Queen Mary University of London

ECS789P
SEMI-STRUCTURED DATA AND ADVANCED DATA MODELLING

Coursework 2

Report

Postgrad, CW2-ECS789P_Group 55

Chung Wing Poon
Erblin Marku
Mousumi Hota

Table of Contents

Basic information

Folders and their usage

- ExplainResult – Outputs of MongoDB explain utility (.js)
- ProfilerResult – Outputs of MongoDB Profiler (.js)
- Queries – A set of 12 MongoDB queries (.js)
- Result – Outputs of 12 MongoDB queries (.js)
- TestData – Test data of this coursework (.json)

Import test data

1. Install MongoDB 5.0.5 Community Edition
2. Install MongoDB Compass
3. Open MongoDB Compass and connect to MongoDB
4. Create 8 collections
    a. Airports
    b. Bookings
    c. Employees
    d. Flights
    e. Passengers
    f. PilotRelatives
    g. Planes
    h. Revenues
5. Navigate to ./TestData
6. Import JSON file one by one to the corresponding collection

Assumption

All collections

- All datetime are ISO formatted and in UTC time
- All collection has an integer id field, which starts from 1, used to facilitate the random data generation and query purpose
- ISO 3166-1 alpha-2 country code are used in nationality / country related fields
- Passport numbers are all digits
- There are only two types of gender (sex): Male (M) or Female (F)

Employees

- Date of birth of all employees are in the range of "1970-01-01" to "1998-12-31"
- Monthly salaries of all employees are from 900.0 to 100000.0.
- There are only 4 types of employees:
    - BOOKC – booking clerks,
    - MAINS – maintenance staff
    - CABIN – cabin staff
    - PILOT – pilots
- Employment date (empDate) are in the range of "2012-01-01" to "2015-12-12"
- Fit-to-flying test (fffTest) field will be an empty string for non-pilot staffs
- Fit-to-flying test date are in the range of "2020-01-01" to "2020-12-31"

PilotRelatives

- Each pilot has exactly 1 relative stored in our database
- Spouse, parents, friend, friends are the relationships that is available

Planes

- Begin service dates are in the range of "2000-01-01" to "2012-12-31"
- There are only 5 types of aircraft status:
    - RPED – Repaired
    - UPED – Upgraded
    - NORM – Normal operation
    - PEND – Pending for repair
    - RETR – Retried
- All planes have minimum 1 and maximum 10 service logs
- Create date of service log are in the range of "2013-01-01" to "2021-12-31"

Airports

- All airports have a unique IATA code
- The range of refuel cost are from 1 to 100
- The range of stay cost are from 1 to 50

Flights

- Co-pilots are drawn from the available qualified pilots
- Qualified pilots are defined as pilots that has a fit-for-flying test date
- All flights have exactly one pilot and one co-pilot

- Departure date and time are in the range of “2021-10-10” to “2021-10-12” and “2021-12-01” to “2021-12-02”
- Arrival date and time are in the range of “2021-10-13” to “2021-10-15” and “2021-12-03” to “2021-12-04”
- Length of flight are from 1000.0 to 10000.0
- Planes will stay in airport for 2 hours max

Bookings

- A booking will have minimum 1 and maximum 4 passengers.
- A booking will have minimum 1 and maximum 4 flights
- All passengers will take the same flight together (i.e., if there are 4 flights and 4 passengers in the booking, they will take all 4 flights together)
- The create date of bookings are in the range of “2021-10-01” to “2021-11-30”

Collections

There are 8 collections in total.

Employee

Employee collection is responsible to store all information about the employees, for example name, date of birth, employee type etc. Each employee will have a 1 to many relationships with bookings and flights, as bookings may involve many employees (i.e., one booking clerk can handle multiple bookings) and a flight has two pilots (1 pilot and 1 copilot). Some employees (i.e. pilots) will have 1 to 1 relationships with PilotRelative as 1 pilot will have 1 relative information stored in this database. One pilot should not be flying two flights at the same time. The application which paired with this database, should handle this situation.

Fields	Data type	Description
id	Int	Employee id
name	Document	Name, see below
dob	Date()	Date of birth
phoneNo	String	Phone number
addr	String	Contact address
email	String	Email
salary	Double	Salary
details	Document	Detailed information of the employee, see below

Employee.name document

Fields	Data type	Description
fname	String	First name
lname	String	Last name
sex	String	Gender

Employee.details document

Fields	Data type	Description
sType	String	Type of employee
nat	String	Nationality
empDate	Date()	Employment date
passportNo	String	Passport number
fffDate	Date()	Contact address

PilotRelative

It is a common practice for airline to obtain the information of the relatives of pilots such that if something bad happened (e.g., plane crash), the airline can contact their relatives. Each pilot will have 1 relative that stored in the database. We decided to store this as a separate collection as not all employees will have this information.

Fields	Data type	Description
id	Int	Pilot Relatives Id
pilotId	Int	Pilot id, references Employees.id
name	Document	First name, last name and gender
phoneNo	String	Phone number
addr	String	Contact address
email	String	Email
relationship	String	Relationship with the pilot

PilotRelative.name document

Fields	Data type	Description
fname	String	First name
lname	String	Last name
sex	String	Gender

Plane

Each plane will have a 1 to many relationships with flights as 1 plane can be in many flights. Each plane may have more than 1 service log which is stored as a list of documents. The latest service log (i.e., the most recent createDate) will represent the current status of the plane.

Fields	Data type	Description
id	Int	Plane Id
mfg	String	Manufacturer
model	String	Model
fullFlyRange	Double	Full fuel load fly range (in KM)
beginServDate	Date()	Begin service date if the aircraft
cap	Integer	Seating capacity of the plane
servLog	List of Document	Service log of the plane, see below

Plane.servLog

It is a list of document which contains the current and previous aircraft status of the plane, create data of such log and the description of such log

Fields	Data type	Description
status	String	Aircraft status of the plane
createDate	String	Create date of this service log
description	String	Descriptions

Airports

An airport will have one-to-many relationships with flights as different flights may use the same or different airport for departure and landing. IATA code (code) is a unique code to identify an airport which provided by the International Air Transport Association.

Fields	Data type	Description
id	Int	Airport id
code	String	IATA (International Air Transport Association) code
name	String	The official name of the airport
location	Document	Country and city
rate	Document	Refuelling cost and stay cost

Airports.location

Fields	Data type	Description
country	String	Country of the airport
city	String	City that the airport is located

Airports.rate

Fields	Data type	Description
refuel	Double	Fixed charge for a plane refuels at an airport
stay	Double	Fixed hourly rate for a plane stops at an airport

Flights

A flight has many to many relationships with bookings as 1 booking may contains many flights and 1 flight can be in many bookings. The departure and arrival airport should not be the same. The pilot and co-pilot should not be the same person and be a qualified pilot (the one with fit-to-fly test). The application which paired with this database, should handle these situations.

Fields	Data type	Description
id	Int	Flight id
plane	Int	Plane that will use in this flight, references Plane.id
dep	Document	Departure information, see below
arv	Document	Arrival information, see below
fLength	Double	Length of the flight (KM)
pilot	Int	Pilot, references Employee.id
coPilot	Int	Co-pilot, references Employee.id

Flight.dep and Flight.arv document

Fields	Data type	Description
airport	Int	Arrival / departure airport, references Airport.id
stay	Int	No. of hours of plane stopped at airport
refuel	Boolean	Refuel or not
dt	Date()	Arrival / departure date and time

Bookings

Booking has many-to-many relationships with passenger as a passenger can have many bookings and a booking can have many passengers. Passengers is a list of integers that contains all the passenger ids that are responsible for a booking. Flights is a list of integers that contains all the flight ids that are responsible for a booking. Bookings should only be created by booking clerks. The application which paired with this database, should handle this situation.

Fields	Data type	Description
id	Int, pk, autoincrement	Booking id
createDate	Datetime	Create date of the booking
byStaff	Int	Created by who, references Employees.id
passengers	List of Int	Passengers for this booking, references Passenger.id
flights	List of Int	Flights used, references Flight.id
price	Float	Booking cost

Passengers

As a passenger can have multiple bookings, we decided to make it a collection instead of an embedded document in booking, such that we can reuse the passenger information and reduce the amount of redundant information.

Fields	Data type	Description
id	Int	Passenger id
name	Document	Name, see below
dob	Date()	Date of birth
phoneNo	String	Phone number
email	String	Email
passportNo	String	Passport number
nat	String	Nationality

Passenger.name document

Fields	Data type	Description
fname	String	First name
lname	String	Last name
sex	String	Gender

Revenue

Responsible to store calculated income, airport costs, salaries and the total revenue of the airline.

Fields	Data type	Description
_id	ObjectId()	Revenue object id
totalBookingCost	Double	Total income (total Bookings cost)
totalAirportCost	Double	Total airport cost
totalSalariy	Double	Total salary
revenue	Double	Revenue
createDate	Date()	Create date

Queries

1. Get the information of all qualified pilots. The airline may want to know how many qualified pilots available so that they could arrange the flights.

```
db.Employees.find(
  {
    'details.sType': "PILOT",
    'details.fffDate': {
      $ne: ''
    }
  }
)
```

Expected result: An array of 50 employees with “PILOT” sType and non-empty fit-to-fly test.

2. Get the number of planes that is in each of the available status. It is important for airline to understand how many planes that are ready to fly, need to repair, retired, or just upgraded. The query first unwinds servLog field and sort the createDate in descending order. Then we make use of $project to add new fields for each of the available status and $group to sum.

```
db.Planes.aggregate(
  [{
    $unwind: {
      path: '$servLog'
    }
  }, {
    $sort: {
      'servLog.createDate': -1
    }
  }, {
    $project: {
      pending: {
        $cond: [{ $eq: ['$servLog.status', 'PEND'] }, 1, 0]
      },
      normal: {
        $cond: [{ $eq: ['$servLog.status', 'NORM'] }, 1, 0]
      },
      retired: {
        $cond: [{ $eq: ['$servLog.status', 'RETR'] }, 1, 0]
      },
      upgraded: {
        $cond: [{ $eq: ['$servLog.status', 'UPED'] }, 1, 0]
      },
      repaired: {
        $cond: [{ $eq: ['$servLog.status', 'RPED'] }, 1, 0]
      }
    }
  }, {
    $group: {
      _id: null,
      pending: { $sum: '$pending' },
      normal: { $sum: '$normal' },
      retired: { $sum: '$retired' },
      upgraded: { $sum: '$upgraded' },
      repaired: { $sum: '$repaired' }
    }
  }]
)
```

Expected result: 200 planes pending to repair, 186 planes working normally, 50 planes retired, 182 planes upgraded and 195 planes repaired.

3. Calculate the monthly revenue of the airline and store it in the revenue collection. The example query below is calculating the revenue in October 2021. The query involves 15 steps:
    a. Matching the time and date range in Flights
    b. Perform 2 lookup to Airports for arrival and departure airports cost
    c. Unwind the lookup result from step 2
    d. Make a project, calculate the stay and refuel cost for each flight
    e. Group and sum as the airport cost
    f. Lookup all employees for the monthly salary of all employees
    g. Unwind the results from step 6
    h. Group and sum as the total salary bill of this month
    i. Lookup all bookings for incomes
    j. Unwind the results from step 9
    k. Matching the time and date range in booking
    l. Make a project, put the total salary bill, total airport cost and each booking income in
    m. Group and sum as the total booking income
    n. Make a project, put the total salary bill, total airport cost and total booking income in and calculate the revenue by total money from bookings – airport costs – total salary bill
    o. Merge the result to Revenue

```
db.Flights.aggregate(
  [{
    $match: {
      'arv.dt': {
        $gte: ISODate('2021-10-01'),
        $lte: ISODate('2021-10-31')
      },
      'dep.dt': {
        $gte: ISODate('2021-10-01'),
        $lte: ISODate('2021-10-31')
      }
    }
  }, {
    $lookup: {
      from: 'Airports',
      localField: 'arv.airport',
      foreignField: 'id',
      as: 'arvAirport'
    }
  }, {
    $lookup: {
      from: 'Airports',
      localField: 'dep.airport',
      foreignField: 'id',
      as: 'depAirport'
    }
  }, {
    $unwind: {
      path: '$arvAirport'
    }
```

```
  }, {
    $unwind: {
      path: '$depAirport'
    }
  }, {
    $project: {
      airportCost: {
        $let: {
          vars: {
            depStay: {
              $multiply: [
                '$dep.stay',
                '$depAirport.rate.stay'
              ]
            },
            depRF: {
              $cond: [
                '$dep.refuel',
                '$depAirport.rate.refuel',
                0
              ]
            },
            arvStay: {
              $multiply: [
                '$arv.stay',
                '$arvAirport.rate.stay'
              ]
            },
            arvRF: {
              $cond: [
                '$arv.refuel',
                '$arvAirport.rate.refuel',
                0
              ]
            }
          },
          'in': {
            $add: [
              '$$depStay',
              '$$depRF',
              '$$arvStay',
              '$$arvRF'
            ]
          }
        }
      }
    }
  }, {
    $group: {
      _id: null,
      totalAirportCost: {
        $sum: '$airportCost'
      }
    }
  }, {
    $lookup: {
      from: 'Employees',
      localField: '*',
      foreignField: '*',
      as: 'emps'
    }
```

```
  }, {
    $unwind: {
      path: '$emps'
    }
  }, {
    $group: {
      _id: '$totalAirportCost',
      totalSalary: {
        $sum: '$emps.salary'
      }
    }
  }, {
    $lookup: {
      from: 'Bookings',
      localField: '*',
      foreignField: '*',
      as: 'bookings'
    }
  }, {
    $unwind: {
      path: '$bookings'
    }
  }, {
    $match: {
      'bookings.createDate': {
        $gte: ISODate('2021-10-01'),
        $lte: ISODate('2021-10-31')
      }
    }
  }, {
    $project: {
      totalAirportCost: '$_id',
      totalSalary: true,
      bookingCost: '$bookings.price'
    }
  }, {
    $group: {
      _id: {
        totalAirportCost: '$totalAirportCost',
        totalSalary: '$totalSalary'
      },
      totalBookingCost: {
        $sum: '$bookingCost'
      }
    }
  }, {
    $project: {
      totalAirportCost: '$_id.totalAirportCost',
      totalSalary: '$_id.totalSalary',
      totalBookingCost: true,
      revenue: {
        $subtract: [{
          $subtract: [
            '$totalBookingCost',
            '$_id.totalAirportCost'
          ]
        },
          '$_id.totalSalary'
        ]
      },
      createDate: ISODate(),
```

```
      _id: ObjectId()
    }
  }, {
    $merge: 'Revenue'
  }]
)
```

Expected result: The result will be inserted into Revenue collection. The revenue is 548732.76, total cost of airport is 12138.29, total cost of booking is 2503443.00 and total salary bill is 1942571.95.

4. Get the planes which have performed the most miles during all flights. This can be used to see overused planes for repair procedures or looking at the least used planes to even out the usage. We have listed the 3 most used planes, but we can manage to see more by increasing the limit or sort ascending to see the least used planes.

```
db.Planes.aggregate(
  [{
    $lookup: {
      from: 'Flights',
      localField: 'id',
      foreignField: 'plane',
      as: 'planeFlights'
    }
  }, {
    $unwind: {
      path: '$planeFlights'
    }
  }, {
    $group: {
      _id: '$id',
      totalFlyingDistance: {
        $sum: '$planeFlights.fLength'
      }
    }
  }, {
    $sort: {
      totalFlyingDistance: -1
    }
  }, {
    $limit: 3
  }]
)
```

Expected result: Plane 94 with total flying distance 23427.87, plane 102 with total flying distance 22864.20, plane 64 with total flying distance 21524.72.

5. Get pilots that are 10 years younger than the co-pilots and the salary of the pilot is less than 80% of the salary of the co-pilot in a flight. Here we will go through the flights pilots and co-pilots and compare their ages and salaries. We set a flag with 2 conditions, if the pilots are 10 years younger than the co-pilots (which is the first indication that the co-pilot will get stressed because someone with less experience than him is a pilot). The second condition is that if the salary of the pilot is less than 80% of the salary of the co-pilot (which means we have not fair salaries). If both these conditions are true, we get the flight number, the pilot id and the co-pilot id so that the Human Resources department can investigate the problem.

```
db.Flights.aggregate(
  [{
```

```
    $lookup: {
      from: 'Employees',
      localField: 'pilot',
      foreignField: 'id',
      as: 'pilots'
    }
  }, {
    $lookup: {
      from: 'Employees',
      localField: 'coPilot',
      foreignField: 'id',
      as: 'copilots'
    }
  }, {
    $unwind: {
      path: '$pilots'
    }
  }, {
    $unwind: {
      path: '$copilots'
    }
  }, {
    $project: {
      id: true,
      pilot: true,
      pilotSalary: '$pilots.salary',
      pilotDOB: '$pilots.dob',
      coPilot: true,
      coSalary: '$copilots.salary',
      copilotDOB: '$copilots.dob',
      flagRedHR: {
        $let: {
          vars: {
            ps: '$pilots.salary',
            cs: {
              $multiply: [
                0.8,
                '$copilots.salary'
              ]
            },
            pdob: '$pilots.dob',
            cdob: {
              $dateAdd: {
                startDate: '$copilots.dob',
                unit: 'year',
                amount: 10
              }
            }
          },
          'in': {
            $and: [
              {
                $lt: [
                  '$$ps',
                  '$$cs'
                ]
              },
              {
                $gt: [
                  '$$pdob',
                  '$$cdob'
```

```
                ]
              }
            ]
          }
        }
      }
    }, {
      $match: {
        flagRedHR: true
      }
    }, {
      $project: {
        coPilot: true,
        pilot: true,
        flight: '$id'
      }
    }]
)
```

Expected result: An array of 26 objects which contains coPilot and pilot employees id, flight id and the object id.

6. Get which countries had the most flights in or out. We eliminate from the result the countries with no flights, so this query can be adjusted to look at the countries with the lowest number of flights by changing the sort of count to 1, or the top 10 countries with largest number of flights by changing the limit to 10.

```
db.Airports.aggregate(
  [{
    $lookup: {
      from: 'Flights',
      localField: 'id',
      foreignField: 'arv.airport',
      as: 'arvFlight'
    }
  }, {
    $lookup: {
      from: 'Flights',
      localField: 'id',
      foreignField: 'dep.airport',
      as: 'depFlight'
    }
  }, {
    $match: {
      $or: [
        {
          arvFlight: {
            $not: {
              $size: 0
            }
          }
        },
        {
          depFlight: {
            $not: {
              $size: 0
            }
          }
        }
```

```
      ]
    }
  }, {
    $project: {
      country: '$location.country',
      cnt: {
        $add: [
          {
            $size: '$arvFlight'
          },
          {
            $size: '$depFlight'
          }
        ]
      }
    }
  }, {
    $group: {
      _id: '$country',
      flightCount: {
        $sum: '$cnt'
      }
    }
  }, {
    $sort: {
      flightCount: -1
    }
  }, { $limit: 10 }]
)
```

Expected result: US with 173 flights, AU with 51 flights, CA with 39 flights, BR with 31 flights, PG with 30 flights, ID with 24 flights, CN with 22 flights, RU with 17 flights, IN with 12 flights, JP with 12 flights.

7. Get the top 10 passengers who travelled the longest distance. The airline may give rewards or vouchers for customers. Again, we can modify the limit to see the least traveling passengers or see more listings by changing the limit in the query.

```
db.Bookings.aggregate(
  [{
    $unwind: {
      path: '$flights'
    }
  }, {
    $unwind: {
      path: '$passengers'
    }
  }, {
    $lookup: {
      from: 'Flights',
      localField: 'flights',
      foreignField: 'id',
      as: 'userFlight'
    }
  }, {
    $lookup: {
      from: 'Passengers',
      localField: 'passengers',
      foreignField: 'id',
      as: 'passengerData'
```

```
    }
  }, {
    $unwind: {
      path: '$userFlight'
    }
  }, {
    $unwind: {
      path: '$passengerData'
    }
  }, {
    $project: {
      id: '$passengerData.id',
      name: '$passengerData.name',
      distance: '$userFlight.fLength'
    }
  }, {
    $group: {
      _id: {
        id: '$id',
        name: '$name'
      },
      totalDistance: {
        $sum: '$distance'
      }
    }
  }, {
    $sort: {
      totalDistance: -1
    }
  }, { $limit: 10 }]
)
```

Expected results: 10 passengers returned with their name, id and total distance of travelling in kilometres. Sir Vladimir Raynor with id 65 should be the passenger with the longest total distance of 327039.41.

8. Get and group by the number of years of employment for all employees.

```
db.Employees.aggregate(
  [{
    $project: {
      id: true,
      name: {
        $concat: [
          '$name.fname', ' ', '$name.lname'
        ]
      },
      sType: '$details.sType',
      salary: true,
      email: true,
      addr: true,
      empDate: '$details.empDate',
      empYears: {
        $dateDiff: {
          startDate: '$details.empDate',
          endDate: ISODate(),
          unit: 'year'
        }
      }
    }
  }, {
```

```
    $group: {
      _id: '$empYears',
      employees: {
        $push: {
          id: '$id',
          name: '$name',
          salary: '$salary',
          email: '$email',
          addr: '$addr',
          sType: '$sType',
          empDate: '$empDate'
        }
      }
    }
  }]
)
```

Expected result: There will be 4 groups (6,7,8,9) which indicate the number of years of employees that is in the group.

9. Showing the refuel and stay cost of airports by interval of 10. This aggregation query, facet aggregate stage deals multifaceted data. In this case, two kinds of rates namely refuel (denoting refuel cost) and stay (denoting cost for at airport). Two buckets of these rate values are created, buckets_by_stay, buckets_by_range. We make use of "group by" clause to group stay, refuelling costs in each bucket based on their values. We create range of values using the "boundaries" field of bucket. if any of the values of refuelling or stay costs doesn't fit into the ranges we have mentioned (say above 100, in this case), will be categorised as 'Others'. Finally count the values as per boundaries mentioned before (ex:20 to 30 ,30 to 40 range, and so on) using the expression $sum:1($sum is the aggregate on the range-based buckets for stay, refuelling). Then we use push aggregate to push the airport details

```
db.Airports.aggregate(
  [{
    $facet: {
      buckets_by_stay: [
        {
          $bucket: {
            groupBy: '$rate.stay',
            boundaries: [
              20,
              30,
              40,
              50,
              60,
              70,
              80,
              90,
              100
            ],
            'default': 'Other',
            output: {
              airport_stay_cost_buckets_by_range: {
                $sum: 1
              },
              airport_details: {
                $push: {
                  code: '$code',
```

```
                    name: '$name',
                    location: '$location'
                  }
                }
              }
            }
          }
        ],
        buckets_by_refuel: [
          {
            $bucket: {
              groupBy: '$rate.refuel',
              boundaries: [
                20,
                30,
                40,
                50,
                60,
                70,
                80,
                90,
                100
              ],
              'default': 'Other',
              output: {
                flight_refuel_buckets_by_range: {
                  $sum: 1
                },
                airport_details: {
                  $push: {
                    code: '$code',
                    name: '$name',
                    location: '$location'
                  }
                }
              }
            }
          }
        ]
      }
    }]
)
```

Expected result: Airports being classified into different boundaries with two categories: stay cost and refuel cost. Please refer to "QueryResult/Q9Result.js" for detailed result.

10. Shows booking by date and price range in a specific range of date. This aggregation query pipeline is created on Booking collection and makes use of project->match->sort aggregate stages. In the project stage we decide upon which fields to include (by setting value as fieldname:1, in our case. Generally, could also be used to exclude fields). In the match stage we make the date ranged (in this case, 1st Oct 2021 to 1st Dec 2021) query on the output of project stage. The user supplied date range (using $gte and $lte aggregates comparing user supplied value to values present in createDate, i.e. booking date) values to ISODate(the type of value the create date field would expect). An additional criterion is introduced here for allowing user to reduce the search space by including price range (we have considered price range as from 2000-4000). Finally, in the sort stage we sort the values by price and date values obtained from match stage, in ascending order, to provide the end users the bookings made starting from lowest cost and earliest booked to relatively more cost and bookings

done later in an increasing order for the given date range, price range in the previous match stage.

```
db.Bookings.aggregate(
  [{
    $project: {
      id: 1,
      flights: 1,
      price: 1,
      createDate: 1
    }
  }, {
    $match: {
      createDate: {
        $gte: ISODate('2021-10-01T00:00:00.000Z'),
        $lte: ISODate('2021-12-01T00:00:00.000Z')
      },
      price: {
        $gte: 2000,
        $lte: 4000
      }
    }
  }, {
    $sort: {
      price: 1,
      createDate: 1
    }
  }]
)
```

Expected result: A array of 195 bookings will be returned.

11. Check if there are over-booked flights by getting the number of passengers in each flight. This is done by using the Flights collection's plane id to lookup Planes collection to get the capacity of the plane that is utilised. It will then lookup the bookings collection to locate the number of bookings that is associated with such flight id. We can then get the capacity of the aircraft and passengers by unwinding planeDetails and booking. Size function is utilised to count the number of passengers in each booking. The result will then be grouped up by the id of the flight and the capacity of the plane. Only those with number of passenger larger than capacity of the plane will be returned as result.

```
db.Flights.aggregate(
  [{
    $lookup: {
      from: 'Planes',
      localField: 'plane',
      foreignField: 'id',
      as: 'planeDetails'
    }
  }, {
    $lookup: {
      from: 'Bookings',
      localField: 'id',
      foreignField: 'flights',
      as: 'booking'
    }
  }, {
    $unwind: {
      path: '$planeDetails'
```

```
    }
  }, {
    $unwind: {
      path: '$booking'
    }
  }, {
    $project: {
      flightId: '$id',
      cap: '$planeDetails.cap',
      passengerCnt: {
        $size: '$booking.passengers'
      }
    }
  }, {
    $group: {
      _id: {
        flightId: '$flightId',
        cap: '$cap'
      },
      numPassengers: {
        $sum: '$passengerCnt'
      }
    }
  }, {
    $project: {
      _id: false,
      flightId: '$_id.flightId',
      remainCap: {
        $subtract: [
          '$_id.cap',
          '$numPassengers'
        ]
      }
    }
  }, {
    $match: {
      remainCap: {
        $lt: 0
      }
    }
  }]
)
```

Expected result: The remaining capacity (remainCap) and the flight id will be returned if such flight has the remaining capacity lower than 0. Nothing will be returned in our dataset as no flight is over-booked.

12. Getting all pilot and their relative details, for instance in case of emergency to contact their next of kin. This aggregate query is written to help notify the next of kin/relative in case of an emergency arises for a pilot, using id. It follows a lookup ->unwind. This query pipeline is created on PilotRelative Collection. In the lookup stage we perform lookup from Employees collection with localfield as pilot_id and foreignField as id which stores id of employee in Employee collection), “as” pilotInfo will be the array of objects for matched s from Employees collection. We then use unwind aggregate stage for deconstructing it for each element in the output of the previous lookup stage (in our case there is only one pilot entry possible for a match of ids from both collection).

```
db.PilotRelatives.aggregate(
  [{
```

```
    $lookup: {
      from: 'Employees',
      localField: 'pilot_id',
      foreignField: 'id',
      as: 'pilotInfo'
    }
  }, {
    $unwind: {
      path: '$pilotInfo'
    }
  }]
)
```

Expected result: A array of 60 pilots relatives and the details of the pilot.

Explain Utility

The explain utility of MongoDB can be used to monitor whether an index is used and how effective it is in reducing query execution time. In this section, we will select query 3 and 7 to see how indexes improve the efficiency of our query. We will run the explain utility with verbosity parameter "executionStats" which will return statistics describing the execution of the winning plan (e.g. time spend in executing part of and entire query)

Query 3 explain

Query 3 involves multiple lookups to Airports, which is a collection with 9225 documents.

Without index

Please refer to "ExplainResult/Q3Explain.js" for the detailed output of the explain utility. This is just the except of the stages Airports lookup section the output.

```
    {
      '$lookup': {
        from: 'Airports',
        as: 'arvAirport',
        localField: 'arv.airport',
        foreignField: 'id'
      },
      totalDocsExamined: Long("922500"),
      totalKeysExamined: Long("0"),
      collectionScans: Long("200"),
      indexesUsed: [],
      nReturned: Long("100"),
      executionTimeMillisEstimate: Long("244")
    },
    {
      '$lookup': {
        from: 'Airports',
        as: 'depAirport',
        localField: 'dep.airport',
        foreignField: 'id'
      },
      totalDocsExamined: Long("922500"),
      totalKeysExamined: Long("0"),
      collectionScans: Long("200"),
      indexesUsed: [],
      nReturned: Long("100"),
      executionTimeMillisEstimate: Long("481")
    },
```

The explainer provided the execution time of each stage of the aggregation in milliseconds. We can see that MongoDB do 200 collection scans to Airports and examined 922500 documents which take nearly 0.5 seconds (481ms)

With index on Airports

Query 3 uses "id" field to lookup Airports. Thus, we will create index for Airports using the "id" field with option "Build index in the background" and "Create unique index".

Please refer to "ExplainResult/Q3Explain_WithIndex.js" for the detailed output of the explain utility. This is just the except of the stages Airports lookup section the output.

```
      {
```

```
        '$lookup': {
          from: 'Airports',
          as: 'arvAirport',
          localField: 'arv.airport',
          foreignField: 'id'
        },
        totalDocsExamined: Long("100"),
        totalKeysExamined: Long("100"),
        collectionScans: Long("0"),
        indexesUsed: ['id'],
        nReturned: Long("100"),
        executionTimeMillisEstimate: Long("5")
      },
      {
        '$lookup': {
          from: 'Airports',
          as: 'depAirport',
          localField: 'dep.airport',
          foreignField: 'id'
        },
        totalDocsExamined: Long("100"),
        totalKeysExamined: Long("100"),
        collectionScans: Long("0"),
        indexesUsed: ['id'],
        nReturned: Long("100"),
        executionTimeMillisEstimate: Long("8")
      },
```

We can see that index has a huge performance improvement on query a collection that have a large amount of data. The time now required to execute these two stages is 8ms, which is 98% faster than query without index. MongoDB now only examinate documents that has the id which provided by Flights and does not do any collection scans.

Query 7 explain

Query 7 involves lookups to Flights and Passengers.

Without index

Please refer to "ExplainResult/Q7Explain.js" for the detailed output of the explain utility. This is just the except of the stages section of the output.

```
stages: [
      {
        '$cursor': {
          queryPlanner: {
            namespace: 'ecs789p.Bookings',
            indexFilterSet: false,
            parsedQuery: {},
            queryHash: '8590D559',
            planCacheKey: '0A506B49',
            maxIndexedOrSolutionsReached: false,
            maxIndexedAndSolutionsReached: false,
            maxScansToExplodeReached: false,
            winningPlan: {
              stage: 'PROJECTION_SIMPLE',
              transformBy: {
                _id: 1,
                flights: 1,
                passengerData: 1,
```

```
                passengers: 1,
                userFlight: 1
              },
              inputStage: { stage: 'COLLSCAN', direction: 'forward' }
            },
            rejectedPlans: []
          },
          executionStats: {
            executionSuccess: true,
            nReturned: 1000,
            executionTimeMillis: 1080,
            totalKeysExamined: 0,
            totalDocsExamined: 1000,
            executionStages: {
              stage: 'PROJECTION_SIMPLE',
              nReturned: 1000,
              executionTimeMillisEstimate: 0,
              works: 1002,
              advanced: 1000,
              needTime: 1,
              needYield: 0,
              saveState: 2,
              restoreState: 2,
              isEOF: 1,
              transformBy: {
                _id: 1,
                flights: 1,
                passengerData: 1,
                passengers: 1,
                userFlight: 1
              },
              inputStage: {
                stage: 'COLLSCAN',
                nReturned: 1000,
                executionTimeMillisEstimate: 0,
                works: 1002,
                advanced: 1000,
                needTime: 1,
                needYield: 0,
                saveState: 2,
                restoreState: 2,
                isEOF: 1,
                direction: 'forward',
                docsExamined: 1000
              }
            }
          }
        },
        nReturned: Long("1000"),
        executionTimeMillisEstimate: Long("1")
      },
      {
        '$unwind': { path: '$flights' },
        nReturned: Long("2488"),
        executionTimeMillisEstimate: Long("1")
      },
      {
        '$unwind': { path: '$passengers' },
        nReturned: Long("6179"),
        executionTimeMillisEstimate: Long("1")
      },
```

```
      {
        '$lookup': {
          from: 'Flights',
          as: 'userFlight',
          localField: 'flights',
          foreignField: 'id'
        },
        totalDocsExamined: Long("2471600"),
        totalKeysExamined: Long("0"),
        collectionScans: Long("12358"),
        indexesUsed: [],
        nReturned: Long("6179"),
        executionTimeMillisEstimate: Long("315")
      },
      {
        '$lookup': {
          from: 'Passengers',
          as: 'passengerData',
          localField: 'passengers',
          foreignField: 'id'
        },
        totalDocsExamined: Long("617900"),
        totalKeysExamined: Long("0"),
        collectionScans: Long("12358"),
        indexesUsed: [],
        nReturned: Long("6179"),
        executionTimeMillisEstimate: Long("1080")
      },
      {
        '$unwind': { path: '$userFlight' },
        nReturned: Long("6179"),
        executionTimeMillisEstimate: Long("1080")
      },
      {
        '$unwind': { path: '$passengerData' },
        nReturned: Long("6179"),
        executionTimeMillisEstimate: Long("1080")
      },
      {
        '$project': {
          _id: true,
          id: '$passengerData.id',
          name: '$passengerData.name',
          distance: '$userFlight.fLength'
        },
        nReturned: Long("6179"),
        executionTimeMillisEstimate: Long("1080")
      },
      {
        '$group': {
          _id: { id: '$id', name: '$name' },
          totalDistance: { '$sum': '$distance' }
        },
        maxAccumulatorMemoryUsageBytes: { totalDistance: Long("7200") },
        totalOutputDataSizeBytes: Long("87731"),
        usedDisk: false,
        nReturned: Long("100"),
        executionTimeMillisEstimate: Long("1080")
      },
      {
        '$sort': { sortKey: { totalDistance: -1 }, limit: Long("10") },
```

```
        totalDataSizeSortedBytesEstimate: Long("27705"),
        usedDisk: false,
        nReturned: Long("10"),
        executionTimeMillisEstimate: Long("1080")
      }
    ]
```

We can see that without indexes we will need around 1 second (1080ms) to execute this query. MongoDB examined 2471600 and 617900 documents in Flights and Passengers respectively with a high number of collection scan 12358.

With index on Passenger and Flights

Query 7 uses “id” field to lookup Passengers and Flights. Thus, we will create index for Passengers and Flights using the “id” field of the respective collections with option “Build index in the background” and “Create unique index”.

Please refer to “ExplainResult/Q7Explain_WithIndex.js” for the detailed output of the explain utility. This is just the except of the stages section of the output.

```
stages: [
      {
        '$cursor': {
          queryPlanner: {
            namespace: 'ecs789p.Bookings',
            indexFilterSet: false,
            parsedQuery: {},
            queryHash: '8590D559',
            planCacheKey: '0A506B49',
            maxIndexedOrSolutionsReached: false,
            maxIndexedAndSolutionsReached: false,
            maxScansToExplodeReached: false,
            winningPlan: {
              stage: 'PROJECTION_SIMPLE',
              transformBy: {
                _id: 1,
                flights: 1,
                passengerData: 1,
                passengers: 1,
                userFlight: 1
              },
              inputStage: { stage: 'COLLSCAN', direction: 'forward' }
            },
            rejectedPlans: []
          },
          executionStats: {
            executionSuccess: true,
            nReturned: 1000,
            executionTimeMillis: 388,
            totalKeysExamined: 0,
            totalDocsExamined: 1000,
            executionStages: {
              stage: 'PROJECTION_SIMPLE',
              nReturned: 1000,
              executionTimeMillisEstimate: 0,
              works: 1002,
              advanced: 1000,
              needTime: 1,
              needYield: 0,
```

```
              saveState: 2,
              restoreState: 2,
              isEOF: 1,
              transformBy: {
                _id: 1,
                flights: 1,
                passengerData: 1,
                passengers: 1,
                userFlight: 1
              },
              inputStage: {
                stage: 'COLLSCAN',
                nReturned: 1000,
                executionTimeMillisEstimate: 0,
                works: 1002,
                advanced: 1000,
                needTime: 1,
                needYield: 0,
                saveState: 2,
                restoreState: 2,
                isEOF: 1,
                direction: 'forward',
                docsExamined: 1000
              }
            }
          }
        },
        nReturned: Long("1000"),
        executionTimeMillisEstimate: Long("0")
      },
      {
        '$unwind': { path: '$flights' },
        nReturned: Long("2488"),
        executionTimeMillisEstimate: Long("0")
      },
      {
        '$unwind': { path: '$passengers' },
        nReturned: Long("6179"),
        executionTimeMillisEstimate: Long("0")
      },
      {
        '$lookup': {
          from: 'Flights',
          as: 'userFlight',
          localField: 'flights',
          foreignField: 'id'
        },
        totalDocsExamined: Long("6179"),
        totalKeysExamined: Long("6179"),
        collectionScans: Long("0"),
        indexesUsed: ['id'],
        nReturned: Long("6179"),
        executionTimeMillisEstimate: Long("212")
      },
      {
        '$lookup': {
          from: 'Passengers',
          as: 'passengerData',
          localField: 'passengers',
          foreignField: 'id'
        },
```

```
        totalDocsExamined: Long("6179"),
        totalKeysExamined: Long("6179"),
        collectionScans: Long("0"),
        indexesUsed: ['id'],
        nReturned: Long("6179"),
        executionTimeMillisEstimate: Long("388")
      },
      {
        '$unwind': { path: '$userFlight' },
        nReturned: Long("6179"),
        executionTimeMillisEstimate: Long("388")
      },
      {
        '$unwind': { path: '$passengerData' },
        nReturned: Long("6179"),
        executionTimeMillisEstimate: Long("388")
      },
      {
        '$project': {
          _id: true,
          id: '$passengerData.id',
          name: '$passengerData.name',
          distance: '$userFlight.fLength'
        },
        nReturned: Long("6179"),
        executionTimeMillisEstimate: Long("388")
      },
      {
        '$group': {
          _id: { id: '$id', name: '$name' },
          totalDistance: { '$sum': '$distance' }
        },
        maxAccumulatorMemoryUsageBytes: { totalDistance: Long("7200") },
        totalOutputDataSizeBytes: Long("87731"),
        usedDisk: false,
        nReturned: Long("100"),
        executionTimeMillisEstimate: Long("388")
      },
      {
        '$sort': { sortKey: { totalDistance: -1 }, limit: Long("10") },
        totalDataSizeSortedBytesEstimate: Long("27705"),
        usedDisk: false,
        nReturned: Long("10"),
        executionTimeMillisEstimate: Long("388")
      }
    ]
```

We can see that there is a massive reduction on the overall execution time of the query (executionStats.executionTimeMillis), from around 1 second (1080ms) to around 0.3 second (388ms) which 64% faster than query without index. The main cause is the increase of efficiency when MongoDB lookup Passengers collection. Passengers contains 100 documents, and it has a many-to-many relationship with Bookings, and a collection scan will need some time as MongoDB will need to scan every document in the collection. By adding “id” field as index of Passengers and Flights, MongoDB now does not need to do ten thousand collections scans to acquire the documents.

Profiler

The profiler of MongoDB captures all detailed information on the write operations, cursors and database commands and store them in “system.profile” collection. Profiler is a useful tool to examinate the performance of queries such that we could identify which queries need to be tuned.

In this section, we will set the profiling level with the following parameters:

```
db.setProfilingLevel(1, {slowms: 50})
```

to collect data for all queries that takes more than 50ms to execute to analyse the performance of our queries. Three queries are selected

Query 5

```
[
  {
    op: 'command',
    ns: 'ecs789p.Flights',
    command: {
      aggregate: 'Flights',
      pipeline: [
        {
          '$lookup': {
            from: 'Employees',
            localField: 'pilot',
            foreignField: 'id',
            as: 'pilots'
          }
        },
        {
          '$lookup': {
            from: 'Employees',
            localField: 'coPilot',
            foreignField: 'id',
            as: 'copilots'
          }
        },
        { '$unwind': { path: '$pilots' } },
        { '$unwind': { path: '$copilots' } },
        {
          '$project': {
            id: true,
            pilot: true,
            pilotSalary: '$pilots.salary',
            pilotDOB: '$pilots.dob',
            coPilot: true,
            coSalary: '$copilots.salary',
            copilotDOB: '$copilots.dob',
            flagRedHR: { '$let': [Object] }
          }
        },
        { '$match': { flagRedHR: true } },
        { '$project': { coPilot: true, pilot: true, flight: '$id' } }
      ],
      cursor: {},
      lsid: { id: UUID("61b5e835-a890-42ab-bb86-0ee6c70e6f42") },
      '$db': 'ecs789p'
    },
    keysExamined: 0,
```

```
    docsExamined: 208400,
    cursorExhausted: true,
    numYield: 0,
    nreturned: 26,
    queryHash: '6DAB46EC',
    planCacheKey: 'D23A0176',
    locks: {
      Global: { acquireCount: { r: Long("1602") } },
      Mutex: { acquireCount: { r: Long("1602") } }
    },
    flowControl: {},
    responseLength: 1706,
    protocol: 'op_msg',
    millis: 79,
    planSummary: 'COLLSCAN',
    ts: ISODate("2021-12-23T08:22:21.638Z"),
    client: '192.168.76.1',
    appName: 'mongosh 1.1.7',
    allUsers: [],
    user: ''
  }
]
```

It is observed that there is a huge number of documents (208400) examined in query 5. If the number of documents increases in Employees, the execute time is expected to be increase also. Indexes should be created in the Employees collection to improve the performance.

Query 6

```
[
  {
    op: 'command',
    ns: 'ecs789p.Airports',
    command: {
      aggregate: 'Airports',
      pipeline: [
        {
          '$lookup': {
            from: 'Flights',
            localField: 'id',
            foreignField: 'arv.airport',
            as: 'arvFlight'
          }
        },
        {
          '$lookup': {
            from: 'Flights',
            localField: 'id',
            foreignField: 'dep.airport',
            as: 'depFlight'
          }
        },
        { '$match': { '$or': [[Object], [Object]] } },
        {
          '$project': { country: '$location.country', cnt: { '$add': [Array] } }
        },
        {
          '$group': { _id: '$country', flightCount: { '$sum': '$cnt' } }
        },
        { '$sort': { flightCount: -1 } },
```

```
        { '$limit': 10 }
      ],
      cursor: {},
      lsid: { id: UUID("61b5e835-a890-42ab-bb86-0ee6c70e6f42") },
      '$db': 'ecs789p'
    },
    keysExamined: 0,
    docsExamined: 7389225,
    hasSortStage: true,
    cursorExhausted: true,
    numYield: 9,
    nreturned: 10,
    queryHash: '3323FF12',
    planCacheKey: 'C75B6B35',
    locks: {
      ReplicationStateTransition: { acquireCount: { w: Long("1") } },
      Global: { acquireCount: { r: Long("36912") } },
      Mutex: { acquireCount: { r: Long("36902") } }
    },
    flowControl: {},
    storage: {},
    responseLength: 475,
    protocol: 'op_msg',
    millis: 2806,
    planSummary: 'COLLSCAN',
    ts: ISODate("2021-12-23T09:43:04.277Z"),
    client: '192.168.76.1',
    appName: 'mongosh 1.1.7',
    allUsers: [],
    user: ''
  }
]
```

This query requires 2.8 seconds to complete. It is observed that there is an unexpected high number of documents (7.3 million) examined in query 6 to complete the aggregation which may caused by using Airports to lookup Flights as Airport has over 9000 documents.

Query 11

```
[
  {
    op: 'command',
    ns: 'ecs789p.Flights',
    command: {
      aggregate: 'Flights',
      pipeline: [
        {
          '$lookup': {
            from: 'Planes',
            localField: 'plane',
            foreignField: 'id',
            as: 'planeDetails'
          }
        },
        {
          '$lookup': {
            from: 'Bookings',
            localField: 'id',
            foreignField: 'flights',
            as: 'booking'
```

```
          }
        },
        { '$unwind': { path: '$planeDetails' } },
        { '$unwind': { path: '$booking' } },
        {
          '$project': {
            flightId: '$id',
            cap: '$planeDetails.cap',
            passengerCnt: { '$size': '$booking.passengers' }
          }
        },
        {
          '$group': {
            _id: { flightId: '$flightId', cap: '$cap' },
            numPassengers: { '$sum': '$passengerCnt' }
          }
        },
        {
          '$project': {
            _id: false,
            flightId: '$_id.flightId',
            remainCap: { '$subtract': [Array] }
          }
        },
        { '$match': { remainCap: { '$lt': 0 } } }
      ],
      cursor: {},
      lsid: { id: UUID("61b5e835-a890-42ab-bb86-0ee6c70e6f42") },
      '$db': 'ecs789p'
    },
    keysExamined: 0,
    docsExamined: 480400,
    cursorExhausted: true,
    numYield: 400,
    nreturned: 3279,
    queryHash: 'E8FDCA30',
    planCacheKey: '8D4C8D7B',
    locks: {
      ReplicationStateTransition: { acquireCount: { w: Long("1") } },
      Global: { acquireCount: { r: Long("2003") } },
      Mutex: { acquireCount: { r: Long("1602") } }
    },
    flowControl: {},
    storage: {},
    responseLength: 104,
    protocol: 'op_msg',
    millis: 186,
    planSummary: 'COLLSCAN',
    ts: ISODate("2021-12-23T09:17:06.938Z"),
    client: '192.168.76.1',
    appName: 'mongosh 1.1.7',
    allUsers: [],
    user: ''
  }
]
```

Query 11 needs to examinate 480400 documents to return the result. numYield in MongoDB is defined as the number of times the operation yielded to allow other operations to complete. Operation yields when it needs to read data that MongoDB has not yet fully read into memory. In another words, there are 400 extra operations that requires to read data from hard disk that caused some latency to slow down the query.