CS2102 AY19/20 S2 Project Report

Team No. 17

Topic: FoodSanta FDS Application

Members:

Name	Matric No
Sim Jun Yuen, Darren	A0136233N
Seah Lynn	A0188825M
Goh Si Ning	A0190367Y
Lee Wah	A0183641E

Table of Contents

Project Responsibilities

Member Name	Responsibilities
Darren	Login System, RestaurantStaff
Si Ning	Customers, Riders, Triggers
Lynn	FDSManager, Customers, Triggers
Lee Wah	Riders, Triggers

Overview

Over the course of 2 months, our team has developed a web-based Food Delivery Service (FDS) application, named FoodSanta, that adheres to this project's requirements.

Data Requirements & Functionalities

Here are our general data constraints for our application:

- All users are required to have registered an account before logging in to FoodSanta. Users are then identified with the username that they have chosen when they sign up.
    - This is done in order to replicate popular delivery service applications such as GrabFood and foodpanda where an account is unique to each user.
- Users can only sign up as one of Customer, Restaurant Staff or Rider in the signup page but not a combination of any of them.
    - This restricts users to only be of one user type and another account with another unique identifier (username) has to be created if a user wants to be a rider and a customer.
    - To prevent people from signing up and editing data without permission or verification, FDSManager accounts cannot be signed up for and can only be used by verified personnel such as our own administrators (Let's assume we are a decently-sized FDS company with employees to manage verification/validation of information).
    - Restaurant Staffs must be approved/added by FDS Managers to a restaurant before they can access the account linked to the restaurant in the database.
- Dates/Time in the application are recorded as either TIMESTAMP or DATE data types in PostgreSQL.
    - Order and delivery related timings are recorded in TIMESTAMP as it is assumed there will be many customers ordering from FoodSanta daily.

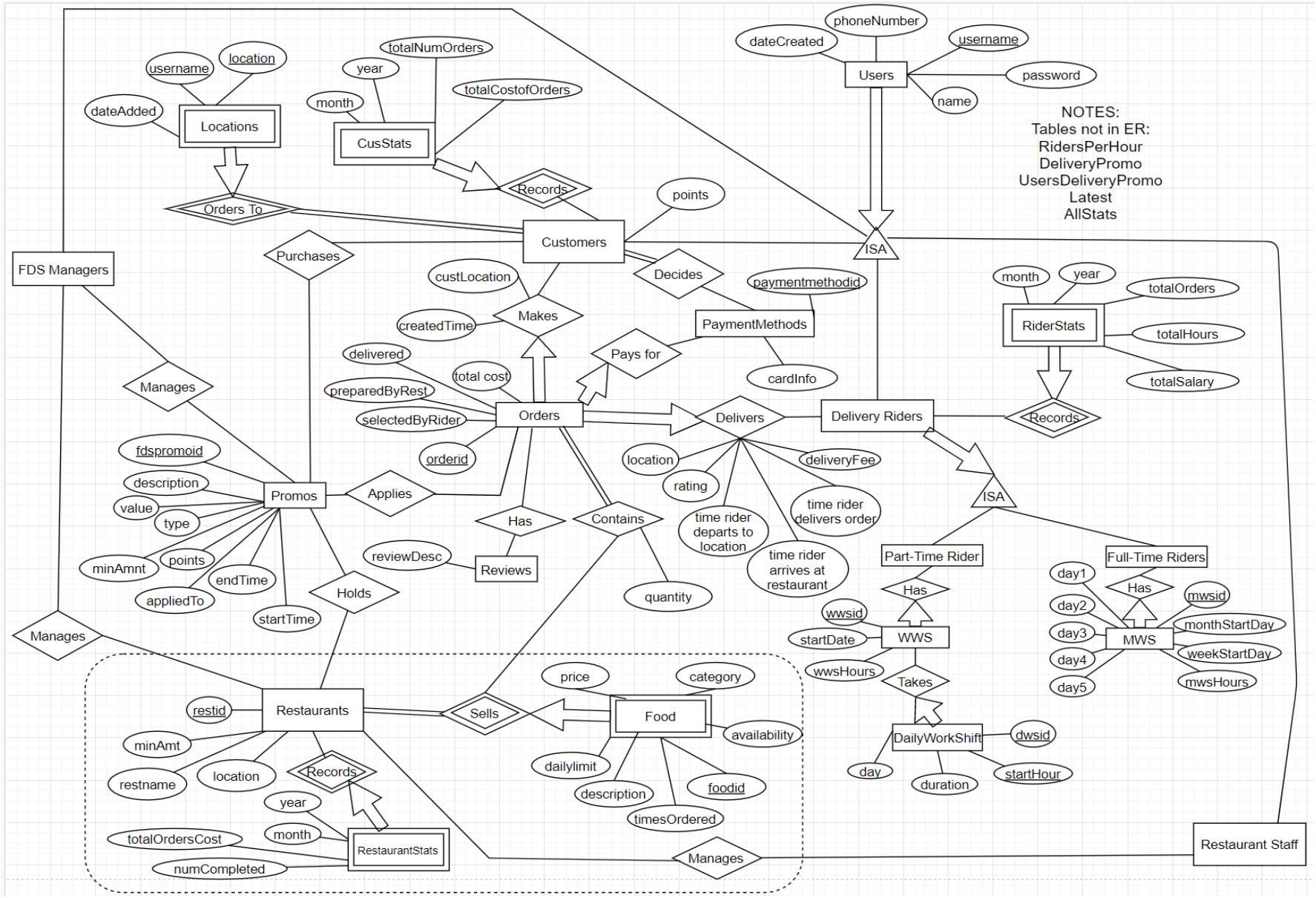
    - Promotion dates and other timings are recorded in DATE as there is no need to be specific to the exact time of creation or update.
- Each restaurant is identified by a unique id and not its name as it is assumed there can be multiple restaurants of the same name but different locations or even menus.
- Promotions are either handed out by FoodSanta (FDS Managers) or the restaurants themselves.
    - Delivery Promos are **fixed** promotions that give customers discounts on the delivery for subsequent orders. These promotions are bought using points that customers earn from spending money in FoodSanta (similar to the other promotions)
- Restaurant Staff can edit/delete/add food items to their restaurant's menu or add/delete restaurant promotions
    - Restaurant promotions cannot be edited to ensure that customers enjoy the same promotion according to their intended benefits and cost.
- Once an order has been confirmed, it cannot be cancelled by the customer. We assume that orders are always intended by the customer since the customer can verify his/her order before checking out.

Interesting Functionalities/Implementation

Here are some of the more interesting/non-trivial aspects of FoodSanta's functionalities/implementation:

- Riders are allocated to undelivered and unallocated orders made by customers according to their availability at that point in time that the order was made and if they are not currently in the middle of delivering an order that has not completed delivery
    - If there are no available riders at that point, the rider that was allocated to the earliest order will then be allocated to that new order
    - This would reduce the amount of time that the customer has to wait after making an order in the event that a rider cannot immediately accept the order
- The FDS manager is also able to add different kinds of promotions that apply to either food orders from restaurants or the delivery fees attached to the order
    - However, since they are not in control of individual restaurants, they do not have the liberty of adding promotions applicable only to certain restaurants
    - This would only be allowed by the restaurant staff of that particular restaurant
- Tables for statistics that the FDSManager and RestaurantStaff can view are included in the database to allow the data addition/update/deletion to be dynamic.
    - For example, when a customer has ordered, the order data will be captured automatically into the statistics table. This prevents runtime for executing complex queries that check various tables for the order data as well as being able to handle present data that could have been deleted later (e.g. when calculating total cost of a previous order that contains a deleted food item).

Entity-Relationship Model



Constraints not captured by the ER diagram

These include:

- Reward points can be used to offset a Customer's delivery charges, but the ER model only reflects points as an attribute of Customers with no other purpose.
- Having at least 5 riders working every hour, but the ER model does not have a numerical limit on the number of riders per hour.
- Once a food item's daily availability drops below 0, it can no longer be selected by Customers. The ER only reflects the availability of a food item as an attribute under the item but has no check in place to ensure food item cannot be selected once availability drops below zero.
- Each order can only be from one restaurant at a time.
- The locations a Customer places their order to is captured under the entity "Locations", but the ER diagram does not reflect that the locations kept are the five more recent locations input by the Customer.
- The total cost of each order is represented as an attribute, but the attributes involved in its calculation are not shown.
- A customer is not allowed to make a new order in the application if he/she does not hit the minimum stipulated threshold of the restaurant. While the ER model does reflect the minimum amount attribute, it does not reinforce that an order will not be added in the above scenario.
- The constraints on working hours of delivery riders
    - Part Time riders having to work for at least 10 hours and at most 48 hours
    - Each shift cannot exceed 4 hours
    - If a rider does take consecutive shifts, there must be at least one hour of break between these shifts
- A week is defined to be a duration consisting of seven consecutive days, and a month consists of four consecutive weeks.
- Constraints regarding monthly work schedule:
    - The ER diagram does not capture the fact that each WWS in MWS must consist of five consecutive workdays. It only captures that the WWS has 5 days.
    - The ER diagram does not reflect the shift timings for each individual shift.
- The rider's salary calculation is not tied to the number of deliveries made, and it is only represented as "totalSalary" under the rider's statistics.
- The FDS application only has operation hours between 10am and 10pm but that is not shown as it is not tied to any one entity in the application.
- There are some specific promotions that are not tied to any restaurants or orders, such as discount coupons to customers who have placed any order for the last 3 months. However, the ER model shows promotions either under FDSPromo or RestaurantPromo.

Relational Schema

Here is our group's relational schema for FoodSanta. All the tables created in our schema **are in BCNF**.

```
CREATE TABLE Users (
    username            VARCHAR(30),
    name                VARCHAR(30) NOT NULL,
    password            VARCHAR(15) NOT NULL,
    phoneNumber         VARCHAR(8) NOT NULL,
    dateCreated         date NOT NULL,

    PRIMARY KEY (username)
);
```

```
CREATE TABLE Customers (
    username            VARCHAR(30),
    points              INTEGER default 0,

    PRIMARY KEY (username),
    FOREIGN KEY (username) REFERENCES Users
);
```

```
CREATE TABLE DeliveryRiders (
    username    VARCHAR(30),

    PRIMARY KEY (username),
    FOREIGN KEY (username) REFERENCES Users
);
```

```
CREATE TABLE FDSManagers (
    username            VARCHAR(30),

    PRIMARY KEY (username),
    FOREIGN KEY (username) REFERENCES Users
);
```

```
-- before insertion, check that customers only has less than 5
-- if not, delete the one with the earliest dateadded and add new one
CREATE TABLE Locations (
    username        varchar(30),
    location        varchar(100),
    dateAdded       DATE NOT NULL,

    PRIMARY KEY (username, location),
    FOREIGN KEY (username) REFERENCES Customers
);
```

```
-- this is so that each customer can have multiple payment methods
-- for every order that requires payment, must look up this table
-- and check username must be the same
CREATE TABLE PaymentMethods (
    paymentmethodid INTEGER,
    username        VARCHAR(30) NOT NULL,
    cardInfo        VARCHAR(60) NOT NULL,

    PRIMARY KEY (paymentmethodid),
    FOREIGN KEY (username) REFERENCES Customers
);
```

```
--insertion of food into Contains table has to decrease
-- availability by one (use trigger under contains)
CREATE TABLE Restaurants (
    restid      INTEGER,
    restName    VARCHAR(50) NOT NULL,
    minAmt      INTEGER NOT NULL,
    location    VARCHAR(100) NOT NULL,

    PRIMARY KEY (restid)
);
```

```sql
CREATE TABLE RestaurantStaff (
    username            VARCHAR(30),
    restid              INTEGER DEFAULT NULL,

    PRIMARY KEY (username),
    FOREIGN KEY (username) REFERENCES Users,
    FOREIGN KEY (restid) REFERENCES Restaurants
);
```

```sql
CREATE TABLE DeliveryPromo (
    deliverypromoid     INTEGER,
    description     VARCHAR(200) NOT NULL,
    amount          INTEGER not null,
    points          INTEGER default 0,

    PRIMARY KEY (deliverypromoid)
);
```

```sql
CREATE TABLE Food (
    foodid          INTEGER,
    description     VARCHAR(50) NOT NULL,
    price           decimal NOT NULL,
    dailylimit      INTEGER NOT NULL CHECK (dailylimit >= 0),
    availability    INTEGER NOT NULL CHECK (availability >= 0),
    category        VARCHAR(20) NOT NULL,
    restid          INTEGER NOT NULL,
    timesordered    INTEGER NOT NULL DEFAULT 0,

    PRIMARY KEY (foodid),
    FOREIGN KEY (restid) REFERENCES Restaurants
);
```

```sql
--insertion into from table needs to check if restid is same as all other restid
CREATE TABLE FDSPromo (
    fdspromoid      INTEGER,
    description     VARCHAR(200) NOT NULL,
    type            VARCHAR(50) NOT NULL,
    CONSTRAINT chck_type CHECK (type IN ('percentoff', 'amountoff')),
    value           INTEGER CHECK (value > 0 AND value <100),
    minAmnt         INTEGER DEFAULT 0,
    appliedto       VARCHAR,
    CONSTRAINT chck_appliedto CHECK (appliedto IN ('total', 'delivery')),
    startTime       DATE NOT NULL,
    endTime         DATE NOT NULL,
    points          INTEGER default 0,

    PRIMARY KEY (fdspromoid)
);
```

```sql
CREATE TABLE UsersPromo (
    fdspromoid      INTEGER,
    username        VARCHAR(30),

    FOREIGN KEY (fdspromoid) REFERENCES FDSPromo ON DELETE CASCADE ON UPDATE CASCADE,
    FOREIGN KEY (username) REFERENCES Customers  ON DELETE CASCADE ON UPDATE CASCADE,
    PRIMARY KEY (fdspromoid, username)
);
```

```sql
CREATE TABLE UsersDeliveryPromo (
    deliverypromoid     INTEGER,
    username        VARCHAR(30),

    FOREIGN KEY (deliverypromoid) REFERENCES DeliveryPromo ON DELETE CASCADE ON UPDATE CASCADE,
    FOREIGN KEY (username) REFERENCES Customers  ON DELETE CASCADE ON UPDATE CASCADE,
    PRIMARY KEY (deliverypromoid, username)
```

```sql
CREATE TABLE RestaurantPromo (
    fdspromoid      INTEGER,
    restid          INTEGER,

    PRIMARY KEY (fdspromoid, restid),
    FOREIGN KEY(fdspromoid) REFERENCES FDSPromo ON DELETE CASCADE
);
```

```sql
CREATE TABLE Contains (
    orderid     INTEGER NOT NULL,
    foodid      INTEGER NOT NULL,
    quantity    INTEGER NOT NULL,

    PRIMARY KEY (orderid, foodid),

    FOREIGN KEY (foodid) REFERENCES Food
);
```

```sql
CREATE TABLE Orders (
    orderid             INTEGER,
    username            VARCHAR(30) NOT NULL,
    custLocation        VARCHAR(100) NOT NULL,
    orderCreatedTime    TIMESTAMP NOT NULL,
    totalCost           decimal NOT NULL,
    fdspromoid          INTEGER,
    paymentmethodid     INTEGER NOT NULL,
    preparedByRest      BOOLEAN NOT NULL DEFAULT FALSE,
    selectedByRider     BOOLEAN NOT NULL DEFAULT FALSE,
    restid              INTEGER NOT NULL,
    delivered           BOOLEAN NOT NULL DEFAULT FALSE,

    PRIMARY KEY (orderid),

    FOREIGN KEY (username) REFERENCES Customers,
    FOREIGN KEY (restid) REFERENCES Restaurants,
    FOREIGN KEY (fdspromoid) REFERENCES FDSPromo
);
```

```sql
CREATE TABLE Reviews (
    orderid         INTEGER,
    reviewDesc      VARCHAR(100),

    PRIMARY KEY (orderid),

    FOREIGN KEY (orderid) REFERENCES Orders
);
```

```sql
CREATE TABLE Delivers (
    orderid                 INTEGER,
    username                VARCHAR(30),
    rating                  INTEGER CHECK ((rating <= 5) AND (rating >= 0)),
    location                VARCHAR(100) NOT NULL,
    deliveryFee             DECIMAL NOT NULL,
    timeDepartToRestaurant  TIMESTAMP,
    timeArrivedAtRestaurant TIMESTAMP,
    timeOrderDelivered      TIMESTAMP,

    PRIMARY KEY (orderid),

    FOREIGN KEY (orderid) REFERENCES Orders,
    FOREIGN KEY (username) REFERENCES DeliveryRiders
);
```

```sql
CREATE TABLE FullTimeRiders (
    username            VARCHAR(30),

    PRIMARY KEY (username),
    FOREIGN KEY (username) REFERENCES DeliveryRiders
);

CREATE TABLE PartTimeRiders (
    username            varchar(30),

    PRIMARY KEY (username),
    FOREIGN KEY (username) REFERENCES DeliveryRiders
);
```

```sql
CREATE TABLE MonthlyWorkSchedule (
    mwsid           INTEGER,
    username        VARCHAR(30),
    mnthStartDay    DATE NOT NULL,
    wkStartDay      INTEGER NOT NULL
                    CHECK (wkStartDay in (0, 1, 2, 3, 4, 5, 6)),
    day1             INTEGER NOT NULL CHECK (day1 in (0, 1, 2, 3)),
    day2             INTEGER NOT NULL CHECK (day2 in (0, 1, 2, 3)),
    day3             INTEGER NOT NULL CHECK (day3 in (0, 1, 2, 3)),
    day4             INTEGER NOT NULL CHECK (day4 in (0, 1, 2, 3)),
    day5             INTEGER NOT NULL CHECK (day5 in (0, 1, 2, 3)),
    mwsHours        INTEGER NOT NULL DEFAULT 0,

    PRIMARY KEY (mwsid),

    FOREIGN KEY (username) REFERENCES FullTimeRiders
);
```

```sql
CREATE TABLE WeeklyWorkSchedule (
    wwsid           INTEGER,
    username        VARCHAR(30),
    startDate       DATE NOT NULL,
    wwsHours        INTEGER NOT NULL DEFAULT 0,

    PRIMARY KEY (wwsid),

    FOREIGN KEY (username) REFERENCES PartTimeRiders
);
```

```sql
CREATE TABLE DailyWorkShift (
    dwsid           INTEGER,
    wwsid           INTEGER,
    day             INTEGER NOT NULL,
    startHour       INTEGER NOT NULL
                    CHECK (startHour >= 10 AND startHour <= 21),
    duration        INTEGER NOT NULL
                    CHECK (duration in (1, 2, 3, 4)),

    PRIMARY KEY (dwsid, startHour),

    FOREIGN KEY (wwsid) REFERENCES WeeklyWorkSchedule
);
```

```sql
CREATE TABLE RidersPerHour (
    username            VARCHAR(30),
    day                 DATE,
    hour                INTEGER,

    PRIMARY KEY (username, day, hour),

    FOREIGN KEY (username) REFERENCES DeliveryRiders
);
```

```sql
CREATE TABLE HoursPerMonth (
    username            VARCHAR(30),
    month               DATE NOT NULL,
    hours               INTEGER NOT NULL DEFAULT 0,

    PRIMARY KEY (username, month),

    FOREIGN KEY (username) REFERENCES DeliveryRiders
);
```

```sql
CREATE TABLE CustomerStats (
    username            VARCHAR(30),
    month               INTEGER,
    year                INTEGER,
    totalNumOrders      INTEGER,
    totalCostOfOrders   INTEGER,

    PRIMARY KEY (username, month, year),
    FOREIGN KEY (username) REFERENCES Customers
);
```

```sql
CREATE TABLE RestaurantStats (
    restid              INTEGER,
    month               INTEGER,
    year                INTEGER,
    numCompletedOrders  INTEGER,
    totalOrdersCost     INTEGER,

    PRIMARY KEY (restid, month, year),
    FOREIGN KEY (restid) REFERENCES Restaurants
);
```

```sql
--use trigger to update the attributes every time
--the rider delivers an order or updates his work schedule
CREATE TABLE RiderStats (
    month           INTEGER,
    year            INTEGER,
    username        VARCHAR(30),
    totalOrders     INTEGER,
    totalSalary     INTEGER,

    PRIMARY KEY (username, month, year),
    FOREIGN KEY (username) REFERENCES DeliveryRiders
);
```

```sql
CREATE TABLE AllStats (
    month               INTEGER,
    year                INTEGER,
    totalNewCust        INTEGER,
    totalNumOrders      INTEGER,
    totalCostOfOrders   INTEGER,

    PRIMARY KEY (month, year)
);
```

```sql
CREATE TABLE Latest (
    orderid         INTEGER,
    restid          INTEGER,

    primary key (orderid),
    foreign key (restid) references Restaurants

);
```

Constraints not captured in Relational Schema

These include:

- Part-time riders can choose not to work a given week
- Part-time riders can work less than 10 hours in a given week
- Part-time riders can work more than 48 hours in a given week
- Full-time riders can choose not to work a given month

- Offering special discount coupons to targeted customers
- Each order's food items must be from a single restaurant
- 5 most recent locations of a Customer
- All the stats constraints may not be fully captured as they are queried or calculated via triggers. This computation is not shown in the schema.
- The constraints whereby food item becomes unavailable to customers once availability drops to 0 is not captured.
- Total cost calculation of orders not captured
- Payment is via cash on delivery (default if customer has no other payment method)
- FDS operation hours not captured

Top 3 Complex Queries Used

**Note that { } syntax refers to user input (e.g. {month} refers to a month value given by the user)*

1. **Filtering out top five items that have been ordered the most for a particular restaurant in a particular month and year**

```
create view topfive (foodid, description, total) as
    select foodid, description, sum(quantity) as total
    from Orders natural join Contains natural join Food
    where restid = {rest_id}
    and extract(month from orderCreatedTime) = {month}
    and extract(year from orderCreatedTime) = {year}
    group by foodid, description
    order by total desc
    limit 5;
```

Further explanation: This query joins Orders, Contains and Food via natural join to retrieve all the foods and their quantities (which can appear more than once as different orders can contain the same food item). Since we are only interested in a particular restaurant in a particular month of a given year, this is filtered out via the where clause where the restid, month and year user input must be the same as that of the orders in the record. The result is then grouped by foodid and description and the total quantity is calculated via the sum aggregate function to get the total number of times the item has been ordered. This result is then sorted in descending order and we can thus extract out the top five items.

2. **Filtering riders currently available to make a delivery for a newly placed order**

```
create view availableRiders (username) as
    select username
    from DeliveryRiders D
    where exists (
        select 1
        from RidersPerHour R
        where D.username = R.username
        and (select extract(isodow from R.day)) =
        (select extract(isodow from current_timestamp))
        and R.hour =
        (select extract(hour from current_timestamp)))
    and not exists (
        select 1
        from Delivers join Orders on (Delivers.orderid = Orders.orderid)
        where Delivers.username = D.username
        and Orders.selectedByRider = True
        and Orders.delivered = False)
```

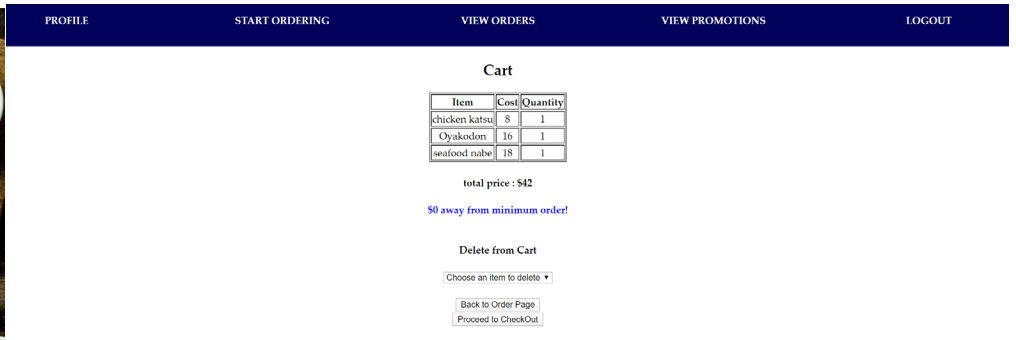
Further explanation: This query filters both full-time and part-time riders who are currently on shift at the time a new order is being made and are not currently busy delivering another order. First, we have to check the Delivery Riders registered in the system with RidersPerHour to find which are the riders working in the current day and time of the week. Then, we need to find those riders who do not have an order under the joint tables of Orders and Delivers that is under his name and has not completed delivery. If there exists such an order, that means the customer is busy delivering another order that has not been completed. Therefore, it would mean he is unable to conduct a delivery.

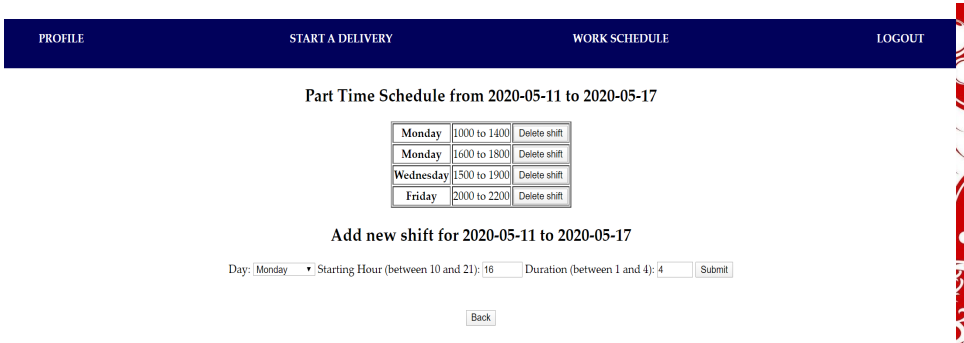
3. **Query for any ongoing promotions at a particular restaurant in the application**

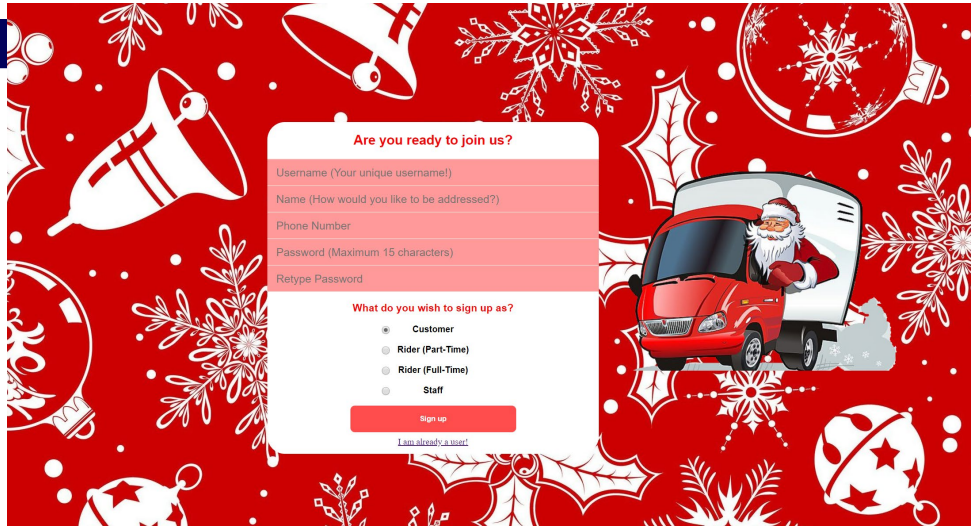
```
create view availableRestaurantPromots (fdspromoid, description, startTime, endTime, points, restname) as
    select P.fdspromoid, P.description, P.startTime, P.endTime, P.points, R.restname
    from FDSPromo P, RestaurantPromo F, Restaurants R
    where P.endTime > (
        select current_date)
    and P.fdspromoid not in (
        select fdspromoid from UsersPromo)
    and P.fdspromoid in (
        select fdspromoid
        from RestaurantPromo)
    and F.restid = R.restid
    and F.fdspromoid = P.fdspromoid
```

Further explanation: This query searches in the FDSPromo table for any promotions at a given restaurant. Since it applies only to a particular restaurant, there should be a matching entry in that of RestaurantPromo that was added by the manager of that particular restaurant. If such a promotion exists and the period of which it is usable has not expired, then it will check whether this promotion has already been applied by this customer. If so, the customer cannot apply the same promotion twice. However, if not, this promotion will be an available promotion for the customer to further add into his cart.

Screenshots of FoodSanta in action







Top 3 Complex Triggers Used

1. **Adding promotions into Customer's account**

```
create or replace function addUsersPromoFunction()
returns trigger as $$
DECLARE
pointsused INTEGER;
customer RECORD;
begin
    select into pointsused (select points from FDSPromo where fdspromoid = NEW.fdspromoid);
    if pointsused = 0 then
        for customer in
            (select username from Customers)
        loop
            insert into UsersPromo values (NEW.fdspromoid, customer.username);
        end loop;
    end if;
return new;
end; $$ language plpgsql;

drop trigger if exists addUsersPromoTrigger on UsersPromo;
create trigger addUsersPromoTrigger
    after insert on FDSPromo
    for each row
    execute function addUsersPromoFunction();
```

Further explanation: In our application, FDSManagers and Restaurants can choose to input new promotions, and determine the points needed to purchase them. This trigger is responsible for entering promotions that are offered 'free-of-charge' (points to redeem is 0) into all of FoodSanta's customer accounts.

Constraints enforced: The constraint enforced here is that if a promotion requires 0 points to redeem, then it should be automatically available to all customers without them having to take any action.

2. Updating Statistics (RestaurantStats in this example)

```
/* Updates restaurant's total number of orders and total cost spent on orders */
create or replace function updateRestaurantStatsFunction()
returns trigger as $$
begin
    /* restid first entry */
    if (not exists(
        select 1
        from RestaurantStats R1
        where R1.restid = NEW.restid
        and R1.month = (select extract(month from current_timestamp))
        and R1.year = (select extract(year from current_timestamp)))) then
        insert into RestaurantStats values(NEW.restid, (select extract(month from current_timestamp)),
         (select extract(year from current_timestamp)), 1, NEW.totalCost);
    /* restid not first entry of monthyear */
    else
        update RestaurantStats R2
        set numCompletedOrders = numCompletedOrders + 1,
            totalOrdersCost = totalOrdersCost + NEW.totalCost
        where R2.restid = NEW.restid
        and R2.month = (select extract(month from current_timestamp))
        and R2.year = (select extract(year from current_timestamp));
end if;
return new;
end; $$ language plpgsql;

drop trigger if exists updateRestaurantStatsTrigger on RestaurantStats;
create trigger updateRestaurantStatsTrigger
    before insert on Orders
    for each row
    execute function updateRestaurantStatsFunction();
```

Further explanation: Along with most of the statistics table, we used triggers to calculate them as they will be fired whenever an order has been placed by a Customer. In the above example, this trigger fires off whenever there is an insert on Orders (i.e. Customer places an order). The trigger function first checks if the restaurant associated with the order has statistics for today's month and year recorded in the table. If it does not, it will insert a new record with the order's total cost and number of completed orders as 1. If there is an existing record, the statistics record is then updated accordingly.

Constraints enforced: Restaurants statistics are computed when a customer makes an order and this computation can only update the record for today's month and year (so that the order is verified to have been made in the current month and year).

3. **Updating labour hours and enforcing constraints**

```
/* update weekly work schedule hours, hours per month and riders per hour upon deletion*/
create or replace function updateDwsDeletionFunction()
returns trigger as $$
DECLARE
totalHours INTEGER;
existingDay TEXT;
monthStart DATE;
oldUsername TEXT;
shiftDate DATE;
hourIterator INTEGER;
hourEnd INTEGER;
count INTEGER;
begin
    -- transforming date into string for error catching
    select case when OLD.day = 0 then 'Monday' when OLD.day = 1 then 'Tuesday' when OLD.day = 2 then 'Wednesday' when OLD.day = 3 then 'Thursday'
                when OLD.day = 4 then 'Friday' when OLD.day = 5 then 'Saturday' when OLD.day = 6 then 'Sunday' end into existingDay
    from DailyWorkShift
    where wwsid = OLD.wwsid;

    -- update wwshours
    update WeeklyWorkSchedule
    set wwsHours = wwsHours - OLD.duration
    where wwsid = OLD.wwsid;

    -- checking new total hours
    select wwsHours into totalHours
        from WeeklyWorkSchedule
        where wwsid = OLD.wwsid;

    -- total hours validity
    if totalHours < 10 then
        raise exception 'FoodSanta: A shift you are trying to delete (%00hrs to %00hrs on %) results in you working less than 10 hours this week! Ho ho ho!',
        OLD.startHour, (OLD.startHour + OLD.duration), existingDay;
    end if;

    -- update hours per month
    select cast(date_trunc('month', startDate + OLD.day) as date) into monthStart from WeeklyWorkSchedule where wwsid = OLD.wwsid;
    select username into oldUsername from WeeklyWorkSchedule where wwsid = OLD.wwsid;
    update HoursPerMonth
    set hours = hours - OLD.duration
    where username = oldUsername
    and month = monthStart;

    -- update riders per hour
    select startDate + OLD.day into shiftDate from WeeklyWorkSchedule where wwsid = OLD.wwsid;
    hourIterator = OLD.startHour;
    hourEnd = OLD.startHour + OLD.duration;
    LOOP
        EXIT WHEN hourIterator >= hourEnd;
        delete from RidersPerHour where username = oldUsername and day = shiftDate and hour = hourIterator;
        -- checking for < 5 riders per hour after deletion
        select count(*) into count from RidersPerHour where day = shiftDate and hour = hourIterator;
        if count < 5 then
            raise exception 'FoodSanta: The shift you are trying to delete (%00hrs to %00hrs on %) results in less than 5 people working at %00hrs! Ho ho ho!',
            OLD.startHour, ((OLD.startHour + OLD.duration) % 24), existingDay, hourIterator;
        end if;
        hourIterator = hourIterator + 1;
    END LOOP;
return new;
end; $$ language plpgsql;

drop trigger if exists updateDwsDeletionTrigger on DailyWorkShift;
create trigger updateDwsDeletionTrigger
    after delete on DailyWorkShift
    for each row
    execute function updateDwsDeletionFunction();
```

Further explanation: Our application allows part time delivery riders to edit their upcoming schedule through insertion and deletion or work shifts. This trigger is responsible for updating the part time rider's hours worked per week and hours worked per month, as well as updating the total number of delivery riders projected to be working for any given hourly interval. Only values relevant to the deleted work shift are calculated and updated.

Constraints enforced: A part time rider should not work less than 10 hours in a given weekly work schedule. There must be at least 5 riders working at each hourly interval.

Specifications of software/frameworks used

As this project is meant to be a simple FDS application, our team has chosen to use the Flask framework (in Python 3.8.2) for its simplicity and convenience for the back-end, and plain HTML-CSS pages for the front-end. We used PostgreSQL 12.0 as our database query language according to our project's specifications and used SQLAlchemy to connect each session in Flask to the local database. Note that we only used SQLAlchemy as a link to execute queries and not any of its Object-Relational Mapping functionalities. Our project also uses the module 'apscheduler' so that the daily limits for food and orders, as well as rider schedules can be generated and updated accordingly.

To start up, the tables and triggers are first loaded into a local database, followed by an .sql file of randomly generated data. In order to boot FoodSanta, start Flask and serve the pages on a local host (usually port 5000 or 5432).

Summary of difficulties and lessons learnt

As our team has little experience in full-stack software engineering and databases, there was a steep learning curve for all of us when trying to implement FoodSanta. We opted to use simpler frameworks such as Flask (and not Django or Javascript-based frameworks) for the back-end and plain HTML-CSS for front-end (instead of ReactJS or jQuery) as it was easier to test and integrate a multi-page web application. This is especially important as all the members are working from home due to the coronavirus pandemic and we could not meet up and integrate/test the application together. We also learnt that local testing can be difficult, especially when members are working on separate branches. Thus, in the future, we could set up one database on a server so that it may be easier to coordinate and test the application when it is running with different users at one time. We tried to host the server on Heroku but after days of trying, we were unable to successfully deploy FoodSanta due to our lack of experience.

Additionally, we initially started to work on our application with the first submission of our ER diagram translated into relational schema. As the applications requirements were not mapped out very clearly in the initial draft, all of us struggled in querying for the results that we needed to display in the application. We eventually had to come together to rewrite the schema more concisely and point out the constraints we think could be enforced

using triggers, and it became a lot easier to insert and query for the appropriate data from the front-end. We hence learnt the importance of a relational schema that reflects the application and data constraints.