Joseph Annand

DSE 6210

Project 1

## Part I: Crow’s Foot Diagram

Below is the Crow’s foot diagram for the flight management system described in the supplied SRS. The entities with the blue headers are entities specifically included in the SRS. The entities with yellow headers are placeholder entities created to satisfy the foreign key constraints originally specified in the SRS document. Note that in the LEGS entity, origin\_airport and destination\_airport remain despite the information in those columns also being stored in the AIRPORTS entity. These columns were included to match the specifications in the SRS document; however, they could be removed to increase data normalization in the schema.

A computer screen shot of a computer

AI-generated content may be incorrect.

## Part 2: Implementation

This section contains the DDL statements for the flight management system database as the schema, flight\_ms; assertions and triggers associated with the schema; and the views and queries to accomplish the customer function in the SRS written in PostgreSQL.

### 2.1 CREATE statements

-- Drop Views

DROP VIEW IF EXISTS flight\_ms.flight\_tracker;

DROP VIEW IF EXISTS flight\_ms.all\_flight\_schedules;

DROP VIEW IF EXISTS flight\_ms.itinerary;

-- Drop foreign keys from all tables

ALTER TABLE IF EXISTS flight\_ms.flight\_schedules DROP CONSTRAINT IF EXISTS flight\_schedules\_airline\_code\_fkey;

ALTER TABLE IF EXISTS flight\_ms.flight\_schedules DROP CONSTRAINT IF EXISTS flight\_schedules\_usual\_aircraft\_type\_code\_fkey;

ALTER TABLE IF EXISTS flight\_ms.flight\_schedules DROP CONSTRAINT IF EXISTS flight\_schedules\_origin\_airport\_code\_fkey;

ALTER TABLE IF EXISTS flight\_ms.flight\_schedules DROP CONSTRAINT IF EXISTS flight\_schedules\_destination\_airport\_code\_fkey;

ALTER TABLE IF EXISTS flight\_ms.flight\_costs DROP CONSTRAINT IF EXISTS flight\_costs\_flight\_number\_fkey;

ALTER TABLE IF EXISTS flight\_ms.flight\_costs DROP CONSTRAINT IF EXISTS flight\_costs\_aircraft\_type\_code\_fkey;

ALTER TABLE IF EXISTS flight\_ms.flight\_costs DROP CONSTRAINT IF EXISTS flight\_costs\_valid\_from\_date\_fkey;

ALTER TABLE IF EXISTS flight\_ms.flight\_costs DROP CONSTRAINT IF EXISTS flight\_costs\_valid\_to\_date\_fkey;

ALTER TABLE IF EXISTS flight\_ms.itinerary\_reservations DROP CONSTRAINT IF EXISTS itinerary\_reservations\_agent\_id\_fkey;

ALTER TABLE IF EXISTS flight\_ms.itinerary\_reservations DROP CONSTRAINT IF EXISTS itinerary\_reservations\_passenger\_id\_fkey;

ALTER TABLE IF EXISTS flight\_ms.itinerary\_reservations DROP CONSTRAINT IF EXISTS itinerary\_reservations\_reservation\_status\_code\_fkey;

ALTER TABLE IF EXISTS flight\_ms.itinerary\_reservations DROP CONSTRAINT IF EXISTS itinerary\_reservations\_ticket\_type\_code\_fkey;

ALTER TABLE IF EXISTS flight\_ms.itinerary\_reservations DROP CONSTRAINT IF EXISTS itinerary\_reservations\_travel\_class\_code\_fkey;

ALTER TABLE IF EXISTS flight\_ms.payments DROP CONSTRAINT IF EXISTS payments\_payment\_status\_code\_fkey;

ALTER TABLE IF EXISTS flight\_ms.reservation\_payments DROP CONSTRAINT IF EXISTS reservation\_payments\_reservation\_id\_fkey;

ALTER TABLE IF EXISTS flight\_ms.reservation\_payments DROP CONSTRAINT IF EXISTS reservation\_payments\_payment\_id\_fkey;

ALTER TABLE IF EXISTS flight\_ms.legs DROP CONSTRAINT IF EXISTS legs\_flight\_number\_fkey;

ALTER TABLE IF EXISTS flight\_ms.itinerary\_legs DROP CONSTRAINT IF EXISTS itinerary\_legs\_reservation\_id\_fkey;

ALTER TABLE IF EXISTS flight\_ms.itinerary\_legs DROP CONSTRAINT IF EXISTS itinerary\_legs\_leg\_id\_fkey;

---Drop tables from the schema

DROP TABLE IF EXISTS flight\_ms.itinerary\_legs CASCADE;

DROP TABLE IF EXISTS flight\_ms.legs;

DROP TABLE IF EXISTS flight\_ms.reservation\_payments;

DROP TABLE IF EXISTS flight\_ms.payments;

DROP TABLE IF EXISTS flight\_ms.itinerary\_reservations;

DROP TABLE IF EXISTS flight\_ms.flight\_costs;

DROP TABLE IF EXISTS flight\_ms.flight\_schedules;

DROP TABLE IF EXISTS flight\_ms.ref\_calendar;

DROP TABLE IF EXISTS flight\_ms.airlines;

DROP TABLE IF EXISTS flight\_ms.travel\_classes;

DROP TABLE IF EXISTS flight\_ms.ticket\_codes;

DROP TABLE IF EXISTS flight\_ms.payment\_statuses;

DROP TABLE IF EXISTS flight\_ms.reservation\_statuses;

DROP TABLE IF EXISTS flight\_ms.airports;

DROP TABLE IF EXISTS flight\_ms.booking\_agents;

DROP TABLE IF EXISTS flight\_ms.passengers;

DROP TABLE IF EXISTS flight\_ms.aircrafts;

DROP SCHEMA IF EXISTS flight\_ms;

CREATE SCHEMA IF NOT EXISTS flight\_ms;

--- Create all tables without foreign keys

CREATE TABLE IF NOT EXISTS flight\_ms.passengers (

passenger\_id INT NOT NULL,

first\_name VARCHAR NOT NULL,

second\_name VARCHAR,

last\_name VARCHAR NOT NULL,

phone\_number VARCHAR NOT NULL,

email\_address VARCHAR NOT NULL,

address\_lines VARCHAR NOT NULL,

state\_province\_county VARCHAR NOT NULL,

country VARCHAR NOT NULL,

other\_passenger\_details VARCHAR,

PRIMARY KEY (passenger\_id)

);

CREATE TABLE IF NOT EXISTS flight\_ms.booking\_agents(

agent\_id INT NOT NULL,

agent\_name VARCHAR NOT NULL,

agent\_details VARCHAR,

PRIMARY KEY (agent\_id)

);

CREATE TABLE IF NOT EXISTS flight\_ms.airports (

airport\_code INT NOT NULL,

airport\_name VARCHAR NOT NULL,

airport\_location VARCHAR NOT NULL,

other\_details VARCHAR,

PRIMARY KEY (airport\_code)

);

--- placeholder table; not outlined in SRS

CREATE TABLE IF NOT EXISTS flight\_ms.reservation\_statuses (

reservation\_status\_code INT NOT NULL,

reservation\_status VARCHAR NOT NULL,

PRIMARY KEY (reservation\_status\_code)

);

--- placeholder table; not outlined in SRS

CREATE TABLE IF NOT EXISTS flight\_ms.payment\_statuses (

payment\_status\_code INT NOT NULL,

payment\_status VARCHAR NOT NULL,

PRIMARY KEY (payment\_status\_code)

);

--- placeholder table; not outlined in SRS

CREATE TABLE IF NOT EXISTS flight\_ms.ticket\_codes (

ticket\_type\_code INT NOT NULL,

ticket\_type VARCHAR NOT NULL,

PRIMARY KEY (ticket\_type\_code)

);

--- placeholder table; not outlined in SRS

CREATE TABLE IF NOT EXISTS flight\_ms.travel\_classes (

travel\_class\_code INT NOT NULL,

travel\_class VARCHAR NOT NULL,

PRIMARY KEY (travel\_class\_code)

);

--- placeholder table; not outlined in SRS

CREATE TABLE IF NOT EXISTS flight\_ms.airlines (

airline\_code INT NOT NULL,

airline\_name VARCHAR NOT NULL,

PRIMARY KEY (airline\_code)

);

CREATE TABLE IF NOT EXISTS flight\_ms.ref\_calendar (

day\_date DATE NOT NULL,

day\_number INT NOT NULL,

business\_day\_yn VARCHAR NOT NULL,

PRIMARY KEY (day\_date)

);

--- placeholder table; not outlined in SRS

CREATE TABLE IF NOT EXISTS flight\_ms.aircrafts (

aircraft\_type\_code INT NOT NULL,

aircraft\_type VARCHAR NOT NULL,

PRIMARY KEY (aircraft\_type\_code)

);

--- Tables that reference each other

CREATE TABLE IF NOT EXISTS flight\_ms.flight\_schedules (

flight\_number INT NOT NULL,

airline\_code INT NOT NULL,

usual\_aircraft\_type\_code INT NOT NULL,

origin\_airport\_code INT NOT NULL,

destination\_airport\_code INT NOT NULL,

departure\_date\_time TIMESTAMP NOT NULL,

arrival\_date\_time TIMESTAMP NOT NULL,

PRIMARY KEY (flight\_number),

FOREIGN KEY (airline\_code) REFERENCES flight\_ms.airlines(airline\_code),

FOREIGN KEY (usual\_aircraft\_type\_code) REFERENCES flight\_ms.aircrafts(aircraft\_type\_code),

FOREIGN KEY (origin\_airport\_code) REFERENCES flight\_ms.airports(airport\_code),

FOREIGN KEY (destination\_airport\_code) REFERENCES flight\_ms.airports(airport\_code)

);

CREATE TABLE IF NOT EXISTS flight\_ms.flight\_costs (

flight\_number INT NOT NULL,

aircraft\_type\_code INT NOT NULL,

valid\_from\_date DATE NOT NULL,

valid\_to\_date DATE NOT NULL,

flight\_cost INT NOT NULL,

PRIMARY KEY (flight\_number, aircraft\_type\_code, valid\_from\_date),

FOREIGN KEY (flight\_number) REFERENCES flight\_ms.flight\_schedules(flight\_number) ON UPDATE CASCADE ON DELETE CASCADE,

FOREIGN KEY (aircraft\_type\_code) REFERENCES flight\_ms.aircrafts(aircraft\_type\_code) ON UPDATE CASCADE,

FOREIGN KEY (valid\_from\_date) REFERENCES flight\_ms.ref\_calendar(day\_date),

FOREIGN KEY (valid\_to\_date) REFERENCES flight\_ms.ref\_calendar(day\_date)

);

--- Tables associated with reservation and payments

CREATE TABLE IF NOT EXISTS flight\_ms.itinerary\_reservations (

reservation\_id INT NOT NULL,

agent\_id INT NOT NULL,

passenger\_id INT NOT NULL,

reservation\_status\_code INT NOT NULL,

ticket\_type\_code INT NOT NULL,

travel\_class\_code INT NOT NULL,

date\_reservation\_made DATE NOT NULL,

number\_in\_party INT NOT NULL,

PRIMARY KEY (reservation\_id),

FOREIGN KEY (agent\_id) REFERENCES flight\_ms.booking\_agents(agent\_id) ON UPDATE CASCADE,

FOREIGN KEY (passenger\_id) REFERENCES flight\_ms.passengers(passenger\_id) ON UPDATE CASCADE,

FOREIGN KEY (reservation\_status\_code) REFERENCES flight\_ms.reservation\_statuses(reservation\_status\_code) ON UPDATE CASCADE,

FOREIGN KEY (ticket\_type\_code) REFERENCES flight\_ms.ticket\_codes(ticket\_type\_code) ON UPDATE CASCADE,

FOREIGN KEY (travel\_class\_code) REFERENCES flight\_ms.travel\_classes(travel\_class\_code) ON UPDATE CASCADE

);

CREATE TABLE IF NOT EXISTS flight\_ms.payments (

payment\_id INT NOT NULL,

payment\_status\_code INT NOT NULL,

payment\_date DATE NOT NULL,

payment\_amount INT NOT NULL,

PRIMARY KEY (payment\_id),

FOREIGN KEY (payment\_status\_code) REFERENCES flight\_ms.payment\_statuses(payment\_status\_code) ON UPDATE CASCADE

);

CREATE TABLE IF NOT EXISTS flight\_ms.reservation\_payments (

reservation\_id INT NOT NULL,

payment\_id INT NOT NULL,

PRIMARY KEY (reservation\_id,payment\_id),

FOREIGN KEY (reservation\_id) REFERENCES flight\_ms.itinerary\_reservations(reservation\_id) ON UPDATE CASCADE ON DELETE CASCADE,

FOREIGN KEY (payment\_id) REFERENCES flight\_ms.payments(payment\_id) ON UPDATE CASCADE

);

--- Tables associated with legs

CREATE TABLE IF NOT EXISTS flight\_ms.legs (

leg\_id INT NOT NULL,

flight\_number INT NOT NULL,

origin\_airport VARCHAR NOT NULL,

destination\_airport VARCHAR NOT NULL,

actual\_departure\_time TIMESTAMP NOT NULL,

actual\_arrival\_time TIMESTAMP NOT NULL,

PRIMARY KEY (leg\_id),

FOREIGN KEY (flight\_number) REFERENCES flight\_ms.flight\_schedules(flight\_number) ON UPDATE CASCADE ON DELETE CASCADE

);

CREATE TABLE IF NOT EXISTS flight\_ms.itinerary\_legs (

reservation\_id INT NOT NULL,

leg\_id INT NOT NULL,

PRIMARY KEY (reservation\_id,leg\_id),

FOREIGN KEY (reservation\_id) REFERENCES flight\_ms.itinerary\_reservations(reservation\_id) ON UPDATE CASCADE ON DELETE CASCADE,

FOREIGN KEY (leg\_id) REFERENCES flight\_ms.legs(leg\_id) ON UPDATE CASCADE ON DELETE CASCADE

);

### 2.2 Assertions and Triggers

---ASSERTIONS

---add date constraint to flight\_costs

--- ensure that the "valid to" date is after the "valid from" date

ALTER TABLE flight\_ms.flight\_costs

ADD CONSTRAINT cost\_dates\_constraint

CHECK (valid\_from\_date < valid\_to\_date)

;

---add time constraint to flight\_schedules

---ensure that departure time is before arrival time

ALTER TABLE flight\_ms.flight\_schedules

ADD CONSTRAINT schedule\_time\_constraint

CHECK (departure\_date\_time < arrival\_date\_time)

;

---add airport constraint to flight\_schedules

---ensure that the destination and origin airports are not the same

ALTER TABLE flight\_ms.flight\_schedules

ADD CONSTRAINT airport\_code\_constraint

CHECK (destination\_airport\_code != origin\_airport\_code)

;

---add time constraint to legs

---ensure that departure time is before arrival time

ALTER TABLE flight\_ms.legs

ADD CONSTRAINT valid\_leg\_time

CHECK (actual\_departure\_time < actual\_arrival\_time)

;

---add cost constraint to flight\_costs

---ensure that the cost of the flight is not negative

ALTER TABLE flight\_ms.flight\_costs

ADD CONSTRAINT positive\_flight\_cost

CHECK (flight\_cost > 0);

---add payment constraint to payments

---ensure that the amount of the payment is not negative

ALTER TABLE flight\_ms.payments

ADD CONSTRAINT positive\_payment\_amount

CHECK (payment\_amount > 0);

---add party constraint in itinerary\_reservations

---ensure that the number ina party is not negative

ALTER TABLE flight\_ms.itinerary\_reservations

ADD CONSTRAINT positive\_number\_in\_party

CHECK (number\_in\_party > 0);

---add date constraint to payments

---ensure that the date of a payment is not in the future

ALTER TABLE flight\_ms.payments

ADD CONSTRAINT valid\_payment\_date

CHECK (payment\_date <= CURRENT\_DATE);

---TRIGGERS

---prevent overlapping periods in flight\_costs

CREATE OR REPLACE FUNCTION prevent\_overlapping\_costs()

RETURNS trigger

AS

$BODY$

BEGIN

IF EXISTS (

SELECT \* FROM flight\_ms.flight\_costs

WHERE flight\_number=NEW.flight\_number

AND aircraft\_type\_code=NEW.aircraft\_type\_code

AND (

(NEW.valid\_from\_date BETWEEN valid\_from\_date AND valid\_to\_date)

OR

(NEW.valid\_to\_date BETWEEN valid\_to\_date AND valid\_from\_date)

)

) THEN

RAISE EXCEPTION 'Range must not overlap with exisiting one for flight'

;

END IF;

RETURN NEW;

END;

$BODY$

LANGUAGE plpgsql;

CREATE OR REPLACE TRIGGER overlap\_costs

BEFORE INSERT OR UPDATE ON flight\_ms.flight\_costs

FOR EACH ROW

EXECUTE FUNCTION prevent\_overlapping\_costs();

---set reservation status code to that of pending status when new reservation is created

CREATE OR REPLACE FUNCTION set\_default\_reservation\_status()

RETURNS TRIGGER AS $$

BEGIN

IF NEW.reservation\_status\_code IS NULL

THEN

NEW.reservation\_status=(

SELECT reservation\_status\_code

FROM flight\_ms.reservation\_statuses

WHERE reservation\_status='Pending'

LIMIT 1)

-- SELECT reservation\_status\_code INTO NEW.reservation\_status\_code

-- FROM flight\_ms.reservation\_statuses

-- WHERE reservation\_status = 'Pending'

-- LIMIT 1;

END IF;

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER default\_reservation\_status

BEFORE INSERT ON flight\_ms.itinerary\_reservations

FOR EACH ROW EXECUTE FUNCTION set\_default\_reservation\_status();

---prevent duplicate reservations by the same passenger

CREATE OR REPLACE FUNCTION prevent\_duplicate\_flight\_reservations()

RETURNS TRIGGER AS $$

BEGIN

-- Check if the passenger already has a reservation for this flight

IF EXISTS (

WITH existing\_flight\_no AS (

SELECT flight\_number

FROM flight\_ms.legs

WHERE leg\_id = NEW.leg\_id

LIMIT 1

)

SELECT 1

FROM flight\_ms.itinerary\_reservations AS r

JOIN flight\_ms.itinerary\_legs AS il ON r.reservation\_id = il.reservation\_id

JOIN flight\_ms.legs AS l ON il.leg\_id = l.leg\_id

WHERE r.passenger\_id = NEW.passenger\_id

AND l.flight\_number = existing\_flight\_no

) THEN

RAISE EXCEPTION 'This passenger is already booked for this flight';

END IF;

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER check\_duplicate\_flight\_reservations

BEFORE INSERT ON flight\_ms.itinerary\_reservations

FOR EACH ROW

EXECUTE FUNCTION prevent\_duplicate\_flight\_reservations();

### 2.3 Queries and Views

#### 2.3.1 View customer itinerary

---CREATE VIEW FOR CUSTOMER (passenger\_id=202) ITINERARY

DROP VIEW IF EXISTS itinerary;

CREATE OR REPLACE VIEW itinerary AS

WITH leg\_schedules AS (

SELECT f.flight\_number

,l.origin\_airport

,l.destination\_airport

,l.actual\_departure\_time

,l.actual\_arrival\_time

,i.reservation\_id

,i.leg\_id

FROM flight\_ms.flight\_schedules AS f

JOIN flight\_ms.legs AS l ON f.flight\_number=l.flight\_number

JOIN flight\_ms.itinerary\_legs AS i ON l.leg\_id=i.leg\_id

), ticket\_details AS (

SELECT r.reservation\_id

,r.passenger\_id

,t.ticket\_type

,c.travel\_class

FROM flight\_ms.itinerary\_reservations AS r

JOIN flight\_ms.ticket\_codes AS t ON r.ticket\_type\_code=t.ticket\_type\_code

JOIN flight\_ms.travel\_classes as c ON r.travel\_class\_code=c.travel\_class\_code

)

SELECT leg\_schedules.reservation\_id,

leg\_schedules.flight\_number,

leg\_schedules.origin\_airport,

leg\_schedules.destination\_airport,

leg\_schedules.actual\_departure\_time,

leg\_schedules.actual\_arrival\_time,

leg\_schedules.leg\_id,

ticket\_details.passenger\_id,

ticket\_details.ticket\_type,

ticket\_details.travel\_class

FROM leg\_schedules

JOIN ticket\_details ON leg\_schedules.reservation\_id=ticket\_details.reservation\_id

WHERE ticket\_details.passenger\_id=202

;

#### 2.3.2 Get all passengers who have seats on a given flight

---Get all customers who have seats on a given flight

CREATE OR REPLACE FUNCTION get\_passengers\_on\_flight(flight\_no INT)

RETURNS TABLE (

flight\_number INT,

passenger\_id INT,

first\_name VARCHAR,

second\_name VARCHAR,

last\_name VARCHAR

) AS $$

BEGIN

RETURN QUERY

WITH passenger\_legs AS (

SELECT

p.passenger\_id,

i.leg\_id,

p.first\_name,

p.second\_name,

p.last\_name

FROM flight\_ms.passengers AS p

JOIN flight\_ms.itinerary\_reservations AS r ON p.passenger\_id = r.passenger\_id

JOIN flight\_ms.itinerary\_legs AS i ON r.reservation\_id = i.reservation\_id

)

SELECT

l.flight\_number,

p.passenger\_id,

p.first\_name,

p.second\_name,

p.last\_name

FROM passenger\_legs AS p

JOIN flight\_ms.legs AS l ON p.leg\_id = l.leg\_id

WHERE l.flight\_number = flight\_no;

END;

$$ LANGUAGE plpgsql

;

#### 2.3.3 View flight schedules

DROP VIEW IF EXISTS all\_flight\_schedules;

CREATE OR REPLACE VIEW all\_flight\_schedules AS

SELECT f.flight\_number

,f.departure\_date\_time

,f.arrival\_date\_time

,o.airport\_name AS origin\_airport\_name

,o.airport\_location AS origin\_location

,d.airport\_name AS destination\_airport\_name

,d.airport\_location AS destination\_location

,l.airline\_name

,c.aircraft\_type

FROM flight\_ms.flight\_schedules AS f

JOIN flight\_ms.airports AS o ON f.origin\_airport\_code=o.airport\_code

JOIN flight\_ms.airports AS d ON f.destination\_airport\_code=d.airport\_code

JOIN flight\_ms.airlines AS l ON f.airline\_code=l.airline\_code

JOIN flight\_ms.aircrafts AS c ON f.usual\_aircraft\_type\_code=c.aircraft\_type\_code

;

#### 2.3.4 Get all flights whose arrival and departures are on time/delayed

DROP VIEW IF EXISTS flight\_tracker;

CREATE VIEW flight\_tracker AS

WITH flight\_data AS (

SELECT f.flight\_number

,l.origin\_airport

,l.destination\_airport

,f.departure\_date\_time

,f.arrival\_date\_time

,l.actual\_departure\_time

,l.actual\_arrival\_time

FROM flight\_ms.flight\_schedules AS f

JOIN flight\_ms.legs as l ON f.flight\_number=l.flight\_number

)

SELECT

d.flight\_number

,d.origin\_airport

,d.destination\_airport

,CASE

WHEN departure\_date\_time < actual\_departure\_time

OR arrival\_date\_time < actual\_arrival\_time THEN 'DELAYED'

ELSE 'ON TIME'

END AS tracking

FROM flight\_data AS d

;

#### 2.3.5 Get total sales for a given flight

---Calculate total sales for a given flight

CREATE OR REPLACE FUNCTION get\_total\_sales(flight\_no INT)

RETURNS TABLE (

flight\_number INT,

total\_sales INT

) AS $$

BEGIN

RETURN QUERY

WITH paid\_flights AS (

SELECT

l.flight\_number,

r.reservation\_id,

r.date\_reservation\_made,

r.number\_in\_party,

ps.payment\_status

FROM flight\_ms.legs AS l

JOIN flight\_ms.itinerary\_legs AS i ON l.leg\_id = i.leg\_id

JOIN flight\_ms.itinerary\_reservations AS r ON i.reservation\_id = r.reservation\_id

JOIN flight\_ms.reservation\_payments AS rp ON r.reservation\_id = rp.reservation\_id

JOIN flight\_ms.payments AS p ON rp.payment\_id = p.payment\_id

JOIN flight\_ms.payment\_statuses AS ps ON p.payment\_status\_code = ps.payment\_status\_code

WHERE l.flight\_number = flight\_no

AND ps.payment\_status = 'PAID'

), with\_costs AS (

SELECT

pf.flight\_number,

pf.reservation\_id,

pf.number\_in\_party,

c.flight\_cost

FROM paid\_flights AS pf

JOIN flight\_ms.flight\_costs AS c ON c.flight\_number = pf.flight\_number

WHERE pf.date\_reservation\_made BETWEEN c.valid\_from\_date AND c.valid\_to\_date

), get\_prices AS (

SELECT

w.flight\_number,

w.number\_in\_party \* w.flight\_cost AS total\_paid

FROM with\_costs AS w

)

SELECT

g.flight\_number,

SUM(g.total\_paid) AS total\_sales

FROM get\_prices AS g

GROUP BY g.flight\_number;

END;

$$ LANGUAGE plpgsql

;

## Part 3: Discussion

Whether Views for the queries described in the CUSTOMER FUNCTIONS section of the supplied SRS are created or not, customers are not likely to experience any differences due to performance constraints. I do not think that the creation of Views on its own reduces the overhead on the database; however, altering Views to have specific characteristics can reduce overhead. Query performance will be difficult to scale as database scales up. There are some strategies that will allow performance to scale with the current relational database management system (RDBMS); however, to optimize efficiency as the database grows, non-relational and Big Data technologies will be most beneficial. Similarly, when evaluating the use of a first-normal form RDBMS in a highly transactional web-based application, performance is likely to improve with denormalization and non-relational technologies despite some advantages.

Views are tables derived from one or more tables in the database and do not always exist physically within the database system. A view can be materialized so that it is precomputed and exists physically in the database system. Non-materialized views do not improve query performance because every time the view is called, the underlying query that creates the view must be performed. Creating a non-materialized view accomplishes two things: reduces the amount of data manipulation language (DML) written by the database administrator and creates user-specific data tables for viewing. The first does nothing to benefit the customer, and although the second allows for an experience tailored to customers while also ensuring data privacy, it does not impact performance constraints that the customer might notice compared to a query existing outside of a view. Views may consist of various complex queries that include JOINs and aggregations. For example, the ITINERARY view consists of five full table JOIN operations. Repeatedly querying the view will put significant strain on the database, especially when multiple customers are requesting to see their itineraries at the same time. A materialized view precomputes the underlying query and physically stores the view, meaning it will eliminate the need to perform the queries for each view every time a customer wants to view their itinerary. Ultimately, the materialized view may improve performance by reducing the total number of complex queries being performed but will also take up more storage space that may be crucial as the database scales up.

As the database scales up, some tactics may help to scale query performance, as well. As mentioned previously, materialized views will reduce the number of complex queries needed to be performed; however, as the amount of data in the database increases, these complex queries will become more computationally expensive. Indexes can be utilized to improve query performance. There are clustered and non-clustered indexes. Clustered indexes are recommended to be used for queries that return large amounts of data and to be created on primary keys while nonclustered indexes are used for all other columns. With several primary keys in the flight management system database, such as flight\_number, leg\_id, and reservation\_id, being queried repeatedly, using a clustered index on these columns would greatly improve query performance for queries to view flight schedules and track the times of flights. Materialized views and clustered indexes, however, share a major drawback that is certainly applicable to the flight management system database. Both tactics to improve query performance struggle in databases that are updated frequently. To ensure data consistency, materialized views need to be refreshed often to reflect any changes to the underlying tables, which increases overhead on the database since these updates need to be applied twice. Clustered indexes need to be rebuilt every time the column changes, and because clustered indexes must be built prior to nonclustered indexes, both types need to be rebuilt if both exist in a table. Hardware modifications can aid in scaling query performance. Changing the CPU, RAM, and storage size and type can improve computing power, increase memory, and accommodate the storage and access of larger amounts of data, respectively. All of these changes would contribute to improved query performance as the database grows; however, there is a practical limit to improving the specifications of a single machine. Vertical scaling, another name for what was just described, can help accommodate the growth of a database to a certain extent. At some point, horizontal scaling becomes more effective.

Horizontal scaling is the process of storing the database across multiple machines rather than a single machine. This means that as the database scales, rather than requiring a more powerful machine to accommodate the growth of the database, additional less powerful machines can be installed. Across the multiple machines in the horizontal-scaling architecture, the data is partitioned, allowing each machine to store some data in the database. Having multiple machines that handle the data in the database can improve query performance and result in faster execution. RDBMSs are more difficult to scale horizontally compared to non-relational database systems. To implement this type of architecture effectively, the database system might need to adjust to a non-relational model. Alternatively, migrating the database to a Big Data environment, like Hadoop, can improve query performance as the database scales up.

Another factor that influences performance constraints on the customer functions is normalization in the dataset. While first-normal form provides the benefits of atomic values in columns and reduced data redundancy throughout the database, it can hinder query performance, especially in a highly transactional web-application. Normalization demands more complex, computationally expensive queries. Since columns are not repeated throughout the schema, multiple join operations must be performed during each query for the customer functions. This will put more strain on a system that is already handling lots of user traffic. A system like this will benefit more from denormalization, in which there is controlled redundancy to mitigate the system’s reliance on join operations to query the database.

Many strategies can be employed to reduce query performance constraints related to the customer functions implemented for the flight management system. Overall, I do not think creating views for the customer functions will improve performance. Materialized views may improve query performance; however, they introduce other challenges. Likewise, creating indexes for frequently queried columns can reduce the number of full table scans and improve query performance. In a highly transactional web-application, though, where there is high user traffic and frequent updates to a growing database, many of these strategies will not be able to keep pace as the database scales up. All these considerations should force consideration that a first-normal form RDBMS is not the best fit for this system.

**References**

Cioloca, C., & Georgescu, M. (2011). Increasing database performance using indexes. *Database Systems Journal*, *2*(2), 13-22.

Gadiraju, K. K., Verma, M., Davis, K. C., & Talaga, P. G. (2016). Benchmarking performance for migrating a relational application to a parallel implementation. *Future Generation Computer Systems*, *63*, 148-156.

Jatana, N., Puri, S., Ahuja, M., Kathuria, I., & Gosain, D. (2012). A survey and comparison of relational and non-relational database. *International Journal of Engineering Research & Technology*, *1*(6), 1-5.

Kharade, K. G., Kharade, S. K., Kumbhar, V. S., & Kamat, R. K. (2020). A comparative analysis of using indexed view to improve the performance of SQL. *Analysis of Using Indexed View to Improve the Performance of SQL*, 6-13.

Rolik, O., Ulianytska, K., Khmeliuk, M., Khmeliuk, V., & Kolomiiets, U. (2021, December). Increase efficiency of relational databases using instruments of second normal form. In *2021 IEEE 3rd International Conference on Advanced Trends in Information Theory (ATIT)* (pp. 221-225). IEEE.