National University of Singapore

CS2102 Group Project Report

AY19/20 Semester 2

Team 16

Hans Kurnia	A0184145E
Lee Yueyu	A0188055X
Liu Chaojie	A0177842U
Liu Jiajun	A0177869B

Table of content

Introduction

FDS is an integrated food delivery service application with an aim of managing a food delivery service and enabling customers to order food from this food delivery service. Users of this application, namely customers of restaurants, delivery riders of restaurants, the restaurants themselves and managers of the food delivery service, are able to access and edit their data easily with this application. Specific details of how these users can use the application can be found in the Functionalities Section.

Notations

- Mark-up text highlights functions, triggers, or tables
- **Note** *note boxes provide additional information*
- **Highlight** *highlight boxes highlight interesting aspects of the application*

Functionalities

The application supports four types of users: customers, delivery riders, restaurants, and managers. The functionalities for each of the user types are detailed below:

All users can
- Create a non-manager account and login.
- Update user information, including generic information (such as username and password), and role specific information (such as restaurant address).

Customers can
- View the menus from all restaurants.
- Order food from the restaurants by cash or card.
- Store up to five recently used addresses.
- Track the delivery status of the orders.
- Review orders and rate delivery service, and browse reviews from other customers.
- Earn reward points and use the points to offset delivery fee.
- Fuzzy search for foods based on food name and filter the results by category and price range.

Restaurants can
- Manage the menu and set limits on food items.
- Track the status of the received orders.
- Manage restaurant-level promotions
- View restaurant statistics, such as monthly orders and earnings.

Riders can
- Opt to be either a part time or full time rider
- Manage weekly or monthly work schedules
- View weekly or monthly salaries
- Be assigned delivery tasks by the system based on availability, distance, and current number of orders
- Update the delivery status
- View rider statistics

Managers can
- Manage users: browse all users; remove users; create manager accounts.
- Manage FDS-level promotions.

- View FDS statistics, such as monthly revenue, monthly new users, and most active customers
- View detailed rider statistics

Noteworthy implementation

Schedule Management Module

The Schedule Management Module (SMM) allows riders to plan their work schedules easily. SMM is designed to be memory efficient, while minimizing the complexity of enforcing constraints on schedules. The SMM consists of two main parts: Weekly Work Schedule (WWS) and Monthly Work Schedule (MWS).

A WWS is stored as multiple hour intervals on different days in a week. Each record in the WWS relation exactly represents one such interval. Each hour interval is specified with start_hour, end_hour and day_in_week in the record. The date start_of_week in the record helps distinguish which week the record belongs to. An illustration of extracting all k intervals of a week starting on date D from WWS is shown below:

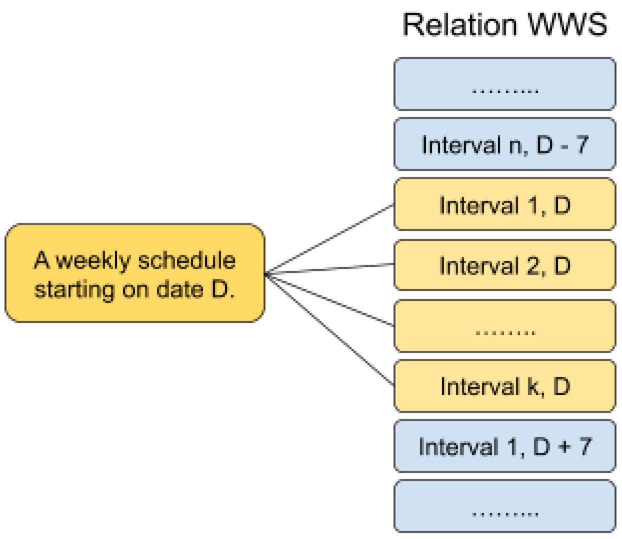


Figure 1. Extracting k intervals of a week starting on date D from WWS

Highlight: This design can easily enforce constraints such as 'maximum four consecutive hours in an interval' and 'one hour break between two hour intervals'.

Given a MWS is composed of four equivalent WWSs, each MWS can be compressed as a prototype week. Each prototype week is stored as a record in the MWS relation, containing seven shift numbers for all days in that week. Relation Shifts can translate shift numbers into hour intervals. A complete MWS can then be obtained by repeating the prototype week for four times. Its logic flow is shown as below:

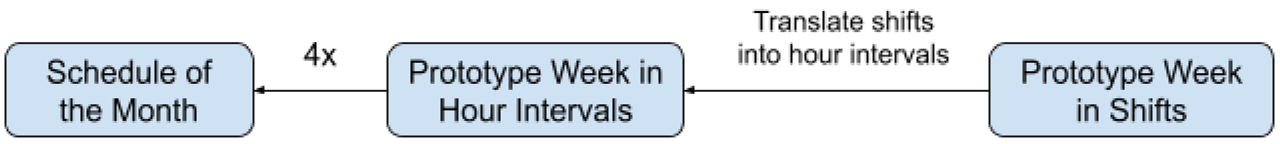

Schedule of the Month
4x
Prototype Week in Hour Intervals
Translate shifts into hour intervals
Prototype Week in Shifts

Figure 2. Obtaining a complete MWS

Highlight:
1. This design simplifies enforcing the constraint 'five consecutive work days in a week' as all shifts are stored in one record.
2. This design is memory efficient because a 28-day schedule can be compressed into a single record, by removing hour intervals from MWS into Shifts to eliminate data redundancy.

Note:
While using start_date to distinguish the weeks and months, check constraints on start_date are also implemented to prevent any overlapping of months or weeks. (e.g. distance among start_dates is constrained to be 28 to synchronize months).

Promotion Management Module

The PMM allows application managers and restaurants owners to manage promotional campaigns easily. The PMM consists of three main entities: **rule**, **action**, and **promotion**. A **rule** defines a series of conditions that an order should satisfy to participate in a promotion. An **action** defines the result of a promotion. A **promotion** is created by combining exactly one rule and exactly one action. A promotion also has its own conditions, including start and end time and eligible domain (FDS-level promotion or restaurant-level promotion). In order for a promotion's associated action to be triggered on an order, the order must satisfy the conditions specified in that promotion and the conditions specified by the rule associated with that promotion. The relationship between the entities are shown in the figure below:

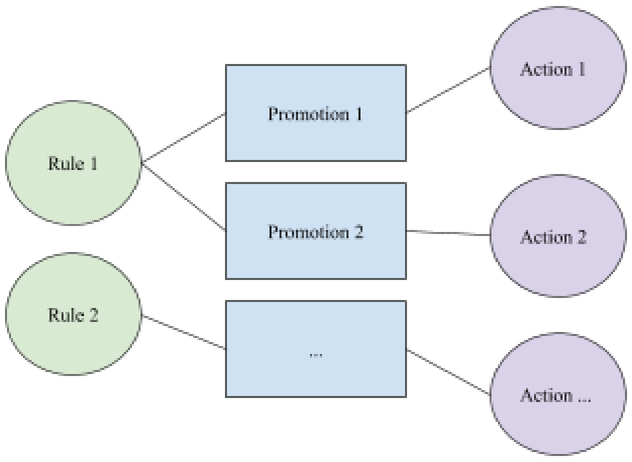


Figure 3. The relationship between Promotion, Rule and Action

Currently, the following types of actions and rules are supported:

Entity	Type	Description
Rule	Nth order	Satisfied when an order is the Nth order that the customer places in a certain restaurant or in FDS
	Order total	Satisfied when order total reaches a certain amount
Action	Food discount	Fixed amount discount or percentage discount on order food cost
	Delivery discount	Fixed amount discount on delivery cost

ℹ️ **Note:** *if an order satisfies multiple actions of the same type, only one of the actions of that type which results in* ***highest discount*** *will be applied to that order. For example, if an order satisfies two food discount actions (a) $5 off, (b)$10 off, only (b) will be applied. However, it is possible for an order to use two actions from different action types.*

🍭**Highlight**: *We choose to decompose promotion into rules and actions because it allows users to reuse their predefined rules and actions to create new promotions. This makes the design more flexible and extensible.*

Data constraints

This section presents important data constraints of the application. Constraints marked with "🌟" are enforced using triggers.

- A user can only have exactly one role (roles are customer, manager, rider and restaurant). 🌟
- A customer has at most five distinct recent addresses and at most one credit card. 🌟
- A credit card is used by at least one customer.
- A restaurant has exactly one address and sells zero or more food items.
- A promotion
  - can only be given by FDS managers or restaurants. Promotion given by managers can be used in all restaurants; promotions given by a restaurant can only be used in the restaurant which sets the promotion. 🌟
  - should have exactly one rule and exactly one action;
  - is applied to an order if and only if certain criteria are satisfied (see Noteworthy Implementation-Promotion) 🌟
- An order
  - can only be placed from 10 a.m. to 10 p.m. daily
  - all food items in that order should be from a single restaurant;
  - must have at least one food item;
  - is delivered to exactly one customer address
  - is delivered by exactly one delivery rider;
  - has at most one review on the foods and at most one review on the delivery service.
- A rider has exactly one role (either part- time(PT) or full-time(FT))
  - FT rider works monthly work schedule (MWS), PT rider works weekly work schedule (WWS)
- In a MWS
  - MWS is a weak entity to FT rider
  - All months of one FT rider do not overlap.
  - Four WWSs in a MWS are equivalent.
  - A week must have exactly five and consecutive work days. 🌟
  - Each work day consists of two four-hour periods and conforms to one of the four given shifts.
- In a WWS
  - WWS is a weak entity to PT rider
  - All weeks of one PT rider do not overlap.
  - The total number of hours is between 10 and 48 hours. 🌟
  - Each rider each day cannot have overlapping time intervals.
  - Each time interval starts and ends on the hour and its duration cannot exceed four hours.
  - There is at least one hour of break between two time intervals.
- Salary is a weak entity to Riders.

Design concerns

Security

Security is an important concern for the application. The security of our application is ensured using multiple approaches.

- *Encryption*: User sensitive information such as passwords are stored in the database only after being hashed with a salt, minimizing the risk of a password leak.
- *Prepared SQL statements:* SQL queries are executed using prepared statements as opposed to direct string concatenation. This limits the risk of SQL injection attacks.

Defensive programming

We strive to actively practice defensive programming in our application. For example, even though we enforce as many constraints as possible for this application in our SQL schema itself, we still check for these same constraints on the backend to make sure our design is robust enough to handle unexpected use cases and future modifications to the application requirements. One example is when we check the format of any credit card information (CVV, 16 digit number) given using SQL constraints while also checking it on the backend.

Entity-Relationship model

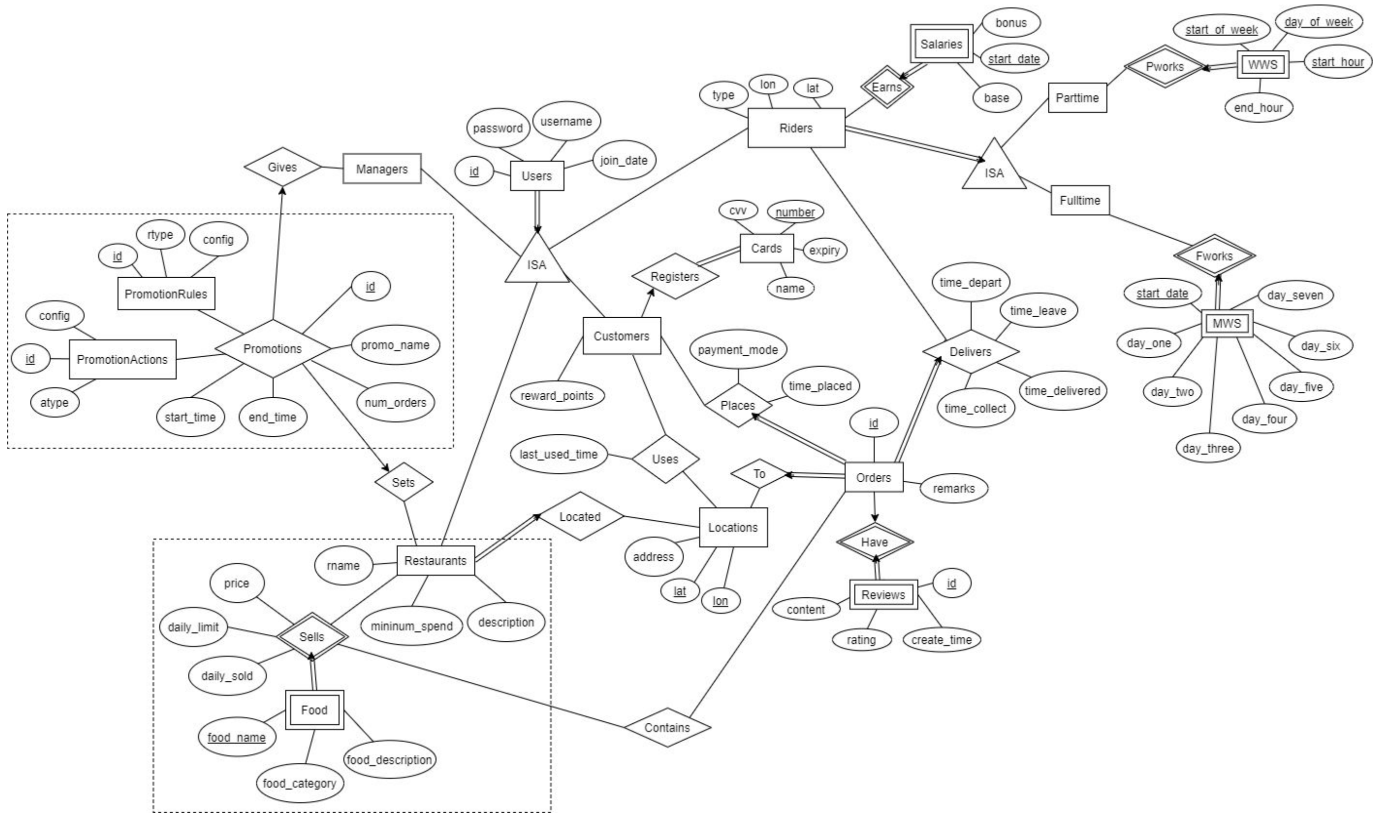


Figure 4. Entity-Relationship Diagram

Schema

Note: All *tables in the schema are in BCNF.*

```
create table Users (
   id        varchar(20) primary key,
    password  text        not null,
    username  varchar(50) not null,
    join_date DATE        not null default CURRENT_TIMESTAMP
);

create table Managers (
    id varchar(20) primary key references Users (id) on delete cascade
);

create table Restaurants (
    id            varchar(20) primary key references Users (id) on delete cascade,
    rname         varchar(50),
    description   varchar(1000), -- brief information of the restaurant,
    minimum_spend money        not null default 0 check (minimum_spend > 0::money),
    address       varchar(100) not null,
    lon           float        not null check (fn_check_lon(lon)),
    lat           float        not null check (fn_check_lat(lat))
);

create table Customers (
    id            varchar(20) primary key references Users (id) on delete cascade,
    reward_points float not null default 0 check (reward_points >= 0)
);

create table Riders (
    id   varchar(20) primary key references Users (id) on delete cascade,
    type rider_type_t not null,
    lon  float check (fn_check_lon(lon)),
    lat  float check (fn_check_lat(lat))
);

create table Sells (
    rid              varchar(20) references Restaurants (id) on delete cascade,
    food_name        varchar(50) check (not food_name = ''),
    food_description text,
    food_category    food_category_t not null,
    daily_limit      integer         not null,
    daily_sold       integer         not null default 0,
    price            money           not null check (price >= 0::money),

    constraint daily_limit check (daily_limit >= 0 and daily_limit >= daily_sold),
    constraint daily_sold check (daily_sold >= 0),
    primary key (rid, food_name)
);

create table CustomerLocations (
    cid            varchar(20),
    lon            float check (fn_check_lon(lon)),
    lat            float check (fn_check_lat(lat)),
    address        varchar(100) not null,
    last_used_time timestamp    not null default CURRENT_TIMESTAMP,

    foreign key (cid) references Customers (id) on delete cascade,
    primary key (cid, lon, lat)
);

create table Orders (
    id             serial,
    rid            varchar(20)    references Restaurants (id) on delete set null,
    delivery_cost  money          not null default 0::money check (delivery_cost >= 0::money),
    food_cost      money          not null default 0::money check (food_cost >= 0::money),

    -- delivery information
    rider_id       varchar(20)             default null references Riders (id),
    cid            varchar(20),
    lon            float check (fn_check_lon(lon)),
    lat            float check (fn_check_lat(lat)),

    -- timing information
    time_placed    timestamp      not null default CURRENT_TIMESTAMP,
    time_depart    timestamp      default null,
    time_collect   timestamp      default null,
    time_leave     timestamp      default null,
    time_delivered timestamp      default null,
    time_paid      timestamp      default null,

    payment_mode   payment_mode_t not null,
    remarks        varchar(500)   default '',

    foreign key (cid, lon, lat) references CustomerLocations (cid, lon, lat) on delete set null,
    primary key (id)
);

create table OrderFoods (
    id        serial,
    oid       integer not null references Orders (id) on delete cascade,
    rid       varchar(20),
    food_name varchar(50),
    quantity  integer not null check (quantity > 0),

    foreign key (rid, food_name) references Sells (rid, food_name) on delete set null,
    unique (oid, rid, food_name),
    primary key (id)
);

create table Reviews  (
   id          serial,
   oid         integer references Orders (id) on delete cascade,
   rating      integer check (rating >= 1 and rating <= 5),
   content     varchar(1000)       not null,
   create_time timestamp           not null default CURRENT_TIMESTAMP,
   primary key (id, oid),
);

create table Registers  (
   cid varchar(20) references Customers(id),
   card_number varchar(19) references Cards(number),
   primary key (cid, card_number),
   unique (cid)
);

create table Cards  (
   number varchar(19) check (number ~ $$\d{4}-?\d{4}-?\d{4}-?\d{4}$$),  -- 16 digits (optionally separated
by hyphens)
   expiry varchar(7)  not null check (expiry ~ $$^(0[1-9]|1[0-2])\/?([0-9]{4}|[0-9]{2})$$), -- valid
formats: MM/YY, MMYY, MM/YYYY, MM/YY
   name   varchar(20) not null,
   cvv    varchar(4)  not null check (cvv ~ $$^[0-9]{3,4}$$)   ,     -- 3 or 4 digits
   primary key (number)
);

create table PromotionRules (
    id       serial primary key,
    giver_id varchar(20) not null references Users (id) on delete cascade
                         check (fn_check_promotion_giver_domain(giver_id)),
    rtype    promo_rule_t,
    config   jsonb
);

create table PromotionActions (
    id       serial primary key,
    giver_id varchar(20) not null references Users (id) on delete cascade
                         check (fn_check_promotion_giver_domain(giver_id)),
    atype    promo_action_t,
    config   jsonb
);

create table Promotions (
    id         serial primary key,
    promo_name varchar(50)       not null,
    rule_id    integer           not null references PromotionRules (id) on delete cascade,
    action_id  integer           not null references PromotionActions (id) on delete cascade,
    num_orders integer default 0 not null,
    start_time timestamp         not null,
    end_time   timestamp         not null check (start_time <= end_time),
    giver_id   varchar(20)       not null references Users (id) on delete cascade
                                 check (fn_check_promotion_giver_domain(giver_id))
);

create table MWS (
    rid        varchar(20) references Riders (id) on delete cascade,
    start_date date,
    day_one    shift_t not null default '0',
    day_two    shift_t not null default '0',
    day_three  shift_t not null default '0',
    day_four   shift_t not null default '0',
    day_five   shift_t not null default '0',
    day_six    shift_t not null default '0',
    day_seven  shift_t not null default '0',

    check (fn_get_rider_type(rid) = 'full_time'),
    check (fn_check_start_date(rid, start_date)),
    primary key (rid, start_date)
);

create table WWS (
    rid           varchar(20) references Riders (id) on delete cascade,
    start_of_week date    not null,
    day_of_week   integer check (day_of_week in (0, 1, 2, 3, 4, 5, 6)),
    start_hour    integer not null check (start_hour >= 10 and start_hour <= 21),
    end_hour      integer not null check (end_hour >= 11 and end_hour <= 22),

    check (fn_get_rider_type(rid) = 'part_time'),
    check (end_hour - start_hour <= 4 and end_hour > start_hour),
    check (fn_check_time_overlap(rid, start_of_week, day_of_week, start_hour, end_hour)),
    check (fn_check_start_of_week(rid, start_of_week)),
    primary key (rid, start_of_week, day_of_week, start_hour)
);

create table Salaries (
    rid        varchar(20) references Riders (id) on delete cascade,
    start_date date,
    base       money not null,
    bonus      money not null default 0,

    check (fn_check_salary_date(rid, start_date)),

    primary key (rid, start_date)
);
```

Triggers

WWS insertion

This trigger (a) ensures that part time riders work at least 10 hours and no more than 48 hours each week and (b) inserts or updates the base salary of the week for this rider according to work hours.

```
create or replace function fn_set_WWS() returns trigger as
$$
declare
   work_hours integer;
begin
   if (not exists (select 1 from Users where id = coalesce(new.rid, old.rid))) then return null;
   end if;

   select sum(end_hour - start_hour)
   into work_hours
   from WWS
   where WWS.rid = coalesce(new.rid, old.rid)
     and WWS.start_of_week = coalesce(new.start_of_week, old.start_of_week);

   if (work_hours > 48)
   then
       raise exception 'Work hours for the week exceed 48 hours';
   elsif (work_hours < 10 or work_hours is null)
   then
       raise exception 'Work hours for the week are less than 10 hours';
   end if;

   if (exists(select 1
              from Salaries
              where Salaries.rid = coalesce(new.rid, old.rid)
                and Salaries.start_date = coalesce(new.start_of_week, old.start_of_week)))
   then
       update Salaries
       set base  = work_hours * 6,
           bonus = 0 -- base salary should be changed.
       where Salaries.rid = coalesce(new.rid, old.rid)
         and Salaries.start_date = coalesce(new.start_of_week, old.start_of_week);
       return null;
   end if;

   insert into Salaries
   values (new.rid, new.start_of_week, work_hours * 6, 0); -- $6 per hour

   return null;
end;
$$ language plpgsql;
```

```
drop trigger if exists tr_set_WWS on WWS cascade;
create constraint trigger tr_set_WWS
   after update or insert or delete
   on WWS
   deferrable initially deferred
   for each row
execute function fn_set_WWS();
```

Five consecutive work days in a week of MWS

This trigger ensures that full time riders work five consecutive days in a week of MWS

```
create trigger tr_MWS_check_shifts
   after update or insert
   on MWS
   for each row
execute function fn_check_shifts();
```

fn_check_shifts is used to check the shifts. It uses window sized 3 to store indexes of days in break.

```
create or replace function fn_check_shifts() returns trigger as
$$
declare
   first_rest  integer := -1;
   second_rest integer := -1;
   third_rest  integer := -1; -- create a window sized 3.
   week_schedule integer[7];
begin
   week_schedule := array [new.day_one, new.day_two, new.day_three, new.day_four, new.day_five,
new.day_six, new.day_seven];
   for counter in 1..7
       loop
           if (week_schedule[counter] = '0')
           then
               third_rest := second_rest;
               second_rest := first_rest;
               first_rest := counter;
               -- slide the rest day into the window.
           end if;
       end loop;

   if (third_rest <> -1 or second_rest = -1) then -- guarantees exact 5 working days
       raise exception 'Exact 5 working days are required in a week.';
   end if;
   if (first_rest - second_rest = 1 or first_rest - second_rest = 6) then -- guarantees consecutive working
days
       return null;
   else
       raise exception '5 working days must be consecutive in a week.';
   end if;
```

```
end;
$$ language plpgsql;
```

Promotion trigger

The promotion trigger checks for eligible promotions based on promotion rules and applies the promotion actions accordingly whenever an order is placed. The detailed data constraint for promotions can be found in the PMM section.

```
create constraint trigger tr_apply_promo
   after insert
   on Orders
   deferrable initially deferred
   for each row
execute function fn_apply_promo();
```

fn_apply_promo is used to apply promotions to an order

```
create or replace function fn_apply_promo() returns trigger as
$$
declare
    eligible_promo_record record;
    new_food_cost         money; -- food cost after discount
    new_delivery_cost     money; -- delivery cost after discount
begin
    select food_cost from Orders where id = new.id into new_food_cost;
    select delivery_cost from Orders where id = new.id into new_delivery_cost;

    for eligible_promo_record in
        with PromotionDiscounts as (
        -- this CTE gets eligible promotions and calculates their discount amounts
        -- a possible tuple in this CTE may be ($6.00, 'Food discount', 6, 3, '6$ off on orders above 10')
        -- meaning promotion with id 3 and name '6$ off...' is eligible for this order, and it will result
in $6 discount on food cost
            select (
                case PromotionActions.atype
                    when 'FOOD_DISCOUNT' then
                        get_food_discount(PromotionActions.id, PromotionActions.atype,
PromotionActions.config, new.id)
                    when 'DELIVERY_DISCOUNT' then
                        get_delivery_discount(PromotionActions.id, PromotionActions.atype,
PromotionActions.config, new.id)
                    end) as amount, -- the discount amount for of a promotion for a given order
                PromotionActions.atype as atype,  -- the promotion action associated with the promotion
                Promotions.id          as pid,
                Promotions.promo_name  as pname
            from Promotions
                join PromotionRules on Promotions.rule_id = PromotionRules.id
                join PromotionActions on Promotions.action_id = PromotionActions.id
            where (select now())
                between Promotions.start_time and Promotions.end_time  -- eligible time period
            and (Promotions.giver_id = new.rid or
                exists(select 1 from Managers where Managers.id = Promotions.giver_id))
                -- promotion domain check, can only be promotion from managers (global) or from restaurant
owner (local)
            and check_rule(PromotionRules.id, PromotionRules.rtype, PromotionRules.config, new.id)
                -- promotion rule eligibility check. E.g. Order Total
        )
        select distinct on (atype) amount, atype, pid, pname
        from PromotionDiscounts
        order by atype, amount desc --gets maximum discount amount from each action type
        loop
            -- loops through each eligible promotion to update new food and delivery cost
            case eligible_promo_record.atype
                when 'FOOD_DISCOUNT' then if (new_food_cost <= 0::money) then continue; end if;
                new_food_cost = new_food_cost - eligible_promo_record.amount;
                when 'DELIVERY_DISCOUNT' then if (new_delivery_cost <= 0::money) then continue; end if;
                new_delivery_cost = new_delivery_cost - eligible_promo_record.amount;
            end case;
            update Promotions set num_orders = num_orders + 1 where id = eligible_promo_record.pid;
            raise notice 'Promotion [%] is applied to order [%]', eligible_promo_record.pid, new.id;
        end loop;
        update Orders set food_cost = new_food_cost, delivery_cost = new_delivery_cost where id = new.id;
        raise notice 'New food cost: %. New delivery cost: %', new_food_cost, new_delivery_cost;
        return null;
end
$$ language plpgsql;
```

fn_apply_promo() uses check_rule() to check if an order satisfied the rules associated with promotions. Note that we store rules' specifications in json format instead of in many table columns. This is because different rule type have very different specification attributes, and storing them as json reduces data redundancy (i.e. avoid many null values in the table) and increases the extensibility of the module:

```
create or replace function check_rule(rid integer, rtype promo_rule_t, rconfig jsonb, oid integer) returns
boolean as
$$
declare
    order_record Orders % rowtype;
begin
    select * from Orders where id = oid into order_record;
    case rtype
        when 'ORDER_TOTAL'::promo_rule_t
            then return (select (rconfig ->> 'cutoff')::money from PromotionRules where PromotionRules.id =
rid) <= order_record.food_cost;
        when 'NTH_ORDER'::promo_rule_t then if (select (rconfig ->> 'domain') = 'restaurant') then
            return (select count(*) from Orders where Orders.rid = order_record.rid and Orders.cid =
order_record.cid)
             = (select (rconfig ->> 'n')::integer);
        elsif (select (rconfig ->> 'domain') = 'all') then
            return (select count(*) from Orders where Orders.cid = order_record.cid)
```

```
                 = (select (rconfig ->> 'n')::integer);
        end if;
        end case;
    return true;
end;
$$ language plpgsql
```

Note: Similarly, the config of actions are stored in json format. get_food_discount() and get_delivery_discount() are used to calculate the discount amount based on action. You can find their implementation here

Complex SQL queries

FDS statistics

For each month in the $month_count months, get the total number of new FDS users, total revenue from orders, and total number of orders, including those months where no orders have been placed and no new users have joined.

```
with recursive MonthlyCalendar as (
   select CURRENT_TIMESTAMP as date
   union all
   select date - interval '1 month'
   from MonthlyCalendar
   where date > CURRENT_TIMESTAMP - interval '$month_count month'
)
select to_char(mc.date, 'YYYY-MM')                                            as yearmonth,
      count(distinct c.id)                                                    as total_new_users,
      coalesce(sum(o.food_cost::numeric + o.delivery_cost::numeric), 0::numeric) as total_order_revenue,
      count(distinct o.id)                                                    as total_order_num
from MonthlyCalendar mc
        left join (Customers natural join Users) c
                  on date_trunc('year', c.join_date) = date_trunc('year', mc.date) and
                     date_trunc('month', c.join_date) = date_trunc('month', mc.date)
        left join Orders o
                  on date_trunc('year', o.time_placed) = date_trunc('year', mc.date) and
                     date_trunc('month', c.o.time_placed) = date_trunc('month', mc.date)
group by mc.date
order by yearmonth desc;
```

ℹ **Note:** The recursive CTE is used to generate a list of past $month_count months. Using a recursive CTE may be time-consuming if $month_count is large. In such cases, it may be better to pre-generate a table of months.

Advanced search

Gets all food name fn, food category fc and restaurant name rn from FDS, where the fn should be similar to $food_name (threshold: 0.1), fc is one of the categories specified in $categories, and the average price of foods from restaurant rn is between $lower_price and $upper_price.

```
select set_limit(0.1)
select food_name, food_category, rname
from Sells S
        join Restaurants R on S.rid = R.id
where food_category in ($categories)
 and (select avg(S2.price::numeric) from Sells S2 where S2.rid = R.id) between $lower_price
 and $upper_price
 and food_name % $food_name
order by similarity(food_name, $food_name) desc
limit 20;
```

During testing, we found that food search could become slow when the table becomes large due to the costly similarity() function. To speed up searching, we created a Generalized Search Tree (GiST) index, which is suitable for text matching, on the Sells table:

```
create index trgm_idx on Sells using gist (food_name gist_trgm_ops);
```

Rider Assignment

This query is about choosing the most suitable riders to deliver an incoming order.
Given a rider x, an order o and timestamp t, *isWorking(r, t)* returns true if x is working (based on the declared schedule) at time t; otherwise it returns false. *distance(r, o)* returns the distance between r's location and the location of the restaurant for which the order is placed. *numDuties(r, t)* is the number of orders that r is delivering at time t.
Given two riders r1 and r2 and an incoming order o, r1 has higher priority if one of the following conditions hold

- *numDuties(r1, o.time_placed) < numDuties(r2, o.time_placed)*
- *numDuties(r1, o.time_placed) = numDuties(r2, o.time_placed)* and *distance(r1, o) < distance(r2, o)*

Given an incoming order o, a set of all riders R, this query finds from the set R' = {r ∈ R | *isWorking(r, o.time_placed)* = true } such that {r4 ∈ R' | ∀ r3∈ R', r4 has higher or equal priority than r3} is the set of most suitable riders.

```
with
AvailableRiders as -- CTE1
(select rid as rider_id from WWS
where start_of_week + day_of_week + (start_hour || ' hour')::interval <= CURRENT_TIMESTAMP
and start_of_week + day_of_week + (end_hour || ' hour')::interval > CURRENT_TIMESTAMP
union -- union both available part time and full time riders.
select rid as rider_id
from MWS F, Shifts S
where CURRENT_DATE - F.start_date < 28 and CURRENT_DATE >= F.start_date
and S.shift_num = case (CURRENT_DATE - F.start_date) % 7
    when 0 then F.day_one when 1 then F.day_two when 2 then F.day_three when 3 then F.day_four
    when 4 then F.day_five when 5 then F.day_six when 6 then F.day_seven
    end
and ((CURRENT_DATE + (S.first_start_hour || ' hour')::interval <= CURRENT_TIMESTAMP
    and CURRENT_DATE + (S.first_end_hour || ' hour')::interval > CURRENT_TIMESTAMP)
    or
    (CURRENT_DATE + (S.second_start_hour || ' hour')::interval <= CURRENT_TIMESTAMP
    and CURRENT_DATE + (S.second_end_hour || ' hour')::interval > CURRENT_TIMESTAMP))),
RiderStatus as -- CTE2
```

```
(select A.rider_id as rider_id, count(O.id) as num_orders,
point(R.lon, R.lat) <@> point(restaurant_lon, restaurant_lat) as distance
from AvailableRiders A join Riders R on A.rider_id = R.id
left join Orders O on A.rider_id = O.rider_id -- left join to preserve riders without any deliveries
and O.time_delivered is null -- remove finished orders
group by A.rider_id, R.id)
select RS.rider_id
from RiderStatus RS
where not exists (
   select 1 from RiderStatus RS2
   where RS2.rider_id <> RS.rider_id
   and
   (
   RS2.num_orders < RS.num_orders or -- less number of deliveries
   (RS2.num_orders = RS.num_orders and RS2.distance < RS.distance) -- same number of deliveries but less
distance
   )
);
```

Project responsibilities

Name	Responsibilities
Liu Jiajun	• Overall architecture design • ER model diagram • Schema and triggers related to restaurant, manager, customer, and orders • Backend and frontend of restaurant, manager, customer, and order related features • Promotion feature • Front end beautification
Liu Chaojie	• ER model diagram • Schema and triggers related to riders and delivery • Frontend and backend for delivery, rider schedules, salaries and rider statistics • Rider scheduling
Hans Kurnia	• ER model diagram • Frontend and backend of customer reviews and password update features
Lee Yueyu	• ER model diagram + updating and beautification of it • Backend of customer reviews and ratings

Conclusion

In this report, we presented the design and implementation of a food delivery application, with a focus on the database design and refinement.

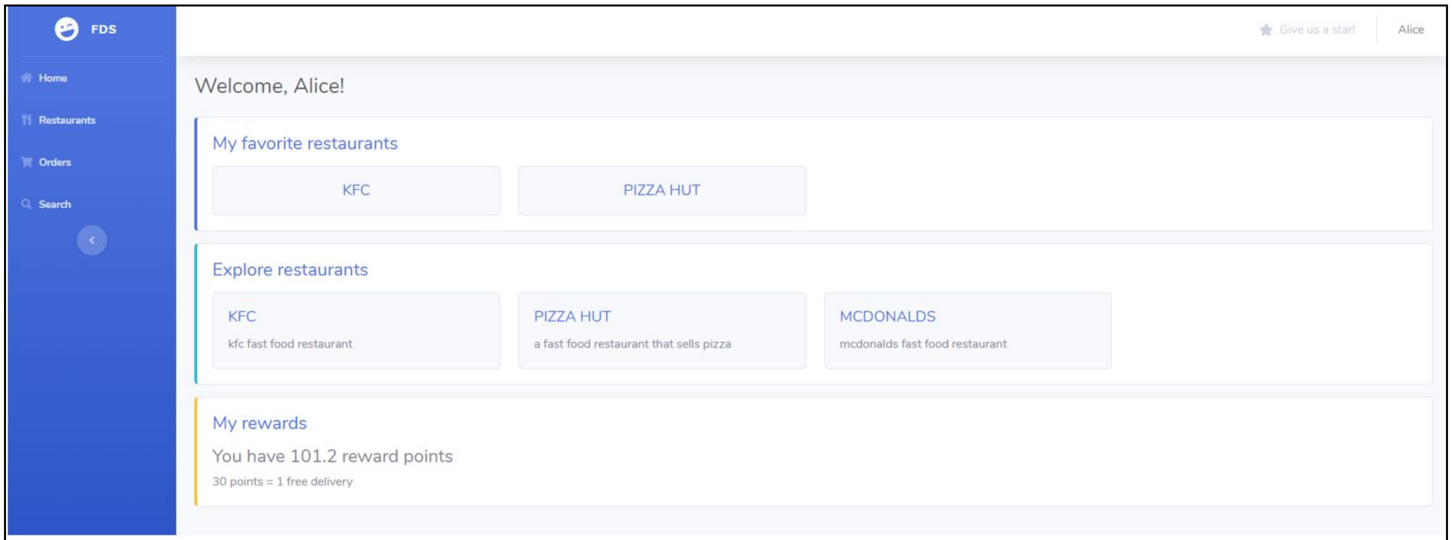
Some difficulties we encountered during the project include: (a) limited experience with front end development - we had to pick up front end development skills in a relatively short time (b) limited experience designing a complex database system - we learned to build things by try-and-error (c) schema refinement - we refactored the schema multiple times to minimize data redundancy.

Overall, it has been a challenging yet rewarding experience developing this product.

Appendix A: Software tools and frameworks

- Database: PostgreSQL
- Backend: Express.js
- Frontend: EJS, Bootstrap 4
- ER diagram: Draw.io

Appendix B: Application screenshots



Welcome, Alice!

My favorite restaurants

KFC

PIZZA HUT

Explore restaurants

KFC
kfc fast food restaurant

PIZZA HUT
a fast food restaurant that sells pizza

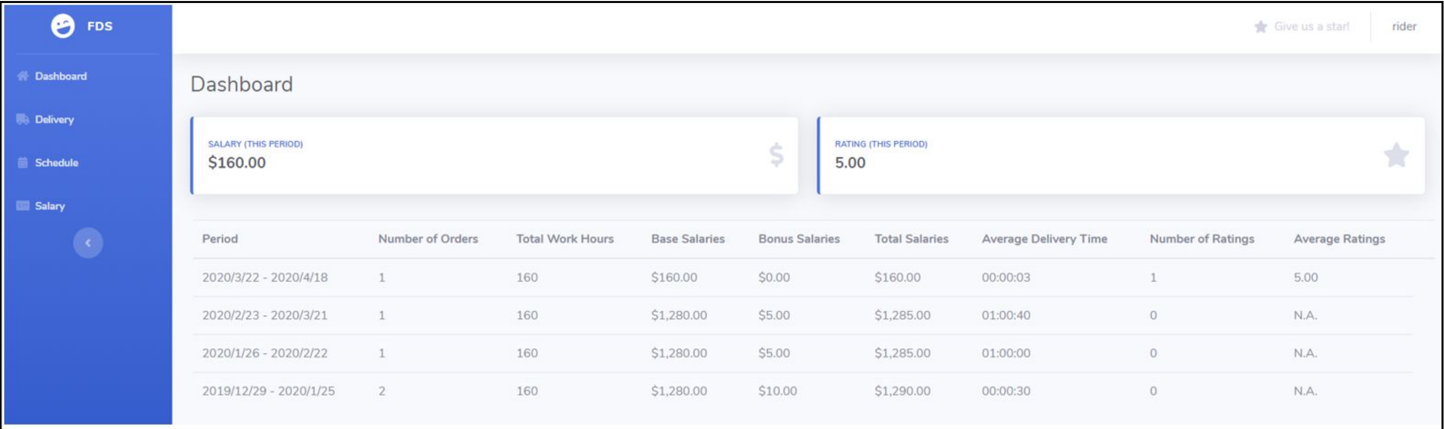
MCDONALDS
mcdonalds fast food restaurant

My rewards

You have 101.2 reward points

30 points = 1 free delivery

Screenshot 1. Customer home page



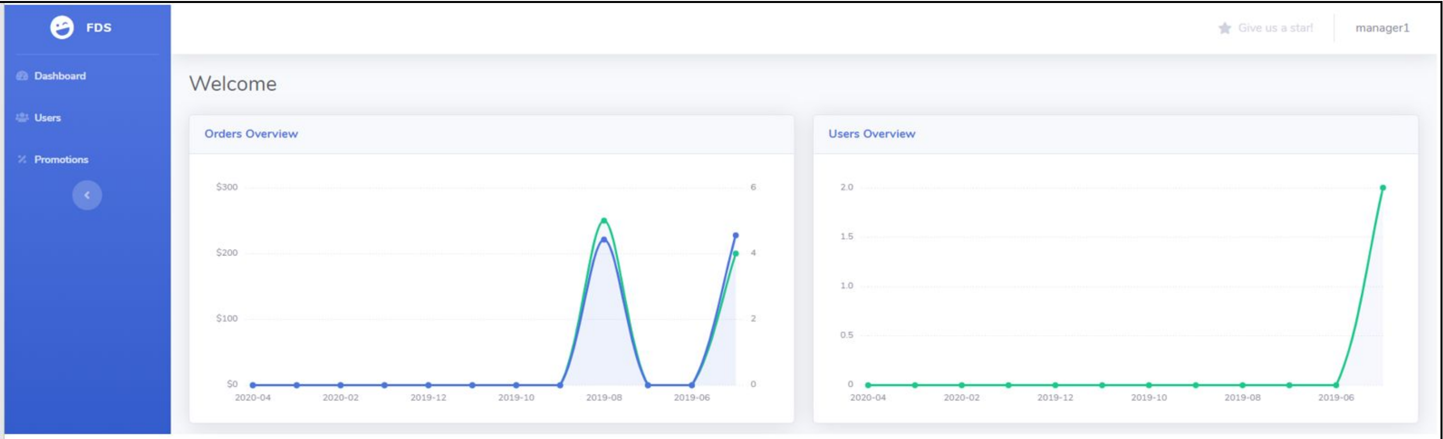
Dashboard

SALARY (THIS PERIOD)
$160.00

RATING (THIS PERIOD)
5.00

Period	Number of Orders	Total Work Hours	Base Salaries	Bonus Salaries	Total Salaries	Average Delivery Time	Number of Ratings	Average Ratings
2020/3/22 - 2020/4/18	1	160	$160.00	$0.00	$160.00	00:00:03	1	5.00
2020/2/23 - 2020/3/21	1	160	$1,280.00	$5.00	$1,285.00	01:00:40	0	N.A.
2020/1/26 - 2020/2/22	1	160	$1,280.00	$5.00	$1,285.00	01:00:00	0	N.A.
2019/12/29 - 2020/1/25	2	160	$1,280.00	$10.00	$1,290.00	00:00:30	0	N.A.

Screenshot 2. Rider dashboard



Screenshot 3. Manager dashboard