Joseph Annand

DSE 6210

Project 1

## Part I: Crow’s Foot Diagram

## 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

### 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

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

#### 2.3.3 View flight schedules

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

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

## 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