ECS650U/ECS789P - SEMI-STRUCTURED DATA AND ADVANCED DATA MODELLING

**CW2: MongoDB Database Development and Performance Tuning**

Group 32

**Assumptions:**

During the development of our system, we made several assumptions. I’ve listed the assumptions we made for each collection below.

**General**

* We created our own custom ids for each collection to facilitate testing relating to the correctness of various queries.
* We deleted the pilot’s collection (mentioned in cw description) and created 1 collection for all AirlineEmployees
* All entries will be one month

**JourneyBooking**

* Additional passengers will be the number of passengers rather than each passengers’ detailed information
* Paying Passenger is the person who made the booking, not included in “addtlPassengers”
* BookingPrice will be recorded in £GBP.

**PlaneFlights**

* Departure airport can’t be the same as arrival airport.

**AirlineEmployees**

* Salaries will be recorded as an annual salary in £GBP.
* Employees will have a home address consisting of postcode, street name, house number and city
* Employees will have contact info consisting of an email address and mobile number.
* We assume the airline employs pilots, clerks, maintenance staff and cabin staff.
* Only AirlineEmployees with the position “Pilot” will have a datetime for fitDate.

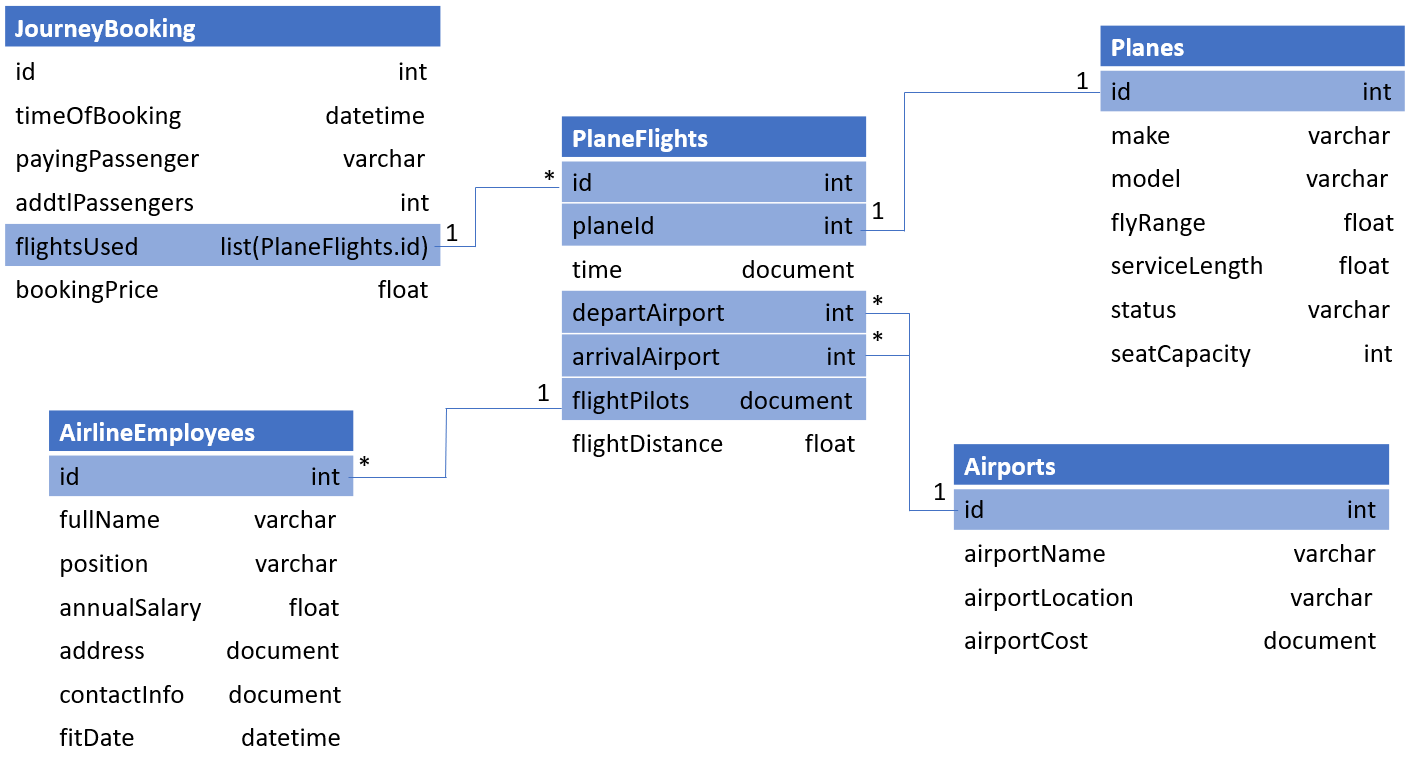
**Planes**

* serviceLength will be recorded in hours e.g., 5.5hours would be 5hours and 30minutes of service.
* Planes will be serviced every 400-600 hours.
* flyRange will be recorded in kilometres.
* Seat capacity includes flight crew.

**Airports**

* Airport costs will be recorded in £GBP hourly.
* Airport locations will be city names.
* We will calculate the monthly airport cost using the hourly cost, which we will then use in our revenue calculation.

**Diagram:**



**Collections:**

(Load “setup.js” for collection creation and document entries.)

JourneyBooking has a one-to-many relationship with PlaneFlights. This means that one journey will have many flights, as it might be a booking with several transitions. The booking passenger will be saved by him/herself and can buy more than one ticked at once which will be saved to “addtlPassengers”. “flightsUsed” is where all the transitions and flights used in the booking will be stored. The last field would be the booking price which is the total earnings of the airline which will be used to calculate the revenue.

**JourneyBooking**

* id int
* timeOfBooking datetime
* payingPassenger varchar
* addtlPassenger int
* flightsUsed [planeFlights\_id]
* bookingPrice float

AirlineEmployees will have a many-to-one relationship with PlaneFlights, this is because there are two employees that will be chosen for PlaneFlights (Pilot and Co-Pilot). Each Employee will have to include their detailed personal information. All employees will have an address and contactInfo which are nested documents. AnnualSalary is found in this collection which will be used to solve for revenue. Although every employee will have fit-to-fly as field, but it will be null unless the employee is in a pilot position.

**AirlineEmployees**

* id int
* fullName varchar
* position varchar
* annualSalary float
* address document
  + postcode varchar
  + streetName varchar
  + houseNo int
  + city varchar
* contactInfo document
  + email varchar
  + mobileNo varchar
* fitDate datetime

Planes have a one-to-one relationship with PlaneFlights, meaning one plane can have one flight. Planes will include the details of the plane, and its maintainability. It includes the serviceLength that will be a float, it increases per use and will be checked before each use to make sure it does not go over the allowed flight hours before it needs servicing, which will be every 400-600 flight hours (standard servicing length). ServiceLength will also be reset to 0 after each service. It has a status field so the employees know where or if the plane is available at all times.

**Planes**

* id int
* make varchar
* model varchar
* flyRange float
* serviceLength float
* status varchar
* seatCapacity int

Airports will have two one-to-many relationships. This is because one airport is connected to many flights as its starting airport and its destination airport. The collection will include the Airport information including the cost of utilizing the airport which is stored per hour, and can be multiplied by the number of hours it is at the airport per month to calculate the revenue.

**Airports**

* id int
* airportName varchar
* airportLocation varchar
* airportCost document
  + parkPrice float
  + refuelPrice float

PlaneFlights is connected to all the tables created, all the relationships can be seen above. The flights will have a planeId that is the primary key of the plane used for transportation, it will have the starting and destination airports which will be the ids of the airport collection. Pilots will be chosen from the AirlineEmployee collection.

**PlaneFlights**

* id int
* planeId int referencing plane collection id
* time document
  + depTime dateTime
  + arrTime dateTIme
* startAirport int referencing airport collection id
* destAirport int referencing airport collection id
* flightPilots document
  + pilot int referencing pilot collection id
  + co-pilot int referencing pilot collection id
* flightDistance float

**Queries:**

1. This query prints out our Airports collection. This is useful as it shows us various important details about each airport, such as the location and costs associated with parking and refueling.

> db.Airports.find();

Expected output: 10 documents

2. This query finds how many planes are working and outputs their total number. This is useful for knowing how many planes are available to be used for flights.

> db.Planes.find({status:"working"}).count();

Expected output: 5

3. This query is used to find the highest paid employee. This is useful as it helps the airline company keep track of costs and helps identify where there may be salary discrepancies amongst employees.

> db.AirlineEmployees.find().sort({annualSalary:-1}).limit(1)

Expected output: 1 document with details about “Kelvin Waters”.

4. This query is used to identify planes that have had a service length of over 400hours. This is useful to airlines as planes that have over 400 hours of service will need to undergo servicing to ensure they are still suitable for safe flight.

> db.Planes.find({

serviceLength : {

$gt:400

}

})

Expected output: 2 documents (1x Boeing 737-800, 1x Airbus A330-300)

5. This query identifies the pilot that has been the main pilot the most times and then prints details about them. This is useful because it helps the airline balance the workload amongst the pool of available pilots.

> db.PlaneFlights.aggregate([

{

$group :{

\_id : "$flightPilots.pilot",

countPilots : {

$count :{}

}

}

},

{$sort : {countPilots: -1}},

{$limit : 1},

{

$lookup :{

from: "AirlineEmployees",

localField: "\_id",

foreignField: "id",

as:"employee"

}

},

{$unwind: "$employee"},

{

$project: {

\_id :0,

"employee" : 1

}

}

])

Expected Output: 1 document with details about the pilot “Stuart Cash”.

6. This query identifies how many times each qualified pilot has been the co-pilot for a flight. This helps to ensure that all responsibilities relating to the flight are distributed amongst the pilots evenly.

> db.PlaneFlights.aggregate([

{

$group :{

\_id : "$flightPilots.co-pilot",

countPilots : {

$count :{}

}

}

},

{

$lookup :{

from: "AirlineEmployees",

localField: "\_id",

foreignField: "id",

as:"employee"

}

},

{$unwind: "$employee"},

{

$project: {

\_id :0,

"employee.fullName" : 1,

"countPilots" : 1

}

}

]);

7. This query identifies the pilot that has worked the most hours as the main pilot. This is important to keep track of as it helps to identify which pilots will need the most breaks, ensuring they remain fit to fly.

> db.AirlineEmployees.aggregate([

{

$project: {

\_id:0,

"id":1,

"fullName": 1,

"position" : 1

}

},

{

$match: {"position": "pilot"}

},

{

$lookup: {from: "PlaneFlights", localField: "id", foreignField: "flightPilots.pilot", as: "flights"}

},

{

$project: {

fullName: 1,

hoursWorked: {

$reduce: {

input: {

$map: {

input: "$flights",

as: "f",

in: {

$divide: [

{ $subtract: [{$toDate: "$$f.time.arrivTime"}, {$toDate: "$$f.time.depTime"}] },

3600000

]

}

}

},

initialValue: 0,

in: {

$add: ["$$value", "$$this"]

}

}

}

}

},

{

$sort: { hoursWorked: -1 }

},

{

$limit: 1

}

]);

Expected output: “Patrik Jacobs, 48.75 hours worked”

8. This query outputs the details for the pilot who’s fit to fly date is coming up next. This is useful to the airline as it allows them to be aware of when pilots will need to renew their fit to fly certificate, ensuring the safety of passengers and staff on their flights.

> db.AirlineEmployees.find({"fitDate": {$exists : true}}).sort({fitDate:1}).limit(1);

Expected output: “Roy Wheatley, date: 2021-01-10"

9. This query outputs the list of paying passengers that have been to Rome. This is useful to the airline as we can then target these passengers with various promotions relating to Rome such as hotel booking discounts to encourage them to visit again.

>db.JourneyBooking.aggregate([

{

$project :{

\_id :0,

"payingPassenger" :1,

"flightsUsed" : 1

}

},

{ $unwind : "$flightsUsed"},

{

$lookup: {

from: "PlaneFlights",

localField: "flightsUsed",

foreignField: "id",

as: "flights"

}

},

{ $unwind : "$flights"},

{

$project :{

"payingPassenger" :1,

"flights.startAirport" : 1,

"flights.destAirport" : 1

}

},

{

$lookup: {

from: "Airports",

localField: "flights.startAirport",

foreignField: "id",

as: "startAirportsNames"

}

},

{

$lookup: {

from: "Airports",

localField: "flights.destAirport",

foreignField: "id",

as: "destAirportsNames"

}

},

{

$project :{

"payingPassenger" :1,

"allAirportNames" : { $concatArrays : ["$startAirportsNames", "$destAirportsNames"]}

}

},

{ $match : { "allAirportNames.airportLocation" : "Rome"}},

{$group : {\_id : "$payingPassenger"}},

{

$project :{

\_id: 0,

"beenToRome" : "$\_id"

}

}

]);

Expected output: “Been to Rome: Jerome Kithinji, Rolly Beya, Nathan Flowers, Arthur Lerev”

10. This query identifies the airport that is the most expensive in terms of total cost (the sum of parking price and refuelling price). This information is useful as it helps the airline set appropriate prices for journeys travelling to and from expensive airports.

>db.Airports.aggregate([

{

"$project": {

\_id : 0,

"id" :1,

"airportName" : 1,

"totalCost" : {

"$add" : ["$airportCost.parkPrice", "$airportCost.refuelPrice"]

}

}

},

{$sort: { result : -1 } },

{$limit: 1}

])

Expected output: “Heathrow Airport, total cost: £550.60”

11. This query calculates the total revenue made or lost by the airline. This is useful as it helps the airline identify whether or not the business is profitable, allowing them to take scale up or scale down operations as appropriate.

>db.JourneyBooking.aggregate([

{ $unwind: "$flightsUsed"},

{

$lookup: {

from: "PlaneFlights",

localField: "flightsUsed",

foreignField: "id",

as: "flights"

}

},

{$addFields:{airportS:"$flights.startAirport"}},

{$addFields:{airportD:"$flights.destAirport"}},

{ $unwind: "$airportS"},

{ $unwind: "$airportD"},

{

$project :{

\_id: 0,

"bookingPrice" :1,

"airportS" : 1,

"airportD" : 1

}

},

{

$lookup: {

from: "Airports",

localField: "airportS",

foreignField: "id",

as: "airp"

}

},

{ $unwind: "$airp"},

{

$project :{

\_id: 0,

"bookingPrice" :1,

"airportS" : 1,

"airportD" : 1,

calcCS : {

$add : [ "$airp.airportCost.refuelPrice", { $multiply: ["$airp.airportCost.parkPrice", 2]}]

}

}

},

{

$lookup: {

from: "Airports",

localField: "airportD",

foreignField: "id",

as: "airp"

}

},

{ $unwind: "$airp"},

{

$project :{

\_id: 0,

"bookingPrice" :1,

"airportS" : 1,

"airportD" : 1,

calcCS : 1,

calcCD : {

$add : [ "$airp.airportCost.refuelPrice", { $multiply: ["$airp.airportCost.parkPrice", 2]}]

}

}

},

{

$group : {

\_id : "$bookingPrice",

totalCost : { $sum: { $add: ["$calcCS", "$calcCD"]} }

}

},

{

$group : {

\_id : null,

totalCost : { $sum: { $subtract: ["$\_id", "$totalCost"]} }

}

},

{

$project :{

\_id: 0,

"Revenue" : "$totalCost",

}

}

])

Expected Output: Revenue: £56256.50

12. This query lists all the airports a specific paying passenger will visit on their journey. This allows the airline to provide them with information specifically related to the airports they are travelling to such as COVID guidelines to be followed.

>db.JourneyBooking.aggregate([

{ $match : {"payingPassenger": "Jack Mach"}},

{ $unwind : "$flightsUsed"},

{

$lookup : {

from: "PlaneFlights",

localField: "flightsUsed",

foreignField: "id",

as: "flights"

}

},

{ $unwind : "$flights"},

{

$project :{

\_id :0,

"payingPassenger" :1,

"flights.startAirport":1,

"flights.destAirport": 1

}

},

{

$lookup : {

from: "Airports",

localField: "flights.startAirport",

foreignField: "id",

as: "startAirport"

}

},

{ $unwind : "$startAirport"},

{$addFields: {startAirportName: "$startAirport.airportName"}},

{

$project :{

\_id :0,

"payingPassenger" :1,

"flights.destAirport": 1,

"startAirportName": 1

}

},

{

$lookup : {

from: "Airports",

localField: "flights.destAirport",

foreignField: "id",

as: "destAirport"

}

},

{ $unwind : "$destAirport"},

{$addFields: {destinationAirportName: "$destAirport.airportName"}},

{

$project :{

\_id :0,

"payingPassenger" :1,

"destinationAirportName": 1,

"startAirportName": 1

}

}

]);

Expected Output: Jack Mach(Starting: Heathrow Airport , Destination: Charles de Gaulle Airport)

**Explainer:**

Explainer is used to retrieve information on query plans and its statistics. We will be executing the explainer on two queries out of the twelve that were generated for the project. To see the difference of the explainer being executed in different queries, we chose one with a search function (find) and an aggregate function. Using such a small sample, all the queries were executed very quickly (below 10ms), so there will not be much difference on whether or not we are using the index. All of the explainer outputs will be saved on the “Explainer\_outputs.js” file.

**Query 4 Explainer**

The we executed the executionStat explainer on a search query first, the output will contain the statistics winning plan information on the completed query. By executing the explainer on a non-indexed query 4, it shows that the found two documents that match the condition provided. It also shows that the query was executed in 0ms, which it means that there might not be room for improvement, but we expected this since the sample data is so small. Since there are no entries yet, totalkeysExamined were 0 and the total documents examined were 8. There were 8 documents examined because that is the total documents in the collection Planes, and it is a COLLSCAN stage which means that it scans the whole collection no matter what. After creating an index for collection Planes on the field serviceLength, we expected the time not to change, but we were mistaken as it takes longer than it did before. The addition of index made the query run in 6ms, which is very slow for the amount of documents that we have, everything improved including the documents examined that changed from eight to 2, the reason it could be slowing down is because in this case it is trying to change the index instantly to the document and it takes longer that if it was already doing it from the document itself using COLLSCAN.

**Query 12 Explainer**

After executing the explainer executionStat on query 12 unindexed, which is now an aggregate function, we can see from the output that it takes 2ms to run. Since it’s still unindexed it is using COLLSCAN again, because it needs to scan through all the documents in the collection. This query is more complex than query 4 and the increase time needed to execute the query shows it. Since this is an aggregate function, we could only put indexes on its sort or match pipeline stage. We indexed the collection JourneyBooking on the payingPassenger field, because that is the field we are matching in the query. We again expect the time needed to execute the query to decrease since we are indexing it (even though it was not the case in the previous query). This time, it did make the query execute faster, as it went from needing 2ms to 1ms, it might not seem like a big change, but the difference would be much more noticeable if we had a much larger data pool. We know that it was covered by the index because the results are retrieved using FETCH and the data is scanned using IXSCAN rather than it all being the COLLSCAN.

**Profiler:**

Profiler is used to gather and analyse the details of queries’ operations and their performance. We will be executing the profiler on six queries out of the twelve that we created. This will give us a valuable insight into the performance of different types of queries such as find() and aggregate(). All of the profiler outputs will be saved in the “Profiler\_outputs.js” file.

**Query 1 profiler output:**

The operation is of type query and the target collection is Airports found in the test database. One positive is the query took zero milliseconds to execute meaning it was quite fast. Another positive is that the numYield is zero, indicating that the operation was not interrupted by any read. One negative is that the planSummary is a COLLSCAN with all ten documents being examined. This is a negative due to it being a potentially very expensive operation if the number of documents in the collection were to be high. No keys were examined due to the lack of complexity in the query meaning there was no index read.

**Query 2 profiler output:**

This operation is of type command with the target collection being Planes in the test database. This query was also quite fast as it took zero milliseconds to execute. Yet again, the numYield was zero, indicating that the operation wasn’t interrupted by read. There were zero keys examined, indicating that the system is performant.

**Query 3 profiler output:**

This operation is of type query and the target collection is AirlineEmployees in the test database. Similarly, the query took 0 seconds to execute, with 0 keys being examined yet again. This indicates that the system is performant.

**Query 5 profiler output:**

This operation is of type command and the target collection is PlaneFlights in the test database. It has similar characteristics to the previous queries.

**Query 6 profiler output:**

This operation is of type command and the target collection is PlaneFlights in the test database. It has similar characteristics to the previous queries.

**Query 7 profiler output:**

This operation is of type command and the target collection is AirlineEmployees in the test database. Unlike the previous queries, this query took one millisecond to execute due to the COLLSCAN going over fifty documents, a large amount compared to the previous queries.