# Coursework 1

CW1\_MSc\_9

# Deliverable A:

# Assumptions:

**Employee Collection:**

* 1. The primary key in this document is the EmployeeID.
  2. We have assumed 5 possible job positions for the employees, denoting which position each employee holds with a field called Roles. The 5 possible roles are: booking clerk, cabin crew, maintenance staff, pilot and captain. The last two denote the difference between the main pilot of a plane and the co-pilot (first officer).
  3. There is a sub-document named as stated as PILOT in the employee document which includes a few other assumptions such as:
     1. There is a license number for each pilot which is a unique 10-digit number, but it is not the primary key.

**Plane Collection:**

* 1. The primary key in this document is the tailnumber.
  2. Each plane has a unique tail number.
  3. There are 3 models of each plane:
     1. A320
     2. A380
     3. Boeing747
  4. There are 3 possible statuses of any airplane:
     1. In Maintenance: The plane is being repaired.
     2. In Use: Is completing a journey.
     3. In Transit: Available for use.
  5. We assume that the seating capacity and the tank capacity are the same for each plane of the same model.

**PlaneFlight Collection:**

* 1. The primary key in this document is the FlightNumber.
  2. We assumed that our company has flights to and from 10 airports across Europe, both direct and indirect.
  3. Flights to the same destination on different dates, need not have the same departure time, however, will take approximately the same time to reach their destination.
  4. Keeping a track of code of each airport along with date and time of arrival/departure details of the plane flights.
  5. The arrival Date-Time of each flight is automatically calculated using a query which is based on estimated duration and departed Date-Time of each flight.
  6. The distance and estimated duration of each flight are real approximations.
  7. Seating cost per passenger is pre-calculated and put in the document manually.
  8. Operational costs include the airport costs and salary bills and it is pre-calculated and stored in the database manually.

**Airport Collection:**

* 1. The primary key in this document is the Airport\_Code.
  2. Each airport has a unique airport code.
  3. Each airport has a fixed hourly wait charge in euros.
  4. Refueling cost per liter is fixed for each airport, in euros.

**Passenger Collection:**

* 1. The primary key in this collection is the email address of the passenger.
  2. All passengers in all the bookings, whether owners of the bookings or not, will be included in this document.

**Flight\_Booking Collection:**

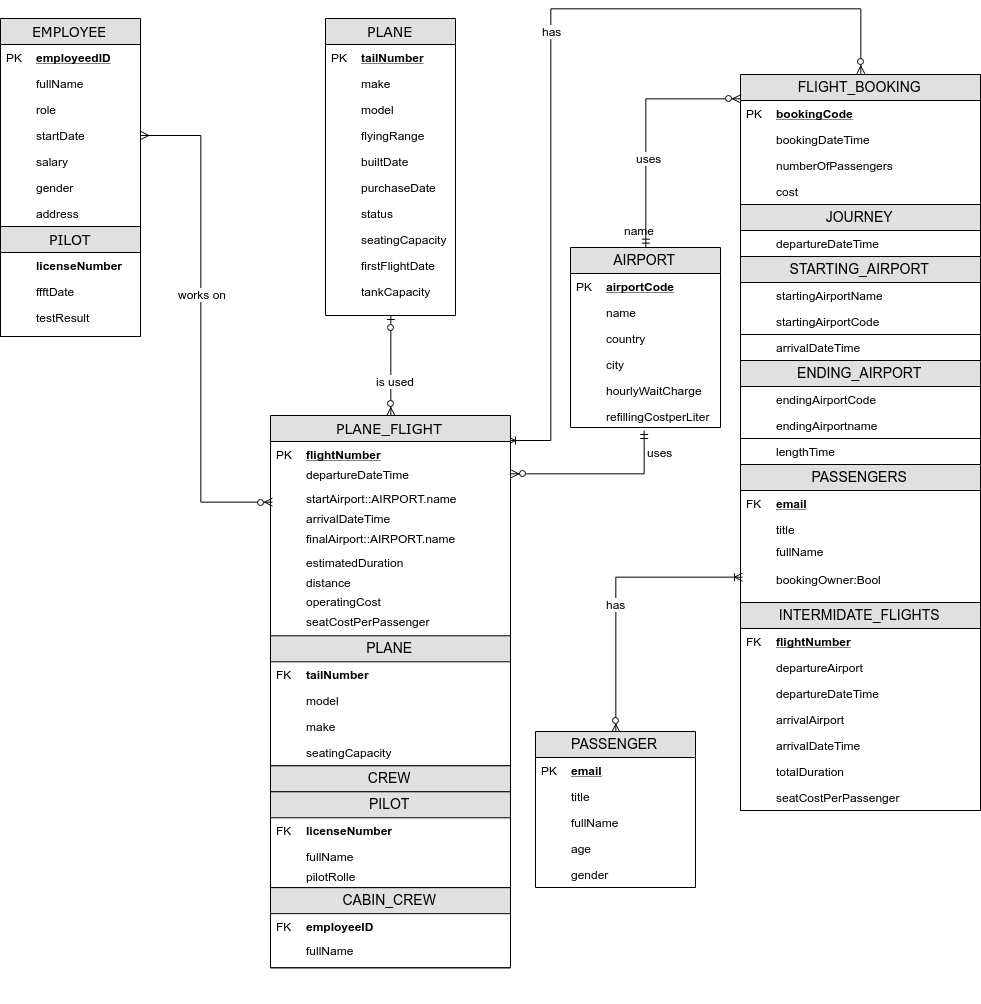
* 1. Each booking has a unique value called bookingCode which is the primary key.
  2. Each booking can be for many passengers, but 1 passenger will be the owner of the booking.
  3. Each booking can have one departure airport and one arrival airport but many intermediate flights in between.
  4. The cost of each booking will be the number of passengers, times the cost for each flight in the booking and will be automatically calculated.

**Revenue Collection:**

* 1. We have assumed that the revenue calculated for a specific period of time needs to be stored in a table.
  2. The total revenue is the money from bookings – operational costs – salary bill
  3. We have assumed that the revenue can be calculated for between any 2 dates. For simplicity reasons however, we have calculated the revenue for 3 months, which are the 3 months for which we also have flight and bookings.

# **Deliverable B:**

ER Diagram:



Can also be found in the delivered folder as ER\_diagram.png

# **Deliverable C:**

Database Dictionary

Employee Collection:

|  |  |  |
| --- | --- | --- |
| Field | Data Type | Comment |
| EmployeeID | String | Primary Key |
| FullName | String | Stores full name of employee |
| Role | String | Stores designation of employee |
| StartDate | Date | Stores the joining date of the employee |
| Salary | Integer | Stores the pay-scale of the employee |
| Gender | String | Stores gender of the employee. |
| Address | String | Stores the address of the employee |
| **pilotDetails** | Array | Optional. Exists only for the pilot and captain roles and contains the pilot details |
| pilotDetails.PilotLicenseNumber | String | 10-digit unique value |
| pilotDetails.ffftDate | Date | Stores the date when the pilot took the fit for flying test |
| pilotDetails.TestResult | String | Stores the result of the test |

Plane Collection:

|  |  |  |
| --- | --- | --- |
| Field | Data Type | Comment |
| TailNumber | Integer | Primary Key |
| Make | String | The make of the plane. Either “Boeing” or “Airbus” |
| Model | String | Stores the model number of the plane |
| FlyingRange | Integer | Stores the distance the plane can cover in kms. |
| BuiltDate | Date | Stores the date when the aircraft was built |
| PurchaseDate | Date | Stores the date the aircraft was purchased |
| Status | String | Stores the current condition of the aircraft |
| SeatingCapacity | Integer | Stores the seating capacity of the aircraft |
| FirstFlightDate | Date | Stores the date, the plane first took flights |
| TankCapacity | Integer | Stores the capacity of the tank. (in litres) |

Airport Collection:

|  |  |  |
| --- | --- | --- |
| Field | Data Type | Comment |
| AirportCode | String | Primary Key |
| Name | String | Stores the name of the airport |
| Country | String | Stores the country the airport is in |
| City | String | Stores the city of the airport |
| HourlyWaitCharge | Integer | Stores the waiting cost the airport charges for the aircraft |
| RefuelingCostperLitre | Integer | Stores the cost of refueling the tank |

Passenger Collection:

|  |  |  |
| --- | --- | --- |
| Field | Data Type | Comment |
| Email | String | Primary Key |
| Title | String | Stores the title of the passenger |
| FullName | String | Stores the full name of the passenger |
| Age | Integer | Stores the age of the passenger |
| Gender | String | Stores the gender of the passenger |

Plane\_Flight Collection:

|  |  |  |
| --- | --- | --- |
| Field | Data Type | Comment |
| FlightNumber | Integer | Primary Key |
| DepartureDateTime | Date | Stores the date and time of departure of planes |
| StartAirport | Object | Object to keep the starting airport details |
| StartAirport.Code | String | The unique code of the airport |
| StartAirpot.Name | String | Stores the name of the starting airport, from where the journey starts |
| FinalAirport | Object | Object to keep the final airport details |
| FinalAirport.Code | String | Unique code of the airport |
| FinalAirport.Name | String | Name of the airport |
| ArrivalDateTime | Date | Stores the date and time of arrival of flight |
| EstimatedDuration | Integer | Stores the estimated duration of the journey |
| EndAirport.Code | String | The unique code of the airport |
| Distance | Integer | Stores the distance of the entire journey in kms. |
| OperatingCost | Integer | Stores the cost of operation of flights in euros |
| SeatCostPerPassenger | Integer | Stores the cost of each seating |
| **PlaneDetails** | Array | Contains the plane details for reference with this table |
| PlaneDetails.tailNumber | String | Foreign Key |
| PlaneDetails.model | String | Stores the model of the aircraft |
| PlaneDetails.make | String | Stored if the plane is an airbus or boeing |
| PlaneDetails.seatingCapacity | Integer | Stores the seating capacity of the aircraft |
| **Crew** | Array | Contain details of different types of staff |
| **Crew.Pilots** | Array | Contains the details of pilots for reference. |
| Crew.Pilots.LicenseNumber | String | Foreign Key |
| Crew.Pilots.FullName | String | Stores the full name of the pilot |
| Crew.Pilots.PilotRole | String | Stores the details of the role of the pilot: Captain or First Officer |
| **Crew.CabinCrew** | Array | Stores the details of the cabin crew |
| Crew.CabinCrew.EmployeeID | String | Foreign Key |
| Crew.CabinCrew.FullName | String | Stores full name of the cabin crew |

Flight\_Booking Collection:

|  |  |  |
| --- | --- | --- |
| Field | Data Type | Comment |
| Booking Code | Integer | Primary Key |
| BookingDateTime | Date | Stores the date and time of when the booking was made |
| NumberOfPassengers | Integer | Stores the number of passengers in one flight |
| Cost | Integer | Stores the cost of each booking |
| **Journey** | Array | Stores the details of the journey made by the aircraft |
| Journey.DepartureDateTime | Date | Stores the time and date of departure |
| ArrivalDateTime | Date | Stores the arrival date and time of the flight |
| **Starting\_Airport** | Array | Stores the details of the starting airport |
| Starting\_Airport.Name | String | Stores the name of the starting airport of the journey |
| Starting\_Airport.tCode | String | Stores the code the airport from where the journey starts |
| **Ending\_Airport** | Array | Stores the details of the ending airport |
| Ending\_Airport.Code | String | Stores the Code of the airport where the journey ends |
| Ending\_Airport.Name | String | Stores the name of the airport where the journey ends |
| EndingAirport.LengthTime |  |  |
| **Passengers** | Array | Stores the details of the passengers traveling |
| Passengers.Email | String | Foreign Key |
| Passengers.Title | String | Stores the title of the passenger |
| Passenger.FullName | String | Stores the first and last name of the passenger |
| Passenger.BookingOwner:Bool | Boolean | Stands true if the passenger is the generator of the booking |
| **Intermediate\_Flights** | Array | Object is to keep the intermediate stop details of the flights |
| Intermediate\_Flights.FlightNumber | String | Foreign Key |
| Intermediate\_Flights.DepartureAirport | String | Stores the name of the departing airport |
| Intermediate\_Flights.DepartureDateTime | Date | Stores the date and time of the departure of the flights |
| Intermediate\_Flights.ArrivalAirport | String | Stores the name of the airport where the flight is arriving |
| Intermediate\_Flights.ArrivalDateTime | Date | Stores the date and time of the arrival of the flights |
| Intermediate\_Flights.totalDuration | Integer | Stores the time of the total duration |
| Intermediate\_Flights.SeatCostPerPassenger | Integer | Stores the cost of seating per passenger |

Revenue Collection:

|  |  |  |
| --- | --- | --- |
| Field | Data Type | Comment |
| fromDate | Date | The date from which the revenue was calculated |
| toDate | Date | The date until which the revenue was calculated |
| OperationalCosts | Double | The operational costs |
| EmployeesCosts | Double | The total salary bill |
| BookingRevenue | Double | The revenue from the bookings |
| Revenue | Double | The final revenue |

# **Deliverable D:**

For populating our database, we have used json files as well as the graphical user interface for MongoDB, Compass. Therefore, we exported the data of each collection as a json file using the command:

mongoexport --db AirlineDB --collection <CollectionName> --out <CollectionName>.json

The data can be imported in a database by using the following command a terminal (not the mongo shell):

mongoimport -d <databaseName> -c <CollectionName> <CollectionName>.json

For our 6 collections that we populated manually it would look something like this:

mongoimport -d AirlineDB -c AIRPORT airports.json

mongoimport -d AirlineDB -c EMPLOYEE employees.json

mongoimport -d AirlineDB -c PLANE planes.json

mongoimport -d AirlineDB -c PASSENGER passengers.json

mongoimport -d AirlineDB -c FLIGHT\_BOOKING bookings.json

mongoimport -d AirlineDB -c PLANE\_FLIGHT flights.json

All the json files can be found in the delivered folder.

# **Deliverable E:**

All the queries can be found in JavaScript file (.js) in the delivered folder, by the name Queries.js

# **Deliverable G:**

Relational Databases:

SQL, standing for Structured Query Language, is the language that these databases are written in and the language used to read and write data from them. These databases are comprised of tables which contain attributes and have relationships between them. Relational databases are great when you have a lot of data that needs to be structured in a very specific way, supporting join operations and being ACID compliant, often guarantying the integrity of our data.

NoSQL Databases:

Instead of tables, NoSQL databases and MongoDB in particular, support collections. A collection has fields instead of attributes, while each field can be an array, object or even array of objects, containing sub-documents. One of the main differences is the lack of relationships and join capabilities. They’re particularly useful when the data is not necessarily structured in a certain way, or if there are multiple sets of data that need to be structured in a way that a table could not easily support, for example nested objects, while the fact that they’re not ACID compliant makes them particularly fast.

Having shortly analyzed some of the main differences between the two databases, it is obvious that there would be some very basic differences in a relational approach to our database model.

In our database model we have decided to keep the details of the Pilots in the Employees collection. That is, to have a sub-document (nested object) in the documents that correspond to a Pilot with the details.

EmployeeID: 1234

Name: Andrew

Role: Pilot,

…,

Pilot Details: {

Pilot License Number: 1234

FfftTest: 2000-10-10

Test Result: success

}

Obviously, such a thing would be impossible with a relational database. If we still had decided to put the pilot details in the employees table then it would look something like this:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| EmployeeID | Name | ... | Role | Pilot License Number | ffftTest | Test Result |
| 1234 | George | … | Cabin Crew | null | null | null |

As seen in the above table, there are many attributes that would remain null in the case an employee was not a pilot.

Another difference would be, for example, in the relationship between the flight booking and the passenger collection. As it stands, each flight booking document has an array of objects for the passengers:

BookingCode: 1234

BookingDateTime: 2018-10-10

Passengers: {

Email: [example@example.com](mailto:example@example.com)

FullName: Andrew Richards

BookingOwner: true

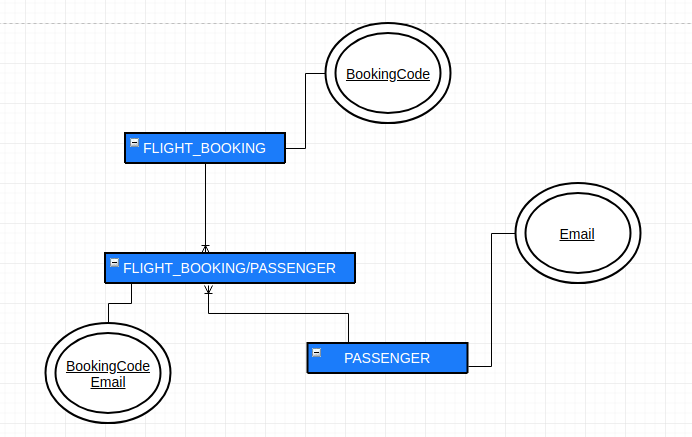
}

The problem if occurs if this was a relational database where there would be a many to many relationship, between the flight booking and the passenger table. In that case, the flight booking table would need to have fields for a possibly large number of passengers:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| BookingID | PassengerEmail1 | PassengerName1 | PassengerEmail2 | …. |
| 1234 | Example@example.com | Andrew Richards | Example2@example.com | ... |

It is obvious that this would be a very bad approach, as there should be a very large number of attributes, for the possibility of having many passengers in a booking, most of which would be null most of the time.

A much better approach in this case, would be to have an intermediate table between the flight booking and passenger tables:



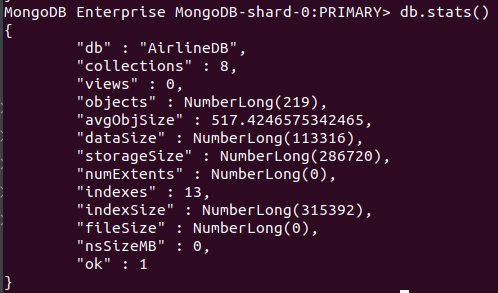
The intermediate table would have a combined key of the bookingCode along with the email address of the passenger.

In conclusion, it is quite understandable why a documented oriented database like MongoDB is a very good option for our database model.

**Deliverable F:**

Explain and profiling utilities

The command “db.stats()” shows the general information in the database. In the specific case of the “AirlineDB”, it shows util information like name, amount of collections, size of objects, database state and number of indexes.



Use of explain() command

Now, executing the command “db.FLIGHT\_BOOKING.find({bookingCode:1}).explain()”, the database returns the next information:

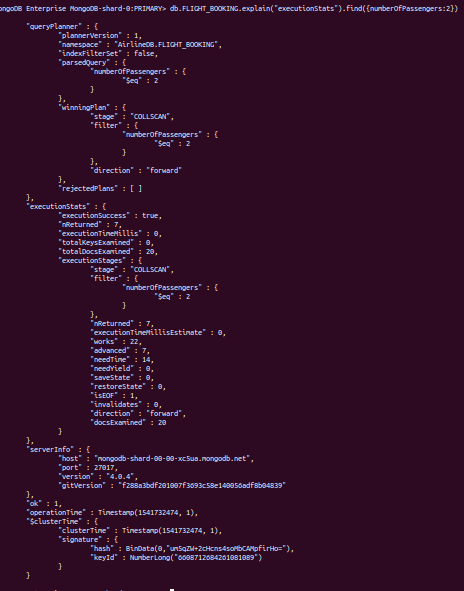


There, apart from the “clusterTime” and “operationTime”, the “winningPlan” is the most important because It gives the selected plan by the query optimizer which are based on a tree of stages. In this case, the query was performed using a field which has and index. A comparison executing a query using a field that does not have index is suitable.



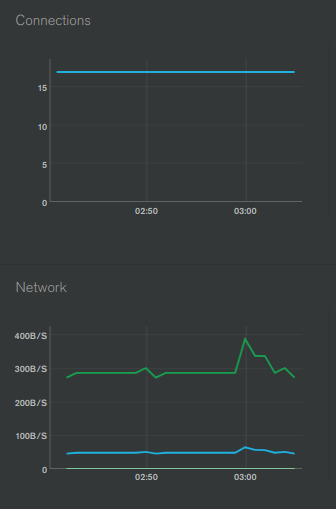
In this case, the "winningplan” given by the query planner is simpler and lacks on all the information regarding the index and its structure. Because it is a database with just a few records, the times are not significant and always 0 ms. The most important difference is that the query planner selects CALLSCAN to scan through collections; on the other hand, the IXSCAN is used in indexed fields with performance boost in massive databases as is citied in the official documentation of mondoDB[[1]](#footnote-30091).

Due this database was deployed in the Cluster Atlas[[2]](#footnote-27565), profiling options were not possible, but some collection of stats so. Executing the next command “db.FLIGHT\_BOOKING.explain("executionStats").find({numberOfPassengers:2})” it is obtained a more complete set of information, including the “executionStats” object which is certanly relevant:



This query shows the number of returned documents, the execution time according to the query planner, the totalDocsExamined as well as the previous information discussed in the previous images.

There is another important tool that is offer by the Atlas cluster, it is a graphical utility where can be seen not only the connections that the database has hosted but also the Network traffic. The next image shows it.



The blue lines, in general, mean the input flow to the database, and the green one the output flow direction.

1. Official documentation: https://docs.mongodb.com/manual/tutorial/analyze-query-plan/ [↑](#footnote-ref-30091)
2. Cluster where AirlineDB was running https://www.mongodb.com/cloud/atlas [↑](#footnote-ref-27565)