
db.init( (err, db) => {
    if (err) {
        console.log(err);
    } else {
        app.listen(3000);
    }
} );
```

We can utilise the initDb const function which will return either an error or a success. Using the if statement we can console.log the error, else we can call app.listen to start the Node.js server when a connection has been established. This setup ensure the application is connected to the database

at startup of the application before it starts listening to the incoming requests and then it will simply keep on running.

The products.js File:

```
const mongodb = require('mongodb');
~~const MongoClient =mongodb.MongoClient;~~
const db = require('../db');
```

We no longer need the MongoClient but we do need mongodb const variable for the decimal import. We can add our own import of db.js file in order to now be able to use the getDb function. Therefore in the place where we work with the database we can now remove the code and use the function.

```
router.get( '/', (res, req, next) => {
    ~~MongoClient.connect('mongodb+srv://<user>:<password>@fromshelltodrivertutorial-~~
    ~~gtajb.mongodb.net/test?retryWrites=true')~~
        ~~.then( client => {~~
            ~~const products = []~~
            ~~client.db( )~~
    const products = [];
```

```
db.getDb()
    .db()
    .collection( 'products' ).find( ).forEach( productDoc => {
        productDoc.price = productDoc.price.toString( );
        products.push(productDoc);
    } );
        .then(result => { res.status(200).json(products) });
        .catch(err => {
            console.log(err);
            res.status(500).json( { message: 'An error occurred.'} )
        })
```

This change will apply to both the get and post request functions as we can now connect to the database using the getDb function from our db.js file we imported. Note the rest of the code can remain as it is.

We should remove all client.close( ) methods from the calls and also the outer .catch( ) promises (not the inner promises) from our get and post functions in the products.js file. Thereafter when reloading the server and application, the application should continue to run and we have the same

functionality as before.

We are now using the same connection all the time and is something which we should do because we will use a concept called connection pooling which is provided by the mongoDB driver by default. This means it actually establishes a couple of connections or at least it is ready to quickly establish them and therefore we can handle multiple incoming requests to our node restAPI simultaneously. We can only send one request per connection to mongoDB normally but since we have a connection pool here of multiple available connections, even if we have multiple incoming connections to node.js, we can forward them to mongodb thanks to the connection pooling.

This is therefore an advantage of this approach where we reuse one client because we have a shared connection pooling.

When we call db.getDB( ), we have to call .db( ) ourselves which will allow us to reuse our connections. Restarting the backend server and refreshing the application we should notice the application is running exactly the same as before but the code is much more optimised compared to before.

Editing and Deleting Products

Below are some snippets for the Editing and Deleting documents using the mongoDB driver.

The products.js File:

Editing Product Code

```
router.patch('/:id', (req, res, next) => {
    const updateProduct = { name: req.body.name, description: req.body.description, price:
    parsefloat Decimal128.fromString(req.body.price.toString( )), image: req.body.image };
    db.getDB( ).db( )
        .collection('products').updateOne( { _id = new ObjectId(req.params.id) },
        { $set: updatedProduct } )
            .then(result => { res.status(200).json( { message: 'Product Updated', productId:
            req.params.id } ); } )
            .catch(err => { console.log(err); res.status(500).json( { message: 'An error occurred'} )
            } );
});
```

Deleting Product Code

```
router.delete('/:id', (req, res, next) => {
    db.getDB( ).db( )
        .collection('products').deleteOne( { _id = new ObjectId(req.param.id) } )
            .then(result => { res.status(200).json( { message: 'Product deleted' } )
            .catch(err => { console.log(err); res.status(500).json( { message: 'An error occurred'} )
            } );
});
```

We have now looked at all the basic CRUD operations using the node.js mongoDB driver and should have a simple CRUD application which interacts with our database to fetch/read single/all data from a collection, update a document within a collection and delete a document from a collection. This should hopefully provide us with some insight as to how we can switch from the shell to drivers.

Implementing Pagination

To implement pagination to our application using the mongoDB driver, we can use the cursor object. There are three methods that can help us with pagination as we will see below:

The products.js File:

```
router.get('/', (req, res, next) => {
    const queryPage = req.query.page;
    const pageSize = 1
    const products []
    db.getDB( ).collection('products').find( )
    .sort( { price: -1 } )
    .skip( ( queryPage -1 ) * pageSize )
    .limit(pageSize)
    …
} );
```

First we can sort our results; either by id, price or product name. In the above we sorted by price. We can then use skip and limit methods.

The skip method allows us to skip previous products. So if we are one page 2 and want to skip the first item then we take the query page which will be 2 and less 1 which will give us one and then multiply by the pageSize i.e. 1 document a page. This forumla will equate to 1 which will tell the skip method to skip the first product.

The limit method allows us, as the name suggests, limit the number of documents fetched from the database so that we only display as many as we have indicated in the pageSize variable. So if we want to display 1 product, we will limit the cursor to returning 1 document.

These three methods (or two methods) allows us to implement pagination by manipulating the cursor and what it returns from our fetch/get command. We can add buttons to allow the user to switch between pages to display different number of products per page.

Useful Links:

Learn how to build a full RESTful API with node.js

https://academind.com/learn/node-js/building-a-restful-api-with/

What is Stitch?

MongoDB stitch is a server-less platform for building applications. It is a collection of services offered by the mongoDB company that we can use when building a web app, mobile app or desktop app so that we can focus on our user interface and core business logic (even the logic which should run on a server on the backend) so that we do not have to write the entire boilerplate for setting up the server, managing the server or create a RESTful API on our own as this is handled by Stitch.

It is built all around Atlas as a cloud database or at least integration with it easily. One core feature of Stitch is authentication i.e. allowing the app authentication to the mongoDB database. Stitch therefore allows us access to our MongoDB Atlas database which will be available from inside our client side application.

Previously we mentioned that we should not access the database from inside the client side because we do not want to expose our database credentials. The idea behind Stitch's solution is that it does not expose our database credentials, instead our application user can sign up and log in through Stitch's service and the user will get temporary credentials for fine-grained access to the database and we can lock down what a single user can do.

We can therefore pass the authentication or the rights to the users of our app because they do not have full access to the database and they would have with our mongoDB credentials which are not exposed.

So Stitch manages this behind the scenes by taking our rules into account and make sure that users do have access to the database but only to exactly the things they have to do such as create a new product (previously we controlled this through node but we can now have stitch to do this).

Stitch allows us to react to events for example if something gets inserted into a database or something gets updated, we can run some code that does something e.g. sends an email or log something into another collection or whatever we need to do.

We can execute code or some kind of functionality in the cloud. Stitch is not just a tool which helps us build beautiful UI's or client side code, we can still execute code in the cloud with so-called function. These are essentially code-snippets which we can define where we do not need to write the boilerplate of parsing incoming requests or exposing routes in the RESTful API, but where we just write a code we want to execute and then we can either call that code directly from inside the client or from inside another function or we setup a trigger to execute a certain code snippet when some event triggers e.g. when something gets inserted. This is similar to Firebase or AWS Lamda cloud functions. Stitch is kind of mongoDB's answer to firebase.

Stitch QueryAnywhere is their solution to allow us to run queries based on the rules we setup for the users of our app directly to our database. There is also mongoDB mobile which we will not cover, but is basically a local mongoDB database which can install on mobile devices so that we can store data and sync it to the cloud even when we are offline. Stitch Triggers is the name for react to events which allows us to configure triggers which will then call Stitch Functions. Additionally there is a feature called Stitch Services which allows us to integrate Stitch with other services for example AWS S3. This will be useful for uploading files because Stitch has no built-in file storage and mongoDB itself is a bad solution as a database should not be used as a file storage and there is not other service that will allow us to store files. This can be solved by using Stitch Services to reach out to AWS S3 which is a file storage and then communicate with that and store files.

So this is the big picture – we have our cloud database which can work with Stitch QueryAnywhere, we got events where we can listen to authentication or database events and then we can execute code in the cloud not only via events but also call the function manually.

The idea is that we build our client side application and we also have some server side logic, we have our mongoDB (Atlas) database, but now our backend (e.g. node.js REST API) will be replaced with the Stitch service. Even though we called it server less, there are still servers somewhere but Stitch manages a lot of the boilerplate/heavy lifting for us and we just communicate to our services, atlas, other services (e.g. database functions) and so on.

Server-less Platform for Building Applications

Cloud Database (Atlas)

Authentication

Our App's Users!

Access to (Atlas) Database

React to Events

Execute Code/
Functionality in the Cloud

Stitch QueryAnywhere

Stitch Triggers

Stitch Functions

MongoDB Mobile

Stitch Services

Client (App)

via SDK

Stitch Servers

MongoDB/Atlas

Database

Functions

App User Auth

Mobile App, Web
App (SPA)

Stitch Preparations

To setup Stitch we need to go into MongoDB Atlas console. Clicking on the Stitch Apps option in the left hand side sidebar will take us to the Stitch Applications section. Simply click on Create New Application and give our application a name, we can link it to a cluster that we have, give a Stitch Service Name, select a deployment model between local and global and finally select a Primary Region. This will take a 1 to 3 minutes to complete the setup and once setup it will redirect us to the Welcome to Stitch! section interface.

On the Stitch Welcome page we can manage the different features that Stitch offers. The getting started page allows us to move through a couple of steps to quickly get started. The clients page allows us to choose which kind of client we are using. If building a web app we should choose JavaScript or if we are building a mobile app we should select native Java (android) or Switft (ios). We would need to install an sdk locally to use the many Stitch features from inside our client and all the steps are outlined in the three tabs depending on the type of application we are building.

Overview of the Stitch Dashboard options:

We can export our configuration using the Export section. Under the configuration section for the Atlas cluster, this is something we can ignore, but allows us to connect our Stitch app differently.

The Rules section will become important for accessing our data and the Triggers section allows us to setup some listeners to database or authentication events and we can execute functions. The Services section are these services where we could integrate AWS S3 for example to add functionality which Stitch does not have built-in, but Stitch gives us a bridge to other services of other companies. The users section allows us to see who signed up to our app and that authentication user management is all handled behind the scenes by Stitch. We do not need to worry about saving the passwords in a secure way and we don't even sore them in our mongoDB collection anymore as this will be handled by Stitch. The Values section are some static values with we could enter which we could use in our rules and functions by accessing it with the special placeholder. The Functions section is the area where we can create a new function to define some JavaScript code that executes on demand on the server side. The Logs section allows us to look at logs for what happened in our Stitch app. Push notification section is something we will not cover but we could push notifications to mobile apps. This is the rough overview of the Stitch dashboard options we have available to us.

On the client section, click on the JS (Browser) tab as we will be creating a web app, and this page will provide the different ways in which we can install the Switch Browser SDK client. We will chose the npm route. In the terminal paste the code on the webpage (below code is as at May 2019):

```
$ npm install --save mongodb-stitch-browser-sdk@"^4.3.1"
```

We should now navigate to our application project folder in the terminal and run npm install (to install the node packages if the dependencies are not already installed) and then run the command npm start to start our application on the localhost.

We no longer require a backend node RESTful API server code as we are now going to use Stitch for our sever-less backend and therefore we also do not need to run the npm run: start server command (i.e. we will not be using our own server anymore).

We are now able to start working with Stitch.

Adding Stitch to our App and Initialising It

On the Stitch Clients page there is a useful example snippet code that we can use in our client side code. Within our project directory we have the src/App.js file which we will have to update.

The src/App.js File:

```
import { Stitch } from mongodb-stitch-browser-sdk;
class App extends Component {
```

```
state = {
        isAuth: ~~false~~ true,
        authMode: 'login',
        error: null
};
constructor( ) { super( ); Stitch.initializeDefaultAppClient( 'stitchtutorial-tjpqh' ); }
```

First we need to import Stitch and other utilities from the mongodb-stitch-browser-sdk module package into the file. We no longer require axios. We would also need to use the JavaScript constructor function which run when the application first starts to initialise the stitch default app client. We would need to pass in a client app id which we get from our Stitch Page. This is the first thing we would need to do on any application that uses Stitch i.e. initialise Stitch using this command. We would also need to ensure isAuth is set to true so that we start in authenticated mode.

With Stitch initialised we can now start using it. We will notice that the application cannot get any data from our mongoDB Atlas database and this is because previously we were using our own backend server code to do the fetch request which no longer works. We would have to update the code and can now start using Stitch to fetch our products from our database.

The src/pages/Product/Products.js File:

```
import { Stitch, RemoteMongoClient } from mongodb-stitch-browser-sdk;

class ProductsPage extends Component {
    …
    fetchData = ( ) => {
        const mongodb = Stitch.defaultAppClient.getServiceClient( RemoteMongoClient.factory,
        ‘mongodb-atlas’ );
        mongodb.db(‘test’).collection(‘products’).find( ).asArray( )
            .then( products => {
                this.setState( { isLoading: false, products: products } );
            } )
            .catch( err => {
                this.setState( { isLoading: false } );
                this.props.onError( ‘Fetching the products failed. Please try again later.’ )
                console.log(err);
            } );
    }
}
```

Again we import Stitch and its utilities into our Products.js file. We would update the fetchData( ) function which uses axios to now use Stitch. We use defualtAppClient (or getAppClient if we do not initialise the defaultAppClient in App.js file) method to get the initialised client and then we get a service client. The Service Client is something like AWS S3, if we want to reach out to some functionality exposed by that, but another service client is also mongoDB i.e. the database. In order to use the mongoDB database with the Service Client we also need to import RemoteMongoClient from the Stitch Browser SDK. We pass in RemoteMongoClient.factory into our getServiceClient which is a reference to a method which will be used internally by Stitch to initialise that mongo client. We pass a second argument which is the service name i.e. mongodb-atlas. We store the result in a const variable called mongodb which we can now reference in our code.

We can now use that mongodb const variable to now work with our database. We can access the database by calling the db function of which we pass in the name of the database and then we can access a collection in the database which is our products collection. Finally we can call the normal CRUD methods we already know from mongoDB such as find, updateOne, deleteMany etc.

We will use the find( ) method and again we now have a cursor again but we get a more simpler to use option here – we use asArray and let stitch do the fast step by step fetching for us. If we needed more control we could use the .iterator which basically allows us to write our own loop. We used the as array to get access to results. This function returns back a promise and therefore we have

the .then and .catch available to us which we can use the catch promise to log the error and the then promise to set the products to the products array within the application state.

When we launch our application we will receive an error and this is expected. We must authenticate first i.e. we must have authenticated users in our application because each user receives a set of things this user may do and then only that interaction may happen with the database. This is our way of protecting our database against unwanted access such as deleting data from the database for example deleting order. These are things we can control on a user's ability with interacting with our database.

Adding Authentication

Within the Stitch Dashboard and under the Users section, we can navigate to the Providers tab and view the various ways in which we could authenticate users for example email/password, third party providers such as facebook and google and even allow anonymous authentication. We will turn on this anonymous user authentication by switching it on for this demonstration. I a production environment, we would probably not use this authentication method and would probably set using email and passwords or third part authentication.

To log in anonymously, we would need to go back into the App.js file and make some changes.

The src/App.js File:

```
import { Stitch, AnonoymousCredential } from mongodb-stitch-browser-sdk;
    constructor( ) {
        super( );
        const client = Stitch.initializeDefaultAppClient( 'stitchtutorial-tjpqh' );
        client.auth.loginWithCredential( new AnonymousCredential( ) );
    }
```

Where we initialised our appClient, we can also login from here. We store the Stitch client in a client const variable and for anonymous authentication we must import from the Stitch Browser SDK the AnonymousCredential. We can now access the .auth features of the sdk and here we can call the loginWithCredential method. We can pass in different credentials and this will depend on the authentication methods we want to use and we unlocked in the Stitch Users > Providers dashboard console. So for now we unlocked the anonymous authentication and we would pass in new AnonymousCredential as an argument. We do not need to pass anything as an argument for the AnonymousCredential. This will now allow us to login anonymously. If we now save and reload our application we should no longer receive the error message we saw previously. We now get a new error of Unexpected token in JSON… we now have to set rules for accessing our database.

Sending Data Access Rules

By default everything is locked down (even for an authenticated users) and we have to inform Stitch which user or which authenticated entity, because we currently have anonymous authentication turned on, what kind of access is allowed.

Within the Stitch Dashboard we need to navigate to the Rules section in order to setup the rules for our data. In here we would need to add a collection. This does not mean adding a collection in the database but adding it to the rules here so that we can set rules for a collection. If nothing is added here then everything is locked down by default and is a great security setting.

So we add the database name and the collection name using the dropdown we want to setup the rule. We have a couple of template rules we could use. We can go with no template and this will setup a rule which we can start to configure.

We can setup validation in the Schema Tab which Stitch will give an error if something does not suffice the criteria set in here. In the permissions tab we can set the default permission which applies to every authenticated entity (including our anonymously signed in user/visitor of our page). Currently everything is locked down but we can enable read access and/or write access. We could add a field and prevent read access but all other fields are readable. This is something we could do

to control which fields are/are not retrievable and which are/are not writeable. Therefore, we have fine-grained control up to the field level. We can save our changes and now anonymous users should be able to read all the products data from our database collection without any errors anymore.

We will now have a new error. The problem we have is that previously we parsed our mongoDB data on the server for example we converted the decimal128 to a number or to a string that we could work with in our JavaScript code. We are now not doing this and are now receiving these mongoDB types which we cannot work with. We now need to convert these on the client because we are doing the entire data fetching on the client. The solution is that we transform our products on the client.

Fetching & Converting Data

To transform our data on the client in order to make sure that we parse the native mongoDB types which are not known by native React/JavaScript into the normal JavaScript primitive values. Here we can create a new const variable and use the native JavaScript .map method on our products which allows us to take an array and map every element in that array into a new element and return a new

array of all these new elements stored in our variable.

```
fetchData = ( ) => {
    const mongodb = Stitch.defaultAppClient.getServiceClient( RemoteMongoClient.factory,
    'mongodb-atlas' );
    mongodb.db('test').collection('products').find( ).asArray( )
        .then( products => {
            const transformedProducts = products.map( product => {
                product._id = product._id.toString( );
                product.price = product.price.toString( );
                return product;
            } );
            this.setState( { isLoading: false, products: products } );
    } )
```

The map method takes in a function which will execute on every element in the array, the input here will be the product (and we could name this whatever we want) and then we want to return the changed version of the product. So here we changed the _id and decimal128 into native JavaScript type values. We use JavaScript's toString method to convert the data into a string. If we save and reload our application the data should be fetched without any errors.

Stitch CRUD Operations Snippets

Below are some snippet examples of using Stitch and CRUD operations such as deleting products, finding a single product, adding products and updating products.

Deleting a Product (Products.js File):

```
productDeleteHandler = productId => {
    const mongodb = Stitch.defaultAppClient.getServiceClient( RemoteMongoClient.factory,
    'mongodb-atlas' );
mongodb.db('test').collection('products').deleteOne( {
    _id: new BSON.ObjectId(productId)
} )
    .then(…).catch(…);
```

Again we get access to the mongodb client with the same code we used in the fetchData function. This will be the same for every CRUD operation where we interact with the database. We will notice that the CRUD operations are syntax the same as we have seen before in the shell and there is no difference.

The problem we have here is that we need to convert the productId from a string into an object id so that mongoDB can filter based on this. We can do this in the browser using a npm package called bson.

In the terminal navigated to the project directory, we can run the following command to install the bson package within our project.

```
$ npm install –save bson
```

This package will allow us to create bson types which are a special type that mongoDB works with in the browser and therefore can create the objectId object in the browser. We can now import this within our Product.js file to use.

```
import BSON from ‘bson’;
```

We have to instantiate the BSON object using the new keyword which creates a new ObjectId passing in the string value from productsId. This will now allow us to successfully delete products from our database.

Finding a Single Product (Product.js File):

```
componentDidMount( ) {
    const mongodb = Stitch.defaultAppClient.getServiceClient( RemoteMongoClient.factory,
    'mongodb-atlas' );
    mongodb.db('test').collection('products').find( {
        _id: new BSON.ObjectId(this.props.match.params.id)
    } ).asArray( )
        .then( productResponse => {
            const product = productResponse[0];
            product._id = product._id.toString( )
            product.price = product.price.toString( )
            this.setState( { isLoading: false, product: product } )
    } )
    .catch(…);
```

Because we used the find method (and not findOne method) which returns a cursor, we need to use the asArray function to return a promise else this will error. ProductResponse no longer holds any data as it now holds an array of products and therefore we must set the state to return the products from the array. We also need to convert the id and price so that JavaScript can work with the types.

Adding Products (EditProduct.js File):

```
const productData = {
    name: this.state.title, price: BSON.Decimal128.fromString(this.state.price.toString( )),
    image: this.state.imageUrl, description: this.state.description
}
const mongodb = Stitch.defaultAppClient.getServiceClient( RemoteMongoClient.factory,
'mongodb-atlas' );
let request;
if { …
} else {
    request = mongodb.db('test').collection('products').insertOne( productData )
}
request.then({…}).catch({…})
```

We need to ensure that the data has been converted to the mongoDB types using BSON before inserting the document into the database. The fromString method will convert a number stored as a string into the Decimal128 value type. Using the toString method will make sure the data is the type of string to prevent any errors on the conversion. We must ensure authenticated entities are setup with Stitch Rules to allow them to write data to the database else we will receive errors.

Updating Products (EditProduct.js File):

```
componentDidMount( ) {
    …//copied from fetching a single product
}
const mongodb = Stitch.defaultAppClient.getServiceClient( RemoteMongoClient.factory,
'mongodb-atlas' );
let request;
if (this.props.match.params.mode === 'Edit' ) {
    request = mongodb.db('test').collection('products').updateOne(
    { _id: new BSON.ObjectId( this.props.match.params.id ) }, productData )
}
```

In the componentDidMount lifecycle method we can copy this from the Finding a Single Product code as this is doing the same. We could refactor the code to follow the DRY principle but this robust method will help to understand and learn the code. We can then copy the code from the Adding Products and use the updateOne command instead and filter by id.

This should hopefully provide some useful insights as to how we can utilise Stitch to control the entities access to our application to perform CRUD action on our database without requiring a backend server.

User Email & Password Authentication

In a production environment we would want users to authenticate using their email and password. To switch to this authentication we would need to change a few settings. Firstly in the App.js file we would need to turn isAuth to false.

```
state = {
    isAuth: false,
    authMode: 'login',
    error: null
};
```

If we were to reload the application it should start on the login page. Stitch offers us a simple way of adding real user authentication. Within the Stitch dashboard Users section and within the Providers tab we need to deactivate anonymous authentication and enable Email/Password authentication. We need to setup the Email Confirmation URL which should point to http://localhost:3000/confirm-account (if we deploy to a real real server then we should update the URL link as it will not use localhost:3000). We have prepared a route to a page which we can work on to confirm the account with Stitch. The Password Reset URL is required and we will use http://localhost:3000/reset-password but this will not work because we have not added a route for this in the app.

The Reset Password Email Subject and Email Confirmation Subject fields are both optional.

We can now save the change and are ready to add Email/Password authentication within our application.

Introducing Stitch

Adding User Sign Up & Confirmation

We can now add our user authentication and logic to our application code. We would first need to update our App.js code:

The src/App.js File:

```
import { Stitch, ~~AnonymousCredential,~~ UserPasswordAuthProviderClient } from mongodb-stitch-browser-sdk;
```

We now need to import the user password authentication functionality from the Stitch Browser SDK and no longer require the AnonymousCredential. We can then go to our authHandler function which checks the validity of the input field. It is here we will use the Stitch Client

```
constructor( ) {
    super( );
    const this.client = Stitch.initializeDefaultAppClient( 'stitchtutorial-tjpqh' );
    this.client.auth.loginWithCredential( new AnonymousCredential( ) );
}

authHandler( )  = (event, authData) => {
    … }
    const emailPassClient =
    this.client.auth.getProviderClient(UserPasswordAuthProviderClient.factory);
        emailPassClient.registerWithEmail(authData.email, authData.password)
            .then( ( ) => { } )
            .catch (err => {
                this.errorHandler('An error occurred.');
                console.log(err);
                this.setState( { isAuth: false } );
            } );
    }
```

Instead of storing the initialised client in a variable, we store it in a property of this class by using the this.client which turns this into a property. This allows us to use it in other functions in the same class. The getProviderClient takes in the argument UserPasswordAuthProviderClient.factory and this basically initialises our UserPasswordAuthProvider functionality. We store this initialiser in a const variable called emailPassClient but can be called anythign we want. We can now use that emailPassClient to call registerWithEmail. In here we need to pass in the email and password the user has provided us in the application login form which we can retrieve from authData.

This will now sign users up with Stitch and again they are stored by Stitch and not within our collection. We have a promise returned in which we can use the .then and .catch promises to find out whether it was successful or not.

We can now sign up to Stitch using a real email and password within our app and we should see a email that has a link that we can click. If we click on the link it will look like we are returned to the Sign up page, but in the developer tools we should actually see this account confirmed output from the src/pages/Auth/ConfirmAccount.js JavaScript file. We parse some data which we find in the url confirm account page, which we now need to send that data to the backend of Stitch to confirm our password. We do this by importing Stitch from the Stitch Browser SDK within our ConfirmAccount.js file and then we can call on some method to work with Stitch.

The src/pages/Auth/ConfirmAccount.js File:

```
import { Stitch, UserPasswordAuthProviderClient } from mongodb-stitch-browser-sdk;
componentDidMount( ) {
    …
    ~~console.log('Account confirmed');~~
    const token = queryParams.get('token'); const tokenId = queryParams.get('tokenId');
    const emailPassClient =
    Stitch.defaultAppClient.auth.getProviderClient(UserPasswordAuthProviderClient.factory);
    emailPassClient.confirmUser(token, tokenId)
        .then(( ) => { this.props.history.replace('/'); } )
        .catch(err => console.log(err) );
}
```

We can replace the console.log command and can now replicate the code that we had in the App.js file. We can now use the emailPassClient to now confirm the user. To confirm the user we need to pass in the token and tokenId which we have already parsed from the URL. Stitch behind the scenes will validate whether the token and tokenId that was mentioned in the email and if all matches it will confirm the user. This will give a promise which either succeeds or fails. If it succeeds we want to redirect the user to the home products page. This completes the Email/Password Auth setup.

Adding User Login

Now that we have added the signup to our application using Stitch, we now need to add the Login for our application. This requires the App.js file code updating.

The src/App.js File:

```
import { Stitch, UserPasswordAuthProviderClient, UserPasswordCredential } from mongodb-stitch-browser-sdk;
authHandler( ){
    …
    let request;
    const emailPassClient = this.client.auth.getProviderClient(
    UserPasswordAuthProviderClient.factory);

    if(this.state.authMode === ‘login’) {
        const credential = new UserPasswordCredential(authDate.email, authData.password);
        request = this.client.auth.loginWithCredential(credential);
```

```
} else {
        request = emailPassClient.registerWithEmail(authData.email, authDate.password)
}
request.then(result => {
        console.log(result);
        if(result) {

        }
} )
.catch(…);
}
```

In the authHandler function we need to find an alternative to signup up a user. We will continue to use the if check to see if the user is authenticating with a email and password. We use the request variable to store the result of the connection and then use the request variable with our promises.

To Login with Stitch we need to import the UserPasswordCredential (this replaces the AnonymousCredential) and we can now use this in our stored credential variable, passing in the two pieces of information: the email and password. With the credential we can now login as we did

before with the anonymous credential.

We use the general stitch client and on auth we can call on the loginWithCredential function passing in the credential variable we created containing the email and password as the argument. This will give us a promise which we can store within our request variable. We would now receive a result for our then promise which will simply notify whether we are logged in. The whole session of the user to the tokens are managed by Stitch behind the scenes. We can see the tokens by going to the browser developer tools within the Applications tab and within the Local Storage, we should see the __stitch.client which stores the information about the user and the token that is managed for us. We now have a way for users to login to our application.

We use a if statement to check if we are logged in and if so we want to change the isAuth state to true. This will then load the products for the user when they have logged into the application. We now have a signup and login to our application using Stitch and would now need to configure the Rules for our authenticated user to interact with the products.

Rules & Real Users

Within the Stitch Dashboard and within the Rules section, we have rules for our products collection but we have these default rules where everyone is able to read and write to our database collection (which we configured when we had Anonymous Entities). Everyone still means logged in users and therefore since we have disabled anonymous authentication, this means only those who have logged in with an email/password can read and write to our product collection. Everything else should be working as before but now only email/password users can interact with the database only.

So we now have a fully working application with Stitch and we do not need to write our own backend which of course can save us a lot of work and time and still have a secure solution.

The Current State of Authentication

With the currently implemented authentication solution, we're NOT taking advantage of Stitch's automatically managed user tokens. Stitch does store the user tokens in the localStorage and it

does detect whether a user is logged in (i.e. if such valid tokens can be found in localStorage) upon initialisation. However, we are not using that in our app, instead we always start off with isAuth: false in our src/App.js file.

You could of course change that and add an auth listener (https://s3.amazonaws.com/stitch-sdks/js/docs/4/interfaces/stitchauth.html#addauthlistener) to Stitch in the constructor (or componentDidMount) of App.js:

```
Stitch.defaultAppClient.auth.addAuthListener(auth => {
    this.setState({isAuth: auth.isLoggedIn});
})
```


Functions & Triggers

Functions and Triggers are two other major features of Stitch. Functions allows us to write some code that we can execute. We can create a new function and in the settings tab give our Function Name. We can then write our function code in the Function Editor. We write our code in JavaScript and all the features are in the official documentation. We can execute JavaScript code, access

other services or access our database using the function. If we write a function for example:

```
console.log('Hello World', arg);
```

We can call this function from inside our frontend/client application through the Stitch Browser SDK for example if we update the App.js constructor function:

```
this.client = Stitch.initializeDefaultAppClient( 'stitchtutorial-tjpqh' );
this.client.callFunction('Hello World', ['John Doe']);
```

We can utilise the client and then use the callFunction method passing in the name of the function we setup in the Stitch Dashboard that we want to run. This will therefore look into our Stitch app which we initialised and stored in this.client variable and then find that function. We can also pass some arguments as a second parameter which is an array of data we pass into the function.

If we reload the application and look at the logs in the Stitch Dashboard Logs section, we should see a bunch of logs but we also see the function call when the application was reloaded. We would see Hello World John Doe as the output. John Doe was the argument that was passed to the function and hence we could use it in our client side code and functions are as simple as that. This function runs totally on a server and users cannot see our code in there and therefore if we have any code that we want to execute and the user should not be able to see, we can put it into a function and call that function from our frontend and then this code is invisible to our users.

We can also put longer taking task in our functions as an example which we do not want to execute in the browser and things of that nature.

We can also execute a function from inside or upon a trigger. On the Stitch Dashboard within the triggers section, we could add an authentication trigger which allows us to run a function when a user was created, logged in or is deleted or alternatively we could add a database trigger.

If we create a database trigger as an example, we would give the Trigger a name, check the Enabled button, add a trigger on the database and collection name and we can check the tick box to listen to either of the CRUD operations (Insert, Update, Delete and Replace operations to be precise). Whenever the operation occurs in the database and collection specified for example a insert operation, we can execute a function by specifying the function name. We can also check the full document to pass the full inserted document as an argument to our function. This will now execute the named function every time we insert a document within the specified database and collection. We can look at the logs to see the function being executed when the trigger condition has been met.

Functions and triggers are very useful features that allow us to write code that runs on the server and even control when the function should run when something happens to one of our collections.