COM661 Full Stack Strategies and Development

BE07. Complex MongoDB Queries

Aims

- To demonstrate how the structure of a MongoDB collection can be modified in code
- To introduce the MongoDB Aggregation Pipeline
- To implement complex database queries through a pipeline of stages
- To investigate options for storage and summary of pipeline output
- To introduce GeoJSON to support location-based queries
- To implement location-based queries using the Aggregation Pipeline

Table of Contents

7.1 Enhancing the Dataset

In the previous practical, we introduced the **PyMongo** library which enables us to embed MongoDB queries into Python applications and process the results obtained. Sometimes, however, we prefer (at least part of) the data processing to be performed by the Database engine rather than the Python application logic so that we can minimize the volume of data returned and return the data in the precise form in which it is required by the application. In such circumstances, the **`find()`** and **`find_one()`** methods do not support sufficiently complex queries, but MongoDB provides a much more flexible query structure known as the **Aggregation Pipeline**. In this practical, we will introduce and examine the Aggregation Pipeline and demonstrate how we can use it to specify more complex retrieval queries.

7.1.1 Adding Additional Fields

Initially, we want to provide a more complex dataset on which to operate, so we will revisit the **biz** collection within the **bizDB** database generated last time to introduce and populate additional fields. Currently, each business is described by a **`name`**, a **`town`** and a **`rating`** – each set to a random value, as well as a **`reviews`** object that is initialised to an empty list. We will extend the definition by adding a field to hold the number of employees in the range 1-100 as well as random gross profit figures for the years 2019, 2020 and 2021. A business will therefore be defined as follows, with the newly added fields highlighted.

```
{
   “name” : “Biz 12”,
   “town” : “Coleraine”,
   “rating” : 3,
   “reviews” : [],
   “num_employees” : 35,
   “profit” : [
      {
         “year” : “2022”,
         “gross” : 50000
      },
      {
         “year” : “2023”,
         “gross” : -10000,
      },
      {
         “year” : “2024”,
         “gross” : 15000
      }
   ]
}
```

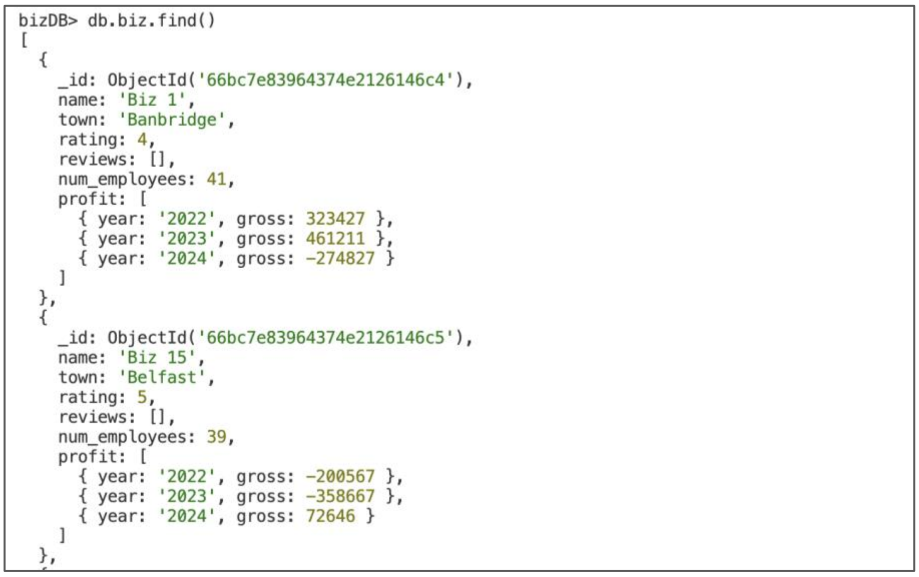
We can make this modification to the structure of each business object by retrieving all businesses with the **`find()`** method and then invoking **`update_one()`** on each in turn, using the **`$set`** command to identify the new fields to be added and the random value for each.

Note: The combination of **`update_one()`** and **`$set`** is exactly the same as already shown when editing the value of a field. If a field already exists, then its current value is changed, otherwise the field is created and initialised to the value provided.

File: add_fields.py

```
from pymongo import MongoClient
import random

client = MongoClient("mongodb://127.0.0.1:27017")
db = client.bizDB
businesses = db.biz

for business in businesses.find():
    businesses.update_one(
        { "_id" : business['_id'] },
        {
          "$set" : {
            "num_employees" : random.randint(1, 100),
            "profit" : [
               { "year": "2022",
                 "gross": random.randint(-500000, 500000)
               },
               { "year": "2023",
                 "gross": random.randint(-500000, 500000)
               },
               { "year": "2024",
                 "gross": random.randint(-500000, 500000)
               },
            ]
          }
        }
    )
```

Do it now!	Add the program ***add_fields.py*** (supplied in the ***BE07 Practical Files***) to the Biz application and run it by the command **`python3 add_fields.py`**. Remember that you will also need to invoke your virtual environment and make sure that the MongoDB server is running. Once the program has run, check that it has had the desired effect by using the MongoDB shell to retrieve a document from the **biz** collection and ensure that the new fields have been added as shown in Figure 7.1, below.



```
bizDB> db.biz.find()
[
  {
    _id: ObjectId('66bc7e83964374e2126146c4'),
    name: 'Biz 1',
    town: 'Banbridge',
    rating: 4,
    reviews: [],
    num_employees: 41,
    profit: [
      { year: '2022', gross: 323427 },
      { year: '2023', gross: 461211 },
      { year: '2024', gross: -274827 }
    ]
  },
  {
    _id: ObjectId('66bc7e83964374e2126146c5'),
    name: 'Biz 15',
    town: 'Belfast',
    rating: 5,
    reviews: [],
    num_employees: 39,
    profit: [
      { year: '2022', gross: -200567 },
      { year: '2023', gross: -358667 },
      { year: '2024', gross: 72646 }
    ]
  },
```

Figure 7.1 New Fields Added

7.1.2 Removing Fields

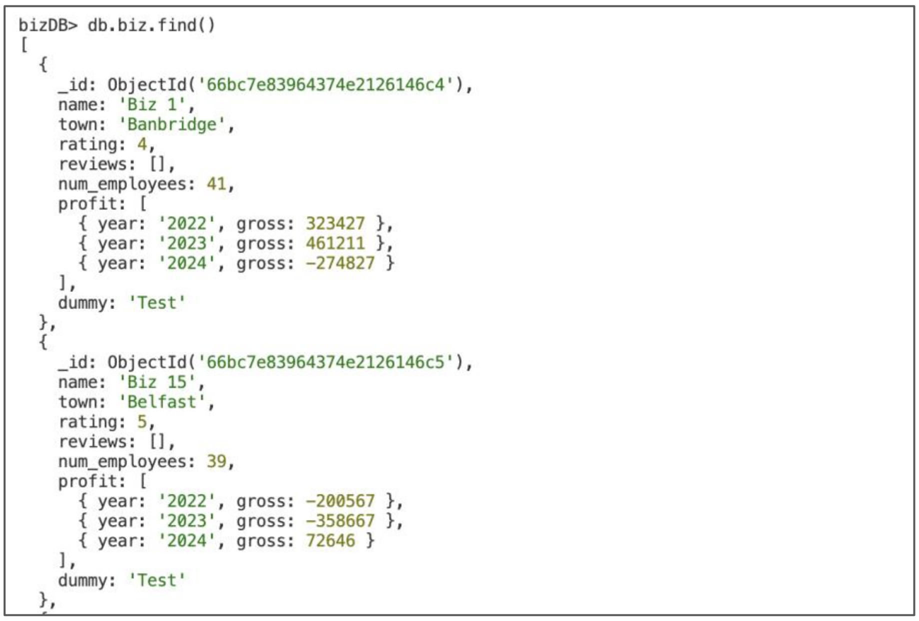
In the same way as we can modify the structure of a document by adding additional fields, so we can also remove fields from a document. To demonstrate this, let's first add an additional field called "**`dummy`**" initialised to the string value "**`Test`**".

Note:	Leaving the **`num_employees`** and **`profit`** fields in the code for this example will result in new values being generated for these fields.

File: add_fields.py

```
...

for business in businesses.find():
    businesses.update_one(
        { "_id" : business['_id'] },
        {
          "$set" : {
            "num_employees" : random.randint(1, 100),
            "profit" : [
               …
            ],
            "dummy" : "Test"
          }
        }
    )
```

Do it now! Change ***add_fields.py*** as shown above to add the additional "dummy" field. Now, run the program again and use the MongoDB shell to repeat the previous retrieval query and verify that the field has been added.



```
bizDB> db.biz.find()
[
  {
    _id: ObjectId('66bc7e83964374e2126146c4'),
    name: 'Biz 1',
    town: 'Banbridge',
    rating: 4,
    reviews: [],
    num_employees: 41,
    profit: [
      { year: '2022', gross: 323427 },
      { year: '2023', gross: 461211 },
      { year: '2024', gross: -274827 }
    ],
    dummy: 'Test'
  },
  {
    _id: ObjectId('66bc7e83964374e2126146c5'),
    name: 'Biz 15',
    town: 'Belfast',
    rating: 5,
    reviews: [],
    num_employees: 39,
    profit: [
      { year: '2022', gross: -200567 },
      { year: '2023', gross: -358667 },
      { year: '2024', gross: 72646 }
    ],
    dummy: 'Test'
  },
```

Figure 7.2 Adding an Extra Field to Demonstrate Removal

Now that the new field has been added to each document in the collection, we can remove it by specifying the **`$unset`** command in the **`update_one()`** method. Note how the name of the field to be removed is provided as the key field of a JSON object, while the value of the field is left as an empty string.

File: add_fields.py

```
...

for business in businesses.find():
    businesses.update_one(
        { "_id" : business['_id'] },
        { "$unset" : { "dummy" : "" } }
    )
```

Do it now! Change ***add_fields.py*** as shown above to remove the additional "dummy" field. Now, run the program again and use the MongoDB shell to repeat the previous retrieval query and verify that the field has been removed and that a MongoDB query once again produces output such as that shown in Figure 7.1.

7.2 The Aggregation Pipeline

Now that our dataset has the required complexity, we can introduce the power and flexibility of the Aggregation Pipeline through the development of some sample queries. In general, the Pipeline is a multi-stage operation, by which the query is divided into a series of individual steps, each of which performs a selection, filtering or transformation operation and passes the result on to the next stage. It can be represented graphically by the illustration in Figure 7.3 which presents the transformation of a source collection into some output by a 3-stage process. In reality, the pipeline can consist of as many stages as are required for the specific query.

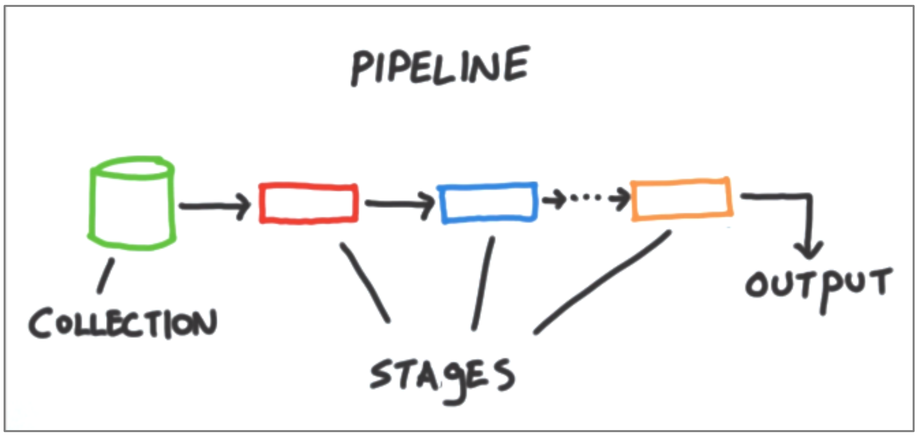


Figure 7.3 The Aggregation Pipeline

7.2.1 Basic Retrieval

An Aggregation Pipeline is constructed by the **`aggregate`** method shown below applied to the **`biz`** collection, which takes a single parameter which is a list of stages.

```
db.biz.aggregate( [

])
```

At the most basic level, the Pipeline can replicate the functionality of the ***PyMongo*** **`find()`** method. For example, to return the collection of businesses where the "**`town`**" field has the value "**`Banbridge`**", we could apply the **`$match`** operator as shown below.

```
db.biz.aggregate( [
     { $match : { "town" : "Banbridge" } }
])
.pretty()
```

This command will have exactly the same effect as the standard MongoDB command

```
db.biz.find( { town : "Banbridge" } )
```

Note: We have chained a **`pretty()`** operation to the query purely in order to format the result in a readable way within the MongoDB console. **`pretty()`** is not part of the Aggregation Pipeline.

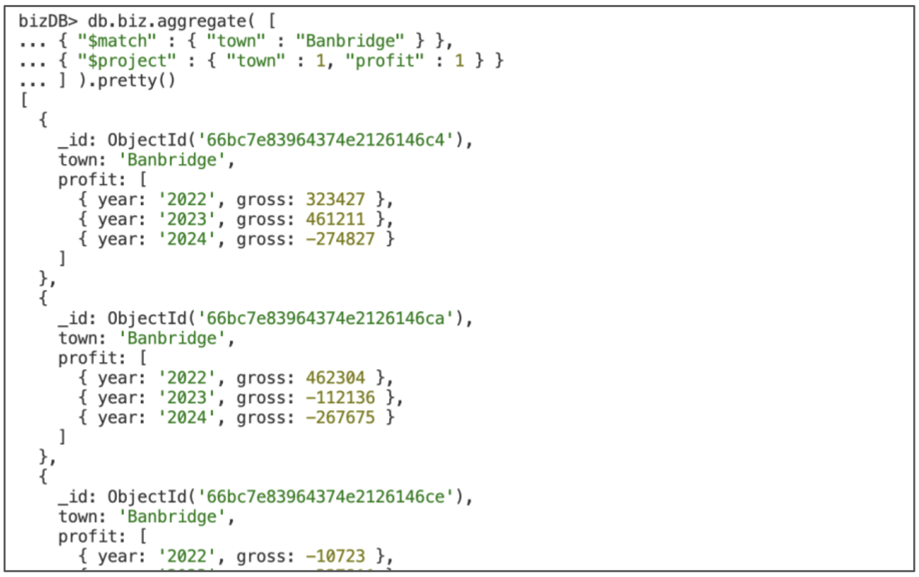
Just as for the **`find()`** query, we can add a projection that specifies the fields to be returned from the query. For example, if the only fields required are "**`town`**" and "**`profit`**", we can specify this by adding a **`$project`** stage to the pipeline as shown below. Note how each stage in the Pipeline is added as a new object in the list of stages, with stages separated by a comma.

```
db.biz.aggregate( [
     { $match : { "town" : "Banbridge" } },
     { $project : { "town" : 1, "profit" : 1  } }
])
.pretty()
```

Again, this could be equally be accomplished by the standard **`find()`** query as

```
db.biz.find( { town : "Banbridge" },
             { town : 1, profit : 1 }
           )
```

Do it now!	Test the Aggregation Pipeline by opening a MongoDB console and specifying an **`aggregate()`** query using the **`$match`** and **`$project`** operations. Ensure that you receive a result similar to that shown in Figure 7.4 below.



```
bizDB> db.biz.aggregate( [
... { "$match" : { "town" : "Banbridge" } },
... { "$project" : { "town" : 1, "profit" : 1 } }
... ] ).pretty()
[
  {
    _id: ObjectId('66bc7e83964374e2126146c4'),
    town: 'Banbridge',
    profit: [
      { year: '2022', gross: 323427 },
      { year: '2023', gross: 461211 },
      { year: '2024', gross: -274827 }
    ]
  },
  {
    _id: ObjectId('66bc7e83964374e2126146ca'),
    town: 'Banbridge',
    profit: [
      { year: '2022', gross: 462304 },
      { year: '2023', gross: -112136 },
      { year: '2024', gross: -267675 }
    ]
  },
  {
    _id: ObjectId('66bc7e83964374e2126146ce'),
    town: 'Banbridge',
    profit: [
      { year: '2022', gross: -10723 },
```

Figure 7.4 The $match and $project Stages

As well as issuing aggregation queries from the MongoDB console prompt, we can also embed the queries in our Python programs. Consider the program ***aggregation.py*** which repeats the above query and prints out the town name and gross profit for 2024 for each business returned by the query.

File: aggregation.py

```
from pymongo import MongoClient

client = MongoClient("mongodb://127.0.0.1:27017")
db = client.bizDB
businesses = db.biz

pipeline = [
    { "$match" : { "town" : "Banbridge"} },
    { "$project" : {"town" : 1, "profit" : 1 } }
]

for business in businesses.aggregate(pipeline):
    print( business["town"], \
           str(business["profit"][2]["gross"]) )
```

Here, we specify the **`pipeline`** as a list of objects before passing it as a parameter to the **`aggregate()`** method. This is purely for reasons of code readability – we could equally specify the pipeline directly as the parameter to **`aggregate()`**. Note also how we gain access to the gross profit value for 2024. As the 2024 figure is the 3rd element in the **`profit`** list, we specify element **`[2]`** in the list before obtaining the **`gross`** attribute from that element. Finally, as this data item is an integer, we convert it to string by the **`str()`** function before outputting it using the **`print()`** statement.

Do it now! Verify the operation of ***aggregation.py*** by adding the file supplied in the ***BE07 Practical Files*** to the ***biz*** folder and running it. Remember that you will need to activate your virtual environment before running the program.

Try it now! Modify ***aggregation.py*** to output the name and number of employees of all businesses in Belfast.

7.2.2 In-query Processing

So far, all examples of **`aggregate()`** have been equally achievable by the standard **`find()`** command. However, **`find()`** has significant limitations that require a more advanced solution – especially when working with sub-documents.

Often, sub-document processing requires a pair of nested loops where the outer loop iterates across the collection of top-level objects and the inner loop iterates across the collection of sub-documents in each. We have already seen this organization in the API endpoint to retrieve the collection of reviews for each business in Practical BE03. The Aggregation Pipeline supports an **`unwind`** operation that generates separate documents by looping over each sub-document in turn. For example, a collection of

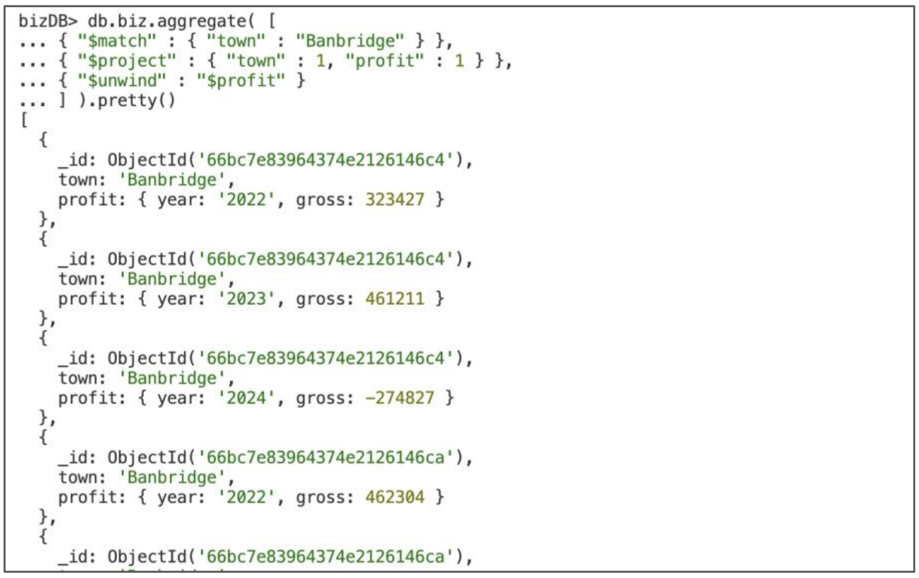
```
[
   {
     "person" : "Adrian"
     "offspring" : [
                      { "name" : "Adam" },
                      { "name" : "Caitlin" }
                    ]
   }
]
```

subjected to an **`unwind`** operation on the **`"offspring"`** field would result in the collection

```
[
   {
     "person" : "Adrian",
     "offspring" : { "name" : "Adam" }
   },
   {
     "person" : "Adrian",
     "offspring" : { "name" : "Caitlin" }
   },
]
```

with a separate document generated for each of the sub-documents in the original collection and all other fields in the top-level document repeated with their original values. We can demonstrate this in the context of our **`biz`** collection by unwinding the collection on the **`profit`** field. Note how the field name **`profit`** is prefixed by **`$`** in the **`$unwind`** command. This is standard in MongoDB to distinguish field names from literal string values.

Do it now!	Test the Aggregation Pipeline by opening a MongoDB console and add an **`$unwind`** operation as shown. Ensure that you receive a result similar to that illustrated in Figure 7.5 below.



```
bizDB> db.biz.aggregate( [
... { "$match" : { "town" : "Banbridge" } },
... { "$project" : { "town" : 1, "profit" : 1 } },
... { "$unwind" : "$profit" }
... ] ).pretty()
[
  {
    _id: ObjectId('66bc7e83964374e2126146c4'),
    town: 'Banbridge',
    profit: { year: '2022', gross: 323427 }
  },
  {
    _id: ObjectId('66bc7e83964374e2126146c4'),
    town: 'Banbridge',
    profit: { year: '2023', gross: 461211 }
  },
  {
    _id: ObjectId('66bc7e83964374e2126146c4'),
    town: 'Banbridge',
    profit: { year: '2024', gross: -274827 }
  },
  {
    _id: ObjectId('66bc7e83964374e2126146ca'),
    town: 'Banbridge',
    profit: { year: '2022', gross: 462304 }
  },
  {
    _id: ObjectId('66bc7e83964374e2126146ca'),
```

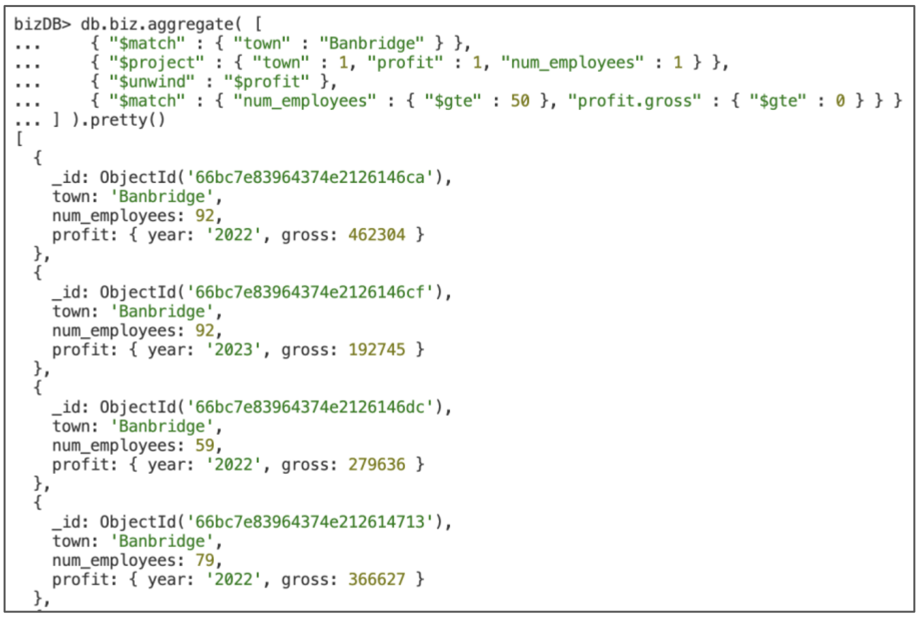
*Figure 7.5 $**`unwind`** on the **`profit`** Sub-document Collection*

Having generated individual documents by the **`$unwind`** operation, we may want to filter them based on some criteria – in the same way in which we filtered the initial collection by only selecting those based in a specific town. We can achieve this by adding another **`$match`** operation as the next stage of the pipeline, to select only those documents for which the number of employees is at least 50 and where the business broke even or turned a profit.

```
db.biz.aggregate([
   { $match : { "town" : "Banbridge" }},
   { $project : {
      "town" : 1, "profit" : 1, "num_employees" : 1 } },
   { $unwind : "$profit" },
   { $match : {
        "num_employees" : { $gte : 50 },
        "profit.gross" : { $gte : 0 } } }
])
.pretty()
```

As the **`$match`** operation refers to **`num_employees`**, so we need to update the **`$project`** object to include this field. Note the **`$gte`** "greater than or equal to" operators applied to the "**`num_employees`**" and "**`profit.gross`**" fields. Other comparison operators such as **`$gt`** (greater than), **`$lt`** (less than), **`$lte`** (less than or equal to), **`$eq`** (equal to) and **`$ne`** (not equal to) are also available and are used in the same way.

Do it now!	Add the new **`$match`** stage to the pipeline as shown above and verify its effect by running the query and confirming that you obtain a result similar to that of Figure 7.6 below.



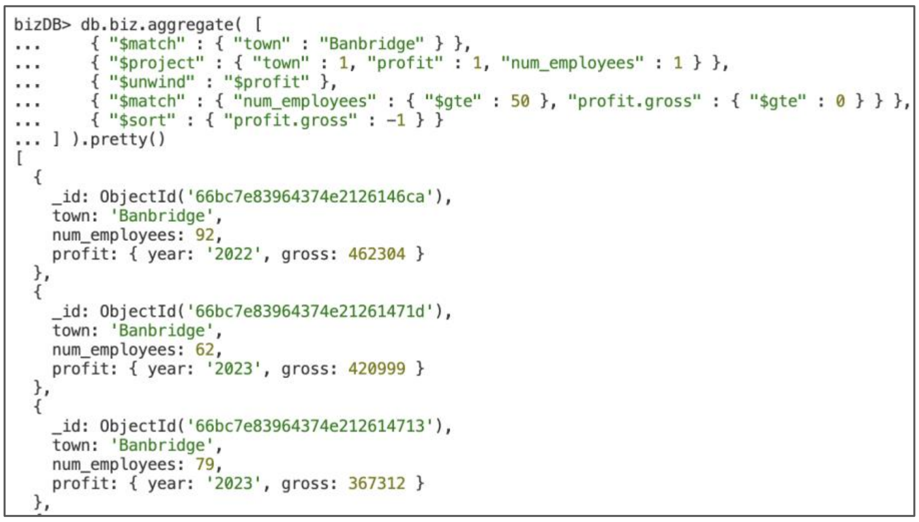
```
bizDB> db.biz.aggregate( [
...     { "$match" : { "town" : "Banbridge" } },
...     { "$project" : { "town" : 1, "profit" : 1, "num_employees" : 1 } },
...     { "$unwind" : "$profit" },
...     { "$match" : { "num_employees" : { "$gte" : 50 }, "profit.gross" : { "$gte" : 0 } } }
... ] ).pretty()
[
  {
    _id: ObjectId('66bc7e83964374e2126146ca'),
    town: 'Banbridge',
    num_employees: 92,
    profit: { year: '2022', gross: 462304 }
  },
  {
    _id: ObjectId('66bc7e83964374e2126146cf'),
    town: 'Banbridge',
    num_employees: 92,
    profit: { year: '2023', gross: 192745 }
  },
  {
    _id: ObjectId('66bc7e83964374e2126146dc'),
    town: 'Banbridge',
    num_employees: 59,
    profit: { year: '2022', gross: 279636 }
  },
  {
    _id: ObjectId('66bc7e83964374e212614713'),
    town: 'Banbridge',
    num_employees: 79,
    profit: { year: '2022', gross: 366627 }
  },
```

Figure 7.6 Adding a $match to the Unwound Dataset

Sometimes, we prefer that the data returned from a query is sorted in a particular order so that it can be easily displayed in a readable fashion. The next stage of the pipeline is to add a **`$sort`** operation to present the data sorted by descending order of gross profit, so that the most profitable business is displayed first.

```
db.biz.aggregate([
   { $match : { "town" : "Banbridge" }},
   { $project : {
      "town" : 1, "profit" : 1, "num_employees" : 1 } },
   { $unwind : "$profit" },
   { $match : {
        "num_employees" : { $gte : 50 },
        "profit.gross" : { $gte : 0 } } },
   { $sort : { "profit.gross" : -1 } }
])
.pretty()
```

Do it now!	Add the new **`$sort`** stage to the pipeline as shown above and verify its effect by running the query and confirming that you obtain a result similar to that of Figure 7.7 below. Try changing the direction of the sort from descending to ascending by replacing the **`-1`** in the **`$sort`** command with **`1`**.



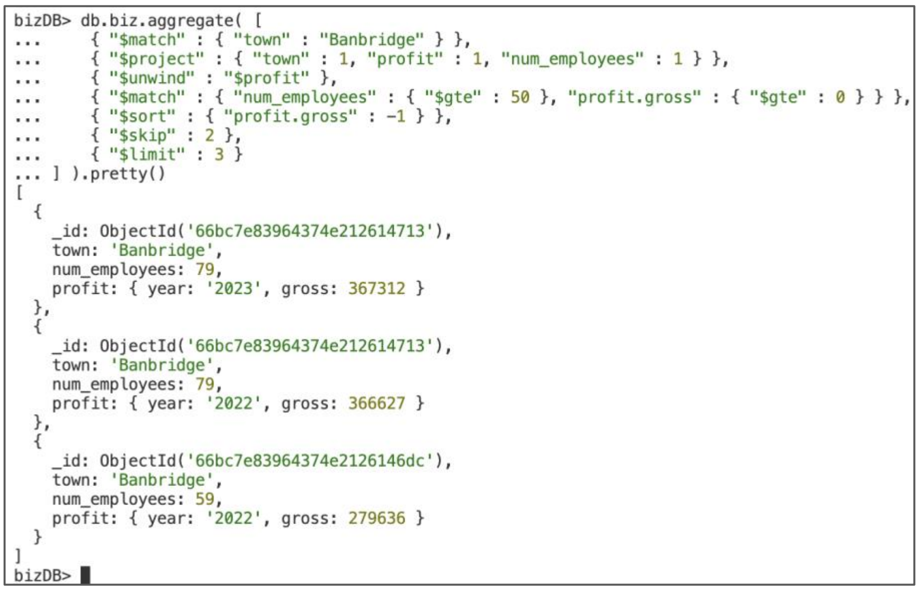
```
bizDB> db.biz.aggregate( [
...      { "$match" : { "town" : "Banbridge" } },
...      { "$project" : { "town" : 1, "profit" : 1, "num_employees" : 1 } },
...      { "$unwind" : "$profit" },
...      { "$match" : { "num_employees" : { "$gte" : 50 }, "profit.gross" : { "$gte" : 0 } } },
...      { "$sort" : { "profit.gross" : -1 } }
... ] ).pretty()
[
  {
    _id: ObjectId('66bc7e83964374e2126146ca'),
    town: 'Banbridge',
    num_employees: 92,
    profit: { year: '2022', gross: 462304 }
  },
  {
    _id: ObjectId('66bc7e83964374e21261471d'),
    town: 'Banbridge',
    num_employees: 62,
    profit: { year: '2023', gross: 420999 }
  },
  {
    _id: ObjectId('66bc7e83964374e212614713'),
    town: 'Banbridge',
    num_employees: 79,
    profit: { year: '2023', gross: 367312 }
  },
```

*Figure 7.7 Adding a **`$sort`** to the Pipeline*

The final processing options that we will examine (though by no means all that the Aggregation Pipeline has to offer) are operations to limit the number of documents that are returned and to select which slice of the document list to return. These are the equivalent of the **`skip()`** and **`limit()`** methods that we have seen chained to the end of the **`find()`** method in Practical C2 and are specified in a similar way. Both accept a single non-negative integer as a parameter which specifies the number of documents to be ignored (in the case of **`$skip`**) and the number of documents to be returned (in the case of **`$limit`**). The following code fragment illustrates **`$skip`** and **`$limit`** stages added to our pipeline.

```
db.biz.aggregate([
   { $match : { "town" : "Banbridge" }},
   { $project : {
      "town" : 1, "profit" : 1, "num_employees" : 1 } },
   { $unwind : "$profit" },
   { $match : {
        "num_employees" : { $gte : 50 },
        "profit.gross" : { $gte : 0 } } },
   { $sort : { "profit.gross" : -1 } },
   { $skip : 2 },
   { $limit : 3 }
])
.pretty()
```

Do it now! Add the **`$skip`** and **`$limit`** stages to the pipeline as shown above and verify their effect by running the query and confirming that you obtain a result similar to that of Figure 7.8 below. Choose parameter values for **`$skip`** and **`$limit`** that enable you to clearly demonstrate that the stages are having the desired effect.



```
bizDB> db.biz.aggregate( [
...     { "$match" : { "town" : "Banbridge" } },
...     { "$project" : { "town" : 1, "profit" : 1, "num_employees" : 1 } },
...     { "$unwind" : "$profit" },
...     { "$match" : { "num_employees" : { "$gte" : 50 }, "profit.gross" : { "$gte" : 0 } } },
...     { "$sort" : { "profit.gross" : -1 } },
...     { "$skip" : 2 },
...     { "$limit" : 3 }
... ] ).pretty()
[
  {
    _id: ObjectId('66bc7e83964374e212614713'),
    town: 'Banbridge',
    num_employees: 79,
    profit: { year: '2023', gross: 367312 }
  },
  {
    _id: ObjectId('66bc7e83964374e212614713'),
    town: 'Banbridge',
    num_employees: 79,
    profit: { year: '2022', gross: 366627 }
  },
  {
    _id: ObjectId('66bc7e83964374e2126146dc'),
    town: 'Banbridge',
    num_employees: 59,
    profit: { year: '2022', gross: 279636 }
  }
]
bizDB> 
```

*Figure 7.8 Adding **`$skip`** and **`$limit`** Stages*

7.2.3 Handling Pipeline Output

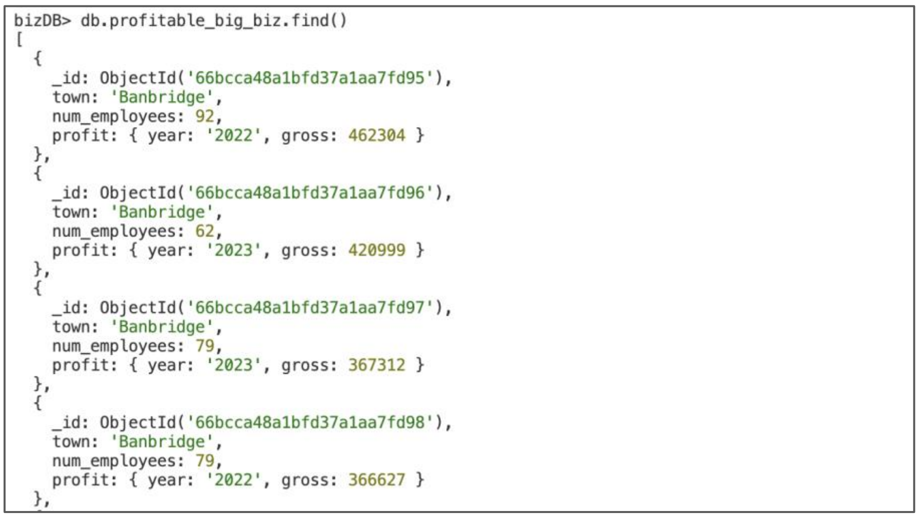
The previous section demonstrated a range of pipeline stages which enable highly flexible retrieval of information from a MongoDB database, and we have seen how the pipeline generates a collection of documents according to the structure specified in the pipeline.

However, there are two other situations in which we may wish to post-process the documents returned – (i) we may want to save the output to a new collection so that we can revisit it for later analysis or share it with another application without having to re-generate it from scratch, and (ii) we may want to generate a summary or other analysis of the results. The Aggregation Pipeline supports both activities, and we will introduce them in this section.

First, we will add an **`$out`** stage to the end of the pipeline to write the results of the query to a new collection called "**`profitable_big_biz`**". As we would like all matching documents to be recorded and we are not concerned with the order of the documents, we also take the opportunity to remove the **`$sort`**, **`$skip`** and **`$limit`** stages. Note also that we need to add an element to the **`$project`** object to explicitly specify that the **`_id`** field is not returned. The **`_id`** is returned by default from every MongoDB query and if we were to allow **`_id`** values to be returned, the database would generate an error when trying to create the new collection since **`_id`** values must be unique and cannot be duplicated across collections.

```
db.biz.aggregate([
   { $match : { "town" : "Banbridge" }},
   { $project : {
      "town" : 1, "profit" : 1, "num_employees" : 1, "_id" : 0 } },
   { $unwind : "$profit" },
   { $match : {
        "num_employees" : { $gte : 50 },
        "profit.gross" : { $gte : 0 } } },
   { $sort : { "profit.gross" : -1 } },
   { $out : "profitable_big_biz" }
])
.pretty()
```

Do it now!	Modify the pipeline as shown to remove the **`$sort`**, **`$skip`** and **`$limit`** stages and add the **`$out`** stage. Run the query and verify that the new "profitable_big_biz" collection is added to the database. Then, issue a **`find()`** query on the new collection to verify that it contains the result set from the aggregation query, as shown in Figure 7.9.



```
bizDB> db.profitable_big_biz.find()
[
  {
    _id: ObjectId('66bcca48a1bfd37a1aa7fd95'),
    town: 'Banbridge',
    num_employees: 92,
    profit: { year: '2022', gross: 462304 }
  },
  {
    _id: ObjectId('66bcca48a1bfd37a1aa7fd96'),
    town: 'Banbridge',
    num_employees: 62,
    profit: { year: '2023', gross: 420999 }
  },
  {
    _id: ObjectId('66bcca48a1bfd37a1aa7fd97'),
    town: 'Banbridge',
    num_employees: 79,
    profit: { year: '2023', gross: 367312 }
  },
  {
    _id: ObjectId('66bcca48a1bfd37a1aa7fd98'),
    town: 'Banbridge',
    num_employees: 79,
    profit: { year: '2022', gross: 366627 }
  },
```

Figure 7.9 Creating a New Collection with the ***$out*** *Stage*

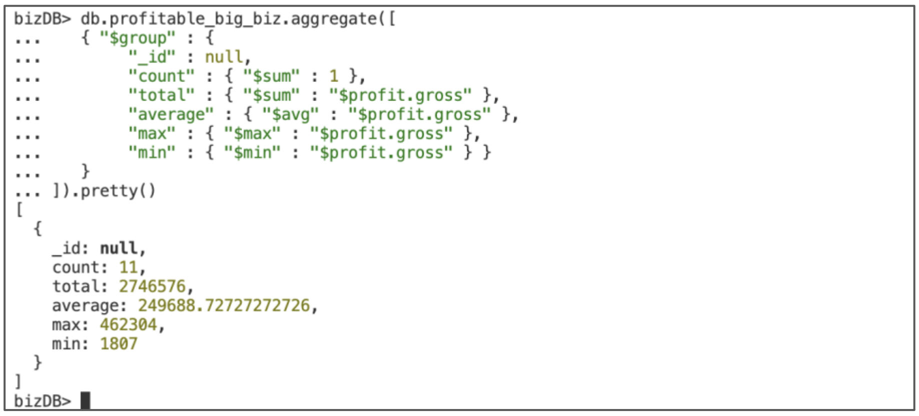
The other possibility is that we wish to generate some summary information on the results generated by the pipeline. This is achieved by the **$group** operation which combines the documents in a collection according to some criteria and outputs a single document for each group that contains the fields specified in the **$group** syntax. As a first example, consider the following query that reports the number of documents output by the pipeline along with the total, average, maximum and minimum profit values.

```
db.biz.aggregate([
   { $match : { "town" : "Banbridge" }},
   { $project : {
      "town" : 1, "profit" : 1, "num_employees" : 1, "_id" : 0 } },
   { $unwind : "$profit" },
   { $match : {
        "num_employees" : { $gte : 50 },
        "profit.gross" : { $gte : 0 } } },
   { $group : {
        _id : null,
        count : { $sum : 1 },
        total : { $sum : "$profit.gross" },
        average : { $avg : "$profit.gross" },
        max : { $max : "$profit.gross" },
        min : { $min : "$profit.gross" } } }
])
.pretty()
```

Note that, since the output of the first 4 stages in the pipeline is exactly that which we have stored in the "**profitable_big_biz**" collection, we could equally write this query making use of this as shown here.

```
db.profitable_big_biz.aggregate([
   { $group : {
        _id : null,
        count : { $sum : 1 },
        total : { $sum : "$profit.gross" },
        average : { $avg : "$profit.gross" },
        max : { $max : "$profit.gross" },
        min : { $min : "$profit.gross" } } }
])
.pretty()
```

Do it now!	Use the $group option to generate a document of summary information as described above. Try both methods and ensure that they generate identical results in the form demonstrated by Figure 7.10 below.



```
bizDB> db.profitable_big_biz.aggregate([
...    { "$group" : {
...         "_id" : null,
...         "count" : { "$sum" : 1 },
...         "total" : { "$sum" : "$profit.gross" },
...         "average" : { "$avg" : "$profit.gross" },
...         "max" : { "$max" : "$profit.gross" },
...         "min" : { "$min" : "$profit.gross" } }
...    }
... ]).pretty()
[
  {
    _id: null,
    count: 11,
    total: 2746576,
    average: 249688.72727272726,
    max: 462304,
    min: 1807
  }
]
bizDB> 
```

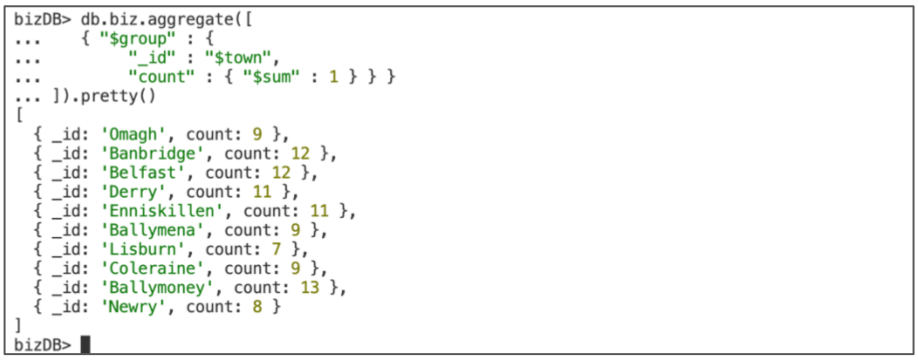
Figure 7.10 Generating Summary Information

There are two particularly notable elements to the **`$group`** stage that can be explored further. First, note how the number of documents is calculated by the expression **`{ $sum : 1 }`**. The effect of this is to create a "running total" by adding 1 for each document that is examined. This can be easily seen by changing the value in the **`$sum`** object from 1 to 2, re-running the query, and examining the result obtained.

Secondly, the **`_id`** field in the **`$group`** output is used to control the grouping. Where the **`_id`** is set to **`null`** (as in the example here), then all documents are bundled into a single group and hence a single document is returned as a result. Alternatively, we can specify one of the field names as the value of the **`_id`** element, which will cause the collection to be grouped into unique values for that field and a separate summary document is produced for each group. We can see this in the example below, which specifies the **"town"** field as the value of the **`_id`** and uses **`$sum`** to return a collection of documents reporting the number of businesses in each town.

```
db.biz.aggregate([
   { $group : {
        _id : "$town",
        count : { $sum : 1 } } }
])
.pretty()
```

Do it now!	Use the **`$group`** option to generate a document specifying each town and the number of businesses as shown above. Run the query and verify that you obtain output such as that shown in Figure 7.11 below.



```
bizDB> db.biz.aggregate([
...     { "$group" : {
...          "_id" : "$town",
...          "count" : { "$sum" : 1 } } }
... ]).pretty()
[
  { _id: 'Omagh', count: 9 },
  { _id: 'Banbridge', count: 12 },
  { _id: 'Belfast', count: 12 },
  { _id: 'Derry', count: 11 },
  { _id: 'Enniskillen', count: 11 },
  { _id: 'Ballymena', count: 9 },
  { _id: 'Lisburn', count: 7 },
  { _id: 'Coleraine', count: 9 },
  { _id: 'Ballymoney', count: 13 },
  { _id: 'Newry', count: 8 }
]
bizDB> 
```

Figure 7.11 Summary Report by Town

Try it now!	Produce the application ***profitable_by_town.py*** that first produces the number of businesses in each town as shown above and then displays the most profitable business in each town for each of the years for which data is available.

7.3 Location-based Queries

A useful feature of MongoDB is the support for queries on GeoJSON data. GeoJSON is an open standard format designed for representing simple geographical features such as points (addresses or locations), lines (streets, roads, etc.) and polygons (areas, counties, provinces countries, etc.). For example, the GeoJSON coordinate location for the Belfast campus of Ulster University might be expressed as

```
{ "location" :
        {
          "type" : "Point”,
          "coordinates" : [54.60388, -5.92937]
        }
}
```

where the **`coordinates`** list specifies the longitude and latitude values for the location as seen in Figure 7.12 below.

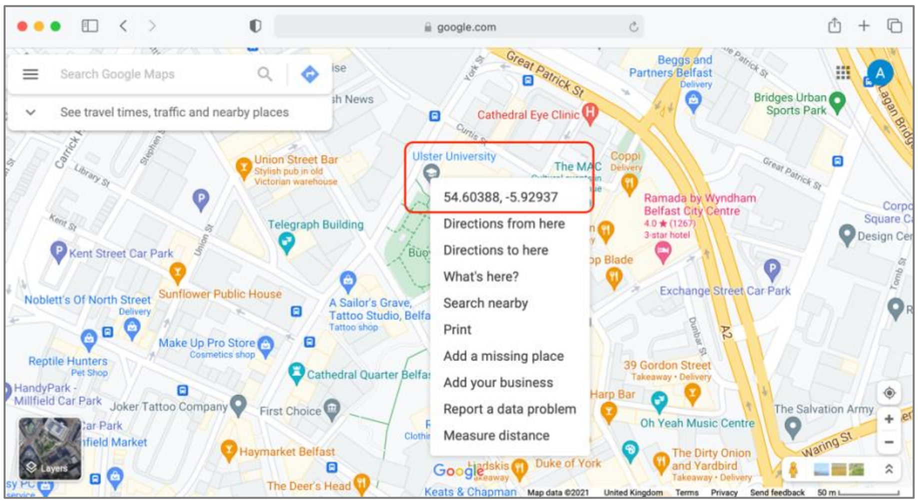


Figure 7.12 Longitude and Latitude

Where GeoJSON data is available, MongoDB provides a range of geospatial queries that allow us to query elements based on their physical location. As an example of this, we will provide a realistic location for each of our business objects and implement a query that retrieves the closest neighbours for any given business.

7.3.1 Preparing the Data

First, we need to generate coordinate data for each business. As our businesses are randomly assigned a location from one of a collection of 10 towns, we can specify a bounding box for each town and generate random longitude and latitude values within the relevant box. For example, the box for Coleraine might be specified as shown in Figure 7.13 with the lower left boundary at the point (55.10653864221481, -6.703013870894064) and the upper right boundary at the point (55.16083114339611, -6.640640630380869).

Note:	The order of coordinate values is always longitude followed by latitude. This can be confusing since it corresponds to y-direction (i.e. north-south) followed by x-direction (i.e. east-west) in a traditional coordinate system.

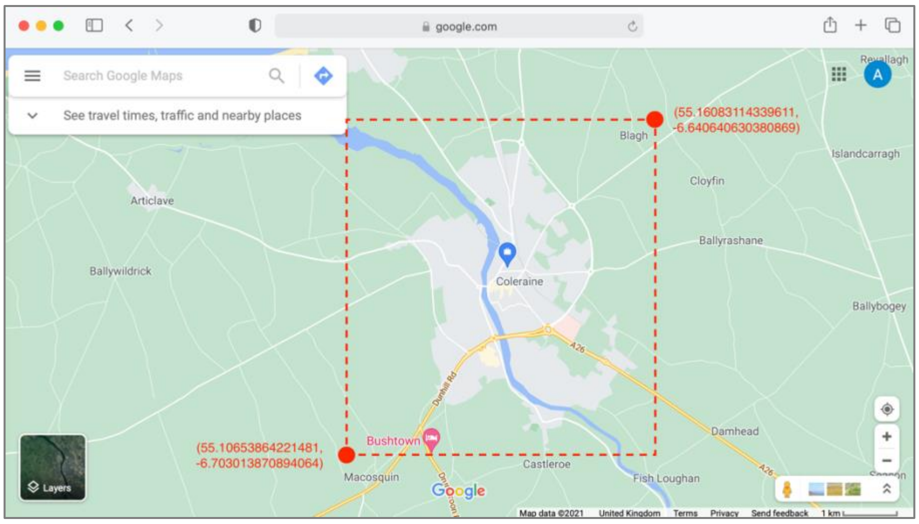


Figure 7.13 Bounding Box for Coleraine

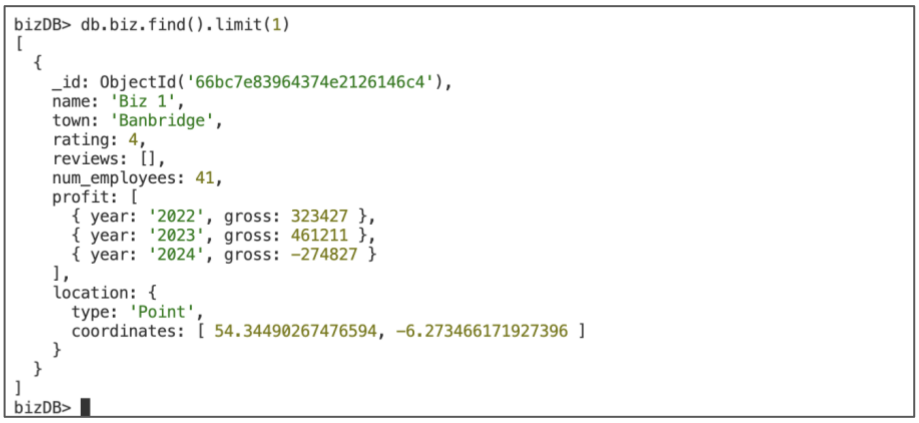
For each of the businesses located in Coleraine, we can then generate a value between the lower and upper boundaries of the longitude and latitude range for the Coleraine area and use a PyMongo **`update_one()`** query to **`$set`** a **`location`** field for the business to the randomly generated value. Repeating this for each of the towns represented in the dataset results in the code presented overleaf.

In ***add_location.py***, the bounding box for each town is represented as a list of 4 values (x1, y1, x2, y2) corresponding to the lower longitude value, lower latitude value, upper longitude value and upper latitude value. The random location for each business is calculated as the lower coordinate value (i.e. x1 or y2) plus a random percentage of the distance between the x and y values. The result of running ***add_location.py*** should be to add a **`location`** field to each business, in GeoJSON format, where the coordinate values are within the bounding box corresponding to that business's **`town`** field.

Do it now!	Run ***add_location.py*** (supplied in the ***BE07 Practical Files***) to add a **`location`** field to each business object. Verify that it has worked by using the MongoDB shell to retrieve the details of a business and make sure that you receive output such as that shown in Figure 7.14 below.

File: add_location.py

```
from pymongo import MongoClient
import random

locations = {
    "Coleraine" : [55.10653864221481, -6.703013870894064,
                   55.16083114339611, -6.640640630380869],
    "Banbridge" : [54.32805966474902, -6.29894073802459,
                   54.36914017698541, -6.238287009221747],
    # ... for every location used in the application
}

client = MongoClient("mongodb://127.0.0.1:27017")
db = client.bizDB
businesses = db.biz

for location in locations:
    for business in businesses.find( { "town" : location } ):
        rand_x = locations[location][0] +
                 ( (locations[location][2] -
                    locations[location][0]) *
                    (random.randint(0,100) / 100) )
        rand_y = locations[location][1] +
                 ( (locations[location][3] -
                    locations[location][1]) *
                   (random.randint(0,100) / 100) )
        businesses.update_one(
            { "_id" : business["_id"] },
            { "$set" :
                {
                    "location" :
                        {
                            "type" : "Point",
                            "coordinates" : [rand_x, rand_y]
                        }
                }
            }
        )
```



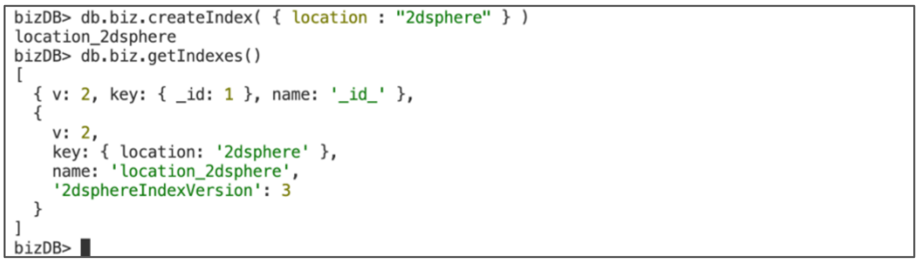
```
bizDB> db.biz.find().limit(1)
[
  {
    _id: ObjectId('66bc7e83964374e2126146c4'),
    name: 'Biz 1',
    town: 'Banbridge',
    rating: 4,
    reviews: [],
    num_employees: 41,
    profit: [
      { year: '2022', gross: 323427 },
      { year: '2023', gross: 461211 },
      { year: '2024', gross: -274827 }
    ],
    location: {
      type: 'Point',
      coordinates: [ 54.34490267476594, -6.273466171927396 ]
    }
  }
]
bizDB> █
```

Figure 7.14 Location Data Generated

The final stage before we can use the GeoJSON data is to construct an index that allows the data to be used in a geospatial query. MongoDB provides a **`2dsphere`** index that supports the calculation of geometries on an earth-like sphere. The index only needs to be created once and can be built by issuing the following command from the MongoDB console prompt.

```
db.biz.createIndex( { "location" : "2dsphere" } )
```

Do it now!	Create the **`2dsphere`** index on the **biz** collection by issuing the **`createIndex()`** command as shown above. You can verify that the index has been created by issuing the **`getIndexes()`** query before and after creating the index as shown by Figure 7.15 below.



```
bizDB> db.biz.createIndex( { location : "2dsphere" } )
location_2dsphere
bizDB> db.biz.getIndexes()
[
  { v: 2, key: { _id: 1 }, name: '_id_' },
  {
    v: 2,
    key: { location: '2dsphere' },
    name: 'location_2dsphere',
    '2dsphereIndexVersion': 3
  }
]
bizDB> █
```

Figure 7.15 Create the 2dsphere Index

7.3.2 Geospatial Queries

With the GeoJSON location data provided for each business and the 2dsphere index created, we can now query businesses based on their location. We will demonstrate this through an application ***neighbours.py*** that selects a business at random and then retrieves the 10 businesses that are located closest to the randomly selected one, ordered by increasing distance.

The MongoDB **`$geoNear`** operation is provided as a stage in the Aggregation Pipeline in the form

```
{ $geoNear :
        {
          { geoNear option 1,
            geoNear option 2,
            ...
            geoNear option n
        }
}
```

The demonstration in ***neighbours.py*** uses the following options:

`near`	The point on which to centre the search. Here, we assign it the GeoJSON location of the randomly selected business.
`maxDistance`	An optional parameter specifying the maximum distance in metres for which to return a result.
`minDistance`	An optional parameter specifying the minimum distance in metres for which to return a result. Here, we set it to 1 to avoid returning a business as a neighbour of itself.
`distanceField`	A string value that will be used as the field name for the element that returns the distance in metres from the search location.
`spherical`	Set to **`True`** to specify that the search should be performed using spherical geometry.

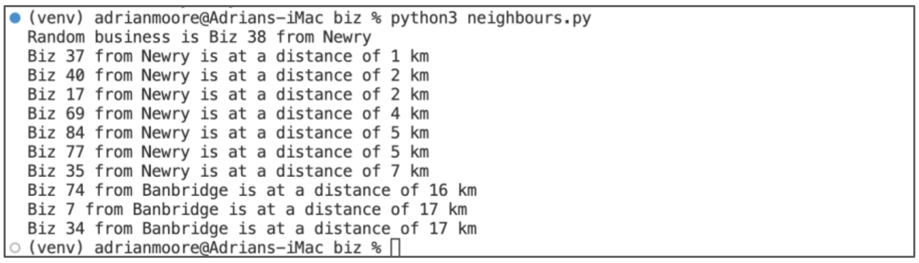
File: neighbours.py

```
from pymongo import MongoClient

client = MongoClient("mongodb://127.0.0.1:27017")
db = client.bizDB
businesses = db.biz

for business in businesses.aggregate( [
                { "$sample" : { "size": 1 } } ] ):
    print("Random business is " + business["name"] + \
          " from " + business["town"])

    for neighbour in businesses.aggregate( [
           { "$geoNear" :
               { "near" :
                  { "type" : "Point",
                     "coordinates" :
                        business["location"]["coordinates"] },
                 "maxDistance": 50000,
                 "minDistance" : 1,
                 "distanceField" : "distance",
                 "spherical" : True
               }
           },
           { "$project" : { "name" : 1, "town" : 1,
                            "distance" : 1 } },
           { "$limit" : 10 }
        ]):
        distance_km = str(round(neighbour["distance"] / 1000))
        print(neighbour["name"] + " from " +
              neighbour["town"] + " is at a distance of " +
              distance_km + "km")
```

Note: The list of **`$geoNear`** options shown here is not exhaustive. Other options are available and are described in the MongoDB manual linked from Section 7.4.

neighbours.py begins by selecting a random business from the **`biz`** collection. This is achieved by using the **`$sample`** stage of the Aggregation Pipeline, providing a **`size`** parameter that dictates the size of the sample to be randomly selected. Once the **`name`** and **`town`** values for the selected business have been output, the application then uses a 2nd **`aggregate()`** query to issue a **`$geoNear`** command to return businesses close to that which was randomly selected. The pipeline also includes a **`$project`** stage to determine that the fields required are the **`name`**, **`town`** and **`distance`** (the name of which was specified in the "**`distanceField`**" parameter of the **`$geoNear`** stage), and a **`$limit`** stage to retrieve only the 10 closest businesses. Finally, we convert the **`distance`** value returned to kilometres and print out the **`name`**, **`town`** and **`distance`** of each business returned.

Do it now!	Add the program ***neighbours.py*** (in the ***BE07 Practical Files***) to your ***biz*** project and run it. Verify that it returns (up to) 10 businesses which are listed in increasing order of distance from the initial random selection. Make sure that you receive output such as that shown in Figure 7.16 below, where the initial selection of a business located in Newry is followed by the remaining Newry-based businesses before offering the 3 closest businesses in the nearest town, Banbridge.



```
● (venv) adrianmoore@Adrians-iMac biz % python3 neighbours.py
  Random business is Biz 38 from Newry
  Biz 37 from Newry is at a distance of 1 km
  Biz 40 from Newry is at a distance of 2 km
  Biz 17 from Newry is at a distance of 2 km
  Biz 69 from Newry is at a distance of 4 km
  Biz 84 from Newry is at a distance of 5 km
  Biz 77 from Newry is at a distance of 5 km
  Biz 35 from Newry is at a distance of 7 km
  Biz 74 from Banbridge is at a distance of 16 km
  Biz 7 from Banbridge is at a distance of 17 km
  Biz 34 from Banbridge is at a distance of 17 km
○ (venv) adrianmoore@Adrians-iMac biz %
```

Figure 7.16 Geospacial Query Output

Try it now!	Modify ***neighbours.py*** so that it returns the closest business from a town that is not the same as that selected at random. For example, for the selection of Biz 35 in Newry shown in Figure 7.16 above, the result returned should be Biz 74 from Banbridge. Make sure that the modification works even if there are 10 or more businesses located in the same town.

7.4 Further Information

- https://docs.mongodb.com/manual/core/aggregation-pipeline/
  MongoDB Aggregation Pipeline manual
- https://docs.mongodb.com/manual/reference/operator/aggregation-pipeline/
  Aggregation Pipeline stages
- https://docs.mongodb.com/manual/reference/operator/aggregation/
  Aggregation Pipeline operators
- https://www.analyticsvidhya.com/blog/2021/04/how-to-create-an-aggregation-pipeline-in-mongodb/
  How to create an Aggregation Pipeline in MongoDB
- https://studio3t.com/knowledge-base/articles/mongodb-aggregation-framework/
  The Beginner's Guide to MongoDB Aggregation (with exercise)
- https://www.tutorialspoint.com/mongodb/mongodb_aggregation.htm
  MongoDB Aggregation – TutorialsPoint
- https://geojson.org/
  The GeoJSON Specification
- https://docs.mongodb.com/manual/reference/operator/aggregation/geoNear/
  $geoNear Manual
- https://thecodebarbarian.com/80-20-guide-to-mongodb-geospatial-queries
  The 80/20 Guide to MongoDB Geospatial Queries