Contents

[**Team** 1](#_Toc126446857)

[**Final business domain write-up** 1](#_Toc126446858)

[**Final ER Model** 2](#_Toc126446859)

[**How the cloud database design meets the system requirements** 3](#_Toc126446860)

[**Screenshots of implementation of cloud storage** 3](#_Toc126446861)

[**Kelven** 9](#_Toc126446862)

[**Denormalisation** 9](#_Toc126446863)

[**Data placement strategy** 11](#_Toc126446864)

[**Selection of cloud storage** 11](#_Toc126446865)

[**Design and content of cloud storage** 12](#_Toc126446866)

[**Business Queries** 14](#_Toc126446867)

[**Kok Kai** 15](#_Toc126446868)

[**Denormalisation** 15](#_Toc126446869)

[**Data placement strategy** 18](#_Toc126446870)

[**Selection of cloud storage** 21](#_Toc126446871)

[**Design and content of cloud storage** 21](#_Toc126446872)

[**Business Queries** 25](#_Toc126446873)

[Other business queries can be: 26](#_Toc126446874)

[**Hafeezur** 26](#_Toc126446875)

[**Denormalisation** 26](#_Toc126446876)

[**Data placement strategy** 27](#_Toc126446877)

[**Selection of cloud storage** 28](#_Toc126446878)

[**Design and content of cloud storage** 29](#_Toc126446879)

[**Business Queries** 33](#_Toc126446880)

[**Ambrish** 34](#_Toc126446881)

[**Denormalisation** 34](#_Toc126446882)

[**Data placement strategy** 38](#_Toc126446883)

[**Selection of cloud storage** 41](#_Toc126446884)

[**Design and content of cloud storage** 42](#_Toc126446885)

[**Business Queries** 46](#_Toc126446886)

# **Team**

## **Final business domain write-up**

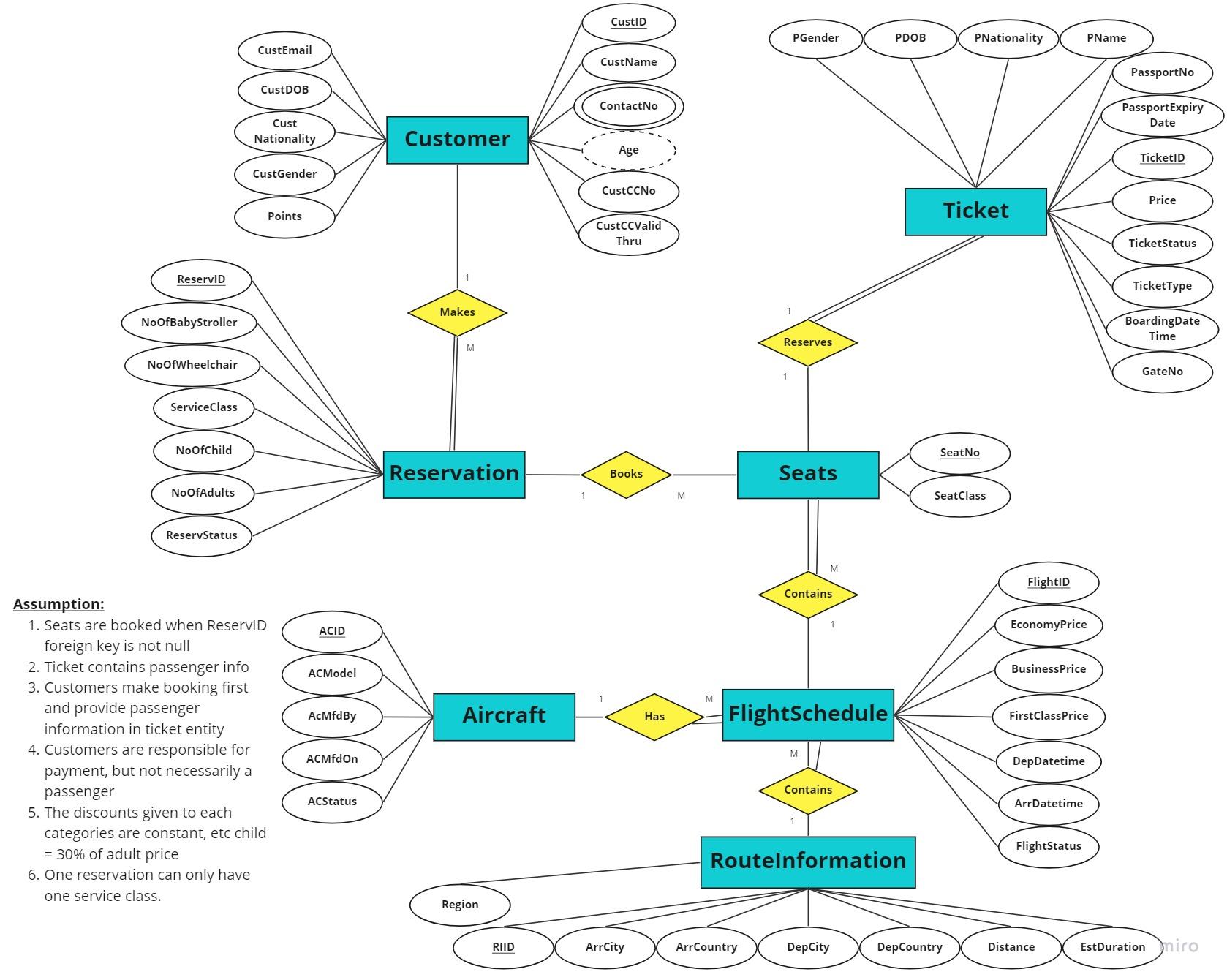
In response to the sharp surge in demand for post-Covid air travel as countries are lifting Covid-19 restrictions, one of the world’s leading airlines has decided to re-design its Airline Reservation & Ticketing System (ARTS). In particular, the airline would like to leverage the benefits of cloud databases to increase performance and reduce storage costs.

This report aims to cover the following sections

1. Team
   1. The final entity relationship diagram
   2. How our chosen cloud database meets the system requirements
   3. Screenshots of implementation of cloud storage
2. Individual
   1. Denormalisation
   2. Data placement strategy
   3. Selection of cloud storage strategies
   4. Design/Schema and content of cloud storage
   5. Business queries

## 

## **Final ER Model**



[Link to ER Model](https://miro.com/app/board/uXjVP2Ysd4Y=/?share_link_id=454619882800)

**Final Relational Mapping**

Customer (CustID, CustName, CustEmail, CustDOB, CustGender, CustNationality, CustCCNo, CustCCValidThru, Points)

CustomerContact (<CustID>, ContactNo)

Ticket (TicketID, PName, PGender, PDOB, PNationality, PassportNo, PassportExpiryDate, Price, TicketType, BoardingDateTime, GateNo, TicketStatus)

Reservation (ReservID, NoOfBabyStroller, NoOfWheelchair, NoOfChild, NoOfAdults, ServiceClass, ReservStatus, <CustID>)

Seats (SeatNo, <FlightId>, SeatClass, <ReservID>)

Aircraft (ACID, ACModel, ACMfdBy, ACMfdOn, ACStatus)

FlightSchedule (FlightID, EconomyPrice, BusinessPrice, FirstClassPrice, DepDateTime, ArrDateTime, FlightStatus, <RIID>, <ACID>)

RouteInfo (RIID, ArrCountry, DepCountry, ArrCity, DepCity, Region, Distance, EstDuration)

## **How the cloud database design meets the system requirements**

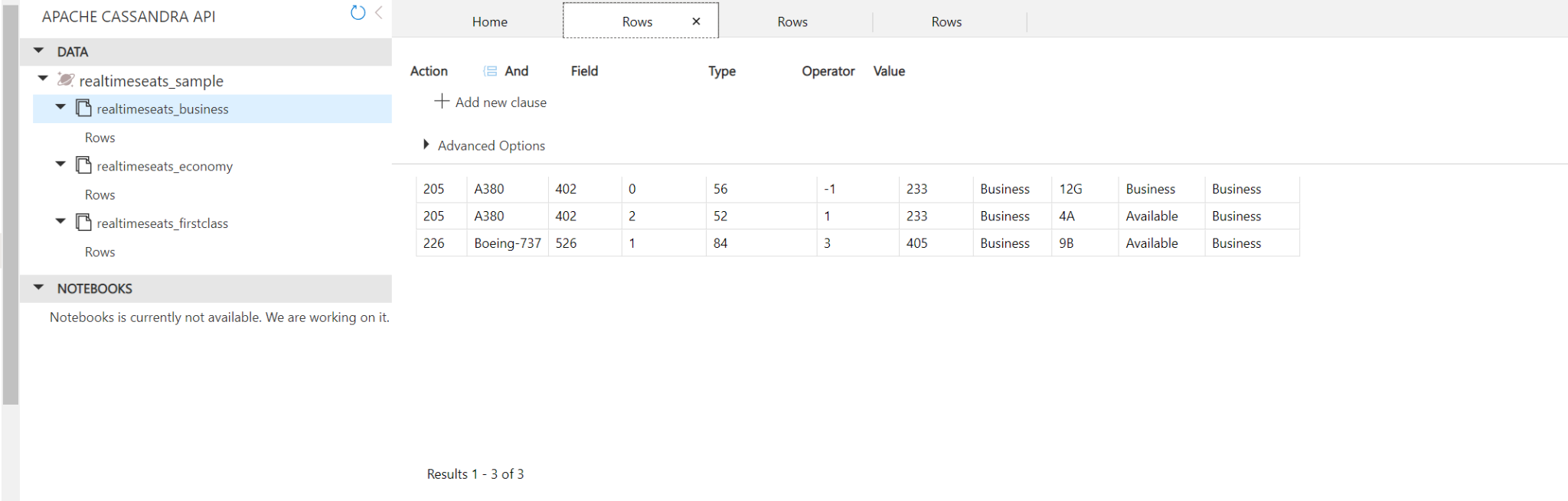
To create the system, we made use of MongoDB (Document DB) and Cassandra (Column-oriented DB) instead of relational databases. This helps the system handle a larger amount of data (scalability) and lessens the cost of the database. Furthermore, with denormalisation and data partitioning, we are able to increase the query performances and the overall scalability of the system.

Through our cloud database design, we were able to meet the following system requirements:

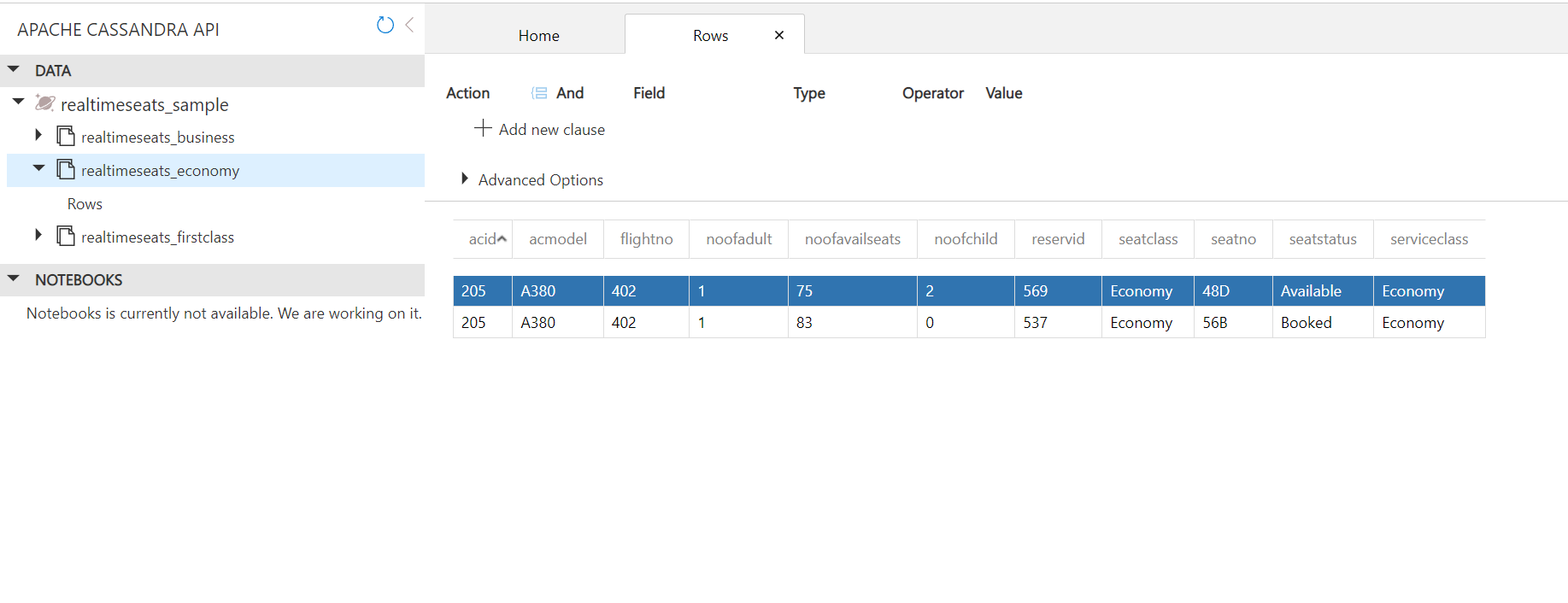
* Getting a real-time view of the available seats on a flight as the airline has to synchronise data from multiple locations periodically
* Getting flight schedules that fits the customer’s requirements
* Cutting high database costs (ie storage, communication)
* Optimal performance and accessibility of data
* Highly available, resilient, and scalable IT infrastructure to guarantee business continuity
* Flight schedule – to view information about the days and times of flights operated by the airline • Seat availability on a flight by service class (e.g. Economy, Business, First class)
* Fare quotes – a consolidated fare for an itinerary based on the flight, day, time, service class, and passenger type, (e.g., adult, child)
* Reservation of seats on a flight
* Ticket information – generating and storing tickets
* Cancellations and refunds – the cancellation of existing reservations and tickets
* Customer loyalty program – to reward frequent flyers to keep them loyal to the airline

## **Screenshots of implementation of cloud storage**

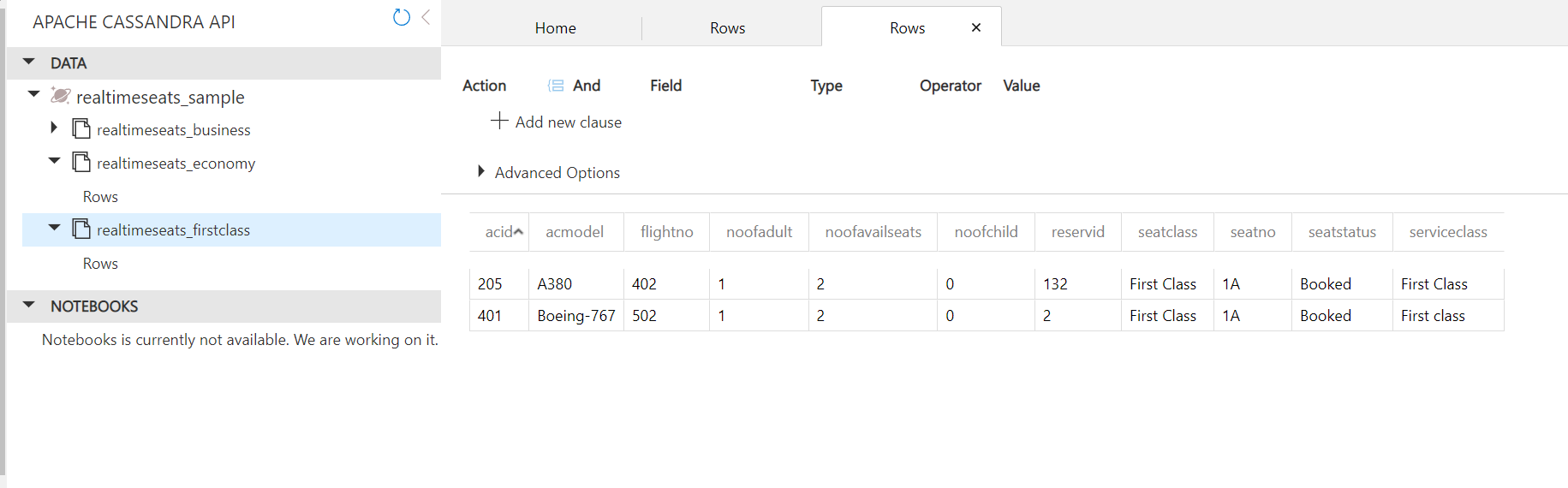
1. **Real-Time View of Seats (Column-Oriented Database) (By Ambrish)**
   1. **Business Class**



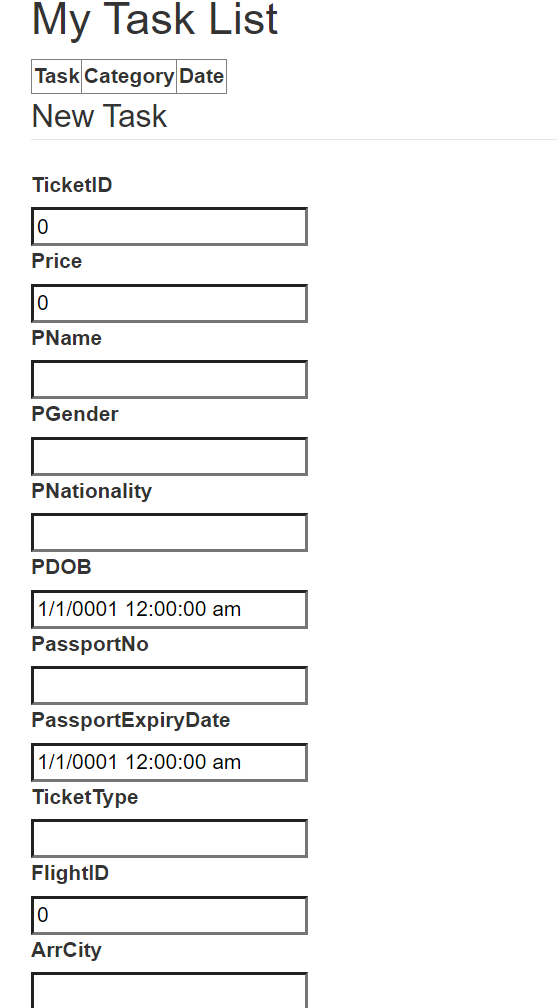
* 1. **Economy Class**



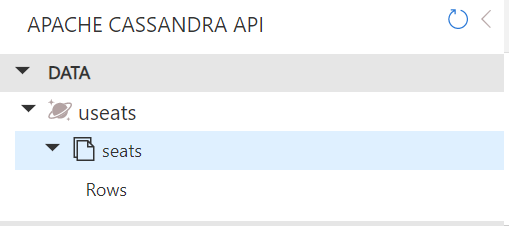
* 1. **First Class**



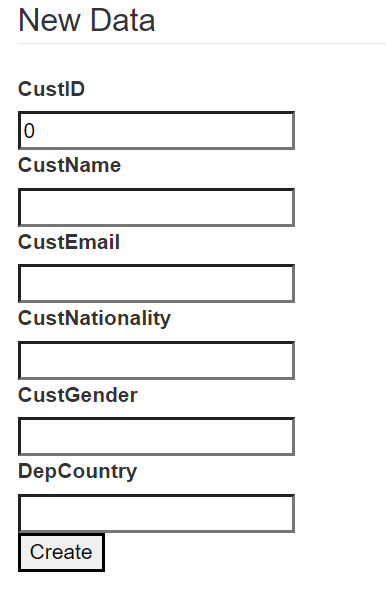
1. **PassengerDetails (MongoDB) (By Ambrish)**

****

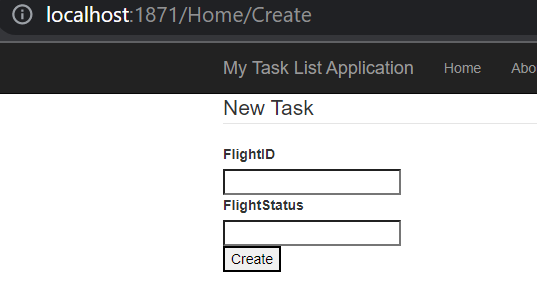
1. **Seats (Column-Oriented Database) (By Ambrish)**

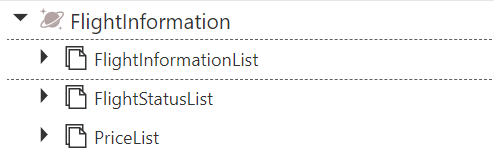


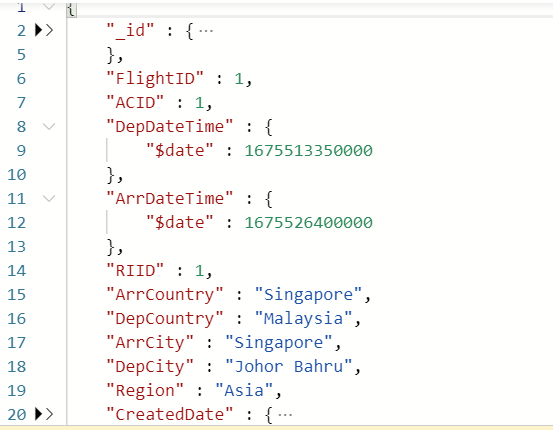
1. **Customer (MongoDB) - Hafeezur**



1. **FlightInformation (Denormalised Table under Kok Kai)**



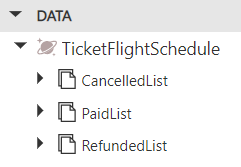








1. **TicketFlightSchedule (MongoDB) (Lim Wee Liang Kelven)**
   1. **TicketFlightSchedule Database**



* 1. **RefundedList**



* 1. **PaidList**



* 1. **CancelledList**



# **Kelven**

## **Denormalisation**

Query: Show the number of paid, cancelled and refunded tickets for a particular flight.

To execute this query, the Ticket table would have to be inner-joined twice with the Seats table and FlightSchedule table to show the number of paid, cancelled and refunded tickets for a particular flight. Since the ARTS would have thousands of data entries, this would severely reduce the system's performance. A new table can help improve performance by putting all relevant data into it. This reduces the need for JOINs and improves performance.

Analysing the ER Diagram, the tables Ticket, Seats, and FlightSchedule are connecting in order of Ticket → Seats → FlightSchedule. This creates an inefficiency if Tickets have to be referred to FlightSchedule often. To join the tables Ticket and FlightSchedule, FlightID can be replicated into Ticket as a foreign key.

|  |
| --- |
| ALTER TABLE Ticket ADD CONSTRAINT ticket\_flightschedule\_fk FOREIGN KEY (FlightID) REFERENCES FlightSchedule (FlightID) |

By adding FlightID as a foreign key to Ticket, a relationship can be formed between Ticket and FlightSchedule without having to join with Seats. But instead of doing a join between Ticket and FlightSchedule, a new table called Ticket\_FlightSchedule will be created.

The Ticket\_FlightSchedule table will consist of 5 columns:

1. TicketID - Unique ID for each ticket
2. PName - Passenger’s name
3. TicketType - Type of ticket of passenger (Child, Adult, Elderly)
4. TicketStatus - Status of ticket (Paid, Refunded, Cancelled)
5. FlightID - Unique ID for each FlightSchedule
6. SeatNo - Unique ID for each Seat

Code used to create the Ticket\_FlightSchedule table

|  |
| --- |
| *IF EXIST (SELECT \* FROM sysobjects WHERE NAME = 'Ticket\_FlightSchedule' AND TYPE = 'U') DROP TABLE Ticket\_FlightSchedule GO  SELECT t.TicketID, t.PName, t.TicketStatus, t.TicketType, fs.FlightID, t.SeatNo INTO Ticket\_FlightSchedule FROM Ticket t INNER JOIN FlightSchedule fs ON t.FlightID = fs.FlightID* |

If new data is created in the database, the SQL code below can be executed along with the other inserts.

|  |
| --- |
| INSERT INTO Ticket\_FlightSchedule column1, column2, ...  SELECT column1, column2, ...  FROM Ticket t INNER JOIN FlightSchedule fs *ON* t.FlightID = fs.FlightID |

## **Data placement strategy**

For the denormalised Ticket\_FlightSchedule table, TicketStatus (Paid, Cancelled, Refunded) can be selected as the PartitionKey and the table will be partitioned horizontally.

Ticket\_FlightSchedule

|  |  |  |
| --- | --- | --- |
| **TicketID** | **…** | **TicketStatus** |
| 1 | … | Paid |
| 2 | … | Cancelled |
| 3 | … | Refunded |
| 4 | … | Cancelled |
| 5 | … | Paid |

After partitioning based on TicketStatus, the Ticket partitions now look like this.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Ticket\_FlightSchedulePaid   |  |  |  | | --- | --- | --- | | **Ticket ID** | **…** | **TicketStatus** | | 1 | … | Paid | | 5 | … | Paid | | Ticket\_FlightScheduleCancelled   |  |  |  | | --- | --- | --- | | **Ticket ID** | **…** | **TicketStatus** | | 2 | … | Cancelled | | 4 | … | Cancelled | | Ticket\_FlightScheduleRefunded   |  |  |  | | --- | --- | --- | | **Ticket ID** | **…** | **TicketStatus** | | 3 | … | Refunded | |

Having over 1200 flights a week creates thousands more tickets. Thus, this partition is beneficial as queries that access only a fraction of the data can run faster because there are fewer data to scan. This reduces the response time to read and load data for particular SQL operations.

After partitioning the Ticket\_FlightSchedule table, the separate tables are JSON objects.

## **Selection of cloud storage**

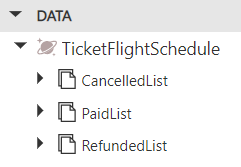
After the data partition step, there are now 3 new tables: Ticket\_FlightSchedulePaid, Ticket\_FlightScheduleCancelled, and Ticket\_FlightScheduleRefunded.

As the business expands globally, more data will be created and stored. A distributed database (DDB) like MongoDB can be used to store data across many countries.

MongoDB is consistent and partition tolerant, meaning some data may not be accessible, but the rest is still accurate. Using MongoDB, the system can continue to operate despite network issues, and operations will be complete even if individual components are unavailable. Using a DDB also distributes the workload to optimise processing power. As an airline, the data has a high velocity and volume, meaning the stored data will grow exponentially and big storage is needed.

MongoDB uses NoSQL. Thus, it is flexible and allows developers to update their fields and schemas if needed. NoSQL databases have more storage flexibility and offer better indexing methods. In a document database like MongoDB, each document is handled as an individual object, and there is no fixed schema, so each document is stored in the way it can be most easily retrieved and viewed. Additionally, your data model can evolve to adapt to changing application requirements. The schema versioning pattern utilises the flexible document model to allow just that.

## **Design and content of cloud storage**



The picture above shows the database, TicketFlightSchedule, which is based on the denormalised table. There are three different collections, CancelledList, PaidList, and RefundedList. They are the 3 partitioned tables shown previously. These lists utilise the same primary key, TicketID, but show different rows based on the TicketStatus (Paid, Cancelled, Refunded).

**RefundedList**



The picture above shows the content of a single document in RefundedList. TicketID, FlightID, and SeatNo are integer values. TicketType, TicketStatus, and PName are string values. Also, the TicketStatus is Refunded based on the status set before creating the MongoDB.

**PaidList**



The picture above shows the content for a single document in PaidList. It contains the same columns, but the row values are different.

**CancelledList**



The picture above shows the content for a single document in CancelledList. It contains the same columns, but the row values are different.

## **Business Queries**

Query: Show the number of paid, cancelled and refunded tickets for a particular flight.

**Before Denormalisation**

|  |
| --- |
| CREATE OR ALTER PROCEDURE uspTicketStatus(@flightid INT = NULL) AS BEGIN  IF (@flightid IS NULL)  SELECT 'FlightID must be provided. Please try again.'  ELSE  SELECT t.TicketStatus, COUNT(t.TicketStatus) as 'Number of TicketStatus'  FROM Ticket t INNER JOIN Seats s ON t.SeatNo = s.SeatNo INNER JOIN FlightSchedule fs ON s.FlightID = fs.FlightID   WHERE fs.FlightID = @flightid  GROUP BY t.TicketStatus END |

**After Denormalisation**

|  |
| --- |
| CREATE OR ALTER PROCEDURE uspTicketStatus(@flightid INT = NULL) AS BEGIN  IF (@flightid IS NULL)  SELECT 'FlightID must be provided. Please try again.'  ELSE  SELECT t.TicketStatus, COUNT(t.TicketStatus) as 'Number of TicketStatus'  FROM Ticket\_FlightSchedule  WHERE fs.FlightID = @flightid  GROUP BY t.TicketStatus END |

# **Kok Kai**

## **Denormalisation**

Query: Show available flight schedule for specific route information where there is enough seat capacity for the total number of people during a reservation in ascending order of depDateTime

Stored Procedure for the query:

|  |
| --- |
| Create or alter procedure uspFlightInfo(@TotalPeople int = NULL, @routeInfo int = NULL) as If (@TotalPeople is null and @routeInfo is null)  begin  Select \* from FlightSchedule  End  Else If (@TotalPeople is null)  begin  Select fs.\* from flightschedule fs inner join routeinformation ri on fs.RIID = ri.RIID where ri.RIID = @routeInfo  End  Else If (@routeInfo is null)  Begin  Select fs.\* from flightschedule fs inner join seats s on fs.flightId = s.FlightId   where @TotalPeople <= (select count(s2.SeatNo) from Seats s2 where s2.ReservId is null and s2.FlightId = fs.FlightId)  End  Else  begin  Select fs.\*, (select count(s2.SeatNo) from Seats s2 where s2.ReservId is null and s2.FlightId = fs.FlightId) as AvailableSeats   from FlightSchedule fs   inner join RouteInformation ri on fs.RIID = ri.RIID   where @TotalPeople <= AvailableSeats  and ri.RIID = @routeInfo and fs.FlightStatus = Scheduled  order by ri.depDateTime asc  end |

**Main query** to focus on from the stored procedure above:

|  |
| --- |
| Select fs.\*, (select count(s2.SeatNo) from Seats s2 where s2.ReservId is null and s2.FlightId = fs.FlightId) as AvailableSeats  from FlightSchedule fs  inner join RouteInformation ri on fs.RIID = ri.RIID  where @TotalPeople <= AvailableSeats and ri.RIID = @routeInfo and fs.FlightStatus = Scheduled order by ri.depDateTime asc |

This query is often used when customers want to reserve a flight. Before reserving a flight, they are required to input two data, total number of passengers (@TotalPeople) and the countries that they are departing from and arriving to (@routeInfo), this is so that the airline can filter the flight schedule to provide the flights that meets the customer’s requirements. Since every customer has to go through this query before the reservation, denormalization is a viable option because this has a high query rate and lower update rate.

Denormalization is a database design technique that breaks down a normalized database structure into a simpler, more redundant structure that improves query performance. The goal of denormalization is to reduce the number of joins required to retrieve data from the database, which will increase the speed and efficiency of queries.

Currently, there are three tables involved, FlightSchedule, Seats and RouteInformation. They can be denormalized into a single table called FlightInformation, which will contain information about the flight schedule, seat availability and the route information.

However, since Seats table will be updated frequently because whenever a reservation is booked, some seats would be taken and be updated to unavailable. It would not be wise to denormalize seats to prevent loss of integrity. FlightSchedule and RouteInformation tables are not updated frequently as they are already set beforehand and will not be affected during the booking system.

FlightInformation table will consist of these columns:

1. FlightId - A unique identifier for each flight schedule, starts from value 1 and incremented
2. ACID - Aircraft identifier, a foreign key that links flight information table to aircraft table
3. DepDateTime - The date and time for departure
4. ArrDateTime - The date and time for Arrival
5. EconomyPrice - Price per ticket for economy class
6. BusinessPrice - Price per ticket for business class
7. FirstClassPrice - Price per ticket for first class
8. FlightStatus - Flight status such as Scheduled, Delayed, Departed, In Air, Expected, Diverted, Recovery, Landed, Arrived, Cancelled
9. RIID - Route information identifier, a foreign key that links flight information table to route information table.
10. ArrCountry - The country for flight’s destination
11. DepCountry - The country for flight’s starting location
12. ArrCity - The city for flight’s destination
13. DepCity - The city for flight’s starting location
14. Region - The region for flight’s starting location

Code for creating FlightInformation table:

|  |
| --- |
| CREATE TABLE FlightInformation (  FlightId int NOT NULL Auto\_increment,  ACID varchar(12) Not Null,  DepDateTime DateTime Not null,  ArrDateTime DateTime Not Null,  EconomyPrice smallmoney not null,  BusinessPrice smallmoney not null,  FirstClassPrice smallmoney not null,  FlightStatus varchar(20) not null,   RIId int NOT NULL,  ArrCountry varchar(99) not null,  DepCountry varchar(99) not null,  ArrCity varchar(99) not null,  DepCity varchar(99) not null,  Region varchar(25) not null check (Region in(“Asia”, “America”, “Europe”, “Africa”)),  Constraint pk\_flightInformation PRIMARY KEY (FlightId)  Constraint fk\_fi\_acid foreign key (ACID) References Aircraft(ACID)  Constraint fk\_fi\_riid foreign key (RIID) References RouteInformation(RIID) ); |

Code for inserting data into FlightInformation table:

|  |
| --- |
| Insert into FlightInformation(FlightId, ACID, DepDateTime, ArrDateTime, EconomyPrice, BusinessPrice, FirstClassPrice, FlightStatus, RIId, ArrCountry, DepCountry, ArrCity, DepCity, Region) Select fs.\*, ri.ArrCountry, ri.DepCountry, ri.ArrCity, ri.DepCity ri.Region from FlightSchedule fs INNER JOIN RouteInformation ri ON fs.RIID = ri.RIID |

After denormalisation, the **main query** would be changed to:

|  |
| --- |
| Select fi.\*, (SELECT COUNT(s.SeatNo) FROM Seats s WHERE s.FlightId = fi.FlightId AND s.ReservId IS NULL) AS AvailableSeats from FlightInformation fi where fi.RIID = @routeInfo and AvailableSeats >= @TotalPeople and fs.FlightStatus = Scheduled order by DepDateTime asc |

In comparison, this new main query reduces the need for inner joins, which helps to improve the performance of the query and speed up retrieval of data. Furthermore, the denormalized table will have a low update rate after omitting the Seats table, making it a high query rate and low update rate.

Sample data:

FlightSchedule

|  |  |  |  |
| --- | --- | --- | --- |
| **FlightID** | **…** | **DepDatetime** | **RIID** |
| 1 | … | Scheduled | 3 |
| 2 | … | Cancelled | 1 |
| 3 | … | Scheduled | 2 |

RouteInformation

|  |  |  |
| --- | --- | --- |
| **RIID** | **…** | **Region** |
| 1 | … | Asia |
| 2 | … | America |
| 3 | … | Europe |

After denormalization is done:

FlightInformation

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **FlightID** | **…** | **FlightStatus** | **RIID** | **…** | **Region** |
| 1 | … | Scheduled | 3 | … | Europe |
| 2 | … | Cancelled | 1 | … | Asia |
| 3 | … | Scheduled | 2 | … | America |

## **Data placement strategy**

Out of all the data placement alternatives, I chose the hybrid alternative because it is designed to optimise data placement where we can take advantage of the benefits of other alternatives while limiting the disadvantages.

FlightInformation table can use horizontal partitioning to distribute flight schedules across different databases based on geographical location. For example, based on the Region attribute, we can separate flight schedules using regions such that for flights departing from Europe will be stored in one database, while flights departing from Asia will be stored in another database. This would reduce the size of each database, improving query performance for users in different regions and increase overall system scalability.

We did horizontal partitioning using region instead of the country because regions usually contain multiple countries while countries are distinct geographical entities. Furthermore, there are only four regions whereas there are 195 countries in the world. If we used countries for horizontal partitioning, we would need to have 195 separate databases. This makes the region a more scalable and flexible option.

Next, we can use vertical partitioning to separate columns such as prices and flightStatus as they may be frequently used in different queries such as “The total price required to pay for this reservation” and “Show flights that are currently delayed”. By storing prices and flightStatus in separate databases or tables, query performance will be improved by reducing the amount of data that needs to be processed.

Example:

This is the original table.

FlightInformation

|  |  |  |
| --- | --- | --- |
| **FlightID** | **…** | **Region** |
| 1 | … | Asia |
| 2 | … | America |
| 3 | … | Europe |
| 4 | … | Asia |
| 5 | … | Africa |

The table will then be separated into different databases based on their region.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| FlightInformationAsia   |  |  |  | | --- | --- | --- | | **FlightID** | **…** | **Region** | | 1 | … | Asia | | 4 | … | Asia | | FlightInformationAmerica   |  |  |  | | --- | --- | --- | | **FlightID** | **…** | **Region** | | 2 | … | America | | FlightInformationEurope   |  |  |  | | --- | --- | --- | | **FlightID** | **…** | **Region** | | 3 | … | Europe |   FlightInformationAfrica   |  |  |  | | --- | --- | --- | | **FlightID** | **…** | **Region** | | 5 | … | Africa | |

Afterwards, vertical partitioning will be done on each of the databases to separate prices and flight status; they are separated tables instead of databases. Only the Asia database will be shown to represent the steps taken for vertical partitioning.

FlightInformationAsia

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **FlightID** | **…** | **Region** | **EconomyPrice** | **BusinessPrice** | **FirstClassPrice** | **FlightStatus** |
| 1 | … | Asia | 200 | 300 | 500 | Scheduled |
| 4 | … | Asia | 220 | 350 | 600 | Cancelled |

After vertical partitioning:

FlightInformationAsia

|  |  |  |
| --- | --- | --- |
| **FlightID** | **…** | **Region** |
| 1 | … | Asia |
| 4 | … | Asia |

PriceAsia

|  |  |  |  |
| --- | --- | --- | --- |
| **FlightID** | **EconomyPrice** | **BusinessPrice** | **FirstClassPrice** |
| 1 | 200 | 300 | 500 |
| 4 | 220 | 350 | 600 |

FlightStatusAsia

|  |  |
| --- | --- |
| **FlightID** | **FlightStatus** |
| 1 | Scheduled |
| 4 | Cancelled |

## **Selection of cloud storage**

After denormalisation and data partitioning, there are four tables(Seats, FlightInformation, Price, FlightStatus) for our query. Using polyglot persistence, which is the usage of multiple databases in a system, we will store the data in the database best suited for each table.

1. Seats Table

Since the Seats table are often used to count the number of available seats and will have a large amount of data since seats are for each flight schedule, it is best to use a column-oriented database. This is because column-oriented databases perform particularly well on aggregation queries as they often only require data from a subset of columns.

Furthermore, column-oriented databases provide scalability by handling large amounts of data and can be horizontally scaled across multiple nodes.

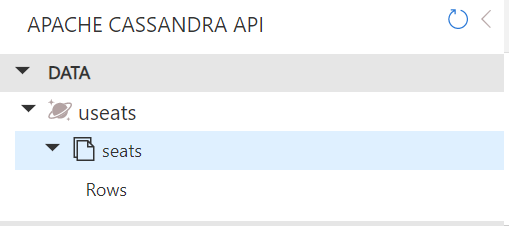
In comparison to relational databases, column-oriented databases are more cost-effective for large-scale data storage and retrieval options, which results in less I/O operations, reduced storage costs, and faster query time. However, column-oriented databases lose when it comes to transactional consistency. To improve transactional consistency, queue storage can be used to implement a task-based architecture where worker services can process tasks in a queue in a reliable and ordered manner.

1. FlightInformation, Price, FlightStatus

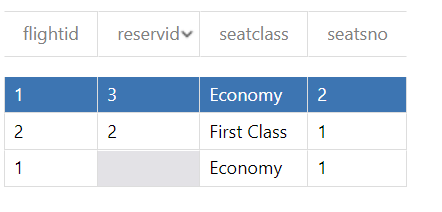
I have decided to use a document database (MongoDB) for the FlightInformation, Price and FlightStatus tables. This is because an airline reservation system is bound to contain a large amount of data, and MongoDB is designed to handle big data and accommodate growth by scaling horizontally. MongoDB is also useful for a wide variety of applications due to the flexibility of the model, and lastly, it can be cost-effective compared to other databases by using a binary JSON format for data storage, which is more compact and efficient than relational databases, reducing storage costs. It can also reduce costs by deploying in a serverless infrastructure, which means that we only pay for what we use.

## **Design and content of cloud storage**

1. Column-Oriented Database

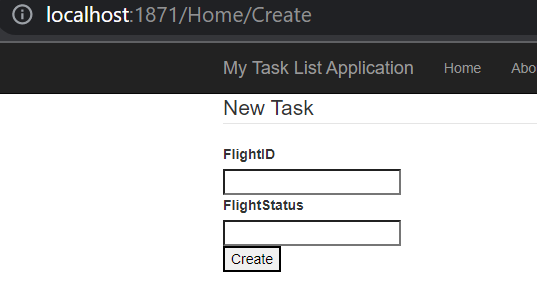


This is the creation of keyspace(useats) and column family(seats).

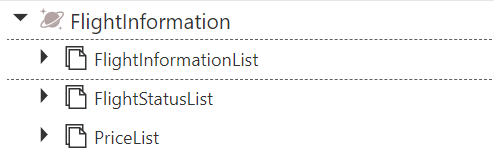


The above picture shows the sample data. Seatsno, flightid and reservid are integer values, with flightid and reservid being referenced from FlightInformation and Reservation table. Seatclass is a string value with constraints that it can only show first class, economy and business. As seen from the column family, seatsno itself cannot be a primary key, it requires flightid to make sure that each column is unique. Lastly, reservid shows a null value, which represents that the seat is available. Whereas if a reservid is present, it represents that the seat is booked. This will help when using the count() aggregate function for the total number of available seats in a flight schedule.

1. Document Database

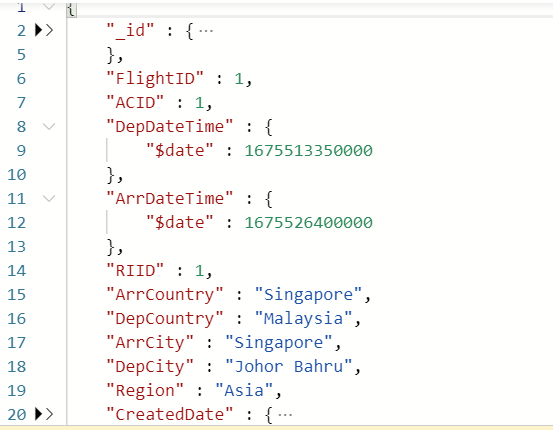


This picture shows an example of me creating a collection and document using visual studio. By manipulating the Dal.cs, TaskModels.cs, Index.cshtml and Create.cshtml, I managed to create a local host that asks for different values based on the collection name.



The picture above shows the database(FlightInformation) with three different collections(FlightInformationList, FlightStatusList, PriceList). The relations between these collections are that they utilise the same primary key(FlightID) to differentiate each document. Essentially, they belong to the same row from the previous denormalized table, just showing different columns from the previous data partitioning to improve query performances in general.

* FlightInformationList



This picture shows the content of a single document in FlightInformationList. FlightID, ACID and RIID are integer values. ArrCountry, DepCountry, ArrCity, DepCity and Region are string values. DepDateTime and ArrDateTime are datetime values stored as a BSON Date type, which is a 64-bit integer, this helps to reduce storage as compared to storing the same information in a human-readable format. Also, the region is Asia based on the location set before creating the MongoDB.

* FlightStatusList



This picture shows the content for a single document in FlightStatusList. It contains FlightID which is an integer value, and FlightStatus which is a string value.

* PriceList



This shows the content of a single document in PriceList. FlightID is an integer value. While EconomyPrice, BusinessPrice and FirstClassPrice are float values.

## **Business Queries**

Query: Show available flight schedule for specific route information where there is enough seat capacity for the total number of people during a reservation in ascending order of depDateTime

Since Cassandra and MongoDB do not support SQL fully, we would utilise LINQ to reconstruct the query.

In the case of the specific route being RIID = 1 (Singapore to Malaysia), we need to first retrieve the total number of available seats in Cassandra:

|  |
| --- |
| var cql = "SELECT FlightID, COUNT(\*) as available\_seats FROM Seats WHERE RIID = ? AND reservid IS NULL GROUP BY FlightID"; var seats = session.Execute(cql, new object[] { 1 }).First(); var availableSeats = seats.ToDictionary(s => s.GetValue<string>("flightid"), s => s.GetValue<int>("available\_seats")); |

Next, retrieve the flight information filtered by availableSeats and sorted by datetime:

|  |
| --- |
| var flights = collection.AsQueryable()  .Where(f => f.RIID == 1 &&   f.AvailableSeats >= availableSeats &&   f.FlightID == seats.GetValue("flightid"))  .OrderBy(f => f.DepartureDateTime)  .ToList(); |

Other business queries can be:

“Retrieve all available flight schedules

# **Hafeezur**

## **Denormalisation**

Query: Show customer details for all flights departing from Singapore in the year 2022 in ascending order of Customer Id.

In order to improve the performance of a relational database, denormalization is introduced to the database. This process involves introducing pre-computed data to the database. It can be performed in two scenarios. The first involves poor system performance, while the second involves a high query rate and low update rate.

Tables: Customer, FlightSchedule, RouteInformation, Reservation

Based on the query the customer table has to be the inner join reservation table on CustId. The reservation table then has to inner join the flight schedule table on FlightNo. Lastly, the flight schedule table should inner join the RouteInfo table on RIID. Since the ARTS would have thousands of data this would drastically reduce the system's performance. To avoid this issue a new table that contains all the data have to be created to reduce the number of inner joins.

Analysing the ER diagram, we need to inner join customer → Reservation → Flight Schedule → Route Info. To reduce the inner join, we could replicate CustId from the customer table to Flight Schedule as a foreign key. By replicating the new flow will be Customer → Flight Schedule → Route Information.

|  |
| --- |
| ALTER TABLE FlightSchedule ADD CONSTRAINT customer\_flightschedule\_fk FOREIGN KEY (CustId) References Customer (CustId) |

After altering the Flight Schedule table add CustID as a foreign key. Next is to create a table that stores all the data. The table created was Customer\_FlightSchedule\_RouteInformation Table. The table consists of 6 columns:

1. Customer ID
2. Customer Name
3. Customer Email
4. Customer Nationality
5. Customer Gender
6. Departing Country

|  |
| --- |
| IF EXISTS (SELECT \* FROM sysobjects WHERE NAME = 'Customer\_FlightShecdule\_RouteInformation') DROP TABLE Customer\_FlightShecdule\_RouteInformation GO Select c.CustID, c.CustName , c.CustEmail, c.CustNationality, c.CustGender, ri.DepCountry INTO Customer\_FlightShecdule  from Customer c inner join FlightSchedule fs on c.custid = fs.custid inner join RouteInformation ri on fs.RIID = ri.RIID WHERE ri.DepDatetime BETWEEN 01-01-2022 AND 31-12-2022 WHERE ri.DepCountry = 'Singapore' ORDER BY c.CustID ASC |

Inserting data into the new table created.

|  |
| --- |
| INSERT INTO Customer\_FlightShecdule\_RouteInformation (CustID, CustName, CustEmail, CustNationality, CustGender) Select c.CustID, c.CustName, c.CustEmail, c.CustNationality, c.CustGender, ri.DepCountry From Customer c inner join FlightSchedule fs on c.custid = fs.custid inner join RouteInformation ri on ri.RIID=fs.RIID WHERE ri.DepCountry = 'Singapore' WHERE ri.DepDatetime BETWEEN 01-01-2022 AND 31-12-2022 ORDER BY c.CustID ASC |

## **Data placement strategy**

As shown in the denormalisation portion CustID is replicated in FlightSchedule Table. Customers do not have to check their Flightschedule through the reservation table. Customers will be able to see their flight schedule from the FlightSchedule table directly. Similarly, the airlines will be able to see the passenger or customer flight schedules easily. After creating the new table Customer\_FlightShecdule\_RouteInformation The amount of inner join is reduced. The system will be more efficient and faster after denormalising.

In the new denormalised table, we can partition CustID, CustName, and DepCountry can be partitioned. The columns will be partitioned vertically. CustID can be viewed with CustName and CustID can also be viewed with DepCountry too.

Customer\_FlightSchedule\_RouteInformation

|  |  |  |  |
| --- | --- | --- | --- |
| **CustID** | **CustName** | **…** | **DepCountry** |
| 1 | Jack | … | Singapore |
| 2 | Bob | … | Japan |
| 3 | Samuel | … | Malaysia |
| 4 | Zack | … | Singapore |
| 5 | Jhon | … | Australia |

|  |  |  |
| --- | --- | --- |
| **CustID** | **CustName** | **…** |
| 1 | Jack | … |
| 2 | Bob | … |
| 3 | Samuel | … |
| 4 | Zack | … |
| 5 | Jhon | … |

|  |  |  |
| --- | --- | --- |
| **CustID** | **…** | **DepCountry** |
| 1 | … | Singapore |
| 2 | … | Japan |
| 3 | … | Malaysia |
| 4 | … | Singapore |
| 5 | … | Australia |

## **Selection of cloud storage**

Cloud Storage Used: Document Storage (MongoDB)

The suite of data products from MongoDB can help build applications that can handle complex data sets. Its database-as-a-service platform, known as Atlas, can be used to serve any type of workload. Its ability to connect to other applications through its API, as well as bidirectional sync between its components, makes it ideal for building applications that run on mobile devices and Edge.

I chose MongoDB due to its ability to store and retrieve related data within a single document. This is very important for developing applications that can take advantage of its built-in features. It allows them to perform various tasks such as searching and analysing data. I opted for a NoSql technique due to its ability to handle large datasets at high speed. It also allows developers to easily update their fields and schemas. MongoDB is an embedded data type. It allows developers to store and retrieve related data within a single document. This is very important for developing applications that can take advantage of its built-in features. Its ability to embed multiple document structures within a single document allows them to perform various tasks.

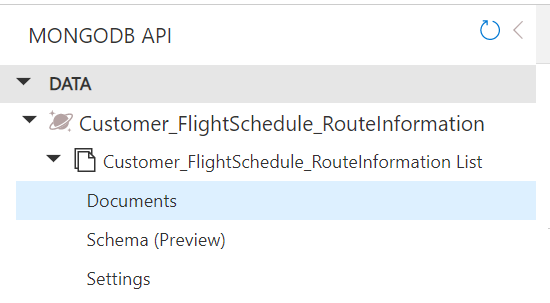
As ARTS wants to achieve an efficient and trustable system. ARTS is also dealing with thousands of data. The Cloud storage I chose for ARTS was MongoDB. NoSql is the best method for handling large amount of data compared to SQL. NoSql is also cheaper method. This allow airlines to save money and spent it on other important factors. Storing the data using MongoDB storage is the best option for ARTS.

After Denormalising we are left with Customer, FlightSchedule, and RouteInformation table.

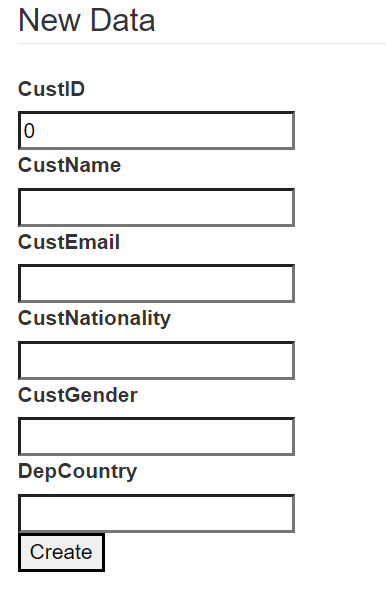
## **Design and content of cloud storage**

Document Storage (MongoDB)

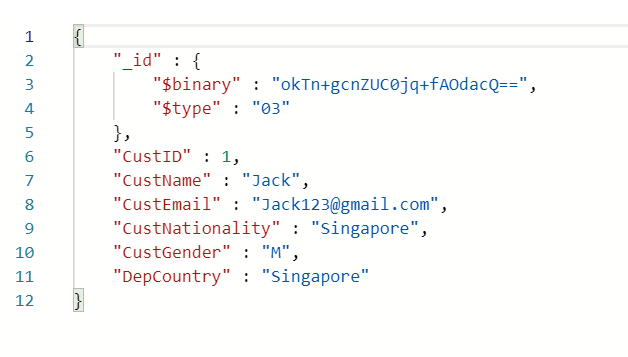
This is the design and content of the cloud storage using MongoDB. Below is the database name Customer\_FlightSchedule\_RouteInformation. I created a collection to input all the data from the denormalised table. The collection is called Customer\_FligtSchedule\_RouteInformation List.



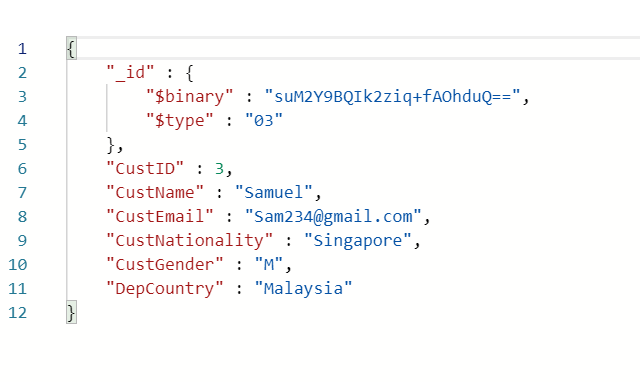
Below is the localhost where the data can be typed in. This is the collection that I created. I changed the code to suite the Assg2 context. The files I updated was Index.cshtml, Create.cshtml, TaskModel.cs, and Dal.cs. I change the attributes to match the new denormalised table attribute.



Below are the sample data. There are five sample data inserted to the database.

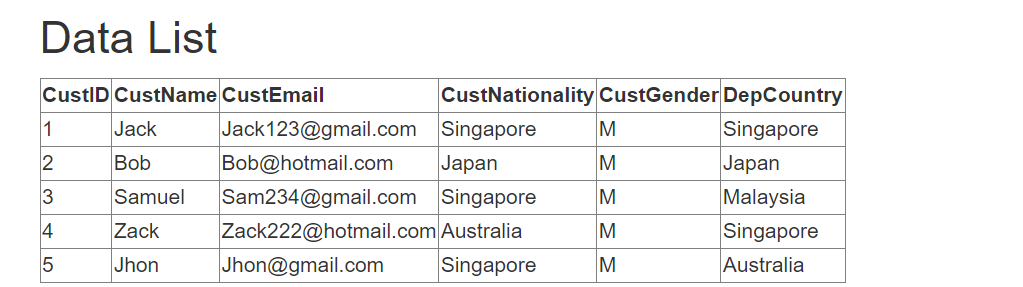








  
Below is the new denormalised table with all the sample data.



## **Business Queries**

Query: Show customer details for all flights departing from Singapore in the year of 2022 in ascending order of Customer Id.

**Before Denormalisation**

Customer (CustID, CustName, CustEmail, CustDOB, CustGender, CustNationality, Points)

Reservation (ReservID, ReservStatus, NoOfChild, NoOfAdult, ServiceClass, <CustID>, <FlightNo>, <ACID>)

FlightSchedule (FlightNo, <ACID>, DepDateTime, EconomyPrice, BusinessPrice,

FirstClassPrice, FlightStatus, <RIID>)

RouteInformation (RIID, DepCity, ArrCity, ArrCountry, DepCountry, Distance, EstDuration)

|  |
| --- |
| Select c.CustID, c.CustName, c.CustEmail, c.CustNationality, c.CustGender, ri.DepCountry From Customer c inner join reservation r on c.CustID = t. CustID inner join FlightSchedule fs on r.Flightno=fs.Flightno inner join RouteInfomation ri on fs.RIID = ri.RIID WHERE ri.DepCountry = 'Singapore' WHERE ri.DepDatetime BETWEEN 01-01-2022 AND 31-12-2022 ORDER BY c.CustID ASC |

**After Denormalisation**

Customer (CustID, CustName, CustEmail, CustDOB, CustGender, CustNationality, Points)

FlightSchedule (FlightNo, <ACID>, DepDateTime, EconomyPrice, BusinessPrice, FirstClassPrice, FlightStatus, <RIID>, <CustID>)

RouteInformation (RIID, DepCity, ArrCity, ArrCountry, DepCountry, Distance, EstDuration)

|  |
| --- |
| Select CustID AS 'Customer Id', CustName AS 'Customer Name', CustEmail AS 'Customer Email', CustNationality AS 'Customer Nationality', CustGender AS 'Customer Gender', DepCountry AS 'Departing Country' From Customer\_FlightShecdule\_RouteInformation WHERE DepCountry = 'Singapore' WHERE ri.DepDatetime BETWEEN 01-01-2022 AND 31-12-2022 ORDER BY CustID ASC |

# **Ambrish**

## **Denormalisation**

**1st Query: Show a real-time view of the available seats departing from Singapore to Dubai**

This query is essential as the customers will want to select which seats they want to reserve when booking their flight to Dubai from Singapore. In order for this to be achieved, the ARTS will need to query the database and receive a quick and accurate response. Moreover, another reason the speed of the query is an extremely important factor is due to the fact that there will be multiple concurrent customers who will be accessing the ARTS at the same time.

Denormalisation refers to introducing precomputed redundant data to a relational database that has otherwise been normalised beforehand to enhance read performance. Denormalisation is used in 2 scenarios. The first scenario involves poor system performance with the second involving a relation having a low update rate and a high query rate.

In order to solve this query, the following table will need to be used:

* Aircraft
* Seats
* FlightSchedule
* Reservation

As these entities will constantly be referred to one another, the tables would need to be inner joined to solve the query.

However, as the ARTS contains several data entries, the system performance will not be up to standard, which would compromise the speed of the Passenger processing. Thus, Denormalisation needs to be done. By creating a new table, we will be able to improve the system performance as the number of JOINs present in the query will reduce significantly, hence leading to a high update and query rate.

Firstly, we will replicate the FlightID, SeatNo and ACID attributes into Reservation, before creating our new table.

|  |
| --- |
| ALTER TABLE Reservation ADD CONSTRAINT reservation\_flightschedule\_fk FOREIGN KEY (FlightID) REFERENCES FlightSchedule (FlightID) |

|  |
| --- |
| ALTER TABLE Reservation ADD CONSTRAINT reservation\_seats\_fk FOREIGN KEY (SeatNo) REFERENCES Seats(SeatNo) |

|  |
| --- |
| ALTER TABLE Reservation ADD CONSTRAINT reservation\_aircraft\_fk FOREIGN KEY (ACID) REFERENCES Aircraft(ACID) |

As we have now successfully completed the Replication process, we can now move on to creating our new table that will be used henceforth. The new table is called RealTimeSeats, which consists of a total of 10 columns. The 10 columns are as follows:

1. ReservID - Unique ID for each reservation
2. ACID - Unique ID for each aircraft
3. ACModel - Name of aircraft model
4. FlightNo - Unique ID for each flight
5. ServiceClass - The service class that was booked consists of First class, business and economy
6. SeatNo - Unique ID for each seat
7. SeatClass - Service class where the seat is located
8. SeatStatus - Service class where the seat is located
9. NoOfChild - Number of children in the booking
10. NoOfAdult - Number of adults in the booking
11. NoAvailSeats - Number of available seats remaining

|  |
| --- |
| CREATE TABLE RealTimeSeats( ReservID INT NOT NULL ACID INT NOT NULL ACModel VARCHAR(99) NOT NULL FlightNo INT NOT NULL ServiceClass VARCHAR(20) NOT NULL SeatNo VARCHAR(3) NOT NULL SeatClass VARCHAR(11) NOT NULL SeatStatus VARCHAR(9) NOT NULL NoOfChild TINYINT NOT NULL NoOfAdult TINYINT NOT NULL  NoAvailSeats SMALLINT NOT NULL Primary Key(ReservID, ACID, FlightID, SeatNo) ); |

After creating the table, we can now insert the data into the table using the following code.

|  |
| --- |
| INSERT INTO RealTimeSeats(ReservID, ACID, ACModel, FlightNo, ServiceClass, SeatNo, SeatClass, SeatStatus, NoOfChild, NoOfAdult, NoAvailSeats) SELECT r.ReservID, r.ACID, a.ACModel, r.FlightNo, r.ServiceClass, r.SeatNo, s.SeatClass, SeatStatus, r.NoOfChild, r.NoOfAdult, (SELECT COUNT(r.SeatNo) FROM Reservation r WHERE r.FlightId = fi.FlightId AND r.ReservId IS NULL) INTO RealTimeSeats FROM Reservation r INNER JOIN FlightSchedule fs ON r.FlightID = fs.FlightID INNER JOIN Aircraft a ON r.ACID = a.ACID INNER JOIN Seats s ON r.SeatNo = s.SeatNo WHERE r.DepCountry = 'Singapore', r.ArrCity = 'Dubai' |

**2nd Query: Show the information of passengers departing from Singapore to London**

This query is essential as it will be used for processing passenger profiles and clearance at the Immigration counter which is essential to check if the passengers' passports and visas are valid while also performing biometric and background checks to verify their identity. Therefore, it is mandatory for ARTS to process queries quickly and efficiently.

The RouteInformation and FlightSchedule tables would need to be inner-joined twice with the Ticket table in order to show this query. However, as the ARTS contains several data entries, the system performance will not be up to standard, which would compromise the speed of the Passenger processing. Thus, Denormalisation needs to be done. By creating a new table, we will be able to improve the system performance as the number of JOINs present in the query will reduce significantly, hence leading to a high update and query rate.

After analysing the ER Diagram that my group has come up with, I have realised that the Tickets, FlightSchedule and RouteInformation tables are connected in the following manner.

Tickets → Seats → FlightSchedule → RouteInformation

As it would be inefficient to constantly refer to the FlightSchedule table when referring to the Tickets table as well as the RouteInformation table, the FlightID column can be replicated into the Ticket and RouteInformation table through the means of a foreign key.

|  |
| --- |
| ALTER TABLE Ticket ADD CONSTRAINT ticket\_flightschedule\_fk FOREIGN KEY (FlightID) REFERENCES FlightSchedule (FlightID) |

|  |
| --- |
| ALTER TABLE RouteInformation ADD CONSTRAINT route\_flightschedule\_fk FOREIGN KEY(FlightID) REFERENCES FlightSchedule (FlightID) |

After replicating the FlightID column into the tables Ticket and RouteInformation, we can now create a new table called PassengerDetails. It consists of 16 columns as listed below.

1. TicketID - Unique ID for each ticket
2. Price - Price of the ticket
3. PName - Name of Passenger
4. PNationality - Nationality of Passenger
5. PGender - Gender of Passenger
6. PDOB - Date of Birth of Passenger
7. PassportNo - Unique ID of Passport
8. PassportExpiryDate - Expiry Date of Passport
9. TicketType - Type of ticket of passenger
10. FlightID - Unique ID for each flight
11. ArrCity - City of Arrival
12. ArrCountry - Country of Arrival
13. DepCity - City of Departure
14. DepCountry - Country of Departure
15. DepDateTime - Departure Date and time
16. ArrDateTime - Arrival Date and time

Here is the code for creating the PassengerDetailsTable:

|  |
| --- |
| CREATE TABLE PassengerDetails ( TicketID INT NOT NULL Price SMALLMONEY NOT NULL PName VARCHAR(99) NOT NULL PNationality VARCHAR(99) NOT NULL PGender CHAR(1) NOT NULL PDOB DATETIME NOT NULL PassportNo CHAR(8) NOT NULL PassportExpiryDate DATETIME NOT NULL TicketType CHAR(7) NOT NULL FlightID INT NOT NULL ArrCity VARCHAR(99) NOT NULL ArrCountry VARCHAR(99) NOT NULL DepCity VARCHAR(99) NOT NULL DepCountry VARCHAR(99) NOT NULL DepDateTime DATETIME NOT NULL ArrDateTime DATETIME NOT NULL Primary Key(TicketID, FlightID) ); |

Below is the code for the insertion of data into the new table created:

|  |
| --- |
| INSERT INTO PassengerDetails(TicketID, Price, PName, PNationality, PGender, PDOB, PassportNo, PassportExpiryDate, TicketType, FlightID, ArrCity, ArrCountry, DepCity, DepCountry, DepDateTime, ArrDateTime) SELECT t.TicketID, t.Price, t.PName, t.PNationality, t.PGender, t.PDOB, t.PassportNo, t.PassportExpiryDate, t.TicketType, fs.FlightID, r.ArrCity, r.ArrCountry, r.DepCity, r.DepCountry, fs.DepDateTime, fs.ArrDateTime INTO Ticket\_RouteInfo\_FlightSchedule FROM Ticket t INNER JOIN FlightSchedule fs ON t.FlightID = fs.FlightID FROM RouteInformation r INNER JOIN FlightSchedule fs ON r.FlightID = fs.FlightID WHERE r.DepCountry = 'Singapore', r.ArrCity = 'London' |

## **Data placement strategy**

**1st Query**

One of the system requirements of the ARTS is to provide a real time view of the available seats on a certain flight. For this to be achieved, the RealTimeSeats table can be partitioned according to the Service class chosen by each customer, to make it easier for the customers to know which seats they are allowed to book based on their chosen service class. Thus, the RealTimeSeats table will be partitioned horizontally according to the Service Class booked by the passenger.

RealTimeSeats

|  |  |  |
| --- | --- | --- |
| **ReservID** | **…** | **ServiceClass** |
| 1 | … | Economy |
| 2 | … | Business |
| 3 | … | Business |
| 4 | … | First Class |
| 5 | … | Economy |

After partitioning based on TicketType, the Ticket table now looks like this.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| RealTimeSeatsEconomy   |  |  |  | | --- | --- | --- | | **ReservID** | **…** | **ServiceClass** | | 1 | … | Economy | | 5 | … | Economy | | RealTimeSeatsBusiness   |  |  |  | | --- | --- | --- | | **ReservID** | **…** | **ServiceClass** | | 2 | … | Business | | 3 | … | Business | | RealTimeSeatsFirstClass   |  |  |  | | --- | --- | --- | | **ReservID** | **…** | **ServiceClass** | | 4 | … | First Class | |

This will be more beneficial to the customers as it makes it easier for them to identify the area of seats where they are allowed to choose and book, making their experiences with the ARTS more convenient. It’s also beneficial to the airline as it allows for a higher read and query rate

**2nd Query**

One of the system requirements of the ARTS is to support all the new ticketing offices and Airport Operations in other countries. In order for this to be achieved, the PassengerDetails table needs to be partitioned according to the nationality of each passenger so that the passengers can be separated according to their respective nationality for easier processing. The BasicDetails table will be partitioned horizontally according to the nationality of the passenger so that all the details of each passenger will be distributed together according to their respective nationality.

PassengerDetails

|  |  |  |
| --- | --- | --- |
| **TicketID** | **…** | **PNationality** |
| 1 | … | Singapore |
| 2 | … | Australia |
| 3 | … | England |
| 4 | … | Singapore |
| 5 | … | Australia |

After partitioning based on TicketType, the Ticket table now looks like this.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| PassengerDetailsSingapore   |  |  |  | | --- | --- | --- | | **Ticket ID** | **…** | **PNationality** | | 1 | … | Singapore | | 4 | … | Singapore | | PassengerDetailsAustralia   |  |  |  | | --- | --- | --- | | **Ticket ID** | **…** | **PNationality** | | 2 | … | Australia | | 5 | … | Australia | | PassengerDetailsEngland   |  |  |  | | --- | --- | --- | | **Ticket ID** | **…** | **PNationality** | | 3 | … | England | |

We can then further partition each table derived according to the FlightID of the flight boarded by each passenger. I will demonstrate what it looks like by conducting this horizontal partition on the PassengerDetailsSingapore table.

PassengerDetailsSingapore\_526

|  |  |  |  |
| --- | --- | --- | --- |
| **Ticket ID** | **…** | **FlightID** | **PNationality** |
| 1 | … | 526 | Singapore |

PassengerDetailsSingapore\_402

|  |  |  |  |
| --- | --- | --- | --- |
| **Ticket ID** | **…** | **FlightID** | **PNationality** |
| 4 | … | 402 | Singapore |

This can be beneficial to the airport staff if there are separate applications according to the passengers’ respective nations and flights so that the procedures set for immigrants from each nation can be dealt with at a faster and more effective rate. Moreover, it is also beneficial for the passengers as it allows them to complete their immigration more quicker.

## **Selection of cloud storage**

**Query 1:**

Cloud Storage Used: Column-Oriented Database (Cassandra)

As the 3 tables RealTimeSeatsEconomy, RealTimeSeatsBusiness and RealTimeSeatsFirstClass make use of aggregation functions which are often used to calculate the number of available seats in each particular flight schedule, it would make the utmost sense to use a Column-Oriented Database for these tables.

In a Column-Oriented database, the columns are divided into groups known as column families. Column families are groups of data that are often accessed together, with each family holding a set of columns that are logically related together and are typically retrieved or manipulated as a unit. The data is also stored as rows under each column family.

We can use a Column-Oriented database such as Cassandra for this scenario as it helps to hold a very large number of columns. As the Cassandra database will be flexible enough to hold the various data in the 3 tables that are updated for each flight schedule, Cassandra would be the best option.

The data for the seats in each service class would be stored in their respective tables and updated in real-time as and when the seats get reserved for each flight schedule. To retrieve the real-time view of the seats, the database would use the service class of the requested seats as the partition key to query the corresponding table and retrieve the information on the available seats. This way, the database can scale easily and maintain high performance even with a large amount of data as the data is partitioned and stored in separate tables based on the service class.

**Query 2:**

Cloud Storage Used: Document Storage (MongoDB)

Firstly, as the ARTS wants to achieve a highly available, resilient, and scalable IT infrastructure to guarantee business continuity, we will use a NoSQL database as it focuses on providing these three benefits. Moreover, as the airline wants to cut down on high database costs, NoSQL helps achieve that aim as well, considering it is cheap and easy to implement. NoSQL databases also do not require schemas in advance and allow the data to be replicated into multiple nodes while also allowing partitioning. Therefore, a NoSQL database is the perfect choice for this ARTS as it allows for improved performance at a lower cost.

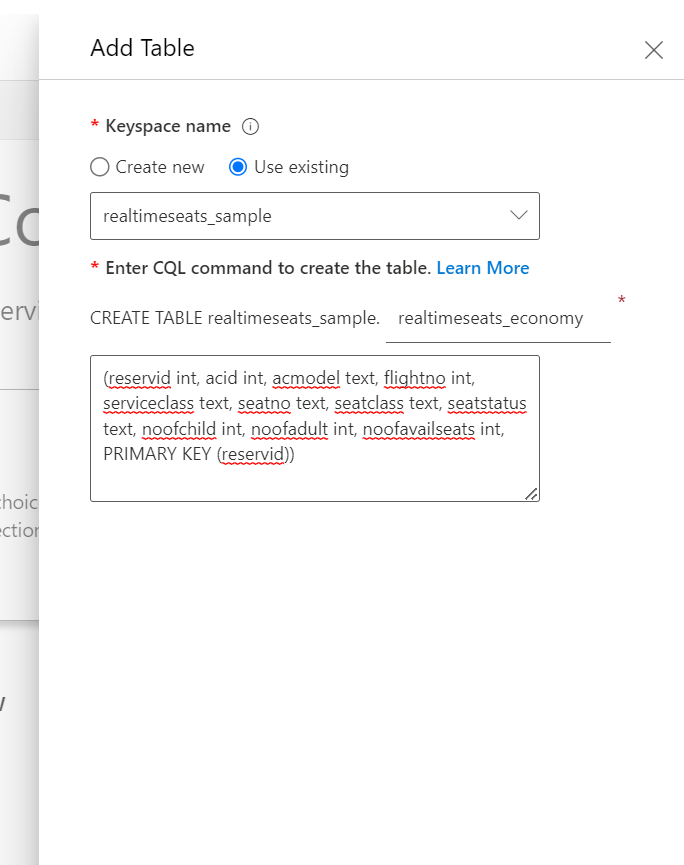
Secondly, as the airline wants the ARTS to process Passenger profiles and clearance, it will be essential for this function to be carried out effectively and quickly. Moreover, as the number of passenger information being stored in the PassengerDetails table will increase day by day, it is essential that the database used has high performance, high availability and automatic scaling. Thus, a Document database such as MongoDB is to be used.

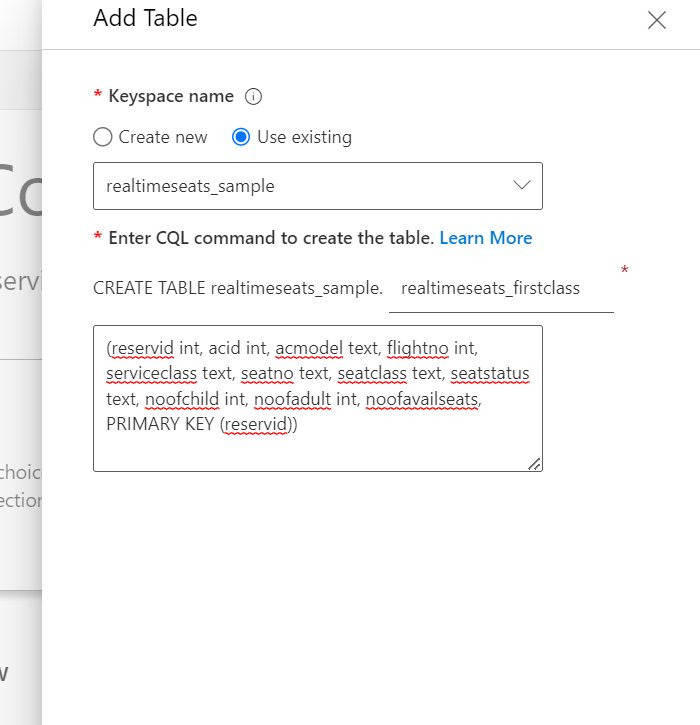
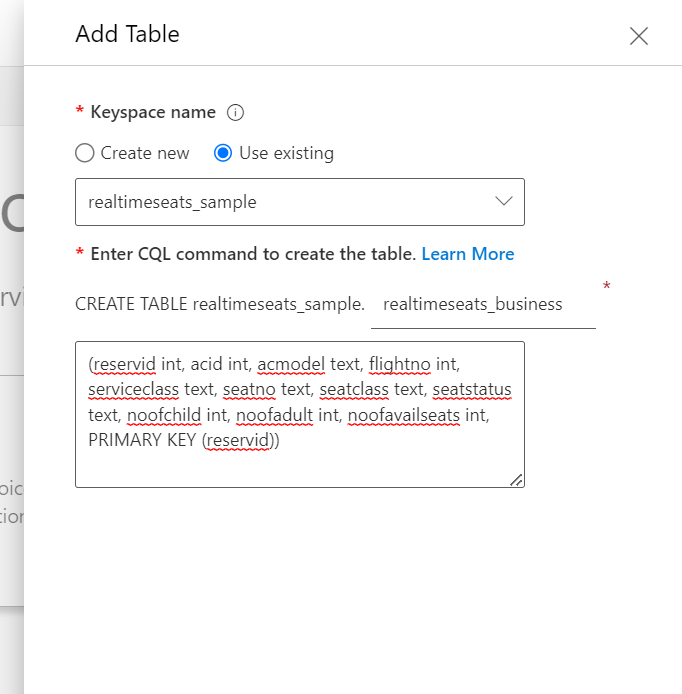
Since MongoDB is consistent and also partition tolerant, even though some data may become unavailable, the remainder will still be accurate and consistent. With MongoDB, the system keeps running even when there are network problems, and actions will be finished even if some components aren't available. Additionally, distributing the workload with a DDB optimizes processing power. Data will grow quickly and require large storage because it is generated at a high velocity and volume for an airline.

## **Design and content of cloud storage**

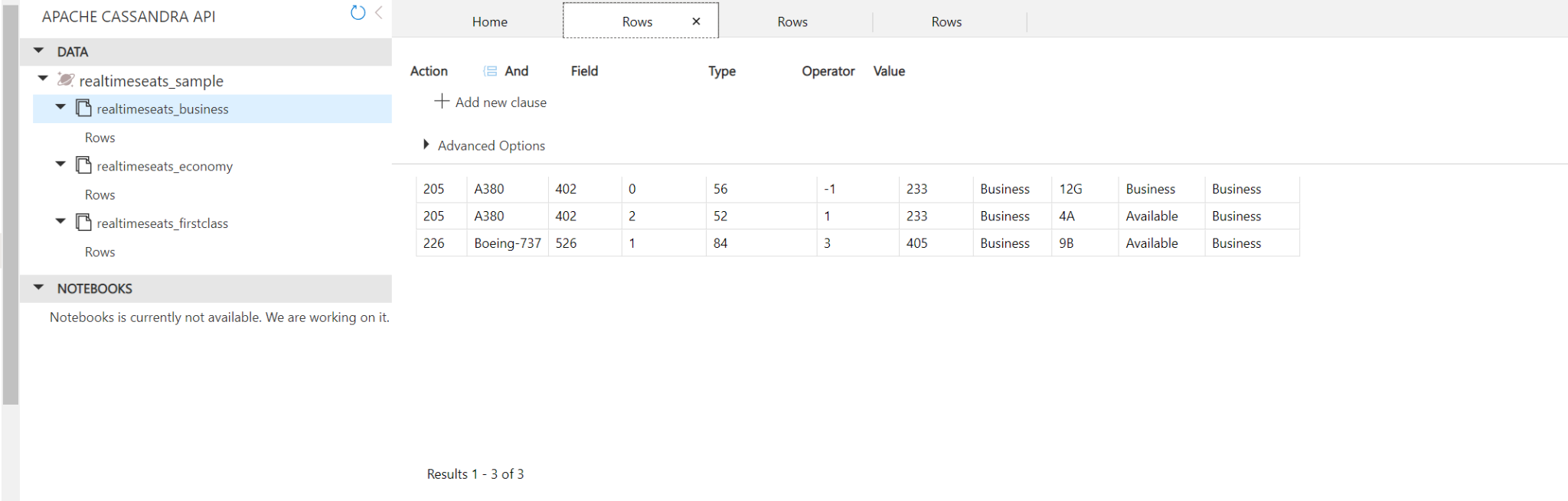
1. **Query 1: Column-Oriented Database (Cassandra)**

These are the steps taken to create the sample data for the Column Oriented Database using Cassandra. I used the Data Explorer function in Microsoft Azure which allows users to create their own datasets from scratch. As seen in the below images I have constructed 3 tables representing RealTimeSeatsEconomy, RealTimeSeatsBusiness and RealTimeSeatsFirstClass.

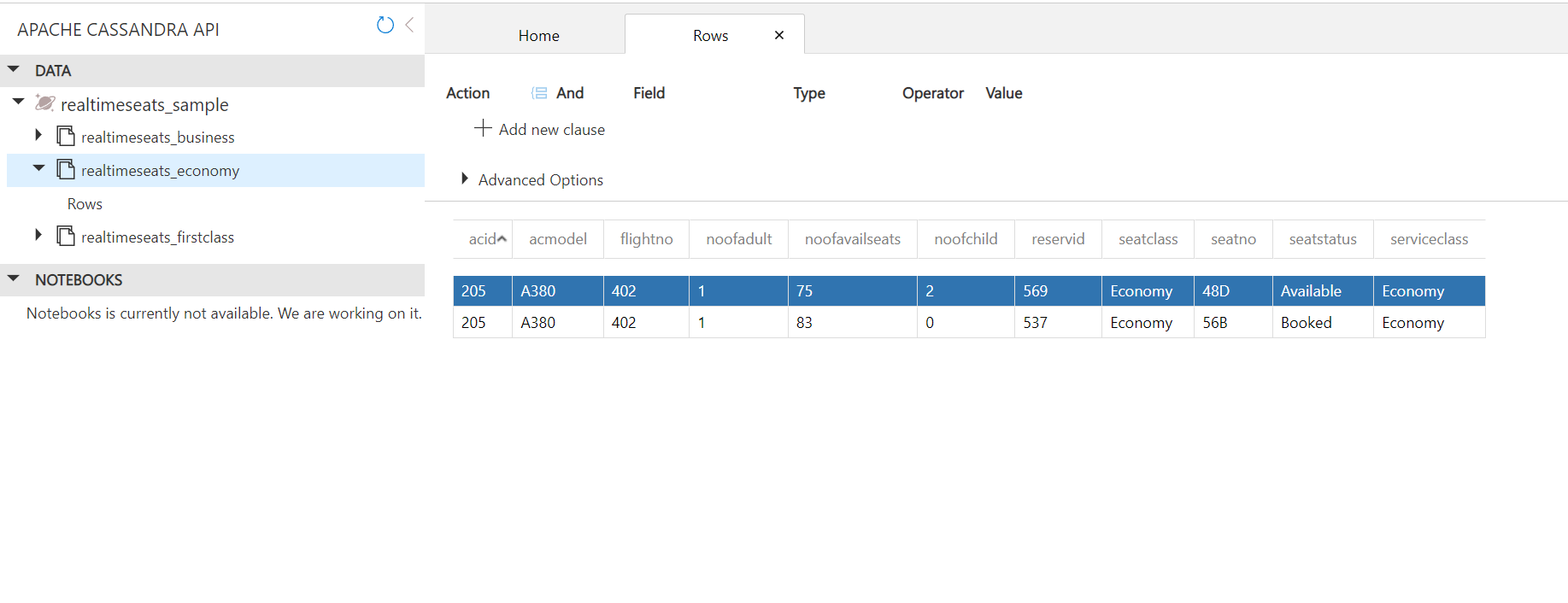




I then inserted multiple rows to each column as sample data. As shown in the image below, you can see the sample data created for the RealTimeSeatsBusiness table.



I then inserted multiple rows to each column as sample data. As shown in the image below, you can see the sample data created for the RealTimeSeatsEconomy table.

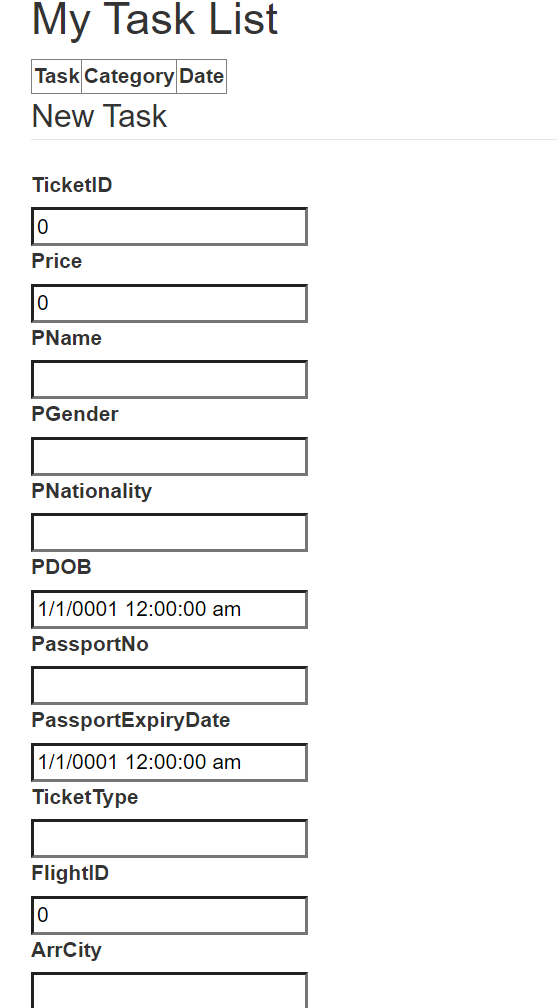


I then inserted multiple rows to each column as sample data. As shown in the image below, you can see the sample data created for the RealTimeSeatsFirstClass table. Graphical user interface, application, table, Excel

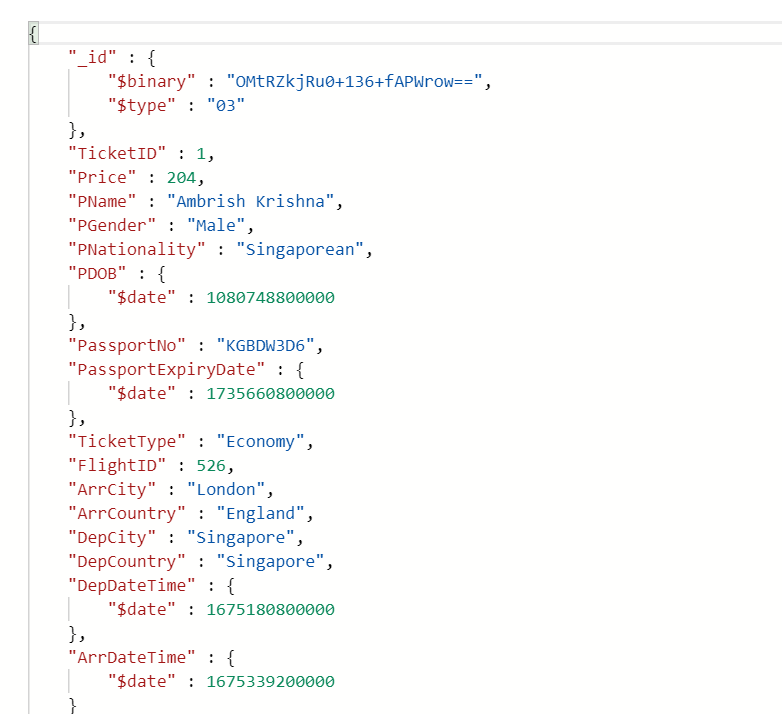
Description automatically generated

1. **Query 2: MongoDB**

These are the steps taken to create the sample data for the Document Database using MongoDB. I used the VS code to do this. As seen in the below image, you can see my local host where I can type in my values.

****

I then created the sample data and the proof can be seen below.





## **Business Queries**

**1st Query: Show a real-time view of the available seats departing from Singapore to Dubai**

**Before Denormalisation**

|  |
| --- |
| SELECT r.ReservID, r.ACID, a.ACModel, r.FlightNo, r.ServiceClass, r.SeatNo, s.SeatClass, SeatStatus, r.NoOfChild, r.NoOfAdult, (SELECT COUNT(r.SeatNo) FROM Reservation r WHERE r.FlightId = fi.FlightId AND r.ReservId IS NULL) FROM Reservation r INNER JOIN FlightSchedule fs ON r.FlightID = fs.FlightID INNER JOIN Aircraft a ON r.ACID = a.ACID INNER JOIN Seats s ON r.SeatNo = s.SeatNo WHERE r.DepCountry = 'Singapore', r.ArrCity = 'Dubai' |

**After Denormalisation**

|  |
| --- |
| SELECT ReservID, ACID, ACModel, FlightNo, ServiceClass, SeatNo, SeatClass, SeatStatus, NoOfChild, NoOfAdult, (SELECT COUNT(r.SeatNo) FROM Reservation r WHERE r.FlightId = fi.FlightId AND r.ReservId IS NULL) FROM RealTimeSeats WHERE r.DepCountry = 'Singapore', r.ArrCity = 'Dubai' |

**2nd Query: Show the information of passengers departing from Singapore to London**

**Before Denormalisation**

This is what the code for the Main Query looks like before doing Denormalisation.

|  |
| --- |
| SELECT t.TicketID, t.PName, t.PGender, t.PDOB, t.PNationality, t.PassportNo, t.PassportExpiryDate, r.DepCity, r.DepCountry, r.ArrCity, r.ArrCountry, fs.DepDateTime, fs.ArrDateTime, t.Price FROM FlightSchedule fs INNER JOIN Ticket t ON fs.FlightID = t.FlightID INNER JOIN RouteInformation r ON fs.FlightID = r.FlightID WHERE r.DepCity = 'Singapore', r.ArrCity = 'London' GROUP BY t.PNationality ORDER BY "t.TicketID" asc |

**After Denormalisation**

This is what the code for the Main Query looks like after conducting Denormalization using Replication and creating a new table.

|  |
| --- |
| SELECT TicketID, PName, PGender, PDOB, PNationality, PassportNo, PassportExpiryDate,DepCity, DepCountry,ArrCity, ArrCountry, DepDateTime, ArrDateTime, Price FROM PassengerDetails WHERE DepCity = 'Singapore', r.ArrCity = 'London' GROUP BY PNationality ORDER BY "TicketID" asc |