Semi-Structured Data and Advanced Data Modelling

COURSEWORK 1 – MONGODB DESIGN AND IMPLEMENTATION

Group 23:
Raj Kadir, Oskar Lasota, Sadeq Rahman, Muhammad Hamza

Table of Contents

Assumptions

Here we will detail the assumptions we have made for our MongoDB database project which were made during the design process of the project.

General	
1.	Anything to do with time is based on a 24-hour clock cycle ranging from 0 to 23, 0 being 00:00 AM and 23 being 11:00PM
2.	All prices are in GBP

Employees	
1.	We assume that everybody works 6-hour shifts, there are 4 static shifts.
2.	We assume employment record does not contain the end date because all the employees are still employed, otherwise deleted off the system.

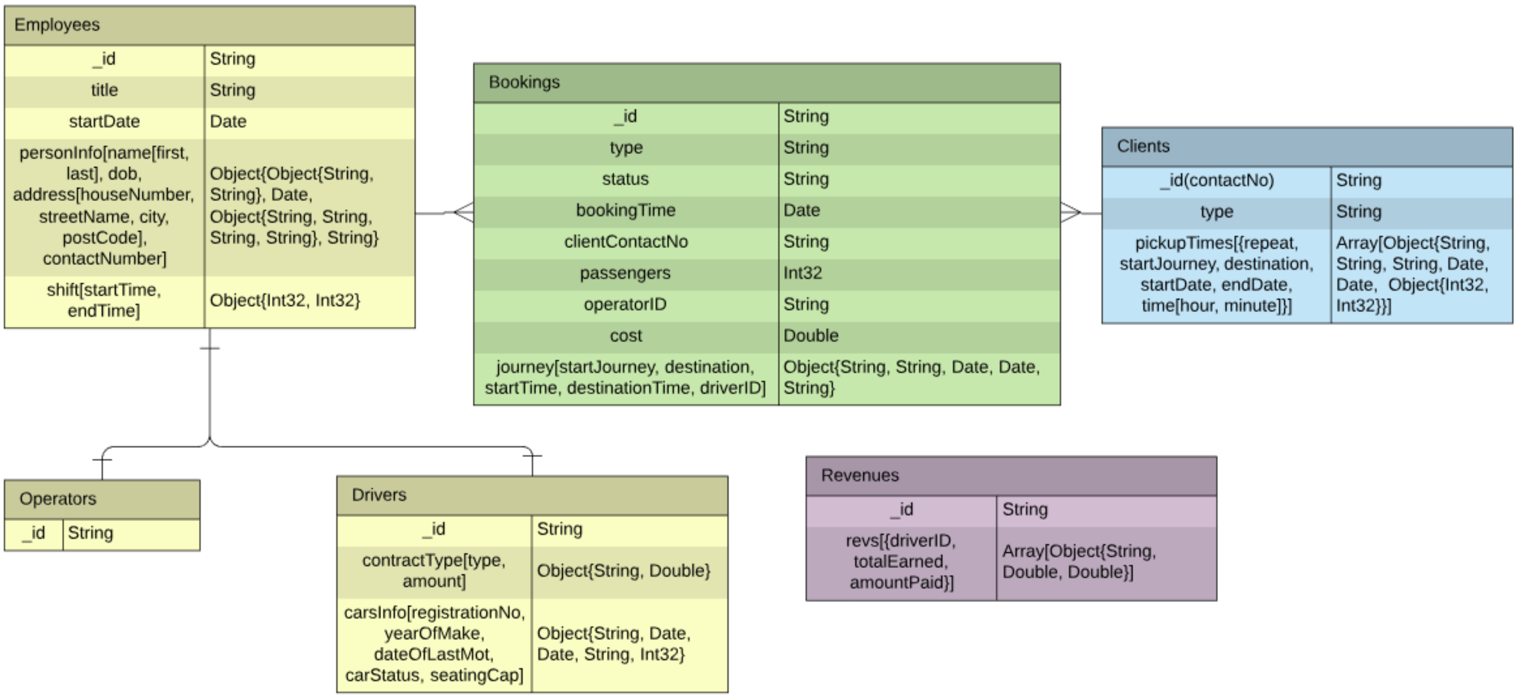
Drivers	
1.	Drivers are Employees
2.	We assume that one car is assigned to one driver.
3.	We assume that the driver owns his car.
4.	We assume that age of car is the ‘year of make’ of the car.

Operators	
1.	Operators are Employees
2.	We assume that the operator sets the cost of the journey.

Bookings	
1.	We assume the payments are made in cash in hand, instead of bank cards.
2.	We assume the payments are fixed and auto-calculated by the operator.
3.	We assume that only the operators working during their shift can make bookings.
4.	We assume all journeys are bookings made by operator, no picking up people from street.
5.	We assume that the operator obtains the number of passengers to match the car seating capacity.
6.	We assume that the driver has a system that records the start and destination of the journey.
7.	We assume that the driver is assigned by the Operator to each booking journey.

Clients	
1.	We assume that each client has one mobile number.
2.	We assume that one mobile number is unique.

ER/NoSQL Diagram



Employees	
_id	String
title	String
startDate	Date
personInfo[name[first, last], dob, address[houseNumber, streetName, city, postCode], contactNumber]	Object{Object{String, String}, Date, Object{String, String, String, String}, String}
shift[startTime, endTime]	Object{Int32, Int32}

Bookings	
_id	String
type	String
status	String
bookingTime	Date
clientContactNo	String
passengers	Int32
operatorID	String
cost	Double
journey[startJourney, destination, startTime, destinationTime, driverID]	Object{String, String, Date, Date, String}

Clients	
_id(contactNo)	String
type	String
pickupTimes[{repeat, startJourney, destination, startDate, endDate, time[hour, minute]}]	Array[Object{String, String, String, Date, Date, Object{Int32, Int32}}]

Operators	
_id	String

Drivers	
_id	String
contractType[type, amount]	Object{String, Double}
carsInfo[registrationNo, yearOfMake, dateOfLastMot, carStatus, seatingCap]	Object{String, Date, Date, String, Int32}

Revenues	
_id	String
revs[{driverID, totalEarned, amountPaid}]	Array[Object{String, Double, Double}]

Entities

Employees, Operators, Drivers, Bookings, Revenues, Clients

Relationships

Drivers **is a** 1 to 1 Employees

Operators **is a** 1 to 1 Employees

Clients **has** 1 to many Bookings

Drivers **has** 1 to many Bookings

Operators **has** 1 to many Bookings

Attributes

Employees	
Field Name	Comments
_id	This field contains a string value; a unique id for each driver/operator
title	This field is a string value
startDate	This field is of **date** type that stores the employees start date at the company
personInfo	This field also contains embedded documents as follows (in bold): **name**[*first*, *last*], dob, **address**[*houseNumber*, *streetName*, *city*, *postCode*, *contactNumber*]
shift	This field is an object with values : [*startTime*, *endTime*].

Drivers	
Field Name	Comments
_id	This field contains a string value; a unique id for each driver that will be the id specified in the Employees *_id* field name.
contractType	This field is a string value to state whether the driver is on a salary or percentage-of-receipts basis.
carsInfo	This field is an embedded document and contains the fields as follows: [registrstionNo, yearOfMake, dateOfLastMot, carStatus, seatingCap]

Operators	
Field Name	Comments
_id	This field contains a string value; a unique id for each operator that will be the id specified in the Employees *_id* field name.

Clients	
Field Name	Comments
_id	This field contains a string value; a unique id for each Client.
type	The type field contains a string value, the client can be either corporate or private.
pickupTimes	The pickupTimes field is an array of collections that consists of the following : [*repeat*, *startJourney*, *destination*, *startDate*, *endDate*, ***time***[*hour*, *minute*]] This field is only filled and updated when the client sets up a new frequency of their pickup.

Bookings	
Field Name	Comments
_id	This field contains a string value ; unique booking
type	This field contains a string value ; type of booking, meaning that it is either pre booked or standard order.
status	This field contains a string value ; this value represents if the booking has been completed/canceled/pending
bookingTime	This field contains a date object ; represents time of the booking
clientContactNo	This field contains a string value ; represents the client
passengers	This field contains an integer value ; represents the amount of passengers
operatorID	This field contains a string value ; represents the unique operator
cost	This field contains a decimal value ; represents the cost of the trip that is assumedly assigned by the operator.
journey	Journey would be stored as an embedded document. It stores the following fields : [*startJourney*, *destination*, *startTime*, *destinationTime*, *driverID*]

Revenues	
Field Name	Comments
_id	Unused by our system and is automatically generated by mongo.
revs	An array that stores the following fields : [*driverID*, *totalEarned*, *amountPaid*]

Design decisions

Employees: Inside this collection we have documents that represent the employees in the system/organisation. Employees are either a Driver or an Operator which is represented by the *title* tag. If an employee document has a *title* tag ‘Driver’ then there will be additional fields *carsInfo* and *contractType.* This was a decision that was influenced by having the option to reduce the amount of collections created, which would not only take memory unnecessarily but also complicate the creation of queries.

Drivers : We are classifying the driver in the Employee collection, as explained in the employee collection. Each driver would contain additional fields in the Employee collection which would be embedded documents and otherwise stored in a separate collection. Each driver contains information about their car, due to the assumption of ‘1 to 1’ relationship.

Operators : Operators will be classified in the Employee collection, as explained in the Employee collection. Compared to the driver, the operator will not contain extra fields the only difference will be that the employee id will represent the operator with a format ‘op001’.

Clients : The client collection is linked to the bookings collection in a ‘one to many’ relationship, we have decided to link these two collections using the client phone number, this will make it easier when dealing with queries and we assume that the phone number is unique enough. Clients will have a ‘PickupTime’ array of collections to set a certain pickup time if they choose to have a certain booking pattern or frequency, this will be in an array to allow multiple frequencies to be set.

Bookings : The booking collection is what is first created when the operator obtains data from the client through the phone. Each booking is linked to a Client by using the phone number which represents the primary key. This represents a many-to-one relationship, to store information about all bookings for each unique client. The booking also has a ‘many-to one’ relationship with Employees, as one employee can have many bookings assigned to themselves. We have also decided to store the journey information inside of bookings as each booking contains a journey instead of having it as a separate collection to make it easier to call queries and improve scalability.

Revenues: We assume that the revenue collection is updated client-side, which leads to having a stand-alone collection that is not connected to any other collection, we have decided this because this data will be static and does not need unnecessary relationships. We store an embedded document with all the drivers, meaning that when queries are ran, they will calculate the amount earned by looking at a unique driver. We assume that this table will be needed every certain amount of time, so it would use a query to update these values whenever necessary.

Queries

Query 1 - Total cost of bookings for a specific month/date

```
db.bookings.aggregate({
    $match: {
        'bookingTime': {
            $gte: ISODate("2018-00-01T00:00:00.000Z")
        },
        'bookingTime': {
            $lte: ISODate("2020-11-31T23:59:59.000Z")
        }
    }
}, {
    $group: {
        _id: null,
        totalBooking: {
            $sum: "$cost"
        }
    }
});
```

Result:

```
{
     "_id" : null,
     "totalBooking" : 258.1
}
```

This query calculates the total cost of bookings for a specified date. By the use of the *$group* aggregation we use the *$sum* accumulator operator which returns the sum of all numerical values stored in the field name *cost* (which is noted as *$cost* in the query above). By the use of the *$match* aggregation I filter through the documents to find documents that fit the parameters set by the use of the *$lte* and *$gte* operations.This query can be useful to the company to see if they are reaching specific company targets for booking costs in the past week or month, for example.

Query 2 - Find the top 10 most expensive bookings in the last 30 days and print them in order of most expensive to least

```
db.bookings.aggregate({
    $match: {
        bookingTime: {
            $gte: new Date(new
Date(ISODate().getTime() - 1000 * 60 * 60 * 24 * 30))
        }
    }
}, {
    $sort: {
        cost: -1
    }
}, {
    $limit: 10
});
```

Result:

```
{
        "_id" : "b010",
        "type" : "regular",
        "status" : "incomplete",
        "bookingTime" : ISODate("2019-10-
30T18:56:39.040Z"),
        "clientContactNo" : "07900000010",
        "passengers" : 6,
        "operatorID" : "op001",
        "cost" : NumberDecimal("54.0000000000000"),
        "journey" : {
                "startJourney" : "Arnos Grove",
                "destination" : "Holborn",
                "startTime" : ISODate("2020-01-
10T12:30:00Z"),
                "destinationTime" : ISODate("2020-01-
10T15:30:00Z"),
                "driverID" : "dr003"
        }
}
{
        "_id" : "b002",
        "type" : "regular",
        "status" : "incomplete",
        "bookingTime" : ISODate("2019-10-
30T18:56:36.208Z"),
        "clientContactNo" : "07900000002",
        "passengers" : 1,
        "operatorID" : "op002",
        "cost" : NumberDecimal("40.7300000000000"),
        "journey" : {
                "startJourney" : "Oxford",
                "destination" : "London",
                "startTime" : ISODate("2019-02-
09T05:30:00Z"),
                "destinationTime" : ISODate("2019-02-
09T07:30:00Z"),
                "driverID" : "dr001"
        }
}
{
        "_id" : "b005",
        "type" : "regular",
        "status" : "incomplete",
        "bookingTime" : ISODate("2019-10-
30T18:56:36.983Z"),
        "clientContactNo" : "07900000005",
        "passengers" : 3,
        "operatorID" : "op005",
        "cost" : NumberDecimal("36.1000000000000"),
        "journey" : {
                "startJourney" : "Stansted Airport",
                "destination" : "Birmingham",
                "startTime" : ISODate("2017-02-
10T01:30:00Z"),
                "destinationTime" : ISODate("2017-02-
10T05:30:00Z"),
                "driverID" : "dr001"
        }
}
.
.
.
```

This query retrieves the top 10 most expensive bookings in the last 30 days. By using the *$match* aggregation and *$gte* again it is used to filter all the documents. With the *new Date()* we can get the current time in milliseconds and then it is subtracted by the calculation *1000*60*60*24*30* (which is the milliseconds in 30 days). We use the *$sort* aggregation to sort all the documents that have been selected and returns them to the pipeline in the specified order. The sorting is specified by selecting the field *cost* to sort by and given the value *-1* to specify a descending order.

This query can be used by the company to find the details of the locations of most expensive trips. By looking at the pickup locations and destinations that customers are taking, the company can focus the advertising in those areas, for example. Targeting the areas that bring the most expensive trips will bring the company more profit.

Query 3 - (Following from the previous query) Specific booking id from the above query can then be used to find the trips that are most profitable by looking at the start/destination of the journeys.

```
db.bookings.find({
    "_id": "b006"
}, {
    "journey.startJourney": 1,
    "journey.destination": 1,
    _id: 0
}).pretty()
```

Result:

```
{
        "journey" : {
                "startJourney" : "Finsbury Park",
                "destination" : "Heathrow Airport"
        }
}
```

This query is used to print the *startJourney* and *destination* for specific bookings. The previous query gave us information of top expensive bookings. Taking the booking id we can use this query to get simpler view of the information we need.

Query 4 - Find booking information for a client which includes important information such as duration and car registration number

```
db.bookings.aggregate({
   $lookup: {
      from: "employees",
      localField: "journey.driverID",
      foreignField: "_id",
      as: "driverInfo"
   }
}, {
   $unwind: "$driverInfo"
}, {
   $lookup: {
      from: "employees",
      localField: "operatorID",
      foreignField: "_id",
      as: "operatorInfo"
   }
}, {
   $unwind: "$operatorInfo"
}, {
   $lookup: {
      from: "clients",
      localField: "clientContactNo",
      foreignField: "_id",
      as: "clientInfo"
   }
}, {
   $unwind: "$clientInfo"
}, {
   $project: {
      _id: 1,
      clientMobNo: "$clientInfo.clientContactNo",
      driverName: "$driverInfo.personInfo.name.first",
      operatorName:
"$operatorInfo.personInfo.name.first",
      departingFrom: "$journey.startJourney",
      departureTime: {
         $hour: "$journey.startTime"
      },
      departingTo: "$journey.destination",
      duration: {
         $toString: {
            $divide: [{
               $subtract: [
                  "$journey.destinationTime",
"$journey.startTime"
               ]
            }, 3600000]
         }
      },
      carRegNo: "$driverInfo.carsInfo.registrationNo",
      costOfJourney: {
         $toInt: "$cost"
      }
   }
}).pretty()
```

```
Result:
{
      "_id" : "b001",
      "departingFrom" : "123 Wakefield Road,
Stratford, London",
      "departureTime" : 7,
      "departingTo" : "Temple Station, London",
      "duration" : "1",
      "carRegNo" : "DB13 WEI",
      "costOfJourney" : 10
}
{
      "_id" : "b002",
      "departingFrom" : "Oxford",
      "departureTime" : 5,
      "departingTo" : "London",
      "duration" : "2",
      "carRegNo" : "CS15 NET",
      "costOfJourney" : 40
}
{
      "_id" : "b003",
      "departingFrom" : "Romford",
      "departureTime" : 21,
      "departingTo" : "Mile End",
      "duration" : "3",
      "carRegNo" : "PO1 TRI",
      "costOfJourney" : 10
}
{
      "_id" : "b004",
      "departingFrom" : "Romford",
      "departureTime" : 17,
      "departingTo" : "Mile End",
      "duration" : "0.5",
      "carRegNo" : "PO1 TRI",
      "costOfJourney" : 16
}
{
      "_id" : "b005",
      "departingFrom" : "Stansted Airport",
      "departureTime" : 1,
      "departingTo" : "Birmingham",
      "duration" : "4",
      "carRegNo" : "CS15 NET",
      "costOfJourney" : 36
}
{
      "_id" : "b006",
      "departingFrom" : "Finsbury Park",
      "departureTime" : 9,
      "departingTo" : "Heathrow Airport",
      "duration" : "2",
      "carRegNo" : "DB13 WEI",
      "costOfJourney" : 32
}
```

```
{
        "_id" : "b007",
        "departingFrom" : "Arnos Grove",
        "departureTime" : 8,
        "departingTo" : "Manchester",
        "duration" : "4.25",
        "carRegNo" : "DB13 WEI",
        "costOfJourney" : 27
}
{
        "_id" : "b008",
        "departingFrom" : "Finsbury Park",
        "departureTime" : 12,
        "departingTo" : "Alexandra Palace",
        "duration" : "-671",
        "carRegNo" : "PO1 TRI",
        "costOfJourney" : 25
}
{
        "_id" : "b009",
        "departingFrom" : "Wood Green",
        "departureTime" : 8,
        "departingTo" : "Tottenham",
        "duration" : "4",
        "carRegNo" : "DB13 WEI",
        "costOfJourney" : 5
}
{
        "_id" : "b010",
        "departingFrom" : "Arnos Grove",
        "departureTime" : 12,
        "departingTo" : "Holborn",
        "duration" : "3",
        "carRegNo" : "PO1 TRI",
        "costOfJourney" : 54
}
```

This query calculates booking information to display for all clients. Information including car’s registration number, departure location and time, estimated duration of journey, and cost of journey. This query is constructed because it displays all necessary information that would be provided to a specific client.
This query involves the use of $*lookup* which performs left outer join in the same database to an ‘unsharded’ collection; it matches the field of the joined collection with the field of input collection, then assigning to a new array named in the “as” field.
The *$unwind* operator deconstructs the array field from the input document and replaces each array field with document of the element of the array. Then, the *$project* operator passes the document to the next pipeline—with only the specified fields.

Query 5 - Given the client’s phone number find the client’s pick-up times.

```
db.clients.find({
    _id: "07900000001"
}, {
    "pickupTimes": 1,
    _id: 0
}).pretty();
```

Result:

```
{
        "pickupTimes" : [
                {
                        "repeat" : "Daily",
                        "startJourney" : "123 Wakefield Road,
Stratford, London",
                        "destination" : "Temple Station,
London",
                        "startDate" : ISODate("2019-08-
01T00:00:00Z"),
                    "endDate" : ISODate("2019-09-
01T00:00:00Z"),
                    "time" : {
                        "hour" : 8,
                        "minute" : 30
                    }
                }
            ]
}
```

Some customers have regular bookings and and those that do will have the relevant information stored inside an embedded document named *pickupTimes*. This query will print all the details about a customer's pickup times. It will retrieve all contents of the embedded document *pickupTimes* for that specific customer.

Query 6 - Given a driver's full name, find the total income earned from all bookings

```
var driverID = db.employees.find({
    "personInfo.name.First": "Jon",
    "personInfo.name.Last": "Targaryen"
}, {
    "_id": 1
}).next()._id

db.bookings.aggregate({
    $group: {
        _id: driverID,
        total: {
            $sum: '$cost'
        }
    }
}, {
    $project: {
        _id: 0,
        total: {
            $toString: {
                $cond: [{
                        $gte: [{
                            $subtract: ["$total", {
                                $floor: "$total"
                            }]
                        }, 0.5]
                    },
                    {
                        $ceil: "$total"
                    },
                    {
                        $floor: "$total"
                    }
                ]
            }
        }
    }
});
```

```
Result:
{ "total" : "258" }
```

This query finds the id by iterating through the documents and then assigns the driver id specific to the driver's name. The assigned variable containing the specified driver id can then be used to to retrieve the total costs of bookings through the aggregation operation using the *$group* operator which sums the cost (*$cost*) using the *$sum* operator. Then, the result is displayed using the *$project* operator which passes the document to display the rounded total.

Query 7 - Given a clients number, find all the bookings associated with it

```
db.bookings.find(
   {
      clientContactNo: "07900000001"
   }
).pretty();
```

```
Result:
{
        "_id" : "b001",
        "type" : "daily",
        "status" : "incomplete",
        "bookingTime" : ISODate("2019-10-
30T18:56:35.882Z"),
        "clientContactNo" : "07900000001",
        "passengers" : 1,
        "operatorID" : "op001",
        "cost" : NumberDecimal("10.2500000000000"),
        "journey" : {
                "startJourney" : "123 Wakefield Road,
Stratford, London",
                "destination" : "Temple Station, London",
                "startTime" : ISODate("2018-09-
12T07:30:00Z"),
                "destinationTime" : ISODate("2018-09-
12T08:30:00Z"),
                "driverID" : "dr002"
        }
}
```

Given a customer number, we use the above query to search the *bookings* collection and it will retrieve any bookings associated with that number.

Query 8 - Query to update a booking to 'complete' status

```
// Show before
db.bookings.find({
   "_id": "b003"
}).pretty();
// Do Update
db.bookings.update({
   "_id": "b003"
}, {
   $set: {
      "status": "Complete"
   }
});
// Show after
db.bookings.find({ "_id": "b003" }).pretty();
```

```
Before:
{
        "_id" : "b003",
        "type" : "regular",
        "status" : "incomplete",
        "bookingTime" : ISODate("2019-11-
01T02:26:22.238Z"),
        "clientContactNo" : "07900000003",
        "passengers" : 1,
        "operatorID" : "op003",
        "cost" : 10.5,
        "journey" : {
                "startJourney" : "Romford",
                "destination" : "Mile End",
                "startTime" : ISODate("2019-07-
02T21:30:00Z"),
                "destinationTime" : ISODate("2019-07-
03T00:45:00Z"),
                "driverID" : "dr004"
        }
}


Result after:
{
        "_id" : "b003",
        "type" : "regular",
        "status" : "incomplete",
        "bookingTime" : ISODate("2019-11-
01T02:26:22.238Z"),
        "clientContactNo" : "07900000003",
        "passengers" : 1,
        "operatorID" : "op003",
        "cost" : 10.5,
        "journey" : {
                "startJourney" : "Romford",
                "destination" : "Mile End",
                "startTime" : ISODate("2019-07-
02T21:30:00Z"),
                "destinationTime" : ISODate("2019-07-
03T00:45:00Z"),
                "driverID" : "dr004"
        }
}
```

This query uses the update method to find a particular booking via its _id value and change the value of status from ‘incomplete’ to ‘complete’

Query 9 - Updates a bookings journey destinationTime for example the driver finishes the journey earlier than expected/later

```
db.bookings.update(
    { "_id": "b003" },
    { $set: { "journey.destinationTime": new Date(2019,
6, 3, 1, 45) } });
db.bookings.find({ "_id": "b003" }).pretty();
```

Result:

```
{
        "_id" : "b003",
        "type" : "regular",
        "status" : "incomplete",
        "bookingTime" : ISODate("2019-11-
01T02:26:22.238Z"),
        "clientContactNo" : "07900000003",
        "passengers" : 1,
        "operatorID" : "op003",
        "cost" : 10.5,
        "journey" : {
                "startJourney" : "Romford",
                "destination" : "Mile End",
                "startTime" : ISODate("2019-07-
02T21:30:00Z"),
                "destinationTime" : ISODate("2019-07-
03T00:45:00Z"),
                "driverID" : "dr004"
        }
}
```

This query uses the update method to find a booking with the _id of b003 and then replace the destinationTime in the journey object with the new Date object that we pass to it, we then run the find query to see the changes to the destinationTime.

Query 10 - Find the average booking price

```
db.bookings.aggregate(
    [{
        $group: {
            _id: "null",
            'averageTravelCost': {
                $avg: "$cost"
            }
        }
    }]
```

Result:

```
25.810000000000002
```

```
).next().averageTravelCost
```

This query uses aggregate method to find average cost of booking by using the operator *$avg,* which is applied to all costs. Then, the method returns the value generated by the document, specifically ‘averageTravelCost’.

Query 11 - A client (07900000002) adds a daily booking every day at 10PM for the next 3 months starting from today's date from Chaplin Road, Wembley, London to Flood Walk, Chelsea, London

```
db.clients.find({ _id: "07900000002" }).pretty();
db.clients.update(
   { _id: "07900000002" },
   {
      $push: {
         pickupTimes: {
            repeat: "Daily",
            startJourney: "24 Chaplin Road, Wembley,
London",
            destination: "17 Flood Walk, Chelsea,
London",
            // YYYY-mm-ddTHH:MM:ss
            startDate: new Date('2019-10-
01T22:00:00'),
            endDate: new Date('2019-01-01T22:00:00'),
            time: {
               hour: NumberInt(22),
               minute: NumberInt(0)
            }
         }
      }
   }
);
db.clients.find({ _id: "07900000002" }).pretty();
```

Before:

```
{ "_id" : "07900000002", "type" : "Private" }
```

Result after:

```
{
        "_id" : "07900000002",
        "type" : "Private",
        "pickupTimes" : [
                {
                        "repeat" : "Daily",
                        "startJourney" : "24 Chaplin Road,
Wembley, London",
                        "destination" : "17 Flood Walk,
Chelsea, London",
                        "startDate" : ISODate("2019-10-
01T21:00:00Z"),
                        "endDate" : ISODate("2019-01-
01T22:00:00Z"),
                        "time" : {
                                "hour" : 22,
                                "minute" : 0
                        }
                }
        ]
}
```

This query uses the update method on a specific client to add an object to the pickupTimes array which is used to store a regular/frequent booking of a client, it does this by pushing the object (using the $push command) to the pickupTimes array.

Query 12 - Calculated revenue—total money earned by the company less total salary paid to employees

```
var bookingTotal = db.bookings.aggregate(
   [{
      $group: {
         _id: "null",
         "bookingsTotal": {
            $sum: "$cost"
         }
      }
   }]
).next().bookingsTotal

var salaryPercentage = db.employees.aggregate(
   [{
      $group: {
         _id: "null",
         "salaryTotal": {
            $sum: {
               $multiply: {
                  $cond: [{
```

Result:

```
166520.875
```

```
                        $eq: ["$contractType.type",
"percentage"]
                    }, "$contractType.amount",
bookingTotal]
                }
            }
        }
    }
  }]
).next().salaryTotal

var salaryFixed = db.employees.aggregate(
  [{
    $group: {
      _id: "null",
      "salaryTotal": {
        $sum: {
          $cond: [{
            $eq: ["$contractType.type", "fixed"]
          }, "$contractType.amount", 0]
        }
      }
    }
  }]
).next().salaryTotal

var totalRevenue = db.revenues.aggregate([{
  $group: {
    _id: "null",
    "sum": {
      $sum: {
        $sum: "$revs.totalEarned"
      }
    }
  }
}]).next().sum

print(totalRevenue - salaryPercentage + salaryFixed)
```

This query calculates the total revenue the taxi company has generated less the amount paid—to the employees, for maintenance, and other costs. Because there are two types of contract for drivers: fixed and receipt-percentage based. Two methods are needed to calculate the total salary paid to the drivers. To calculate the total salary that is percentage based, the total cost of booking is calculated using the *$sum* operator and then assign it to the variable ‘bookingTotal’. This variable is used in the method to calculate the percentage earnings of individual drivers—which are differentiated amongst the different contract types by using the *$cond* and *$eq* operators—and then adding them all using the *$sum* operator. However, before summation, the individual earnings are calculated using the *$multiply* operator. Similarly, the total earnings of the fixed contract types and total revenue are calculated by grouping them for the output and using *$sum* for the total. Then, lastly the total salaries are subtracted from totalRevenue to give the net profit.

Query 13 - This query finds all the journeys a particular car registration number has done in its lifetime

```
var driverRegID = db.employees.find({
    "carsInfo.registrationNo": "DB13 WEI",
    }, {
    "_id": 1
}).next()._id

db.bookings.find({
    "journey.driverID" : driverRegID
    }, {
    "journey.startJourney": 1,
    "journey.destination": 1,
    _id: 0
}).pretty();
```

Result:

```
{
        "journey" : {
                "startJourney" : "123 Wakefield Road,
Stratford, London",
                "destination" : "Temple Station, London"
        }
}
{
        "journey" : {
                "startJourney" : "Finsbury Park",
                "destination" : "Heathrow Airport"
        }
}
{
        "journey" : {
                "startJourney" : "Arnos Grove",
                "destination" : "Manchester"
        }
}
{
        "journey" : {
                "startJourney" : "Wood Green",
                "destination" : "Tottenham"
        }
}
```

This query finds the registration of a specific car and is stored inside the temporary variable *driverRegID*. Using this variable that has stored the vehicle registration we search the bookings table to print every journey the car has taken.

Query 14 - Deleting a driver that has left the company

```
db.employees.deleteOne(
    { _id : "dr009"}
);
```

Result:

```
{ "acknowledged" : true, "deletedCount" : 1 }
```

Assuming that a driver of _id: “dr009” exists, then using the delete method we can remove the associated document driver from the system, while keeping to our assumptions this could happen whenever the employee leaves the company.

Profile command

Before performing a profile command on the following queries, we created a new database to ensure the output results is as accurate as possible. Then, after insert all documents into their corresponding collections, we set profiling level by typing **db.setProfilingLevel(2)** into the mongo shell. Afterwards, after performing a query, we get the system profile by entering **db.system.profile.find()** into the mongo shell.

Query 4:

```
db.bookings.aggregate({
    $lookup: {
        from: "employees",
        localField: "journey.driverID",
        foreignField: "_id",
        as: "driverInfo"
    }
}, {
    $unwind: "$driverInfo"
}, {
    $lookup: {
        from: "employees",
        localField: "operatorID",
        foreignField: "_id",
        as: "operatorInfo"
    }
}, {
    $unwind: "$operatorInfo"
}, {
    $lookup: {
        from: "clients",
        localField: "clientContactNo",
        foreignField: "_id",
        as: "clientInfo"
    }
}, {
    $unwind: "$clientInfo"
}, {
    $project: {
        _id: 1,
        clientMobNo: "$clientInfo.clientContactNo",
        driverName: "$driverInfo.personInfo.name.first",
        operatorName:
"$operatorInfo.personInfo.name.first",
        departingFrom: "$journey.startJourney",
        departureTime: {
            $hour: "$journey.startTime"
        },
        departingTo: "$journey.destination",
        duration: {
            $toString: {
                $divide: [{
                    $subtract: [
                        "$journey.destinationTime",
"$journey.startTime"
                    ]
                }, 3600000]
            }
        },
        carRegNo: "$driverInfo.carsInfo.registrationNo",
```

```
{
        "op" : "command",
        "ns" : "taxi1.bookings",
        "command" : {
                "aggregate" : "bookings",
                "pipeline" : [
                        {
                                "$lookup" : {
                                        "from" : "employees",
                                        "localField" :
"journey.driverID",
                                        "foreignField" : "_id",
                                        "as" : "driverInfo"
                                }
                        },
                        {
                                "$unwind" : "$driverInfo"
                        },
                        {
                                "$lookup" : {
                                        "from" : "employees",
                                        "localField" : "operatorID",
                                        "foreignField" : "_id",
                                        "as" : "operatorInfo"
                                }
                        },
                        {
                                "$unwind" : "$operatorInfo"
                        },
                        {
                                "$lookup" : {
                                        "from" : "clients",
                                        "localField" :
"clientContactNo",
                                        "foreignField" : "_id",
                                        "as" : "clientInfo"
                                }
                        },
                        {
                                "$unwind" : "$clientInfo"
                        },
                        {
                                "$project" : {
                                        "_id" : 1,
                                        "clientMobNo" :
"$clientInfo.clientContactNo",
                                        "driverName" :
"$driverInfo.personInfo.name.first",
                                        "operatorName" :
"$operatorInfo.personInfo.name.first",
        costOfJourney: {
          $toInt: "$cost"
        }
    }
});
```

```
                                        "departingFrom" :
"$journey.startJourney",
                                        "departureTime" : {
                                                "$hour" :
"$journey.startTime"
                                        },
                                        "departingTo" :
"$journey.destination",
                                        "duration" : {
                                                "$toString" : {
                                                        "$divide" : [
                                                                {

"$subtract" : [

"$journey.destinationTime",

"$journey.startTime"
                                                                        ]
                                                                },
                                                                3600000
                                                        ]
                                                }
                                        },
                                        "carRegNo" :
"$driverInfo.carsInfo.registrationNo",
                                        "costOfJourney" : {
                                                "$toInt" : "$cost"
                                        }
                                }
                        }
                ],
                "cursor" : {

                },
                "lsid" : {
                        "id" : UUID("4f35aa69-66f7-472b-
9106-d212f0fff4dd")
                },
                "$db" : "taxi1"
        },
        "keysExamined" : 0,
        "docsExamined" : 10,
        "cursorExhausted" : true,
        "numYield" : 0,
        "nreturned" : 10,
        "queryHash" : "A300CFDE",
        "planCacheKey" : "A2B33459",
        "locks" : {
                "ReplicationStateTransition" : {
                        "acquireCount" : {
                                "w" : NumberLong(63)
                        }
                },
                "Global" : {
                        "acquireCount" : {
                                "r" : NumberLong(63)
                        }
                },
                "Database" : {
                        "acquireCount" : {
                                "r" : NumberLong(63)
                        }
                },
                "Collection" : {
                        "acquireCount" : {
```

```
				"r" : NumberLong(62)
			}
		},
		"Mutex" : {
			"acquireCount" : {
				"r" : NumberLong(63)
			}
		}
	},
	"flowControl" : {

	},
	"responseLength" : 1717,
	"protocol" : "op_msg",
	"millis" : 0,
	"planSummary" : "COLLSCAN",
	"ts" : ISODate("2019-10-31T17:56:55.195Z"),
	"client" : "127.0.0.1",
	"appName" : "MongoDB Shell",
	"allUsers" : [ ],
	"user" : ""
}
```

op and command
The output of this query shows executed information by the db.bookings.aggregate() query. It shows the type of operation executed is *command* operation, which is aggregate command, in this case, using the *bookings* collection.

keysExamined
The field *keysExamined* stores the number of index keys that are examined by MongoDB in order to execute the query. However, in this case the number of index keys were 0 because there were no indexes set.

docsExamined, nReturned and responseLength
The *docsExamined* field, however, stores a total of 10 *bookings* documents that were examined by MongoDB and the *nReturned* field shows that they were all returned with a *responseLength* of 1717 bytes.

Millis, Ts and planSummary
The *millis* is the execution time of a query which is 0 for this query because there were small number of documents that needed to be scanned and returned. The *planSummary* shows that MongoDB needed to perform a COLLSCAN which means collection scan. *Ts* is the timestamp of when the query is executed.[1]

[1] https://www.percona.com/blog/2018/09/06/mongodb-investigate-queries-with-explain-index-usage-part-2/

Query 10:

```
db.bookings.aggregate({
    $lookup: {
        from: "employees",
        localField: "journey.driverID",
        foreignField: "_id",
        as: "driverInfo"
    }
}, {
    $unwind: "$driverInfo"
}, {
    $lookup: {
        from: "employees",
        localField: "operatorID",
        foreignField: "_id",
        as: "operatorInfo"
    }
}, {
    $unwind: "$operatorInfo"
}, {
    $lookup: {
        from: "clients",
        localField: "clientContactNo",
        foreignField: "_id",
        as: "clientInfo"
    }
}, {
    $unwind: "$clientInfo"
}, {
    $project: {
        _id: 1,
        clientMobNo: "$clientInfo.clientContactNo",
        driverName:
"$driverInfo.personInfo.name.first",
        operatorName:
"$operatorInfo.personInfo.name.first",
        departingFrom: "$journey.startJourney",
        departureTime: {
            $hour: "$journey.startTime"
        },
        departingTo: "$journey.destination",
        duration: {
            $toString: {
                $divide: [{
                    $subtract: [
                        "$journey.destinationTime",
"$journey.startTime"
                    ]
                }, 3600000]
            }
        },
        carRegNo:
"$driverInfo.carsInfo.registrationNo",
        costOfJourney: {
            $toInt: "$cost"
        }
    }
});
```

```
{
        "op" : "command",
        "ns" : "ocean.bookings",
        "command" : {
                "aggregate" : "bookings",
                "pipeline" : [
                        {
                                "$group" : {
                                        "_id" : "null",
                                        "averageTravelCost" : {
                                                "$avg" : "$cost"
                                        }
                                }
                        }
                ],
                "cursor" : {

                },
                "lsid" : {
                        "id" : UUID("9e39c43f-494b-447c-9b3f-
481fb4fd9b58")
                },
                "$db" : "ocean"
        },
        "keysExamined" : 0,
        "docsExamined" : 10,
        "cursorExhausted" : true,
        "numYield" : 0,
        "nreturned" : 1,
        "locks" : {
                "ReplicationStateTransition" : {
                        "acquireCount" : {
                                "w" : NumberLong(2)
                        }
                },
                "Global" : {
                        "acquireCount" : {
                                "r" : NumberLong(2)
                        }
                },
                "Database" : {
                        "acquireCount" : {
                                "r" : NumberLong(2)
                        }
                },
                "Collection" : {
                        "acquireCount" : {
                                "r" : NumberLong(2)
                        }
                },
                "Mutex" : {
                        "acquireCount" : {
                                "r" : NumberLong(2)
                        }
                }
        },
        "flowControl" : {

        },
        "responseLength" : 152,
        "protocol" : "op_msg",
        "millis" : 0,
```

	"planSummary" : "COLLSCAN", "ts" : ISODate("2019-10-31T21:41:02.415Z"), "client" : "127.0.0.1", "appName" : "MongoDB Shell", "allUsers" : [ ], "user" : "" }

op and command
Just as the previous output for query 4, profiling query 10 shows the results for executing db.bookings.aggregate() where we executed the aggregate command for the *bookings* collection.

keysExamined
Query 10, just like query 4, also shows that the field *keysExamined* is 0 as there were no indexes set.

docsExamined, nReturned and responseLength
MongoDB again examined 10 *bookings* documents and the *nReturned* field shows that they were all returned with a *responseLength* of 152 bytes.

Millis, Ts and planSummary
Similarly, query 10 also had an execution time of 0 as there was a very small number of documents that were scanned and returned. Again, COLLSCAN was performed by MongoDB for this query profiling.[2]

[2] https://docs.mongodb.com/manual/reference/database-profiler/

Explain command

Query 6:

```
Var exp = db.bookings.explain("executionStats")

exp.aggregate({
    $group: {
        _id: driverID,
        total: {
            $sum: '$cost'
        }
    }
}, {
    $project: {
        _id: 0,
        total: {
            $toString: {
                $cond: [{
                        $gte: [{
                            $subtract: ["$total", {
                                $floor: "$total"
                            }]
                        }, 0.5]
                    },
                    {
                        $ceil: "$total"
                    },
                    {
                        $floor: "$total"
                    }
                ]
            }
        }
    }
});
```

```
{
    "stages": [{
            "$cursor": {
                "query": {

                },
                "fields": {
                    "cost": 1,
                    "_id": 0
                },
                "queryPlanner": {
                    "plannerVersion": 1,
                    "namespace": "new.bookings",
                    "indexFilterSet": false,
                    "parsedQuery": {

                    },
                    "queryHash": "8B3D4AB8",
                    "planCacheKey": "8B3D4AB8",
                    "winningPlan": {
                        "stage": "COLLSCAN",
                        "direction": "forward"
                    },
                    "rejectedPlans": []
                },
                "executionStats": {
                    "executionSuccess": true,
                    "nReturned": 10,
                    "executionTimeMillis": 0,
                    "totalKeysExamined": 0,
                    "totalDocsExamined": 10,
                    "executionStages": {
                        "stage": "COLLSCAN",
                        "nReturned": 10,
                        "executionTimeMillisEstimate": 0,
                        "works": 12,
                        "advanced": 10,
                        "needTime": 1,
                        "needYield": 0,
                        "saveState": 1,
                        "restoreState": 1,
                        "isEOF": 1,
                        "direction": "forward",
                        "docsExamined": 10
                    }
                }
            }
        },
        {
            "$group": {
                "_id": {
                    "$const": "dr002"
                },
                "total": {
                    "$sum": "$cost"
                }
            }
        },
        {
```

```
      "$project": {
        "_id": false,
        "total": {
          "$convert": {
            "input": {
              "$cond": [{
                  "$gte": [{
                      "$subtract": [
                        "$total",
                        {
                          "$floor": [
                            "$total"
                          ]
                        }
                      ]
                    },
                    {
                      "$const": 0.5
                    }
                  ]
                },
                {
                  "$ceil": [
                    "$total"
                  ]
                },
                {
                  "$floor": [
                    "$total"
                  ]
                }
              ]
            },
            "to": {
              "$const": "string"
            }
          }
        }
      }
    }
  ],
  "ok": 1
}
```

queryPlanner.winningPlan.stage = “COLLSCAN”
The *queryPlanner* document provides default mode information regarding the winning plan, which includes whether a COLLSCAN (collection scan) is needed, which in this case. This means that a collection scan is needed to be done by MongoDB—which is done without an index.

queryPlanner.winningPlan.rejectedPlans = []
This is empty because there aren’t any rejected plans. There is no other execution plan when the query is executed with COLLSCAN because of not having indexes in the collection, except _id.

executionStats.nReturned = 10
The total number of documents that were returned from this query are 10.

executionStats.totalDocsExamined = 10
It is 10 because the total number of documents in the bookings collection is 10. The reason the output was expected to be the same is because the query utilises COLLSCAN.

executionStats.executionTimeMillis = 0
The query took a mere 0 seconds to execute. However, this is also due to the small number of collections

Query 5:

```
db.clients.find({
    _id: "07900000001"
}, {
    "pickupTimes": 1,
    _id: 0
}).explain("executionStats");
```

```
"queryPlanner" : {
        "plannerVersion" : 1,
        "namespace" : "ocean.clients",
        "indexFilterSet" : false,
        "parsedQuery" : {
                "_id" : {
                        "$eq" : "07900000001"
                }
        },
        "winningPlan" : {
                "stage" : "PROJECTION_SIMPLE",
                "transformBy" : {
                        "pickUpTimes" : 1,
                        "_id" : 0
                },
                "inputStage" : {
                        "stage" : "IDHACK"
                }
        },
        "rejectedPlans" : [ ]
},
"executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 1,
        "executionTimeMillis" : 0,
        "totalKeysExamined" : 1,
        "totalDocsExamined" : 1,
        "executionStages" : {
                "stage" : "PROJECTION_SIMPLE",
                "nReturned" : 1,
                "executionTimeMillisEstimate" : 0,
                "works" : 2,
                "advanced" : 1,
                "needTime" : 0,
                "needYield" : 0,
                "saveState" : 0,
                "restoreState" : 0,
                "isEOF" : 1,
                "transformBy" : {
                        "pickUpTimes" : 1,
                        "_id" : 0
                },
                "inputStage" : {
                        "stage" : "IDHACK",
                        "nReturned" : 1,
                        "executionTimeMillisEstimate" : 0,
                        "works" : 2,
                        "advanced" : 1,
                        "needTime" : 0,
                        "needYield" : 0,
                        "saveState" : 0,
                        "restoreState" : 0,
                        "isEOF" : 1,
                        "keysExamined" : 1,
                        "docsExamined" : 1
                }
        }
}
```

queryPlanner.winningPlan.stage = "PROJECTION_SIMPLE"
Unlike COLSCAN where we don't have the index given, projection_simple is able to immediately get a reference to the document which is expected since we have given it enough data to single out a document i.e using the unique clients id of "07900000001"

queryPlanner.winningPlan.rejectedPlans = []
The rejected plans array is empty. When the query is executed with PROJECTION_SIMPLE the only execution plan is the winning plan. We don't have any indexes in the collection, apart from _id, so there is only one execution plan.

executionStats.nReturned = 1
The number of documents returned is 1

executionStats.totalDocsExamined = 1
The number of documents examined is 1 this is expected because the query uses PROJECTION_SIMPLE, the low number shows that it is optimized

executionStats.executionTimeMillis = 0
The execution time of the query. It's just 0 millisecond, keep in mind it only had to search through 1 document, so it is expected to be this fast.

How can we improve the query?
The query is already quite optimized as it is, in order to further optimize this, we would probably have to work with the C++ source code of mongodb itself

Validation

Collection validation - Employees, Clients, Bookings, Revenues (see **setup.js** for the full scripts of validations on other collections)

```
db.createCollection("bookings", {
    validator: {
        $jsonSchema: {
            bsonType: "object",
            required: [
                "_id",
                "type",
                "status",
                "bookingTime",
                "clientContactNo",
                "passengers",
                "operatorID",
                "cost",
                "journey"
            ],
            properties: {
                _id: {
                    bsonType: "string",
                    description: "must be a string and is required"
                },
                type: {
                    bsonType: "string",
                    description: "must be a string and is required"
                },
                status: {
                    bsonType: "string",
                    description: "must be a string and is required"
                },
                bookingTime: {
                    bsonType: "date",
                    description: "must be a date and is required"
                },
                clientContactNo: {
                    bsonType: "string",
                    description: "must be a string and is required"
                },
                passengers: {
                    bsonType: "int",
                    description: "must be an int and is required"
                },
                operatorID: {
                    enum: [
                        "op001",
                        "op002",
                        "op003",
                        "op004",
                        "op005",
                        "op006",
                        "op007",
                        "op008"
                    ],
                    description: "must be one of the enum values and is required"
                },
                cost: {
                    bsonType: "double",
                    description: "must be a double and is required"
                },
                journey: {
                    bsonType: "object",
                    required: [
                        "startJourney",
                        "destination",
                        "startTime",
                        "destinationTime",
                        "driverID"
                    ],
                    properties: {
                        startJourney: {
                            bsonType: "string",
                            description: "must be a string and is required"
                        },
                        destination: {
                            bsonType: "string",
                            description: "must be a string and is required"
                        },
                        startTime: {
                            bsonType: "date",
                            description: "must be a date and is required"
                        },
                        destinationTime: {
                            bsonType: "date",
                            description: "must be a date and is required"
                        },
                        driverID: {
                            enum: [
                                "dr001",
                                "dr002",
                                "dr003",
                                "dr004"
                            ],
                            description: "must be one of the enum values and is required"
                        }
                    }
                }
            }
        }
    },
    validationLevel: "strict",
    validationAction: "error"
});
```

[Note: This code is only for the validation of the collection, bookings. Validation code for the rest of the collections is available in the setup.js file]

Invalid documents
The above code shows validation on one of the collections: bookings, which ensures that no invalid booking document can be inserted into the database.

Required values and their data types
How it does this is ensuring that booking have their required values and once the required values are included also check their data types to match the correct ones.

Foreign keys
We also simulated a foreign key scenario here by having the driverID and operatorID an enum of the values that currently exist in the database (at the moment hard coded ids) to ensure that bookings do not have a driverID/operatorID that doesn’t exist in the database.

Settings
We then applied the validationLevel of strict and validationAction of error to ensure that no document can be inserted if they do not match the validation requirements, we could have set this to warning to allow the insert but give warnings however this would stop the system from keeping its integrity of valid documents

Triggers

Scheduled Trigger - Generate a daily report on bookings (see **generateDailyReportsTrigger.js** for full script)

```
exports = function() {
   // Instantiate MongoDB collection handles
   const mongodb = context.services.get("Cluster0"); // Cluster name on atlas - CHANGE ME to match yours
   const bookings = mongodb.db("taxi").collection("bookings"); // taxi database, bookings collection - CHANGE
ME to match yours
   const reports = mongodb.db("store").collection("reports"); // store database, reports collection - CHANGE
ME to match yours


   // Get an aggregate that looks like this { todaysBookings: "1", date: 2019-...} and insert it into the reports
collection
   // Generate the daily report
   return bookings.aggregate([
   // Only report on bookings placed since yesterday morning
   {
      $match: {
         bookingTime: {
            $gte: getYesterdayMorningDate(),
            $lt: getThisMorningDate()
         }
      }
   },
      {
         $count: "todaysBookings"
      },
      {
         $addFields: { "date" : new Date()}
      }
   ]).next()
   .then(dailyReport => {
      reports.insertOne(dailyReport)
      .then(result => console.log(`Successfully inserted item with _id: ${result.insertedId}`))
      .catch(err => console.error(`Failed to insert item: ${err}`))
   })
   .catch(err => console.error("Failed to generate report:", err));
};

function getThisMorningDate() {
   …..
}

function getYesterdayMorningDate() {
   ….
}

function getMorningDate(date) {
   ….
}
```

The above code is a trigger that can be ran onto the database on a scheduled time. We used the advanced mode on the atlas cluster and placed this code to be ran every 24 hours, it will return us every 24 hours the amount of bookings that occured within the time frame set and the current date of when the trigger was called.	Result: { "todaysBookings" : 1, "date" : ISODate("2019-11-01T01:43:31.538Z") }

Database trigger - Send an SMS to the driver whenever a new booking is inserted into the database (see **textDriverUpdateTrigger.js** for full script)

```
exports = function(dbEvent) {
    // Only run if this event is for a newly inserted document
    if(dbEvent.operationType !== "INSERT") { return }

    // Format a message with the inserted document's `_id` and `journey` field.
    const { journey } = dbEvent.fullDocument;

    // Create the message to send to the driver via text message
    const msg = `New trip, pickup location is : ${journey.startJourney} and the destination is:
${journey.destination}`;

    // Get the Twilio messaging service number and the number that
    // should receive the SMS from stored global values.
    const fromPhone = context.values.get("twilioPhoneNumber");

    // store driverID into a variable
    const driverID = journey.driverID;

    // Get the drivers phone number from the driverID, this booking has been assigned to
    const toPhone = getDriversNumber(driverID);

    // Get the twilio service
    const twilio = context.services.get("myTwilio");

    // Send the SMS Message
    twilio.send({ To: toPhone, From: fromPhone, Body: msg });
};


function getDriversNumber(driverID){

    const mongodb = context.services.get("Cluster0"); // Cluster name on atlas - CHANGE ME to match yours
    const employees = mongodb.db("taxi").collection("employees"); // taxi database, employees collection -
CHANGE ME to match yours

    var number =
    employees.findOne({
        _id: driverID
    }, {
        "personInfo.contactNumber": 1
    }).personInfo.contactNumber;

    return number;
}
```

The above code is applied whenever an insert occurs onto the bookings collection where it creates a message based off the journey object. We then grab the drivers phone number through the driverID that is stored on the journey and pass the values to the twilio api, which should in practise send an SMS message to the driver about any new journeys that has been assigned to them by the operator.	Expected result: SMS: “New trip, pickup location is Aldgate East station, London and the destination is Mile End Hospital, London”

Extras

Programmatically generating - over 10,000 employee drivers documents (see **insertsProgrammatically.js** for the full script)

```
for(i = 0;  i <= 10000; i++){
    db.employees.insert({
        _id: "dr" + i,
        title: "Driver",
        startDate: randomDate(new Date('2010-01-9'), new Date('2019-01-01')),
        personInfo : {
            name: {
                first: randomizeMe(rgfirstNames),
                last: randomizeMe(rglastNames),
            },
            dob: randomDate(new Date('1960-11-9'), new Date('1990-01-01')),
            address: {
                houseNumber: randomNumber(1, 200),
                streetName: randomizeStreet(),
                city: randomizeMe(rgCity),
                postCode : randomPostCodeNumber()
            },
            contactNumber: randomPhoneNumber()
        },
        shift: {
            startTime: 0,
            endTime: 6
        },
        contractType: {
            type: "fixed",
            amount: randomNumber(18000, 35000)
        },
        carsInfo:{
            registrationNo: randomLicenceNumber(),
            yearOfMake: randomDate(new Date('1990-11-9'), new Date()),
            dateOfLastMot: randomDate(new Date('2016-11-9'), new Date()),
            carStatus: randomizeMe(rgCarStatus),
            seatingCap: randomizeMe(rgSeatCaps)
        }
    });
}
```

The above code is only a snippet of the code that can generate 10,000 driver employees into the database while keeping the database validation in check. See the insertsProgramatically.js file for the full version.

Expected result:
Inserts of 10,000 employees, all drivers with random variables in the fields and passing the validation test

Programmatically generating - over 100,000 bookings documents (see **insertsProgrammatically.js** for the full script)

```
for(i = 0; i < 100000; i++){
    db.bookings.insert({
        _id: "b" + i,
        type: "regular",
        status: randomizeMe(rgBookingStatus),
        bookingTime: randomDate(new Date('1990-01-01'), new Date()),
        clientContactNo: randomPhoneNumber(),
        passengers: NumberInt(1),
        operatorID: randomizeMe(rgRandomOperator),
        cost: randomNumber(10, 100),
        journey: {
            startJourney: randomizeStreet(),
            destination: randomizeStreet(),
            startTime: randomDate(new Date('1990-01-01'), new Date()),
            destinationTime: randomDate(new Date('1990-01-01'), new Date()),
            driverID: randomizeMe(rgRandomDriver)
        }
    });
}
```

The above code again is only a snippet of the main iteration that will generate over 100,000 booking documents programmatically, see the insertsProgramatically.js file for more information

Expected result:
Insertion of 100,000 bookings, all with random data populated and passing the validation test

Conclusion – including summary of MongoDB experience compared to known relational database design.

Throughout this entire process of setting up the database with queries for the Taxi firm, we have used a nosql database and have experienced how it differentiates from a relational database, in this section we will talk about the difficulties we have encountered and the advantages and downsides compared to a relational database.

Starting off with the advantages that really stood out when we used mongodb, the one that stood out the most was its schema-less models. Although this is not always the safest feature as data could end up inserted incorrectly. MongoDB allows schema-less models which means that documents in a collection are not required to have fixed and predetermined set of fields; similar fields are not required to have the same types of data inserted. Each document can have a different structure befitting to the way a particular application uses the data. Even if this may not be pertinent to an application, the flexibility that MongoDB offers is useful for most situations.

Another benefit of using mongodb is that MongoDB is horizontally scalable, allowing 'sharding' which is a method whereby the system dataset is distributed across multiple machines, whereas relational databases are only vertically scalable meaning that it enlarges the capacity of a particular server, increasing the RAM. The dividing of the workload provides better efficiency making MongoDB much faster than a single server with more RAM and capacity.[3]

One of the biggest differences between the two different database systems is that MongoDB allows for embedded subdocuments and arrays which we heavily used in our design and we believe that this is a huge advantage of using mongodb. Denormalizing our data like this heavily reduces the number of collections required in the system. This is useful as it allows for less queries needing to be made, information is more easily retrieved. Whereas in a relational design there is no use of embedded documents, information is more spread out as they're kept in separate tables and queries are made via joins. There are no joins in MongoDB, but MongoDB's queries allow javascript functions to be used inside of them to gift a lot flexibility.[4]

For example, in our *Employees* document, we denormalise our data and insert embedded documents such as *personInfo*, *address*, *shift*, *contractType*, *carsInfo*. Whereas in a relational database, we would have created separate tables such as Driver and Operator. For example, a Driver table in a relational database would have contained the fields that are held by the embedded documents, *contractType* and *carsInfo,* separate from an Employees table. So, in a relational system queries would have to make use of joins to retrieve data from these different tables.

Now we shall speak about the downsides of using mongodb in comparison to our experience of a relational database. When creating the collections and populating them, having no foreign key was error-prone when linking objects that had relationships whereas in a

[3] https://docs.mongodb.com/manual/sharding/
[4] https://docs.mongodb.com/manual/core/databases-and-collections/

relational database that wouldn't be a problem and tables could be linked together with a foreign key being forced to be inserted.

Another difficulty that we have encountered during the development stage was the learning curve, it was very difficult to adapt to the nosql model after being accustomed to the relational database model. The design stage of the database had to be approached differently, as there was no foreign keys and each object had an automated object id. We were also surprised by the possibility of having embedded documents as well as arrays in our collections, as this reduced the amount of collections that we had to create, especially when considering one-to-one relationships.

Overall, this experience of designing and implementing a database system for the taxi firm has educated us a lot about nosql databases and made us open our minds on all of the advantages specified above and many more. The steep learning curve is worth the hustle when implementing a system that is dealing with high amounts of data as our mongodb queries had really good performance numbers when ran on highly populated collections. Moreover, having the option of using embedded documents gave us many options and views on how we can design our system in many different ways.