CMSC 661 Database Systems Concepts

Skyline- Airline Reservation System

Final Report

Dr. Yelena Yesha

Team Mates:

Amruta Deshmukh – GH15093 – amu1@umbc.edu
Phani Teja Kesha – KP38691 – phani1@umbc.edu
Sowmya Ramapatruni – BG80177 – sowmya1@umbc.edu
Vamshi Krishna Sai Nagabandi – SO03137 – nvamshi1@umbc.edu

Table of Contents

Section 1 Introduction

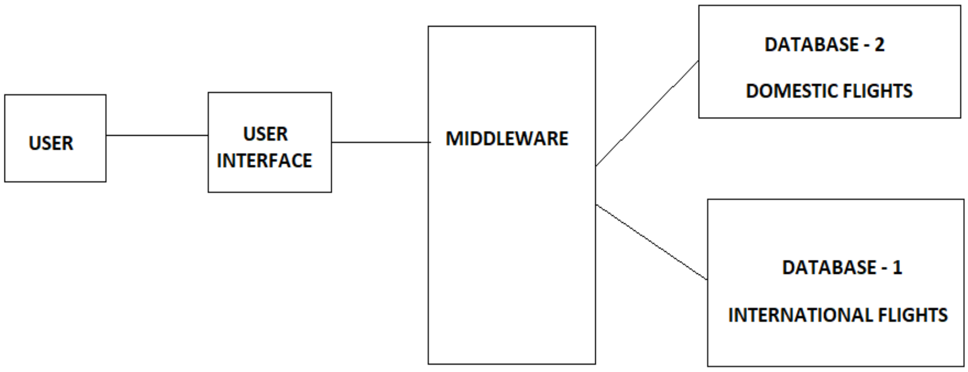
Our project is an airline reservation system, which provides a candid interface and an optimized system, enabling user to browse for flights and book flight tickets online. The datasets are obtained from the source "openflights.org", which provides real-time and accurate data. The data is divided into two databases, one containing data of domestic flights and the other one containing data for international flights. The flights operating within the United States are considered as domestic flights. All other flights have been considered as international flights.

The search page shows the various flights for a one-way trip or a round trip. Search works on the inputs given by the user namely Source and destinations, type of trip, number of passengers, category of travel class, type of passengers, etc. The user can select the flight deciding upon the hours of travel and fare displayed on the search page. After the booking is done, the user can check for the booking history to see previous bookings for that account. The user can cancel the tickets. The booking reference is provided for the user to identify the booking.

Finally search results are compared against the naïve approach of Airlines reservation systems which has all its transactions load in a single database. The experiments and results showed that Airlines with multiple databases has achieved better performance and scalability.

Section 2 System Requirements

2.1 System Architecture Diagram



A User tries to interact with the application through the user interface provided. The middleware layer receives the request from the user interface layer and tries to extract the requested data from the databases. There are two databases, one consisting the data for the domestic flights and one for the data consisting the international flights. Flights operating within the United States are considered as domestic flights. Flights operating between locations outside the United States are considered as international flights. HTML, CSS and Bootstrap are used for the development of user interface. Node.js and Express.js is used in the middleware layer. MySQL is used as a database.

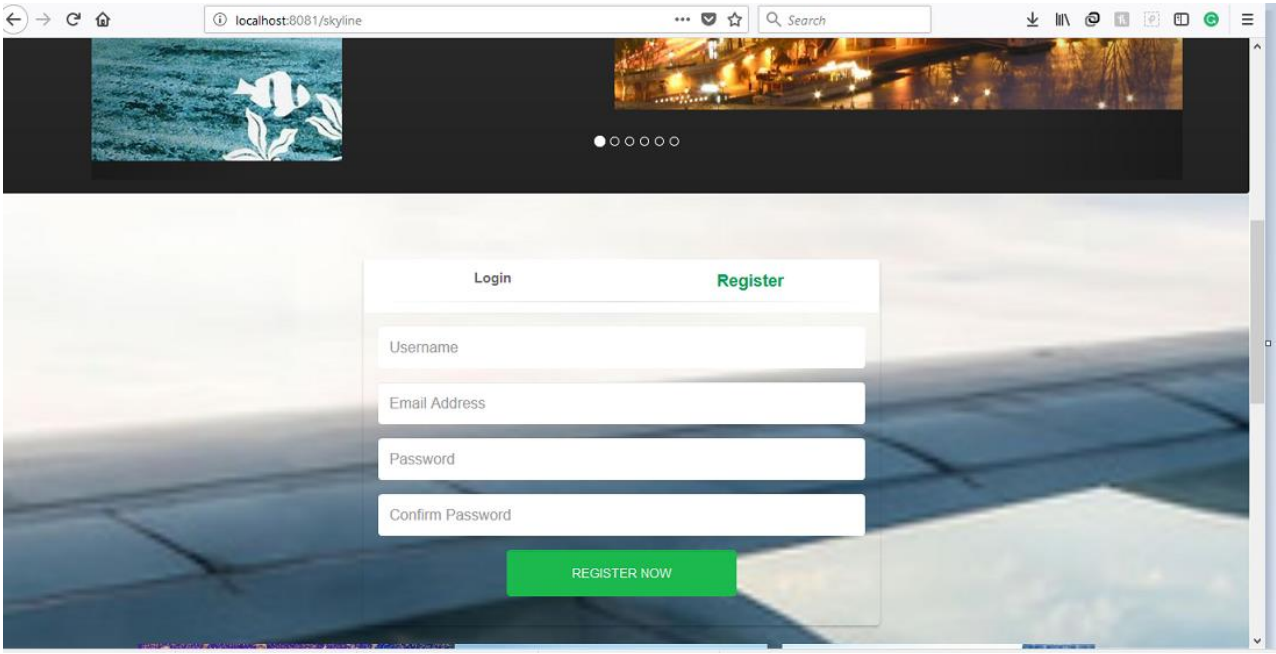
2.2 Interface Requirements

- The web application should allow an easy flow between the pages and provide precise but necessary information.
- There must be a proper gateway for authorizing a user access to the application through the login credentials.
- Confidentiality of the customer information must be maintained for better working of the web application
- The user interface must be elegant, user friendly and reduce latencies of web page transitions.
- The user should easily understand the interface of searching for flights, and easily input his'/her's criteria of travel.

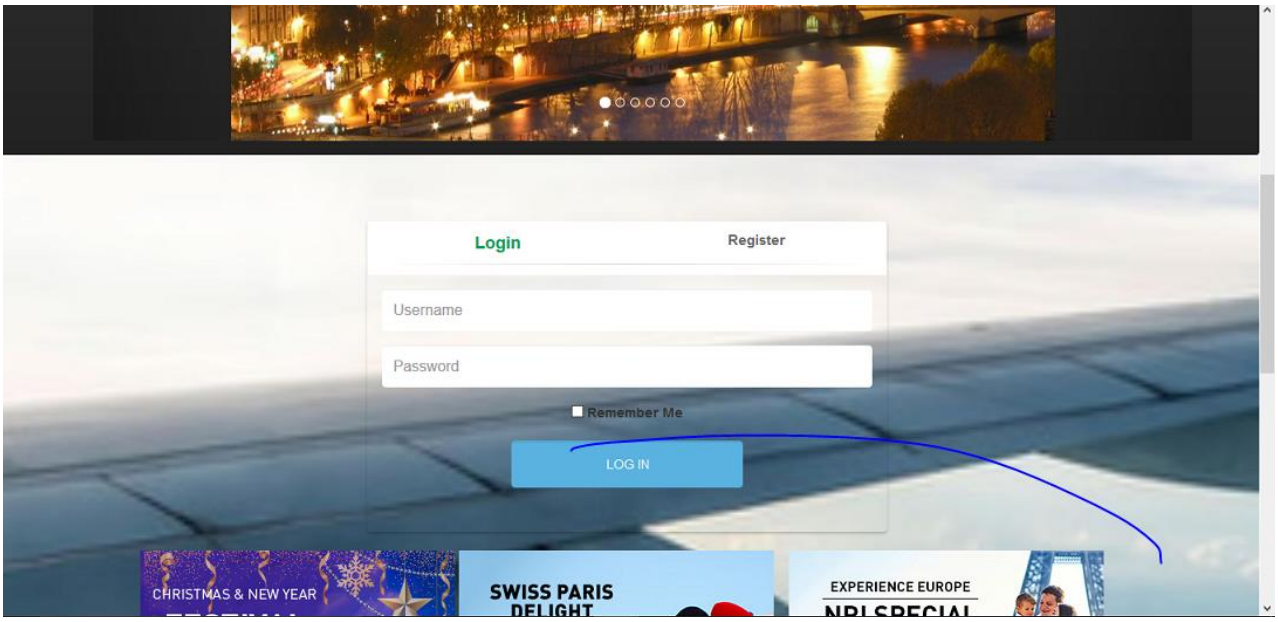
- The application must display all the relevant flights according to the user inputs, along with essential information of the individual flights.
- The users should easily be able to book a ticket in the flight of their choice.
- The user should have access to the booking history, and be able to cancel a ticket.
- The user interface must be elegant, user friendly and reduce latencies of web page transitions.
- The overhead of database must be handled in the backend and should not affect performance of the system in the front-end.

2.3 Functional Requirements

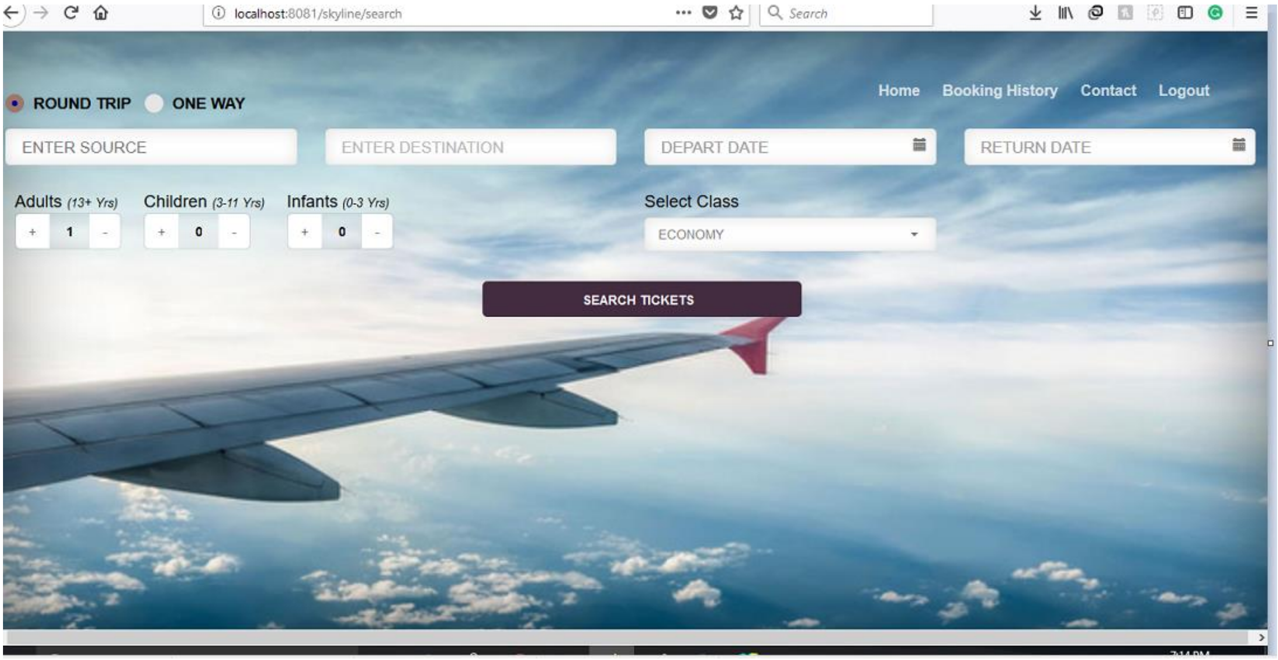
- The registration page allows the user to register with email id and password, and to specify other contact details required for the registration.



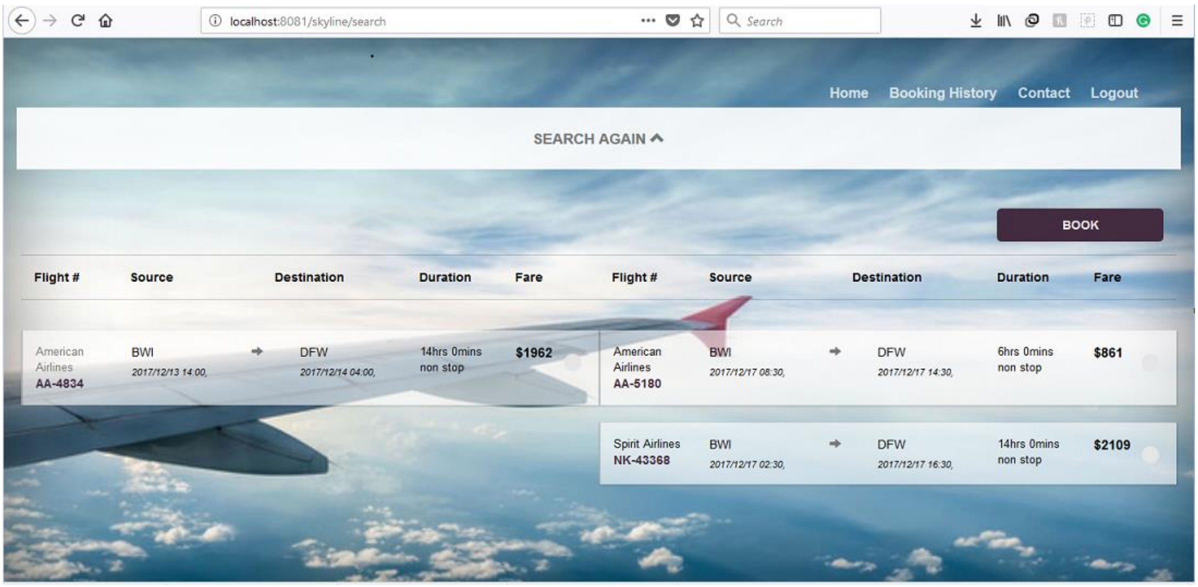
- The login page allows the user to sign in with the user credentials.



- The search page allows the user to search for flights based on the source airport, destination airport, category, type of trip and date of trip. Only on successful login can a user search for flights.



- The search request should be redirected to the relevant database, based on if it’s a domestic flight or an international flight.



- After the search, flights relevant to the search should be displayed allowing the user to select the flight he/she wishes for.
- The number of tickets should be according to user’s wish.
- The user should be able to book a flight of choice and a unique reference number is generated for the booking.
- When multiple users try to book a same ticket, only one of the customer’s transaction should be successful.
- The round trip should allow the user to book tickets for departure and return flights.
- The application should allow user to view his/her booking history and cancel tickets.

Section 3 Conceptual Design of the Database

3.1 Entity-Relationship (ER) Model

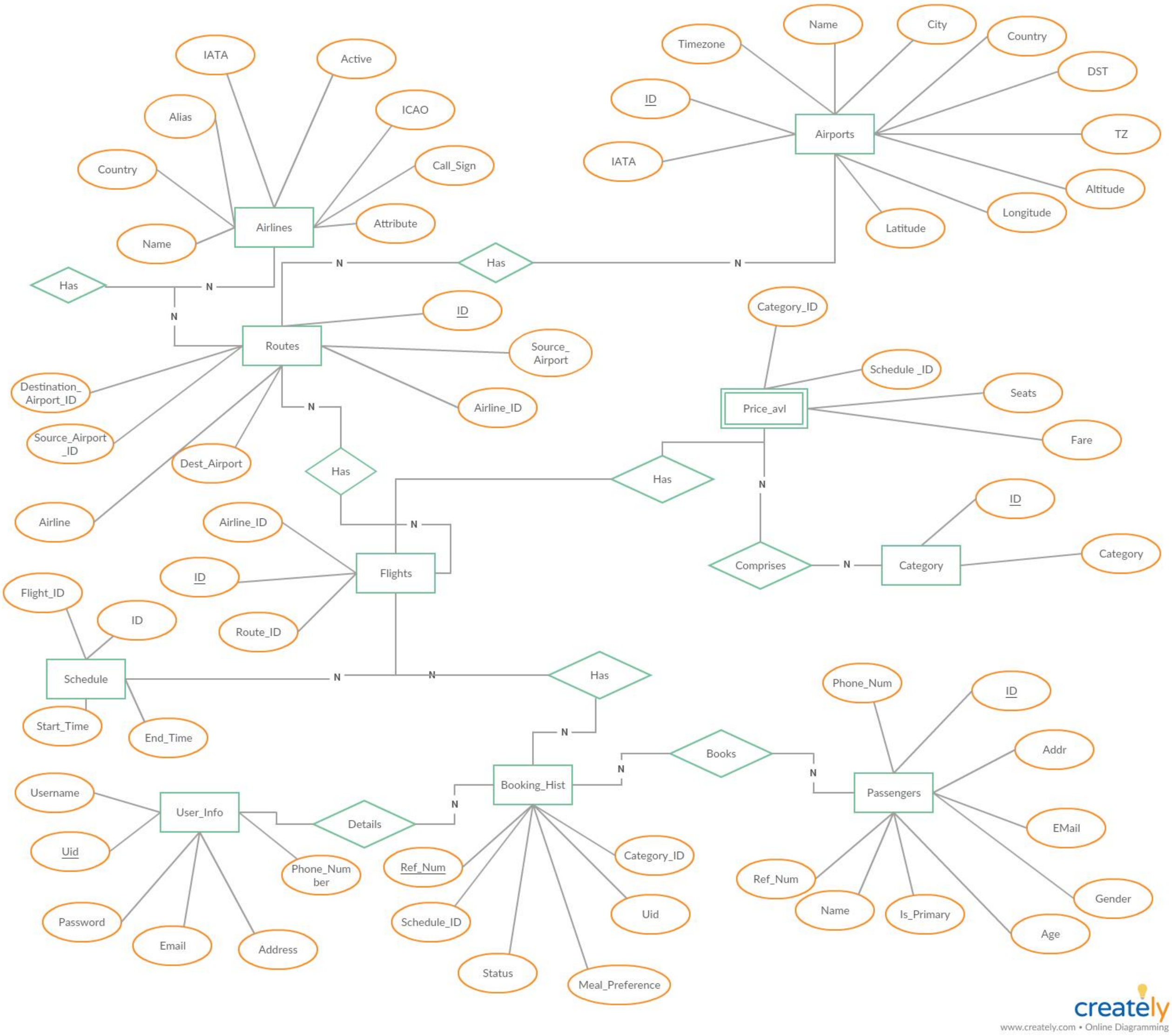


Fig: E-R Diagram for Database-1

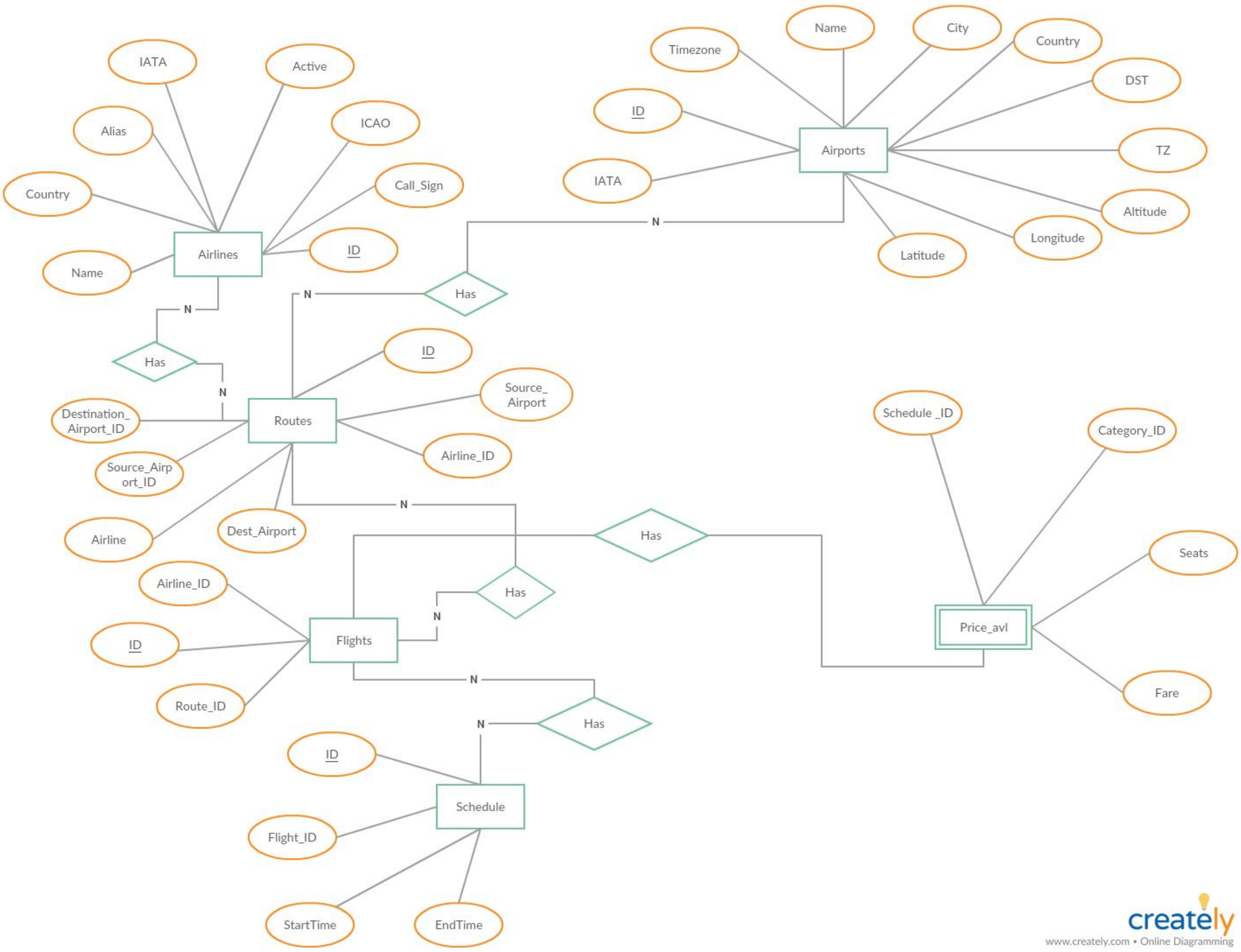


Fig 2: E-R Diagram for Database-2

3.2 Data Dictionary and Business Rules

AIRLINES

Attribute	Data Type	Size	Contraints
ID	INT	20	PRIMARY KEY
ALIAS	VARCHAR	200	
IATA	VARCHAR	5	NOT NULL
ICAO	VARCHAR	5	NOT NULL
CALL_SIGN	VARCHAR	200	NOT NULL
NAME	VARCHAR	150	
COUNTRY	VARCHAR	150	NOT NULL
ACTIVE	VARCHAR	10	

AIRPORTS

Attribute	Data Type	Size	Contraints
ID	INT	20	PRIMARY KEY
NAME	VARCHAR	150	NOT NULL
CITY	VARCHAR	150	NOT NULL
COUNTRY	VARCHAR	150	NOT NULL
IATA	VARCHAR	10	NOT NULL
LATITUDE	DOUBLE		
LONGITUDE	DOUBLE		
ALTITUDE	DOUBLE		
TIMEZONE	VARCHAR	15	
DST	VARCHAR	15	
TZ	VARCHAR	100	

ROUTES

Attribute	Data Type	Size	Contraints
ID	INT	20	PRIMARY KEY
AIRLINE	VARCHAR	5	NOT NULL
AIRLINE_ID	INT	20	FOREIGN KEY
SOURCE_AIRPORT	VARCHAR	10	NOT NULL
SOURCE_AIRPOT_ID	INT	20	FOREIGN KEY
DESTINATION_AIRPORT	VARCHAR	10	NOT NULL
DESTINATION_AIRPORT_ID	INT	20	FOREIGN KEY

BOOKING HISTORY

Attribute	Data Type	Size	Contraints
REF_NUM	VARCHAR	250	PRIMARY KEY
UID	INT	15	FOREIGN KEY
CATEGORY _ID	INT	15	FOREIGN KEY
STATUS	VARCHAR	150	NOT NULL
SCHEDULE_ID	INT	15	FOREIGN KEY
MEAL PREFERENCE	VARCHAR	100	

PASSENGER_INFO

Attribute	Data Type	Size	Contraints
ID	INT	20	PRIMARY KEY
REF_NUM	INT	250	FOREIGN KEY
NAME	VARCHAR	250	NOT NULL
PHONE_NUMBER	INT	15	
EMAIL	VARCHAR	60	NOT NULL
ADDR	VARCHAR	200	
AGE	INT	3	
GENDER	CHAR	10	

PRICE_AVL

Attribute	Data Type	Size	Contraints
ID	INT	15	PRIMARY KEY
SCHEDULE_ID	INT	15	FOREIGN KEY
CATEGORY_ID	INT	15	FOREIGN KEY
FARE	INT	30	NOT NULL
SEATS	INT	15	NOT NULL

USERINFO

Attribute	Data Type	Size	Contraints
ID	INT	20	PRIMARY KEY
EMAIL	VARCHAR	200	UNIQUE
USERNAME	VARCHAR	250	UNIQUE
PASSWORD	VARCHAR	200	NOT NULL
PHONE_NUMBER	VARCHAR	20	
ADDRESS	VARCHAR	200	

CATEGORY

Attribute	Data Type	Size	Contraints
ID	INT	20	PRIMARY KEY
CATEGORY	VARCHAR	35	NOT NULL

FLIGHTS

Attribute	Data Type	Size	Contraints
ID	INT	20	PRIMARY KEY
ROUTE_ID	INT	20	FOREIGN KEY

SCHEDULE

Attribute	Data Type	Size	Contraints
ID	INT	20	PRIMARY KEY
FLIGHT_ID	INT	20	FOREIGN KEY
STARTIME	VARCHAR	20	NOT NULL
ENDTIME	VARCHAR	20	NOT NULL

PASSENGER_CATEGORY

Attribute	Data Type	Size	Contraints
ID	INT	20	PRIMARY KEY
TYPE	VARCHAR	20	NOT NULL
DETAILS	VARCHAR	100	

Business Rules:

The following are the business rules which we have considered:

- A user must provide details only which are valid.
- A user can search for flights and make a booking only on logging into the application.
- A user can book multiple tickets, and in multiple flights according to their willingness.
- A user can only cancel tickets only before trip starts.
- There is no limit to the number of bookings a user can make.
- A minimum luggage allowance of 15lbs is provided in each flight.

Section 4 Logical Database Schema

4.1 Schema of the Database

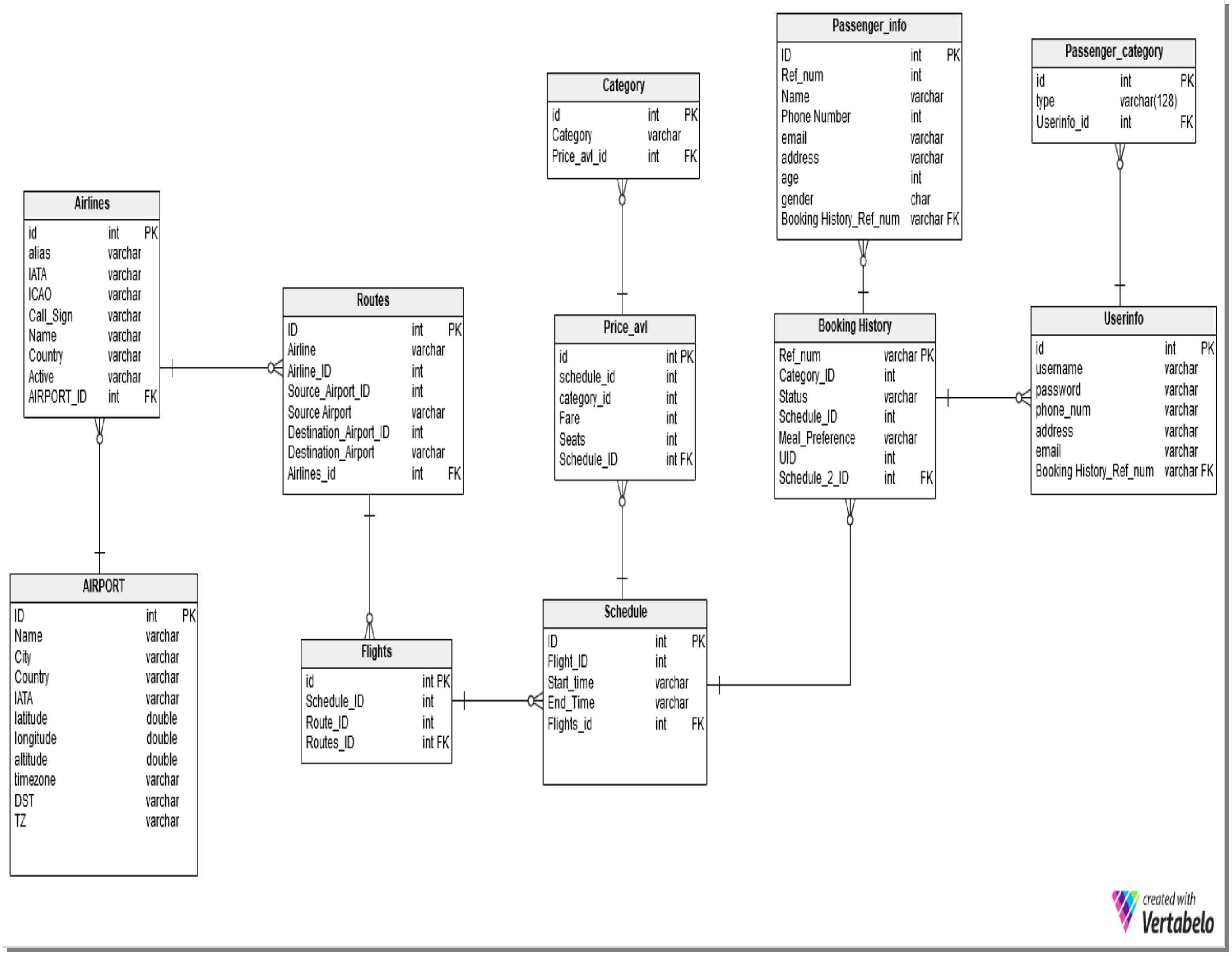


Fig: Schema of Database-1

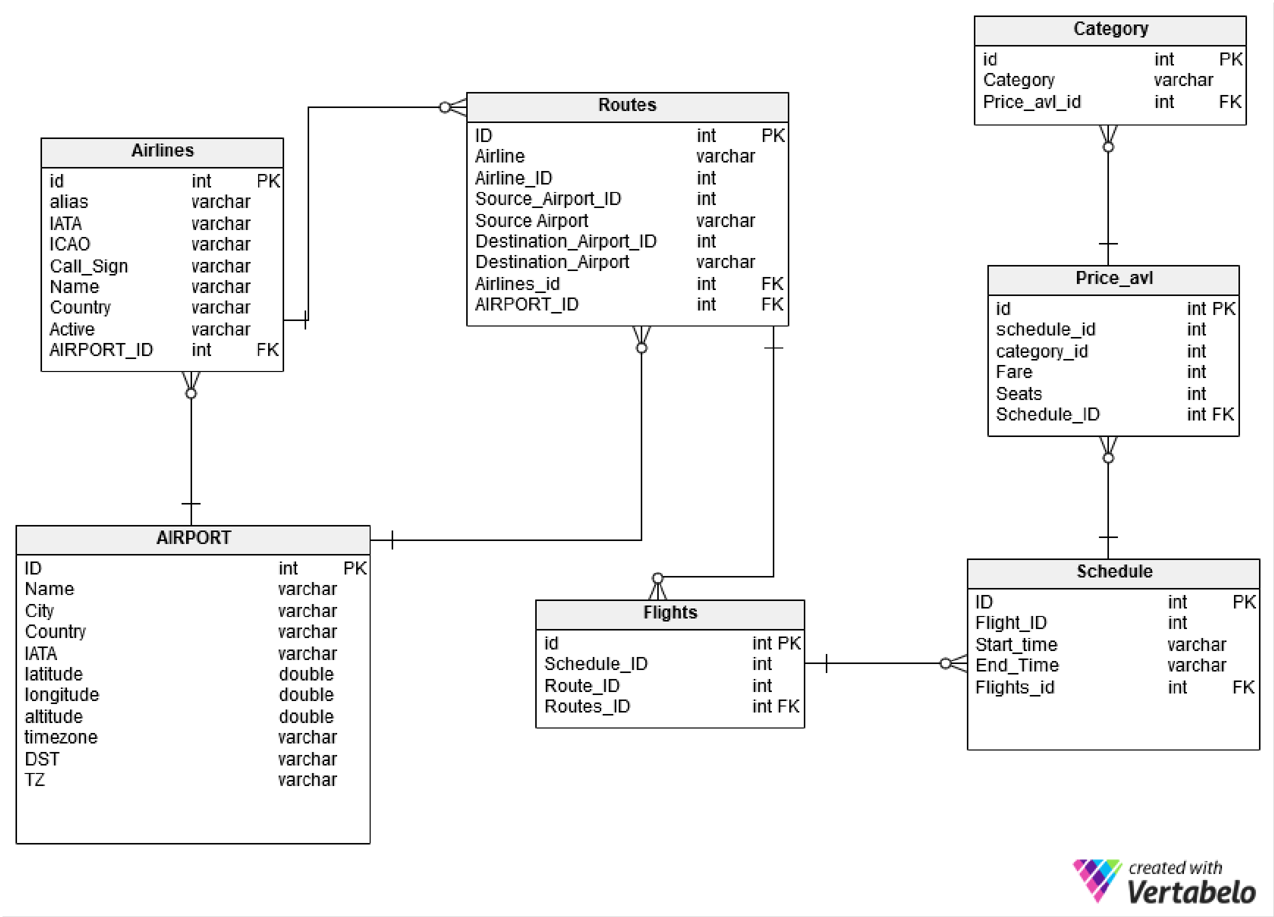


Fig: Schema of Database-2

4.2 SQL Statements Used to Construct the Schema

1) CREATE TABLE AIRPORTS (ID int (20), NAME varchar (150) NOT NULL, CITY varchar (150) NOT NULL, COUNTRY varchar (150) NOT NULL, IATA varchar (10) NOT NULL, LATITUDE DOUBLE, LONGITUDE DOUBLE, ALTITUDE DOUBLE, TIMEZONE varchar (15), DST varchar (15), TZ varchar (100), PRIMARY KEY (ID));

2) CREATE TABLE AIRLINES (ID int (20), NAME varchar (150) NOT NULL, ALIAS varchar (200), IATA varchar (5) NOT NULL, ICAO varchar (5) NOT NULL, CALL_SIGN varchar (200), COUNTRY varchar (150) NOT NULL, ACTIVE varchar (10), PRIMARY KEY (ID));

3) CREATE TABLE ROUTES (ID int(20) NOT NULL, AIRLINE_NAME varchar (5) NOT NULL, AIRLINE_ID int(20) , SOURCE_AIRPORT varchar(10) NOT NULL, SOURCE_AIRPORT_ID int(20) NOT NULL, DESTINATION_AIRPORT varchar(10) NOT NULL, DESTINATION_AIRPORT_ID int(10) NOT NULL, PRIMARY KEY (ID), FOREIGN KEY (SOURCE_AIRPORT_ID) REFERENCES AIRPORTS(ID), FOREIGN KEY (DEST_AIRPORT_ID) REFERENCES AIRPORTS(ID), FOREIGN KEY AIRLINE_ID REFERENCES AIRLINES(ID));

4) CREATE TABLE BOOKING_HISTORY (REF_NUM varchar (250), UID int(15), CATEGORY_ID int(15), STATUS varchar (150) NOT NULL, SCHEDULE_ID int(15), MEAL_PREFERENCE varchar(100), PRIMARY KEY (REF_NUM), FOREIGN KEY(SCHEDULE_ID) REFERENCES SCHEDULE(ID), FOREIGN KEY CATEGORY_ID REFERENCES CATEGORY(ID), FOREIGN KEY(UID) REFERENCES USERINFO(ID));

5) CREATE TABLE PASSENGER_INFO (ID int (20), REF_NUM int(250), NAME varchar(250) NOT NULL, PHONE_NUMBER int(15), EMAIL varchar(60) NOT NULL, ADDRESS varchar(200), AGE int(3), GENDER char(10) PRIMARY KEY (ID), FOREIGN KEY REF_NUM REFERENCES BOOKING_HISTORY(REF_NUM));

6) CREATE TABLE PRICE_AVL (ID int(15), SCHEDULE_ID int(15), CATEGORY_ID int(15), FARE int(30), SEATS int(15) NOT NULL, PRIMARY KEY(ID), FOREIGN KEY SCHEDULE_ID REFERENCES SCHEDULE(ID), FOREIGN KEY CATEGORY_ID REFERENCES CATEGORY(ID));

7) CREATE TABLE USERINFO (ID int(20), EMAIL varchar(200) UNIQUE, USERNAME varchar(250) UNIQUE, PASSWORD varchar(200) NOT NULL, PHONE_NUMBER varchar(20), ADDRESS varchar(200), PRIMARY KEY (ID));

8) CREATE TABLE CATEGORY (ID int(20), CATEGORY varchar(35) NOT NULL, PRIMARY KEY(ID));

9) CREATE TABLE FLIGHTS (ID int(20), ROUTE_ID int(20), PRIMARY KEY(ID), FOREIGN KEY ROUTE_ID REFERENCES ROUTES(ID));

10) CREATE TABLE SCHEDULE (ID int(20), FLIGHT_ID int(20), STARTTIME varchar(20) NOT NULL, ENDTIME varchar(20) NOT NULL, PRIMARY KEY(ID), FOREIGN KEY FLIGHT_ID REFERENCES FLIGHT(ID));

11) CREATE TABLE PASSENGER_CATEGORY (ID int(20), TYPE varchar(20) NOT NULL, DETAILS varchar(100), PRIMARY KEY(ID));

Section 5 Tables, Views and Queries

STORED PROCEDURES:

FOR SEARCHING FLIGHTS

```
CREATE DEFINER=`root`@`localhost` PROCEDURE `search_tickets`(IN starttime
VARCHAR(20), IN endtime VARCHAR(20), IN category_id int(20), IN source_airport_id int(20),
IN destination_airport_id int(20), IN seats int(15))
BEGIN
SELECT fare,
airlines.NAME AS airline_name,
airline,
schedule_id,
flight_id,
source_IATA AS source,
destination_IATA AS destination,
starttime,
endtime
FROM (
SELECT routes.airline_id,
route.source_airport_id AS source_id,
route.destination_airport_id AS destination_id,
route.airline AS airline,
schedule.id AS schedule_id,
flights.id AS flight_id,
schedule.starttime,
schedule.endtime,
price_avl.fare AS fare
FROM schedule
JOIN price_avl
ON schedule.id = price_avl.schedule_id
JOIN flights
ON schedule.flight_id = flights.id
JOIN routes
ON routes.id = flights.route_id
WHERE schedule.starttime = starttime
AND schedule.endtime = endtime
AND price_avl.category_id =category_id
```

```
AND price_avl.seats >= seats
AND routes.source_airport_id = source_airport_id
AND routes.destination_airport_id = destination_airport_id) res
JOIN flights on res.flight_id = flights.id
JOIN airports
ON res.source_id = airports.id
JOIN airports
ON res.destination_id =airports.id;

END
```

RETRIEVING BOOKING HISTORY

```
CREATE DEFINER=`root`@`localhost` PROCEDURE `booking_history`(IN user_id
VARCHAR(20))
BEGIN
SELECT status,
Ref_num,
Source_IATA AS source,
Destination_IATA AS destination,
starttime,
endtime,
airlines.NAME AS airline_name,
airline,
flight_id
FROM (
SELECT booking_history.status,
Booking_history.ref_num,
routes.source_airport_id AS source_id,
routes.destination_airport_id AS destination_id,
routes.airline_id,
routes.airline AS airline,
flights.id AS flight_id,
schedule.starttime,
schedule.endtime
FROM booking_history
JOIN schedule
ON booking_history.schedule_id = schedule.id
JOIN flights
```

```
ON schedule.flight_id = flights.id
JOIN routes
ON flights.route_id = routes.id
WHERE booking_history.uid = user_id) res
JOIN airports
ON res.source_id = airports.id
END
```

CANCELLATION

```
CREATE DEFINER=`root`@`localhost` PROCEDURE `update_fares`(IN seats_update int(20), IN category_id int(20), IN schedule_id int(20))
BEGIN
UPDATE price_Avl
SET seats = seats + seats_update
WHERE category_id = category_id
AND schedule_id = schedule_id;

END
```

BOOKING

```
CREATE DEFINER=`root`@`localhost` PROCEDURE `update_status`(IN status VARCHAR(20), IN booking_id VARCHAR(40))
BEGIN
UPDATE booking_history
SET status = status
WHERE ref_num = booking_id;
END
```

```
CREATE DEFINER=`root`@`localhost` PROCEDURE `update_passenger_info`(IN booking_id VARCHAR(20), IN name VARCHAR(40), IN phone_num int(20), IN email VARCHAR(40),IN is_primary int(5), meal_pref int(5))
BEGIN
INSERT into PASSENGER_INFO
Values (ref_num, name, phone_number,
email, is_primary, meal_pref)
END
```

TRIGGERS

1) FOR DYNAMIC FARE (INCREASING FARE ON MAKING A BOOKING)

```
DELIMITER //
CREATE TRIGGER Dynamic_fares
BEFORE UPDATE
    ON price_Avl FOR EACH ROW
BEGIN
        SET NEW.fare = NEW.fare +20;
END; //
DELIMITER;
```

INDEXES

1) CREATE INDEX START_TIME ON SCHEDULE(STARTTIME);

2) CREATE INDEX CATEGORY ON PRICE_AVL(CATEGORY_ID);

3) CREATE INDEX SOURCE_ID ON ROUTES(SOURCE_AIRPORT_ID);

Section 6 The Use of the Database System

Mysql is to be downloaded from the https://dev.mysql.com/downloads/ and installed in the system. Mysql workbench is used for importing the data into the databases. Data from the airports, airlines and routes are loaded into the database by the command **"LOAD DATA/file/"** into the table. The other tables are generated by scripts and the data is loaded into the database system. The server is initiated by Node.JS.

Stored Procedures are created in the database for searching tickets, booking history, updating the fares, updating the status. These stored procedures are used to take the input from the user interface and gives the required output. The implementation of these will stop SQL injections into the system as the controller does not directly call the query but only the input parameters are sent to the database and has no access to change or manipulate the sql query.

Indexes are added to the database to make the search query optimised. Indexes are used as constraints for the databases and make the search quick. The number of rows examined in a database will be enormously decreased by indexing some of the attributes which we can segregate the data into different clusters.

Locking techniques like strict two-phase locking is implemented on the query where the tickets are booked. The table is locked when there is a transaction going on and committed if the transaction is executed with zero errors. If there is any error in any of the transaction the database is rolled back to the state before booking the tickets and the transaction starts again.

Triggers are used for implementing the dynamic fares in the system. The trigger will manipulate the cost of the ticket depending on the number of bookings.

Section 7 Transactions

Transactions are needed to maintain the database integrity when multiple users access the same table simultaneously. We implemented transactions using nodejs builtin API'S. Booking is one of the feature's which needs secure transactions as multiple users can try to book tickets at same time.

These are the following built-in nodejs functions which ensure transactions.

```
Connection.beginTransaction()
Connection.commit()
Connection.rollback()
```

Below is the code snippet which uses nodejs transactions in booking tickets. The function first begins the transaction and books the ticket. If the database encounters any issue in between, it rollbacks to the original state without commiting the transaction. And finally, when operation is completed successfully, it commits the transaction to the database.

```
function book_tickets(insert_statements,cb){
   var i = 0 ;
   connection.beginTransaction(function(err){
      if(err){
         cb(false);
      }else{
         recursive();
      }
   })
   function recursive(){
      if( i >= insert_statements.length){
         connection.commit(function(err){
            if(err){
               cb(false);
            }else{
               cb(true);
            }
         })
         return;
      }
```

```
        connection.query(insert_statements[i],function(err,rows){
         if(err){
         connection.rollback(function(err){
            cb(false);
         })
         }
         else{
            i++;
            recursive();          }
    });
}}
```

8. Concurrency Control

Concurrency to the system is provided at two levels. One from the application point of view and the other from the database point of view.

The first level is handled by nodejs framework, which uses a single thread with an event-loop. In this way, Node can handle 1000s of concurrent connections without any of the traditional detriments associated with threads. There is essentially no memory overhead per-connection, and there is no context switching. Many web servers, for example achieve concurrency by creating a new thread for every connection. In most platforms, this comes at a substantial cost. The default stack size in Java is 512KB, which means that if you have 1000 concurrent connections, your program will consume half a gigabyte of memory just for stack space. Additionally, forking threads in most systems costs an enormous amount of time, as does performing a context switch between two threads.

We used Strict two-phase locking protocol to ensure concurrency with in the system. A transaction cannot write into database until it reached its commit point. Similarly, a transaction cannot release any locks until it finishes writing into database, therefore locks are not released until after the commit point.

The above mentioned nodejs transactions when used with Sequilize supports managed transactions. The difference is that the managed transaction uses a callback that expects a promise to be returned to it. The callback passed to transaction returns a promise chain, and does not explicitly call t.commit() nor t.rollback(). If all promises in the returned chain are resolved successfully the transaction is committed. If one or several of the promises are rejected, the transaction is rolled back.

The possible isolations levels to use when starting a transaction:

Sequelize.Transaction.ISOLATION_LEVELS.READ_UNCOMMITTED // "READ UNCOMMITTED"
Sequelize.Transaction.ISOLATION_LEVELS.READ_COMMITTED // "READ COMMITTED"
Sequelize.Transaction.ISOLATION_LEVELS.REPEATABLE_READ // "REPEATABLE READ"
Sequelize.Transaction.ISOLATION_LEVELS.SERIALIZABLE // "SERIALIZABLE"

9. Conclusions and Future Work

Conclusions:

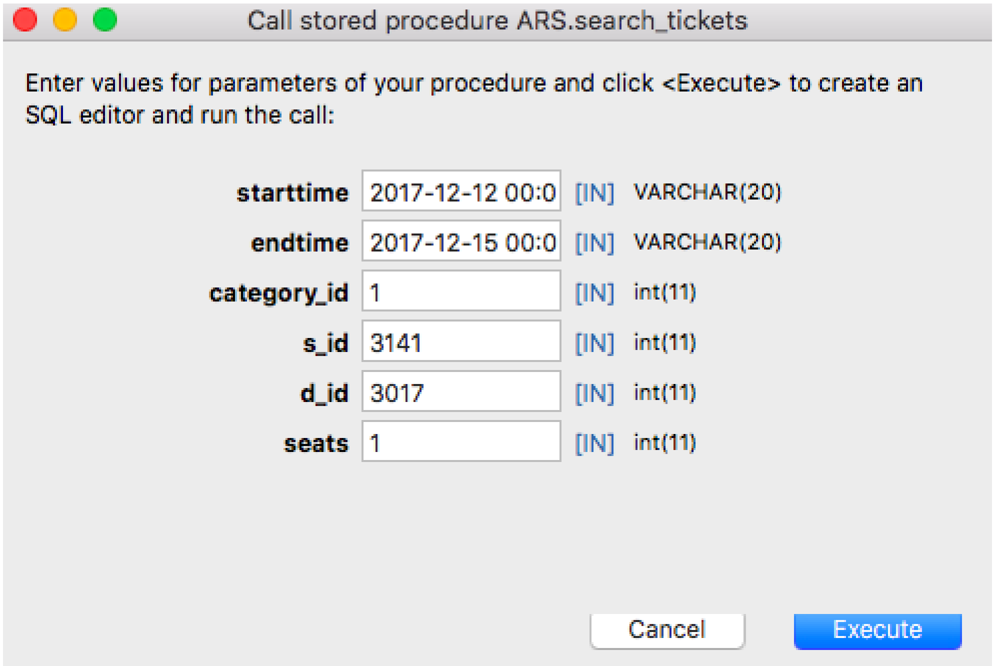
We have tried to develop a user friendly and hassle-free web application that enables users to easily search for flights and make bookings. We have tried to optimize the performance of the system by implementing the usage of multiple databases, dividing the data among themselves based on domestic and international flights. Following are the baselines which we feel have achieved.

Profiling and Query Execution Time

Search Query Performance:

The time taken for a given search for flights between HYD to PUN in the database without any indexes is 2.18 seconds. The time taken for the same search after indexing the schedule and prices table is 0.026 seconds. That means the search query is optimised by 83 times. All these searches are executed in the International Database.

Stored Procedure for flight search from Hyderabad to Pune



Query Statistics

Timing (as measured at client side):
Execution time: 0:00:2.18241215

Timing (as measured by the server):
Execution time: 0:00:2.18212883
Table lock wait time: 0:00:0.00041400

Errors:
Had Errors: NO
Warnings: 0

Rows Processed:
Rows affected: 0
Rows sent to client: 5
Rows examined: 1942950

Temporary Tables:
Temporary disk tables created: 0
Temporary tables created: 1

Joins per Type:
Full table scans (Select_scan): 0
Joins using table scans (Select_full_join): 1
Joins using range search (Select_full_range_join): 0
Joins with range checks (Select_range_check): 0
Joins using range (Select_range): 1

Sorting:
Sorted rows (Sort_rows): 0
Sort merge passes (Sort_merge_passes): 0
Sorts with ranges (Sort_range): 0
Sorts with table scans (Sort_scan): 0

Index Usage:
No Index used

Other Info:
Event Id: 94
Thread Id: 1209

Rows Processed in HYD to PUNE - INTERNATIONAL Database - 1942950
Rows Returned in HYD to PUNE - INTERNATIONAL Database - 5

Stored Procedure for flight search from Hyderabad to Pune after indexing

Query Statistics

Timing (as measured at client side):
Execution time: 0:00:0.02632403

Timing (as measured by the server):
Execution time: 0:00:0.02618464
Table lock wait time: 0:00:0.00025300

Errors:
Had Errors: NO
Warnings: 0

Rows Processed:
Rows affected: 0
Rows sent to client: 5
Rows examined: 183

Temporary Tables:
Temporary disk tables created: 0
Temporary tables created: 1

Joins per Type:
Full table scans (Select_scan): 0
Joins using table scans (Select_full_join): 0
Joins using range search (Select_full_range_join): 0
Joins with range checks (Select_range_check): 0
Joins using range (Select_range): 1

Sorting:
Sorted rows (Sort_rows): 0
Sort merge passes (Sort_merge_passes): 0
Sorts with ranges (Sort_range): 0
Sorts with table scans (Sort_scan): 0

Index Usage:
At least one Index was used

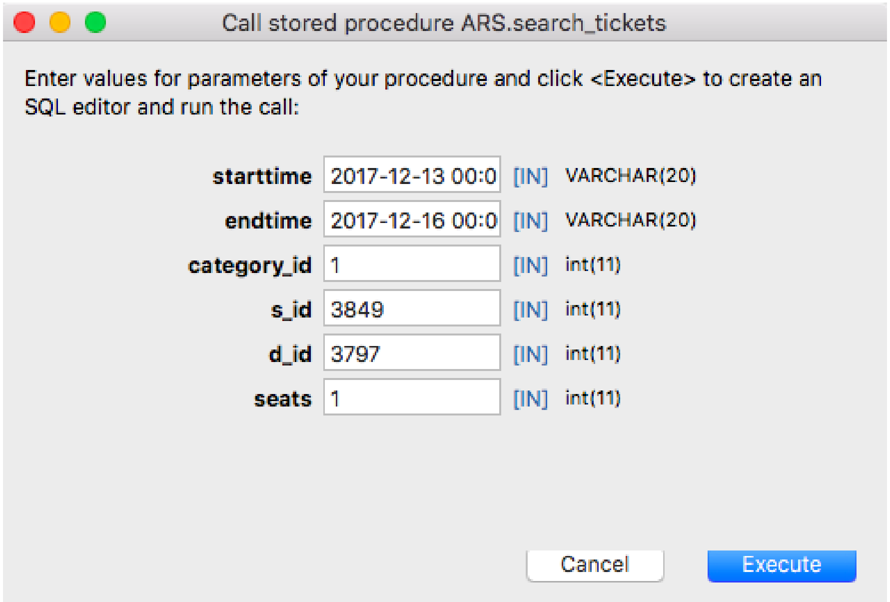
Other Info:
Event Id: 112
Thread Id: 1209

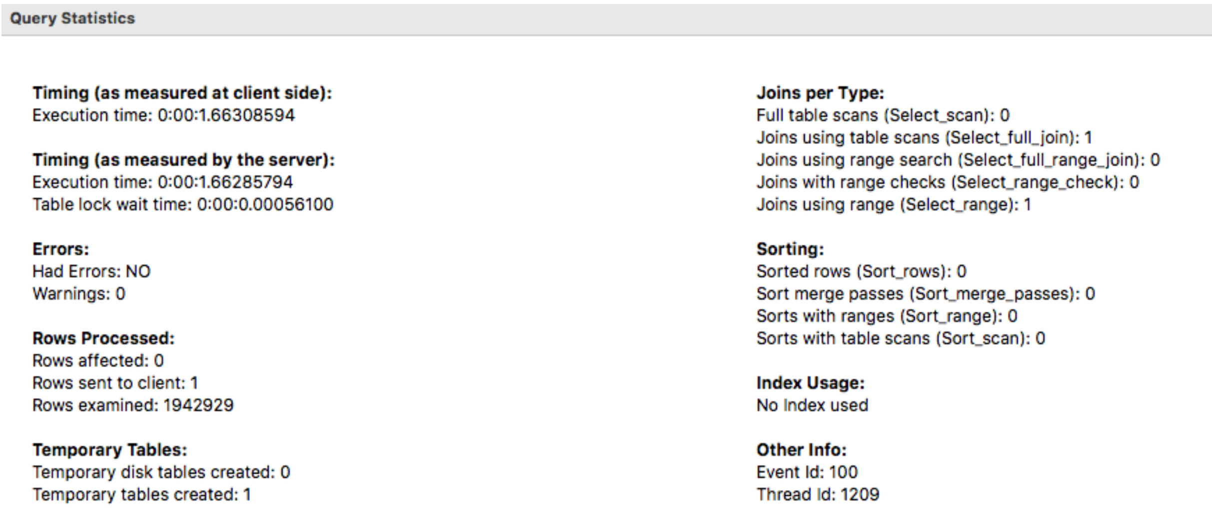
AFTER INDEXING

Rows Processed in HYD to PUNE - INTERNATIONAL Database - 183
Rows Returned in HYD to PUNE - INTERNATIONAL Database - 5

The time taken for the given search for flights between BWI to JFK in the International Database without any index is 1.66 seconds. The time taken for the same search after indexing the schedule and prices table is 0.0085 seconds. That means the search query is optimised by 195 times. All these Searches are executed in the common database with all the data. Let’s check the time taken for the domestic flights search after they are segregated into the domestic database. In the domestic database the time taken is 0.0028 seconds. The search query is optimised by 3 times.

Stored Procedure for flight search from BWI to JFK





Rows Processed in BWI to JFK - COMMON DATABASE - 1942929

Rows Returned in BWI to JFK - COMMON DATABASE - 1

AFTER INDEXING

Stored Procedure for flight search from BWI to JFK

Query Statistics

Timing (as measured at client side):
Execution time: 0:00:0.00858593

Timing (as measured by the server):
Execution time: 0:00:0.00844307
Table lock wait time: 0:00:0.00032700

Errors:
Had Errors: NO
Warnings: 0

Rows Processed:
Rows affected: 0
Rows sent to client: 1
Rows examined: 77

Temporary Tables:
Temporary disk tables created: 0
Temporary tables created: 1

Joins per Type:
Full table scans (Select_scan): 0
Joins using table scans (Select_full_join): 0
Joins using range search (Select_full_range_join): 0
Joins with range checks (Select_range_check): 0
Joins using range (Select_range): 1

Sorting:
Sorted rows (Sort_rows): 0
Sort merge passes (Sort_merge_passes): 0
Sorts with ranges (Sort_range): 0
Sorts with table scans (Sort_scan): 0

Index Usage:
At least one Index was used

Other Info:
Event Id: 118
Thread Id: 1209

Rows Processed in BWI to JFK - COMMON DATABASE - 77

Rows Returned in BWI to JFK - COMMON DATABASE - 1

Query Statistics

Timing (as measured at client side):
Execution time: 0:00:0.00000000

Timing (as measured by the server):
Execution time: 0:00:0.00289304
Table lock wait time: 0:00:0.00000000

Errors:
Had Errors: NO
Warnings: 0

Rows Processed:
Rows affected: 0
Rows sent to client: 1
Rows examined: 14

Temporary Tables:
Temporary disk tables created: 0
Temporary tables created: 2

Joins per Type:
Full table scans (Select_scan): 1
Joins using table scans (Select_full_join): 2
Joins using range search (Select_full_range_join): 0
Joins with range checks (Select_range_check): 0
Joins using range (Select_range): 1

Sorting:
Sorted rows (Sort_rows): 0
Sort merge passes (Sort_merge_passes): 0
Sorts with ranges (Sort_range): 0
Sorts with table scans (Sort_scan): 0

Index Usage:
No Index used

Other Info:
Event Id: 42
Thread Id: 24

Rows Processed in BWI to JFK - DOMESTIC DATABASE - 14

Rows Returned in BWI to JFK - DOMESTIC DATABASE - 1

Thus, there is an optimization to some extent, when the data is segregated into domestic and international flights.

Future Work:

The following things could be done which would enhance the application in a much better way:

- Creating an interface to select a specified seat for the given flight.
- To incorporate a web check-in which makes the travel of the passenger easy and hassle free.
- Load balancing of the databases by depending on the search query in different seasons.
- Optimising the Heavy load on the search query by creation of multiple databases not only confining to two databases.
- Adding fare calendar to each route by which the passenger can get the glance of the flight cost all over the month.
- Providing the details of Cabs and Hotels as packages when flights are booked for a given destination.

References

1) https://thenewboston.com/videos.php?cat=355

2) https://www.tutorialspoint.com/mysql/

3) https://www.tutorialspoint.com//nodejs/nodejs_restful_api.htm

4) https://dev.mysql.com/doc/workbench/en/

5) https://easyengine.io/tutorials/mysql/remote-access/

6) https://stackoverflow.com/questions/14779104/how-to-allow-remote-connection-to-mysql

7) https://dba.stackexchange.com/questions/64945/airline-reservation-system

Appendix

1) https://github.com/itsvamshiks/Skyline