BOSTON UNIVERSITY

CS 669 Term Project

"Monitor-Daily Food Record" Application

Database

Authored By: Lingyan Jiang
Student ID: U10514676
Email: lingyanj@bu.edu

Table of Content

Term Project Overview

For the CS669 project, I want to build a database for the "Monitor-Daily Food Record" application I designed. Simply put, the "Monitor" app records the user's daily diet and gives reasonable fitness exercises and diet suggestions. Many people don't know whether their daily diets are healthy or not, whether their calorie intake exceeds the recommended limit, or what exercises they can do to expend the calories they have consumed. I am also concerned that many people don't have access to a nutritionist or a doctor, so the diet problem continues to get worse. That's the reason I'm going to build this app and its database. This app will solve this problem by analyzing the user's daily calories, vitamins, sugar, and salt content, and water through the information that users input about their diet and weight, as well as other food delivery app/website data, so as to recommend the proper diet and fitness methods.

In terms of data and vision, the following is a brief example of how someone would use the application. Amy logs into the "Monitor" app to record today's weight, breakfast, and lunch, and the app database saves Amy's breakfast and lunch food and drink and approximate meal time. At dinner time, Amy uses Uber Eats to order food delivery, and the "Monitor" app directly saves Amy's dinner food and drinks and meal time through the Uber Eats delivery software. The "Monitor" app will analyze users in real-time. Once Amy opens the "Monitor" app, she will be able to see the calories, vitamins, sugar, salt, water intake, and daily weight changes. "Monitor" will also recommend corresponding fitness exercises, changes to her diet, missing vitamins, etc.

The user data directly obtained from users' inputs or users' software memory and the data analyzed by the algorithm will be stored in the database and will be used to recommend diet and fitness for each user. In this project, I improved the design and structure of the entire "Monitor" application database through 5 iterations. In these 5 iterations, I have continuously learned about database creation, management, design, and other issues, and my design has become more complex. I hope that one day this "Monitor" application can really be launched on Google Play and use this database design.

Iteration 1

Iteration 1 overview

For the CS669 project, I want to build a database for the "Monitor-Daily Food Record" application I designed. Simply put, the "Monitor" app records the user's daily diet and gives reasonable fitness exercises and diet suggestions. In iteration 1, I will provide an overview of the application design and database design. Then, I will provide use cases that enumerate steps of how the "Monitor" application database will be typically used, and also identify significant database fields needed to support the use case. Finally, I will concisely summarize my project and the work I have completed thus far, and additionally record my questions, concerns, and observations, so that I can be aware of them and can communicate about them with my instructor and classmates.

Use Cases and Fields

I. First important usage of the database is when a user signs up for an account and installs the application.

Account Signup/Installation Use Case

1. The users visit the app store and install the "Monitor" application.
2. The "Monitor" application asks them to create an account when it's first run.
3. The application also asks users to grant privacy and storage authorization to make it more convenient to obtain users' information.
4. The users enter their information and the account is created in the database.
5. The application asks users to install browser plugins so that their food delivery purchases can be automatically tracked when they make them.

For the database of first use case, this use case requires storing information about users' accounts (in steps #2 and #4). Steps #1, #2 and #5 apply to the users and application but not the database directly. Significant fields for an account for this application are listed in the table below.

Account		
Field	What is Stored	Why it's Needed
Username	This is the unique summary name or abbreviation associated with each account.	This is necessary as a key in the database in case some people have the same name.
FirstName	This is the first name of the account holder.	This is necessary for displaying the person's name on screens and addressing them

		when sending them emails or other communications.
LastName	This is the last name of the account holder.	This is necessary for displaying the person's name on screens and addressing them when sending them emails or other communications.
Gender	This is the gender of the account holder.	This is useful so that the algorithm can calculate correct calorie consumption and recommend the correct fitness exercise.
Height	This is the height of the account holder.	This is useful so that the algorithm can calculate correct calorie consumption and recommend the correct fitness exercise.
InitialBodyWeight	This is the initial body weight of the account holder.	This is useful so that the algorithm can calculate the change of weight every day and recommend the correct fitness exercise.
MedicalHistory	This is the medical history of the account holder.	This is important so that the application can take note of users with medical conditions at the backend and can take more care of their recommended diets and fitness exercises.
Allergy	This is the allergy of the account holder.	This is important so that the application can take note of users with medical conditions at the backend and can take more care of their recommended diets and fitness exercises.
AimGoal	This is the target or aim that shows why account holders install the application.	This is useful so that the algorithm will adjust the recommendations to meet the users' goals like keep fit or lose weight.

II. Second important usage of the database is when users input their daily diet and record by "Monitor" application in its database.

User Input Use Case

1. The users input their daily meals/snacks/water they ate and drank or they take a picture of food or receipt in the database.
2. The users input the approximate time they took meals/snacks/water in the database.
3. (Optional) The users record and update their weight in the database.
4. (Optional) The users input today's exercise time and types in the database.

All steps are about the relevant daily diet information of the user from their own inputs. Significant fields are detailed below.

Daily_Meal		
Field	What is Stores	Why it's Needed
BreakfastTime	This is the time that the account holder takes breakfast approximately.	This is useful so that algorithms can calculate correct calorie consumption, and so that they can recommend correct diet plans and fitness exercise.
BreakfastEat	This is the food that account holders eat at breakfast.	This is useful so that algorithms can calculate correct calorie consumption, and so that they can recommend correct diet plans and fitness exercise.
BreakfastDrink	This is the drink that account holders have at breakfast.	This is useful so that algorithms can calculate correct calorie consumption, and so that they can recommend correct diet plans and fitness exercise.
LunchTime	This is the time that the account holder takes lunch approximately.	This is useful so that algorithms can calculate correct calorie consumption, and so that they can recommend correct diet plans and fitness exercise.
LunchEat	This is the food that account holders eat at lunch.	This is useful so that algorithms can calculate correct calorie consumption, and so that they can recommend correct diet plans and fitness exercise.
LunchDrink	This is the drink that account holders have at lunch.	This is useful so that algorithms can calculate correct calorie consumption, and so that they can recommend correct diet plans and fitness exercise.
DinnerTime	This is the time that the account holder takes dinner approximately.	This is useful so that algorithms can calculate correct calorie consumption, and so that they can recommend correct diet plans and fitness exercise.
DinnerEat	This is the food that account holders eat at dinner.	This is useful so that algorithms can calculate correct calorie consumption, and so that they can recommend correct diet plans and fitness exercise.
DinnerDrink	This is the drink that the account holder has at dinner.	This is useful so that algorithms can calculate correct calorie consumption, and so that they can recommend correct diet plans and fitness exercise.
Snack	This is the snack including food and drinks the account holder takes outside of the dinner time.	This is useful to divide the snack calories into daily consumption, so that algorithm can calculate correct calorie consumption, and so that they can recommend correct diet plans and fitness exercise.

SnackTime	This is the time that the account holder takes a snack approximately.	This is useful to know that the user eats the snack so that algorithm can calculate correct calorie consumption, and so that they can recommend correct diet plans and fitness exercise.
Water	This is the water account holder takes outside of the dinner time.	This is necessary to know how much water users took everyday to make sure users daily water needs.
DailyBodyWeight	This is the daily body weight of the account holder.	This is useful to track user's daily weight changes and help algorithms analyze the effectiveness of diet plans that are recommended by application, and so that the application can adjust the fitness plan and diet plan.
ExerciseTaken	This is the exercise that the account holder has taken.	This is useful so that the user's calorie value can be offset, and so that the calorie consumption will be correct.
ExerciseTakenTime	This is the length of time that the account holder exercised.	This is useful so that the user's calorie value can be offset, and so that the calorie consumption will be correct.

III. Third important usage of the database is when a food delivery purchase is made and automatically recorded in the "Monitor" application database via browser extensions.

Automatic Purchase Tracking Use Case

1. The user visits a food delivery application or website and makes a purchase.
2. The "Monitor" browser plugin detects that the purchase is made, and records the relevant information in the database such as the arriving time, food product, drinks, etc.

Step #2 highlights that the database will store relevant information about a food delivery purchase. Significant fields are detailed below.

Food_Delivery_Purchase		
Field	What is Stores	Why it's Needed
FoodDelivery	This is the food of online food delivery.	This is useful so that algorithms can calculate correct calorie consumption, and so that they can recommend correct diet plans and fitness exercise.
DrinkDelivery	This is the drink of online food delivery.	This is useful so that algorithms can calculate correct calorie consumption, and so that they can recommend correct diet plans and fitness exercise.

DeliveryArrivingTime	This is the arriving time of online food delivery.	This is useful to know the approximate time that users eat food, and so that algorithms can calculate correct calorie consumption, and they can recommend correct diet plans and fitness exercise.

IV. Fourth important usage of the database is analyzed by the algorithm in the "Monitor" application and the result of each user will be recorded in the database. This step is not involved with the user and it is only involved with the system. Fields are detailed below.

Balance		
Field	What is Stores	Why it's Needed
CaloriesBalance	This is the status of the user's calories balance after algorithm analyzing.	This is important because it shows directly if the user is below the daily calorie limit or not and how far the user is beyond the limit.
VitaminsBalance	This is the status of the user's vitamins balance after algorithm analyzing.	This is important because it shows directly if the user has met vitamins daily requirement or not and how far the user is from the requirement.
SugarIntakeBalance	This is the status of the user's sugar intake balance after algorithm analyzing.	This is important because it shows directly if the user is below the sugar daily limit or not and how much the user exceeded the limit.
SaltIntakeBalance	This is the status of the user's salt intake balance after algorithm analyzing.	This is important because it shows directly if the user is below the salt daily limit or not and how much the user exceeded the limit.
WaterIntakeBalance	This is the status of the user's water intake balance after algorithm analyzing.	This is important because it shows directly if the user needs to drink more water or not.
ExerciseBalance	This is the status of the user's exercise balance after algorithm analyzing.	This is important because it shows directly if the user needs to do more exercise or not and what kind of exercises are recommended.
WeightBalance	This is the status of the user's weight balance after algorithm analyzing.	This is important because it shows directly if the user reaches the target weight or not and how many kilos and exercises are needed to reach the goal.

V. Another important usage is when a person decides to look up their past weight and diet history changes and look for the exercise suggestions.

History Lookup and Suggestion Use Case

1. The user signs into the “Monitor” application.
2. The person selects the option to look up past diet history.
3. The “Monitor” application shows the diagram of recent calorie, vitamin, sugar intake, salt intake, water intake, weight changes from the database, and also gives the user the option to search.
4. The user searches based upon a date range, which causes a database search.
5. The application also displays all exercise suggestions from the database.

The database would make use of the fields for the second use case to the fifth use case. As the user will update the data day by day, all data will be recorded and stored by date.

Summary and Reflection

The database I built is used in the "Monitor-Daily Food Record" application, which is a diet and fitness monitoring application, and gives reasonable fitness recommendations through algorithm analysis. Due to the daily user data update, all data will be saved along with the corresponding date. However, I still have some concerns about the algorithms’ accuracy and the size of the database. The database of diet monitoring may require a more complete classification. The distinction between water and food alone may not be accurate enough. Also, because the database needs to be updated every day, there are certain concerns about the size and extraction of the database. It is my hope that I will be able to solve this problem in the course of this semester.

Iteration 2

Iteration 2 overview

In the iteration 2, I will define structural database rules which formally specify the entities, relationships, and constraints for my "Monitor" application database design. For the structural database rule analysis of my project, I will start with use cases to analyze it. Next, I will also create an initial ERD, the universally accepted method of modeling and visualizing database designs, to visualize the entities and relationships defined by the structural database rules.

Structural Database Rules

I. The first important usage of the database is when a user signs up for an account and installs the application.

Account Signup/Installation Use Case

1. The users visit the app store and install the "Monitor" application.
2. The "Monitor" application asks them to create an account when it's first run.
3. The application also asks users to grant privacy and storage authorization to make it more convenient to obtain users' information.
4. The users enter their information and the account is created in the database.
5. The application asks users to install browser plugins so that their food delivery purchases can be automatically tracked when they make them.

In this use case, we can find that only step #4 starts to be associated with the database. Step #4 makes it clear that there is an entity - Account in this use case. Although one Account entity can also be broken down into multiple entities with relationships, I don't have enough information from this use case alone. So we keep in mind the Account entity is needed for the database, and move on to the next use case.

II. The second important usage of the database is when users input their daily diet and record by "Monitor" application in its database.

User Input Use Case

1. The users input their daily meals/snacks/water into the database or they can take a picture of food or receipt, so the algorithm can analyze what they ate or drank.
2. The users input the approximate time they took meals/snacks/water in the database.
3. (Optional) The users record and update their weight in the database.
4. (Optional) The users input today's exercise time and types in the database.

From this use case, I see three critical data points: daily meal, daily body weight and daily exercise. The three entities corresponding to these three data points can be determined. The “Monitor” application takes records of those daily data and each data point is associated with the Account entity. I now have enough information to create some structural database rules. I’ll number them so that they can later be referred to by number.

1. Each daily meal is associated with an account, each account may be associated with many daily meals.

I create this structural rule because I infer from the use case that each daily meal record is associated with an account. While each user of the “Monitor” application may have zero to many daily meal records. The reason why I use “may” in this structural business rule is because some users may forget to use the application someday and leave the daily meal records empty. It is also possible that an account exists in the database without any daily meal record being made if the user only uses the "Monitor "application to keep record of his/her body weight. Clearly, this relationship is plural.

2. Each daily body weight is associated with an account, each account may be associated with one daily body weight.

This rule indicates that each daily body weight is associated with an account. However, different from structural business rule #1, each account may be associated with zero or one body weight data. The reason why I use “may” in this structural rule is the same as above that some users may not use the daily body weight record function in the “Monitor” application. That is why the daily body weight data is zero. As to “one daily body weight”, the daily body weight data will be updated once the user submits his/her latest body weight, so each account may be associated with one daily body weight. Obviously, this relationship is optional.

3. Each exercise is associated with an account, each account may be associated with many exercises.

For this structural business rule, we can clearly see that each exercise is associated with an account from the use case. Same as the situation in the rule #1, each account may be associated with many exercises since the users of the “Monitor” application may do different types of exercises in one day. Through structural business rule, we can know that this relationship is also plural.

III. The third important usage of the database is when a food delivery purchase is made and automatically recorded in the “Monitor” application database via browser extensions.

Automatic Purchase Tracking Use Case

1. The user visits a food delivery application or website and makes a purchase.

2. The “Monitor” browser plugin detects that the purchase is made, and records the relevant information in the database such as the arriving time, food product, drinks, etc.

From this automatic purchase tracking use case, although this entity can also be broken down into multiple entities with relationships, I still would like to keep it in one entity – Food_Delivery_Purchase. The step #2 in the use case is the most important part related with the database. The entity also has a relationship with the Account entity shown as below.

4. Each food delivery purchase is associated with an account, each account may be associated with many food delivery purchases.

This rule indicates that each food delivery purchase is associated with an account. Just as I mentioned above, users may order zero or several food delivery purchases in one day. That’s the reason I use “may” and “many” in this structural rule. Also, from the rule, we can know that this relationship is plural.

IV. The fourth important usage of the database is analyzed by the algorithm in the “Monitor” application and the result of each user will be recorded in the database. This step is not involved with the user and it is only involved with the system. Although this usage doesn’t have a use case, the data result from the algorithm will be stored in the database. I would like to use one entity – Balance to keep the data. Balance entity is also associated with the Account entity and there is also a structural business rule between them.

5. Each balance is associated with an account, each account may be associated with one balance.

Of course, we can infer that each balance is associated with an account, because each user's balance is one-to-one corresponding to its accountID. Conversely, each account may be associated with one balance. Users may use the "Monitor" application in many situations. If users never use this application after downloading and registration, the algorithm cannot analyze the balance data due to the incomplete basic data. In addition, every time the data in the Daily_Meal, Daily_Body_Weight, Exercise and Food_delivery_Purchase entities are updated, the data is updated in the balance entity. So users can only associate with zero or one balance. Of course, this relationship is also optional.

V. The last important usage is when a person decides to look up their past weight and diet history changes and look for the exercise suggestions.

History Lookup and Suggestion Use Case

1. The user signs into the “Monitor” application.
2. The person selects the option to look up past diet history.
3. The “Monitor” application shows the diagram of recent calorie, vitamin, sugar intake, salt intake, water intake, weight changes from the database, and also gives the user the option to search.
4. The user searches based upon a date range, which causes a database search.

5. The application also displays all exercise suggestions from the database.

This use case is interesting in particular because, although it describes functionality of the application, it does not describe any additional data that must be stored in the database. This search feature would search over past diet history from the second use case to the fifth use case we defined. We do not need additional entities in order to support this search feature. As the user will update the data day by day, all data will be recorded and stored by date.

So from the five use cases I have thus far, I have these five structural database rules as following.

Structural Database Rules
1. Each daily meal is associated with an account, each account may be associated with many daily meals.
2. Each daily body weight is associated with an account, each account may be associated with one daily body weight.
3. Each exercise is associated with an account, each account may be associated with many exercises.
4. Each food delivery purchase is associated with an account, each account may be associated with many food delivery purchases.
5. Each balance is associated with an account, each account may be associated with one balance.

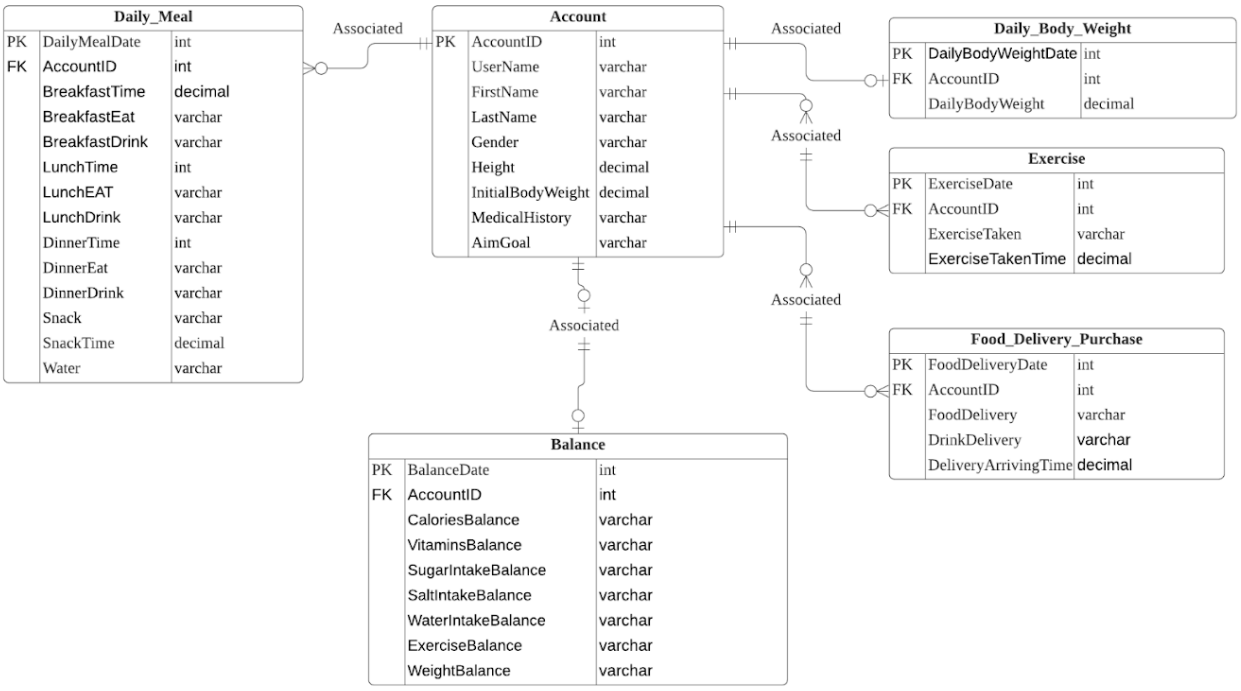
Conceptual Entity Relationship Diagram (ERD)

The following is the ERD I came up with for the five rules, using crow's foot diagram to make it clearer. As you can see in the ER diagram below, the Account entity is associated with all five entities. The crow's foot signs show clearly the relationship between each of the two entities. In this ERD, I use  to represent CFN zero or many relationship,  to represent CFN exactly one relationship and  to represent CFN zero or one relationship.

CS669 Project Iteration 2 _ Database ER diagram

Lingyan Jiang | October 5, 2020

CS669 Project iteration 2 Lingyan Jiang U10514676



Structural Business Rules:

1.Each daily meal is associated with an account, each account may be associated with many daily meals.
2.Each daily body weight is associated with an account, each account may be associated with one daily body weight.
3.Each exercise is associated with an account, each account may be associated with many exercises.
4.Each food delivery purchase is associated with an account, each account may be associated with many food delivery purchases.
5.Each balance is associated with an account, each account may be associated with one balance.

Summary and Reflection

Usually, when users start to use the "Monitor" application, they need to enter their account information, daily diet, daily weight and daily fitness. The “Monitor” application can automatically obtain the user's food delivery purchase information through the user's authorization, so that all the data is stored in the database and analyzed to give the user reasonable fitness/diet recommendations. The search function of the "Monitor" application can also retrieve the daily data information from the database.

The structural database rules and ERD for my database design contains Account, Daily_Meal, Daily_Body_Weight, Exercise and Food_Delivery_Purchase, five featured entities and the relationship between them. To my surprise, my four entities are all associated with one Account

entity. And for my five use cases, only five entities were obtained. This iteration showed me that because the use cases focus on the system, even large use cases of people who use the system and the database can result in a small number of entities being stored in the database. The number of use cases and entities are not necessarily proportional.

At the same time, there are ambiguities when I set up the database structure. For some structural database rule definitions, there is a part worthy of more in-depth thinking. For example, for the relationship between the Daily_Body_Weight entity and the Account entity, is it also possible to change it into a one-to-many relationship? Or how can I change the database structure to change the relationship between the two entities? I hope to investigate it more deeply in the next project iterations.

Iteration 3

Iteration 3 overview

Based on the last two iterations, I have created the application use cases, business rules, and ERD diagram. In the iteration 3, I will add the specialization generalization relationships, which allows one entity to specialize an abstract entity to my structural database rules and ERD diagram. Additionally, I will create an initial DBMS physical ERD, which will be tied to a specific relational database vendor and version, with SQL-based constraints and data types.

Specialization-Generalization Relationships

For the CS669 project, I want to build a database for the "Monitor-Daily Food Record" application I designed. Based on the last two iterations, I have created the application use cases, business rules, and ERD diagram below. For the specialization-generalization relationships of my project database, I have started to look again at the use cases and tried to improve it and add some more business rules to allow one entity to specialize in an abstract entity.

Structural Database Rules (From iteration 2)
1. Each daily meal is associated with an account, each account may be associated with many daily meals.
2. Each daily body weight is associated with an account, each account may be associated with one daily body weight.
3. Each exercise is associated with an account, each account may be associated with many exercises.
4. Each food delivery purchase is associated with an account, each account may be associated with many food delivery purchases.
5. Each balance is associated with an account, each account may be associated with one balance.

I. The first important usage of the database is when a user signs up for an account and installs the application.

Account Signup/Installation Use Case (From iteration 2)

1. The users visit the app store and install the “Monitor” application.
2. The “Monitor” application asks them to create an account when it’s first run.
3. The application also asks users to grant privacy and storage authorization to make it more convenient to obtain users’ information.

4. The users enter their information and the account is created in the database.
5. The application asks users to install browser plugins so that their food delivery purchases can be automatically tracked when they make them.

Something that would be useful for the "Monitor" application is to have different kinds of accounts. As application designers, we also want to obtain financial support from users to better improve the application's database, codes, and algorithms. I want to offer a free account that has some limitations on the functionality, and a paid account which offers all of the features. So I have modified the use case as follows.

Account Signup/Installation Use Case (new)

1. The users visit the app store and install the "Monitor" application.
2. The "Monitor" application asks them to create either a free or paid account when it's first run.
3. The application also asks users to grant privacy and storage authorization to make it more convenient to obtain users' information.
4. The users select the type of account and enter their information and the account is created in the database.
5. The application asks users to install browser plugins so that their food delivery purchases can be automatically tracked when they make them.

From the new step #2 of the use case, I derive the sixth structural database rule to support the change to the use case as follows.

6. A new account is a free account or a paid account.

From the use case above, we know that there are two choices for creating an account - free and paid, and that is the complete list. Based on the specialization-generalization rule, the relationship between free/paid accounts is complete and disjoint. Because a new account is identified as the supertype, free and paid choices are identified as the subtypes. Every new account must be one of those subtypes, and can only be one. I did not put any of the verbiages such as "several of these" or "none of these" since the rule is totally complete and disjoint.

II. Another use case that relates to specialization-generalization relationships is the third important usage of the database. In this use case, when a food delivery purchase is made and automatically recorded in the "Monitor" application database via browser extensions.

Automatic Purchase Tracking Use Case (From iteration 2)

1. The user visits a food delivery application or website and makes a purchase.
2. The "Monitor" browser plugin detects that the purchase is made, and records the relevant information in the database such as the arriving time, food product, drinks, etc.

Considering the specialization-generalization relationship in the third use case, I would like to treat different kinds of food delivery purchase approaches differently. There are several food delivery purchase methods. Users can purchase food delivery from food delivery applications, online websites, phone call ordering or none of the above. Food delivery purchase is identified as the supertype; food delivery application, online website, phone call ordering are described as multiple subtypes. Several food delivery purchase methods indicate the relationship is overlapping, and "none of the above" indicates that the relationship is partially complete. The modified new use case is as follows.

Automatic Purchase Tracking Use Case (new)

1. The user visits a food delivery application or website and makes a purchase.
2. The TrackMyBuys browser plugin detects that the purchase is made, and records the relevant information in the database such as whether the purchase is from a food delivery application or an online website or a phone call order, the purchase date, price, product, store, etc.

From the new step #2 of the third use case, I derive the seventh structural database rule to support the change to the use case as follows.

7. A food delivery purchase is from a food delivery application, online website, phone call ordering, both, or none of the above.

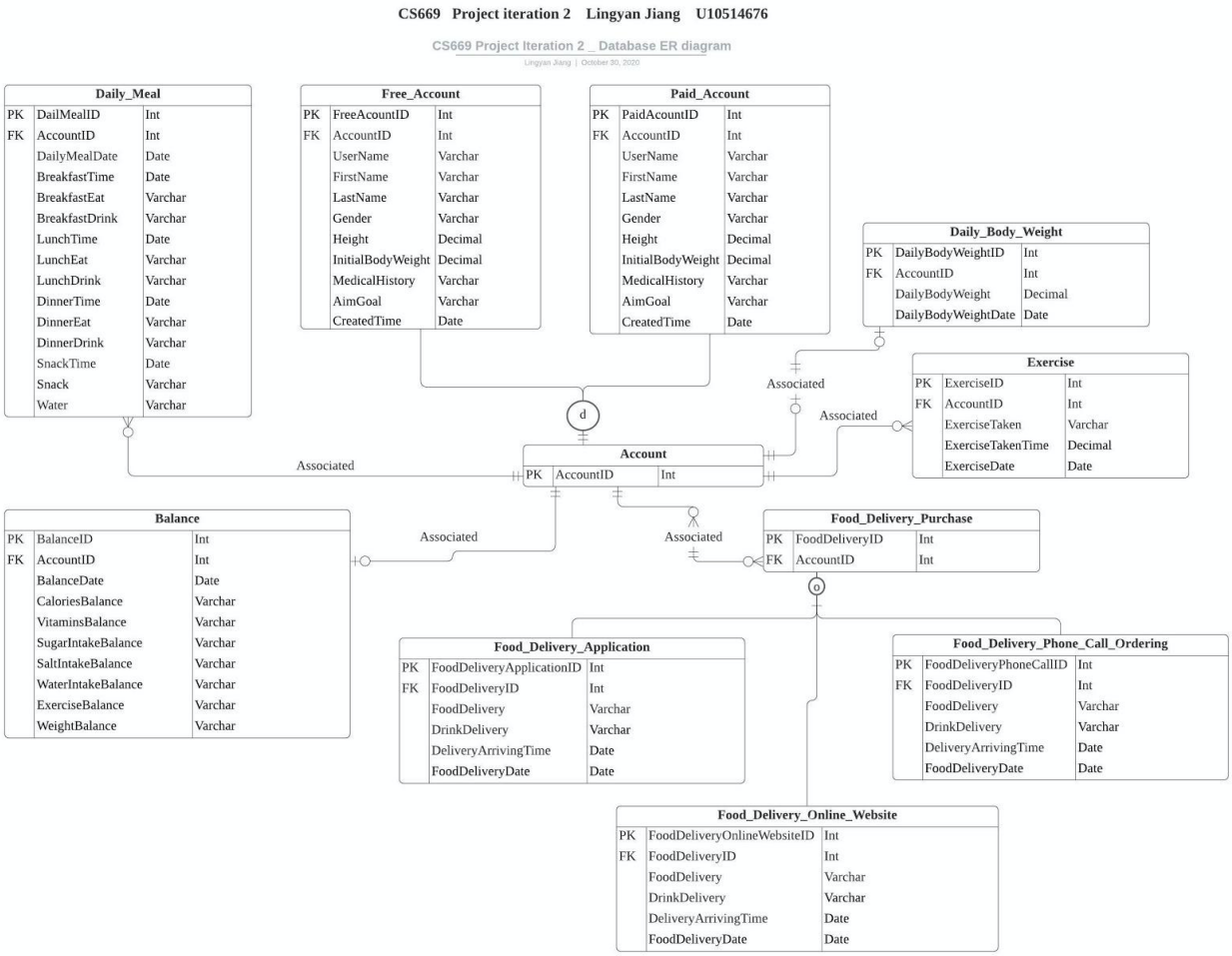
This relationship allows my database to have three different types, but I am not sure that these are the only types available. That's the reason why I add "or none of the above" at the last of the seventh business rule. Also, the food delivery method may change day by day, in order to meet the purchase method changes, I add "both " in the business rule just in case.

Now I have seven structural database rules, including my original five plus the two I just created. The new structural database rules are as follows.

Structural Database Rules (iteration 3)
1. Each daily meal is associated with an account, each account may be associated with many daily meals.
2. Each daily body weight is associated with an account, each account may be associated with one daily body weight.

3. Each exercise is associated with an account, each account may be associated with many exercises.
4. Each food delivery purchase is associated with an account, each account may be associated with many food delivery purchases.
5. Each balance is associated with an account, each account may be associated with one balance.
6. A new account is a free account or a paid account.
7. A food delivery purchase is from food delivery application, online website, phone call ordering, both, or none of the above.

The new ERD diagram with two additional structural database rules is below.



For the Crow's Foot diagram, "d" in the circle is used to indicate that the relationship is disjoint, and an "O" is used to indicate that the relationship is overlapping. Notice that extensions to Crow's Foot use a single bar to indicate partial completeness and two bars for total completeness. As to the two new entities under Account_Free_Account and Paid_Account, the relationship is totally complete and disjoint. As to another three new entities under Purchase _

Food_Delivery_Application, Food_Delivery_Online_Website, and Food_Delivery_Phone_Call_Ordering, the relationship is partially complete and overlapping. These additions capture the two new structural database rules and use specialization-generalization.

Relationship Classification and Associative Mapping

From the initial five business rules and ERM diagram from iteration 2, I created six entities. The associative relationships in my conceptual ERD are Daily_Meal/Account, Exercise/Account, Daily_Body_Weight/Account, Food_Delivery_Purchase/Account, and Balance/Account.

The Daily_Meal/Account relationship is 1:M. Each daily meal data can be associated with exactly one user account, but one user account is related to many daily meal data.

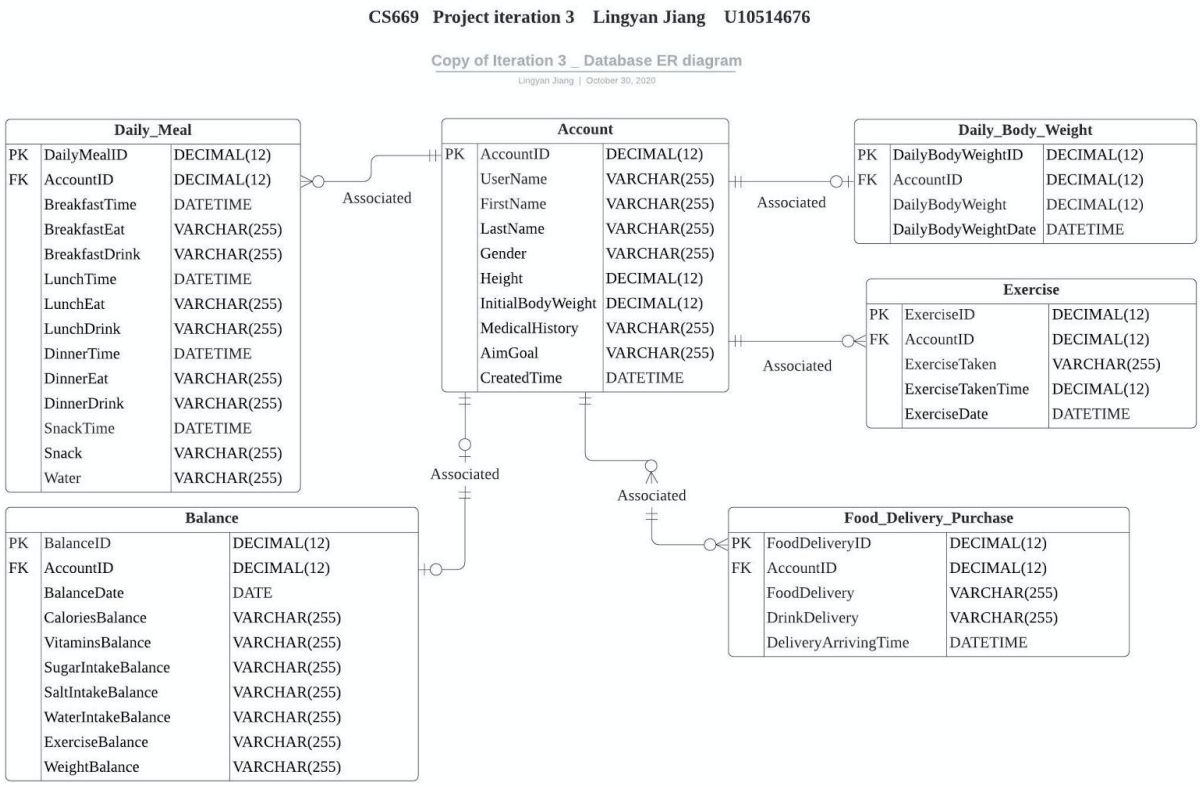
The Daily_Body_Weight/Account relationship is also 1:M. Many daily body weight data can be tied to exactly one account, but each account may be associated with one daily body weight data.

The Exercise/Account relationship is also 1:M. Many exercise data can be associated with exactly one account, each account may be associated with many exercises.

The Food_Delivery_Purchase/Account relationship is also 1:M. Many food delivery purchase data can be tied to exactly one account, each account may be associated with many food delivery purchases.

The Balance/Account relationship is also 1:M. Many balance data are associated with one account, each account is associated with exactly one balance.

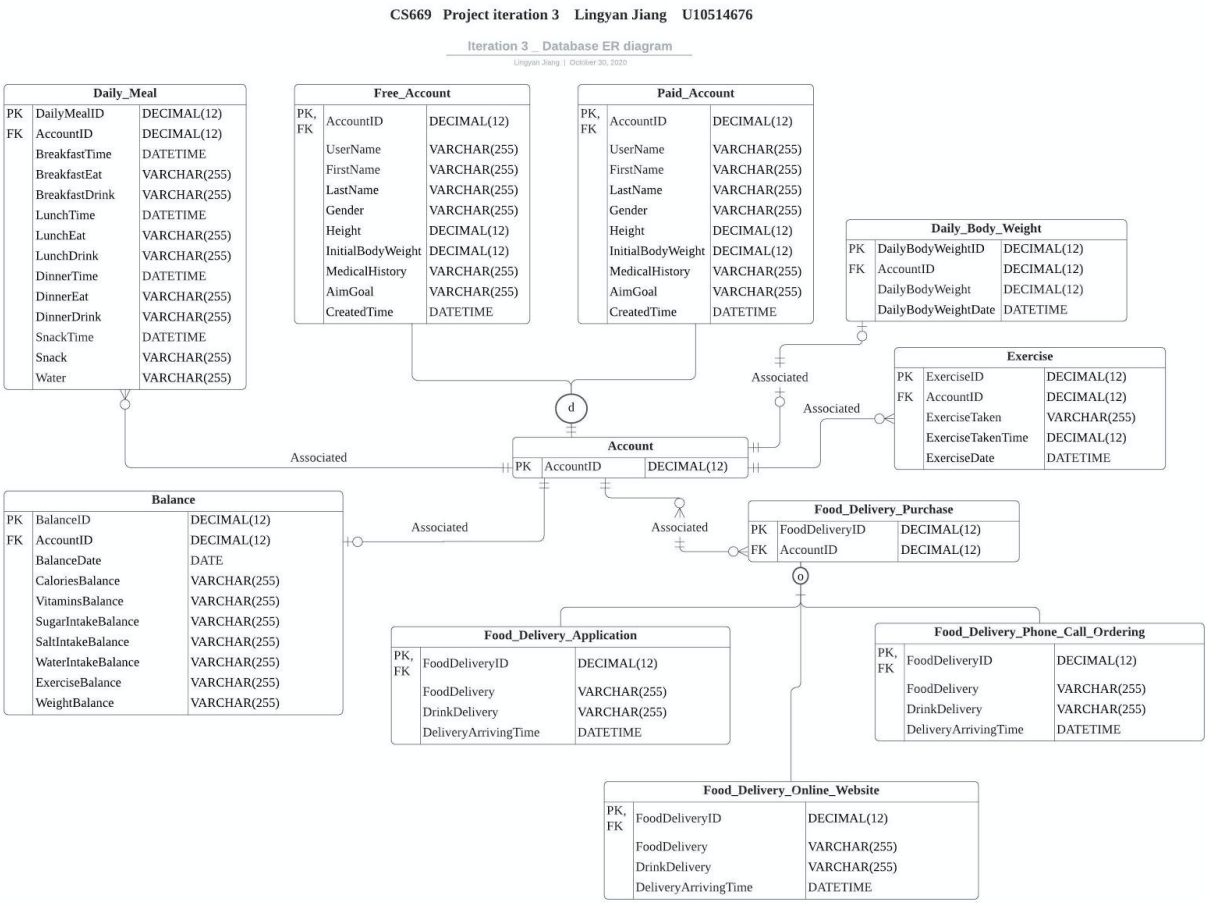
In my ERD diagram, I didn't create an M:N relationship. So there is no need to create a bridging or linking entity. All associative relationships in my conceptual ERD are Daily_Meal/Account, Exercise/Account, Daily_Body_Weight/Account, Food_Delivery_Purchase/Account, and Balance/Account, which are all 1:M relationships. Here is the DBMS physical ERD I created from the conceptual ERD for these relationships.



Taking into account the lecture I learned in the class and the real situation of the “Monitor” application, I opted to make the primary keys of the DECIMAL(12) datatype to allow for a lot of records in my database, such as AccountID, DailyMealID, DailyBodyWeightID, ExerciseID, BalanceID, and FoodDeliveryID. Since all six entities are related with a 1:M relationship, the DBMS physical ERD retains the related six entities, and a foreign key is placed in the entity that is related to at most one. For the other data in the entities, since the “Monitor” application is to record the user's daily diet and give reasonable fitness exercises and diet suggestions, the dataset needs to store dates, daily diet data, and fitness/diet suggestions data. For the dates, I used the DATETIME datatype instead of DATE or TIME, so that the dataset doesn’t need to store dates and time separately. As to the daily diet data, and fitness/diet suggestions data, users may import decimal and varchar together, so I used the VARCHAR(255) datatype and allowed more column space to store more data information.

Specialization-Generalization Mapping

There are two specialization-generalization relationships in my conceptual ERD, one for the Food_Delivery_Purchase entity and one for the Account entity. The following is my DBMS physical ERD with these relationships mapped onto them.



As we can see above, the two additional entities under Food_Delivery_Purchase are Food_Delivery_Application, Food_Delivery_Online_Website, and Food_Delivery_Phone_Call_Ordering, each of which has a primary and foreign key of FoodDeliveryID which reference the primary key of Food_Delivery_Purchase. The additional entities under Account are FreeAccount and PaidAccount, which have primary and foreign keys of AccountID which reference the primary key of Account. With these additional mappings, this DBMS physical now has all of the relationships in the conceptual ERD.

Summary and Reflection

From iteration 2, I created five initial business rules from the use cases. Based on the business rules and real use case situation for users, I created 6 entities and an initial ERD diagram.

In this project iteration, I started to check my use cases again to see if they contain a specialization-generalization relationship or not. In the first use case, the account entity can split into a free account or a paid account so that users have a choice whether to pay for extra functions of the “Monitor” application or only use the application for free. According to this specialization-generalization relationship, I got the sixth business rule - “A new account is a free

account or a paid account". In addition, considering the specialization-generalization relationship in the third use case, there are three entities Food_Delivery_Application, Food_Delivery_Online_Website, and Food_Delivery_Phone_Call_Ordering under the Food_Delivery_purchase entity. Since there are several food delivery purchase methods, users can purchase food delivery from food delivery applications, online websites, phone call ordering, or none of the above. Based on that, I got the seventh business rule - "A food delivery purchase is from a food delivery application, online website, phone call ordering, both, or none of the above".

For the second part of the iteration, I created a DBMS physical ERD, which is tied to relational database vendors and versions with SQL-based constraints and data types. The associative relationships in my conceptual ERD are Daily_Meal/Account, Exercise/Account, Daily_Body_Weight/Account, Food_Delivery_Purchase/Account, and Balance/Account, which are all 1:M relationships. I didn't use the bridging or linking entity because there is no M:N relationship in my database. The corresponding ERD diagrams are above.

I still have some confusion about the design of the database, especially for data other than primary keys or foreign keys. I need to think more about the datatype for the database of the "Monitor" application so that the database can be further simplified. I hope that in future iterations, I will start to look at the attributes and the exact data to be placed in the database to better understand the way forward.

Iteration 4

Iteration 4 overview

During iteration 4, I am now adding attributes and their data types table by table. My choices and reasoning are in the table below. Then, I will normalize the DBMS physical ERD to reduce or eliminate data redundancy in my database design. Based on the normalized ERD diagram, I will start to create the tables and constraints in Oracle. In addition, I will also identify columns with Index placement in order to speed up the performance of my database.

Full DBMS Physical ERD

Now I am adding attributes and their data types table by table. My choices and reasoning are in the table below.

Table	Attribute	Datatype	Reasoning
Account	UserName	VARCHAR(64)	Every account has a username associated with it, which will be used to login into the "Monitor" application. I allow usernames to be up to 64 characters.
	FirstName	VARCHAR(255)	This is the first name of the account holder, up to 255 characters of the first name.
	LastName	VARCHAR(255)	This is the last name of the account holder, up to 255 characters of the last name.
	Encrypted Password	VARCHAR(64)	Every account has a password. It will be stored in encrypted text format in the database. 64 characters should be a safe limit to store encrypted text.
	Email	VARCHAR(255)	This is the email address of the account holder. 255 characters should be a safe upper bound.
	Gender	VARCHAR(64)	This is the gender of the account holder. 64 characters should be safe for gender type.
	Height	DECIMAL(3,2)	This is the body height of the account holder. Since we will store centimeters, 3 digits and 2 decimal points will be safe for height type.
	Medical History	VARCHAR(1024)	This is the medical history of the account holder. Since some users may have a different medical history, 1024 characters will be safer.

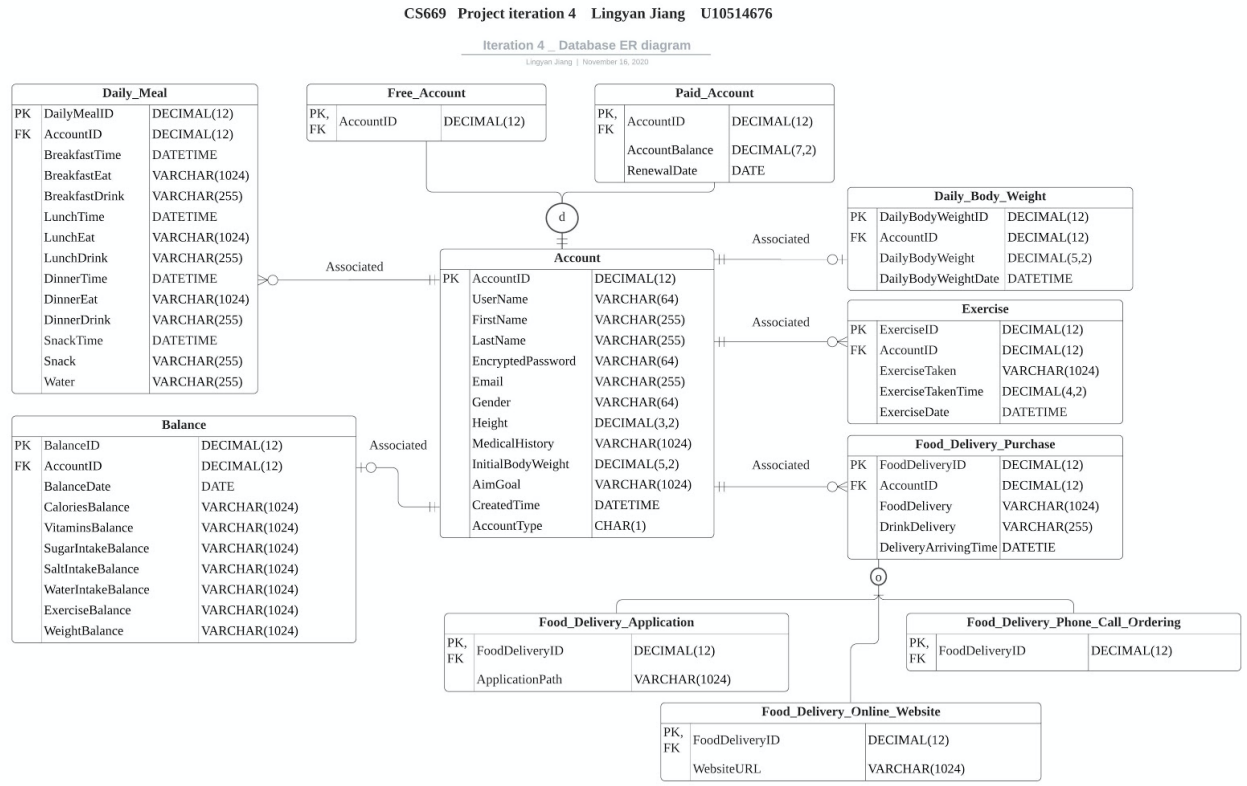
	InitialBodyWeight	DECIMAL(5,2)	This is the initial body weight of the account holder. Since we will store kilograms, 5 digits and 2 decimal points will be safe for body weight.
	AimGoal	VARCHAR(1024)	This is the aim/goal for using the "Monitor" application, such as keeping fit, losing weight, or keeping a healthier diet. 1024 characters will be safer.
	CreatedTime	DATETIME	This is the date-time when the account holder creates the account. DATETIME can save the date and time so that the application can keep a record of the user's initial body weight and be helpful for calculating the weight changes in the future.
	AccountType	CHAR(1)	As identified in a prior iteration, there are two types of accounts – free and paid. This attribute is the subtype discriminator indicating which it is.
Paid_Account	AccountBalance	DECIMAL(7,2)	This is the unpaid balance, if any, for the paid account. I allow for up to 7 digits and 2 decimal points, though it will likely never get this high.
	RenewalDate	DATE	This is the date on which the account needs to be renewed with a new payment.
Daily_Meal	BreakfastTime	DATETIME	This is the date-time that the account holder takes breakfast approximately. DATETIME can save the date and time so that the application can keep a record of everyday changes.
	BreakfastEat	VARCHAR(1024)	This is the food that the account holder eats at breakfast. Since the data is the record of breakfast, 1024 characters will be safer.
	BreakfastDrink	VARCHAR(255)	This is the drink that the account holder has at breakfast. 255 characters should be a safe upper bound.
	LunchTime	DATETIME	This is the date-time that the account holder takes lunch approximately. DATETIME can save the date and time so that the application can keep a record of everyday changes.
	LunchEat	VARCHAR(1024)	This is the food that the account holder eats at lunch. Since the data is the record of lunch, 1024 characters will be safer.
	LunchDrink	VARCHAR(255)	This is the drink that the account holder has at lunch. 255 characters should be a safe upper bound.
	DinnerTime	DATETIME	This is the date-time that the account holder takes dinner approximately. DATETIME can save the date and time so that the application can keep a record of everyday changes.

	DinnerEat	VARCHAR(1024)	This is the food that the account holder eats at dinner. Since the data is the record of dinner, 1024 characters will be safer.
	DinnerDrink	VARCHAR(255)	This is the drink that the account holder has at dinner. 255 characters should be a safe upper bound.
	Snack	VARCHAR(255)	This is the snack including food and drinks the account holder takes outside of dinner time. 255 characters should be a safe upper bound.
	SnackTime	DATETIME	This is the date-time that the account holder takes a snack approximately. DATETIME can save the date and time so that the application can keep a record of everyday changes.
	Water	VARCHAR(255)	This is the amount of water the account holder drinks outside of meal times. 255 characters should be a safe upper bound.
Daily_Body_Weight	DailyBodyWeight	DECIMAL(5,2)	This is the daily bodyweight of the account holder. Since we will store kilograms, 5 digits and 2 decimal points will be safe for weight.
	DailyBodyWeightDate	DATETIME	This is the date-time of the account holder's daily body weight. DATETIME can save the date and time so that the application can keep a record of the user's everyday body weight changes.
Exercise	ExerciseTaken	VARCHAR(1024)	This is the exercise that the account holder has taken. Users may perform different exercises every day, so I allow for 1024 characters so that people can type in something long if they need to.
	ExerciseTakenTime	DECIMAL(4,2)	This is the length of time that the account holder exercised. I allow for up to 4 digits and standard 2 decimal points.
	ExerciseDate	DATETIME	This is the date-time when the account holder does exercise. DATETIME can save the date and time so that the application can keep a record of the user's exercise.
Food_Delivery_Purchase	FoodDelivery	VARCHAR(1024)	This is the food of online food delivery. Since the data is the record of the food delivery, 1024 characters will be safer.
	DrinkDelivery	VARCHAR(255)	This is the drink of online food delivery. 255 characters should be a safe upper bound.
	DeliveryArrivingTime	DATETIME	This is the arrival time of online food delivery. DATETIME can save the date and time so that the application can keep a record of delivery arrival time.

Food_Delivery_Application	ApplicationPath	VARCHAR(1024)	This is the application path of the food delivery application record so that the food delivery record can directly be stored in the “Monitor” application database instead of being typed in by the account holder.
Food_Delivery_Online_Website	WebsiteURL	VARCHAR(1024)	This is the website URL path of the food delivery record so that the food delivery record can directly be stored in the “Monitor” application database instead of being typed in by the account holder.
Balance	BalanceDate	DATE	This is the date on which the account holder has different balance status and suggestions for every day.
	CaloriesBalance	VARCHAR(1024)	This is the status of the user's calorie balance after algorithm analysis. I allow for 1024 characters just in case there may be some long suggestions and balance for the user.
	VitaminsBalance	VARCHAR(1024)	This is the status of the user's vitamin balance after algorithm analysis. I allow for 1024 characters just in case there may be some long suggestions and balance for the user.
	SugarIntakeBalance	VARCHAR(1024)	This is the status of the user's sugar intake balance after algorithm analysis. I allow for 1024 characters just in case there may be some long suggestions and balance for the user.
	SaltIntakeBalance	VARCHAR(1024)	This is the status of the user's salt intake balance after algorithm analysis. I allow for 1024 characters just in case there may be some long suggestions and balance for the user.
	WaterIntakeBalance	VARCHAR(1024)	This is the status of the user's water intake balance after algorithm analysis. I allow for 1024 characters just in case there may be some long suggestions and balance for the user.
	ExerciseBalance	VARCHAR(1024)	This is the status of the user's exercise balance after algorithm analysis. I allow for 1024 characters just in case there may be some long suggestions and balance for the user.
	WeightBalance	VARCHAR(1024)	This is the status of the user's weight balance after algorithm analysis. I allow for 1024 characters just in case there may be some long suggestions and balance for the user.

I think that I have captured all of the necessary fundamental attributes for the “Monitor” application in the table above. I see that I could be more detailed with storing account

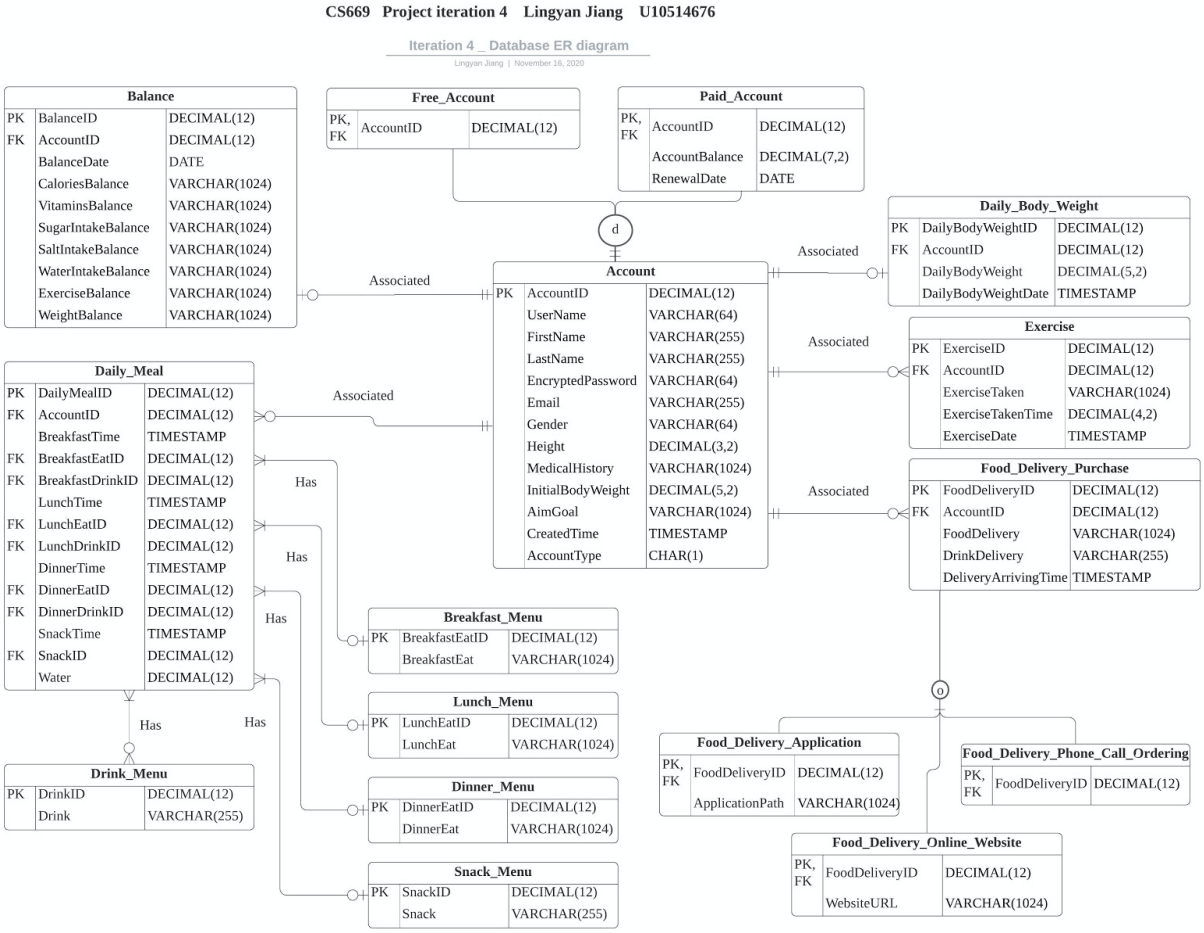
information balances and payment information. But given the use cases and structural database rules I've developed thus far, these attributes suffice. The following diagram is my ERD with the new attributes included.



In the above ERD diagram, each of the attributes has been added to the respective entities. The previously added primary and foreign keys have also been retained. There are only two primary keys in the Free_Account entity and the Food_Delivery_Phone_Call_Ordering entity. I may identify some more as the application is further developed. While there is room for more detail, I feel that this is a solid DBMS physical ERD for the use cases and structural database rules I have added thus far in the design.

Normalization

When looking through the whole ERD diagram, I found that there are some places that are redundant in my physical ERD. In the Daily_Meal entity, users sometimes may have the same breakfast/lunch/dinner/drink. Instead of typing in the same breakfast/lunch/dinner/snack/drink repeatedly, I normalize this entity into five additional entities as follows.



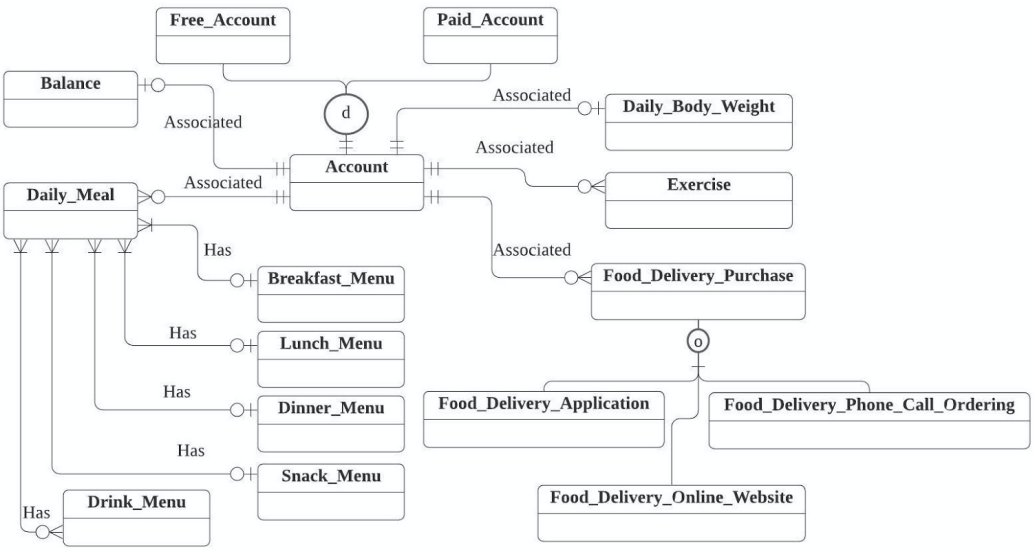
There are five additional entities after normalization – Breakfast_Menu, Lunch_Menu, Dinner_Menu, Snack_Menu, and Drink_Menu. I moved the BreakfastEat, LunchEat, DinnerEat, Snack, Drink into its own entity, and used BreakfastEatID, LunchEatID, DinnerEatID, SnackID, DrinkID as the primary keys in their entities. Thus, I do not need to repeat the breakfast/lunch/dinner/snack/drink data when users are having the same meals or drinks.

For the drink in the Daily_Meal entity, I used different foreign keys such as BreakfastDrinkID, LunchDrinkID, DinnerDrinkID, and associated with DrinkID in the Drink_Menu entity. Although in the Food_Delivery_Purchase, there are also meal records, it is unnecessary to normalize it, because the food delivery record is retrieved directly from the application/ website/typing submission. So there is no need to add complexity.

Below are my structural database rules modified to reflect the new entities. The new ones are italicized. I added #8, #9, #10, #11, #12 to reflect the new meal menu and drink menu.

Structural Database Rules (iteration 4)
1. Each daily meal is associated with an account, each account may be associated with many daily meals.
2. Each daily body weight is associated with an account, each account may be associated with one daily body weight.
3. Each exercise is associated with an account, each account may be associated with many exercises.
4. Each food delivery purchase is associated with an account, each account may be associated with many food delivery purchases.
5. Each balance is associated with an account, each account may be associated with one balance.
6. A new account is a free account or a paid account.
7. A food delivery purchase is from the food delivery application, online website, phone call ordering, both, or none of the above.
8. Each daily meal may have zero or one breakfast, each breakfast is associated with one or many daily meals.
9. Each daily meal may have zero or one lunch, each lunch is associated with one or many daily meals.
10. Each daily meal may have zero or one dinner, each dinner is associated with one or many daily meals.
11. Each daily meal may have zero or one snack, each snack is associated with one or many daily meals.
12. Each daily meal may have zero or many drinks, one or many drinks are associated with one daily meal.

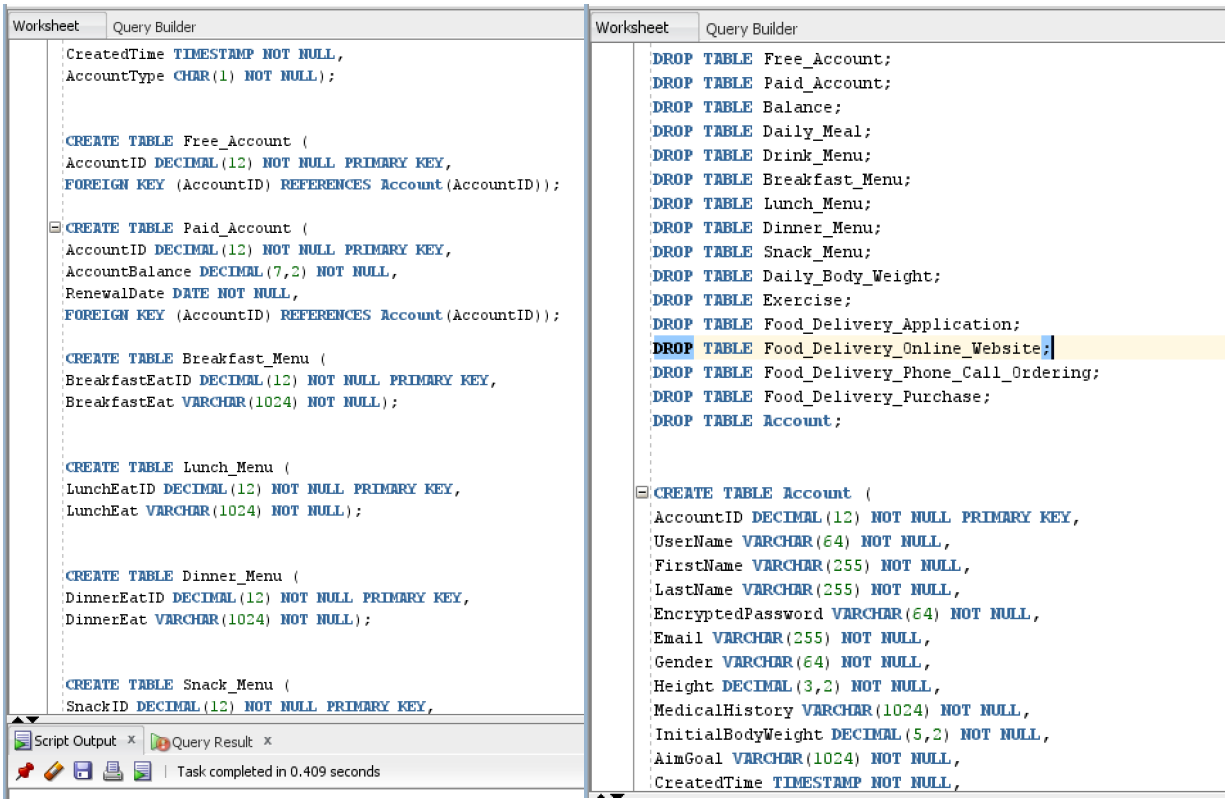
Below is my new conceptual ERD to reflect the new entities.

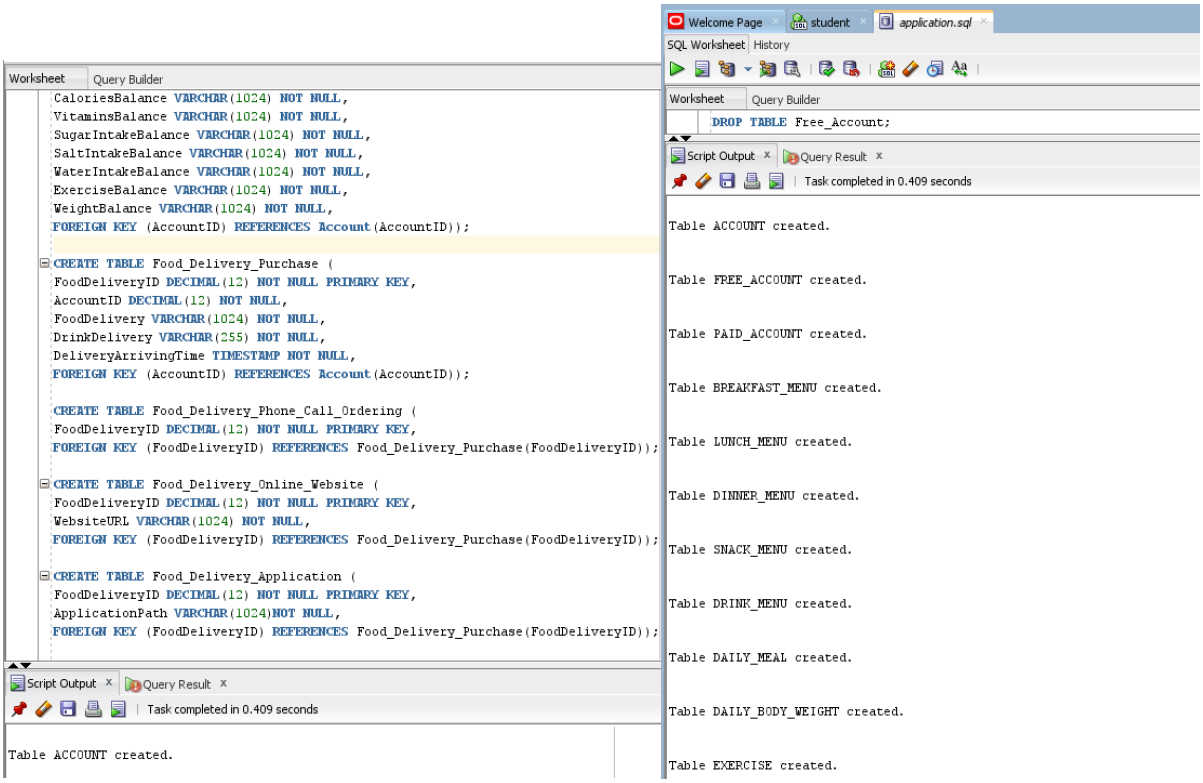


The Breakfast_Menu, Lunch_Menu, Dinner_Menu, Snack_Menu, and Drink_Menu entities are now included in the conceptual ERD, and the conceptual ERD is in sync with the structural database rules and the DBMS physical ERD.

Tables Create Script

The following screenshots are the scripts for creating the table in the Oracle for the “Monitor” application. According to the examples, I also used DROP TABLE commands at the top so that the script is re-runnable, then followed with the CREATE TABLE commands. All columns and constraints are included as illustrated in the ERD.





Index Placement

As far as primary keys that are already indexed, here is the list.

Account.AccountID

Free_Account.AccountID

Paid_Account.AccountID

Daily_Meal.DailyMealID

Breakfast_Menu.BreakfastEatID

Lunch_Menu.LunchEatID

Dinner_Menu.DinnertEatID

Snack_Menu.SnackID

Drink_Menu.DrinkID

Daily_Body_Weight.DailyBodyWeightID

Exercise.ExerciseID

Food_Delivery_Purchase.FoodDeliveryID

Food_Delivery_Application.FoodDeliveryID

Food_Delivery_Phone_Call_Ordering.FoodDeliveryID

Food_Delivery_Online_Website.FoodDeliveryID

Balance.BalanceID

As far as foreign keys, I know that all of them need an index. Below is a table identifying each foreign key column, whether or not the index should be unique or not, and why.

Column	Unique?	Description
Daily_Meal.AccountID	Not unique	The foreign key in Daily_Meal referencing Account is not unique because there can be many daily meals from the same account.
Daily_Meal.BreakfastEatID	Not unique	The foreign key in Daily_Meal referencing Breakfast_Menu is not unique because the same breakfast type can be had many times.
Daily_Meal.BreakfastDrinkID	Not unique	The foreign key in Daily_Meal referencing Drink_Menu is not unique because the same breakfast drink can be had many times.

Daily_Meal.LunchEatID	Not unique	The foreign key in Daily_Meal referencing Lunch_Menu is not unique because the same lunch type can be had many times.
Daily_Meal.LunchDrinkID	Not unique	The foreign key in Daily_Meal referencing Drink_Menu is not unique because the same lunch drink can be had many times.
Daily_Meal.DinnerEatID	Not unique	The foreign key in Daily_Meal referencing Dinner_Menu is not unique because the same dinner type can be had many times.
Daily_Meal.DinnerDrinkID	Not unique	The foreign key in Daily_Meal referencing Drink_Menu is not unique because the same dinner drink can be had many times.
Daily_Meal.SnackID	Not unique	The foreign key in Daily_Meal referencing Snack_Menu is not unique because the same snack can be ated many times.
Daily_Body_Weight.AccountID	Not unique	The foreign key in Daily_Body_Weight referencing Account is not unique because there can be many body weights from the same account.
Exercise.AccountID	Not unique	The foreign key in Exercise referencing Account is not unique because there can be many exercises from the same account.
Food_Delivery_Purchase.AccountID	Not unique	The foreign key in Food_Delivery_Purchase referencing Account is not unique because there can be many food deliveries from the same account.
Balance.AccountID	Not unique	The foreign key in Balance referencing Account is not unique because there can be many balance statuses for the same account.

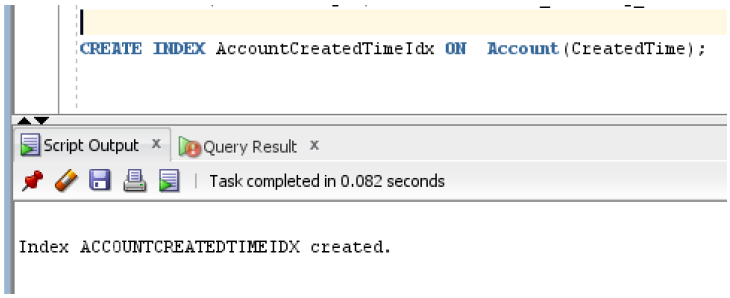
First, I select the Account.CreatedTime to be the first index. This would be a non-unique index because many accounts could have the same creation date and time.

It's also reasonable that there will be many queries that limit by account balances, to see what accounts are over or under a certain limit. So I select Paid_Account.AccountBalance to be

indexed. This would be a non-unique index because many accounts could have the same balance.

Index Creation

The following is the screenshot demonstrating the creation of a foreign key index for the "Monitor" application.



The above screenshot is the demonstration of creating a query-driven index, an index on the CreatedTime attribute in the Account entity. I named the index "AccountCreatedTimeIdx" to help identify what it's for and put it on the CreatedTime column in Account.

Summary and Reflection

In iteration 4, due to the daily user data update, all data will be saved through different records of date. Thus, the database must support a person recording, retrieving, and analyzing their daily food and body record among everyday changes.

The structural database rules and conceptual ERD for my database design contain the important entities of Account, Daily_Meal, Daily_Body_Weight, Exercise, Food_Delivery_Purchase, Daily_Meal, as well as relationships between them. The design contains a hierarchy of Food_Delivery_Purchase/Food_Delivery_Application, Food_Delivery_Purchase/Food_Delivery_Online_Website, and Food_Delivery_Purchase/Food_Delivery_Phone_Call_Ordering to reflect the three primary ways people purchase food delivery. The design also contains a hierarchy of Account/Paid_Account and Account/Free_Account to reflect the fact that people can sign up for a free account or a paid account for the "Monitor" application. The DBMS physical ERD contains the same entities and relationships, uses the best practice of synthetic keys, and contains the important attributes needed by the database to support the application.

The SQL script that creates all tables follows the specification from the DBMS physical ERD exactly. Important indexes have been created to help speed up access to my database and are also available in an index script.

During the four iterations, I put one initial idea of a database for application into reality. In each step of the iterations, I continue to improve my ERD diagram, business rules, and attributes. In continuous improvement, I slowly realized the true meaning of data structure design. For the next step, I will try to code and launch my “Monitor” application. Once launched, users can create accounts and input their data in my “Monitor” application database.

Iteration 5

Iteration 5 overview

This iteration is about putting data into your database and answering questions from the data. My database design structure was implemented in Iteration 4 through the creation of the tables, attributes, and constraints. In this iteration 5, I will create and execute reusable stored procedures that complete the steps of transactions necessary to add data to my "Monitor" application database. Next, I will create the history table to track changes to values and develop a trigger to maintain it. Finally, to see my database working, I will define questions useful to the organization or application that will use my "Monitor" application database, then write queries to address the questions.

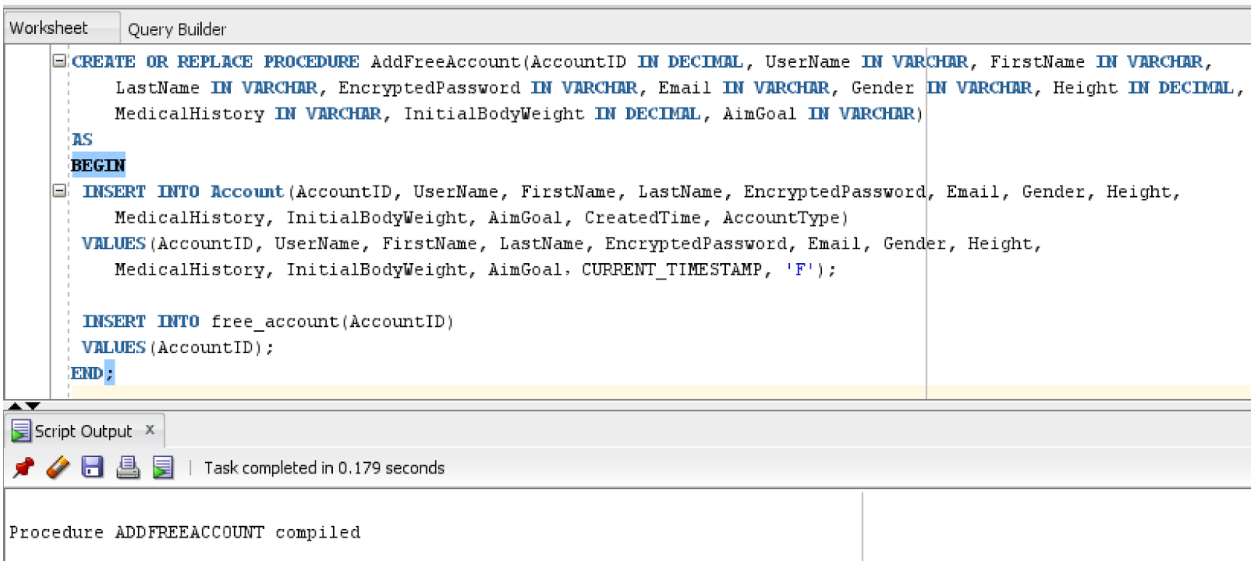
Reusable, Transaction-Oriented Store Procedures

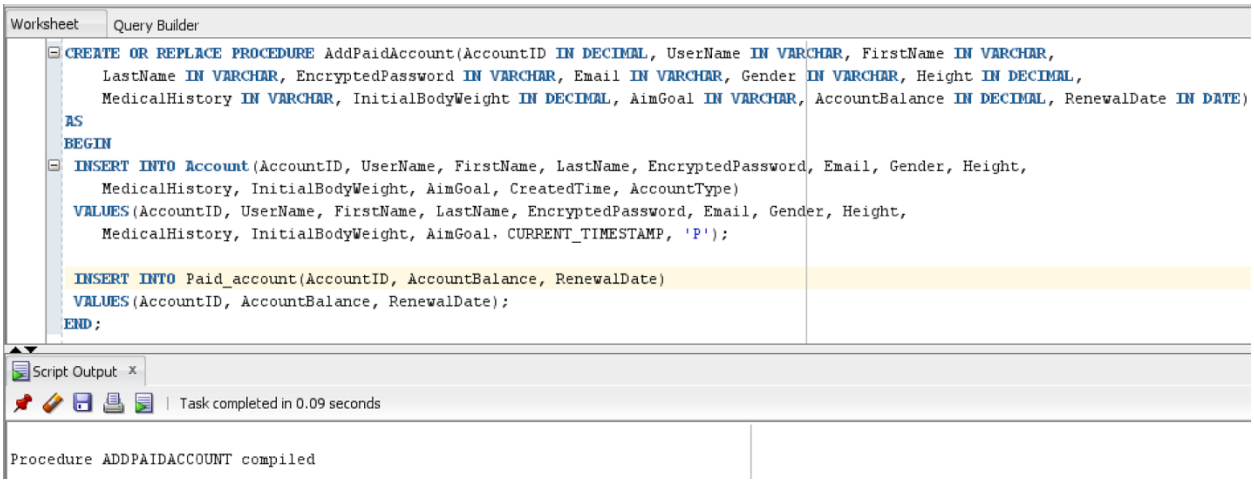
The first use case for the “Monitor” application is the account signup use case listed below.

Account Signup/Installation Use Case

1. The users visit the app store and install the “Monitor” application.
2. The “Monitor” application asks them to create either a free or paid account when it’s first run.
3. The application also asks users to grant privacy and storage authorization to make it more convenient to obtain users’ information.
4. The users select the type of account and enter their information and the account is created in the database.
5. The application asks users to install browser plugins so that their food delivery purchases can be automatically tracked when they make them.

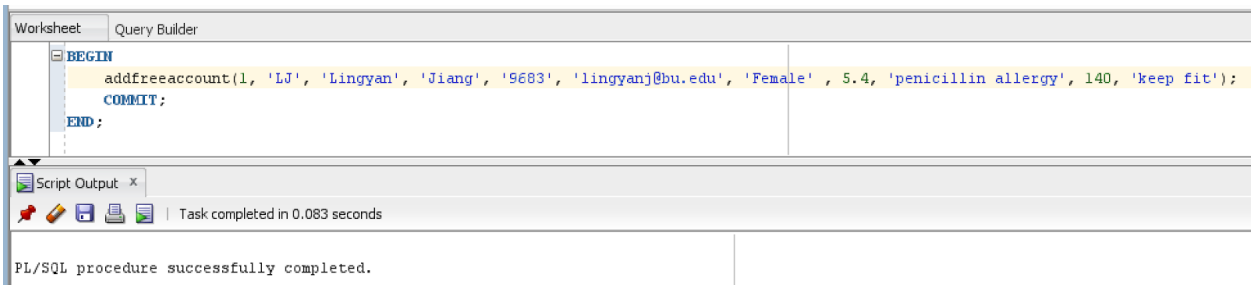
For this use case, I will implement transactions that create a free account and a paid account using Oracle. Here is a screenshot of my stored procedure definition.

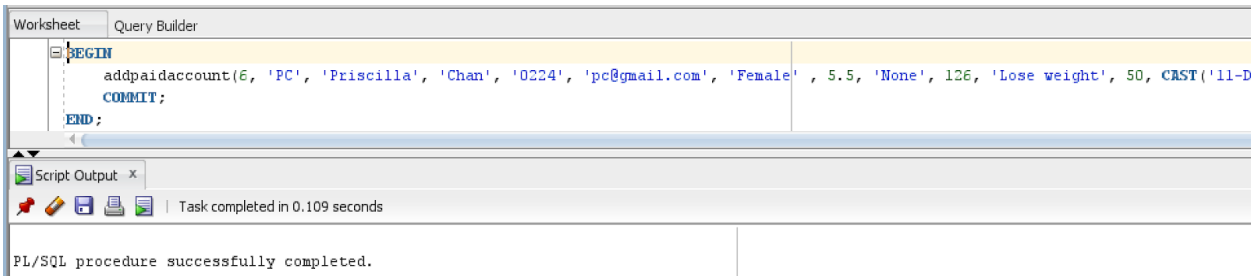




I name the stored procedure “AddFreeAccount” and give it parameters that correspond to the Account and FreeAccount tables. Since CreatedTime is always the current date, I do not need a parameter for that but instead using the CURRENT_TIMESTAMP function in Oracle. Since this procedure is always for a free account, I do not use a parameter for AccountType, but hardcode the character “F”. I also name the store procedure “AddPaidAccount” and give it parameters that correspond to the Account and PaidAccount tables. I also use the CURRENT_TIMESTAMP function to get the current date and time of storing the data. For the AccountType, I use the character “P” to identify the account.

Then, I start to insert data to insert my database using tables stored procedure execution. I also use some fictional persons with the corresponding information to make the dataset more realistic. Here is a screenshot of my stored procedure execution.





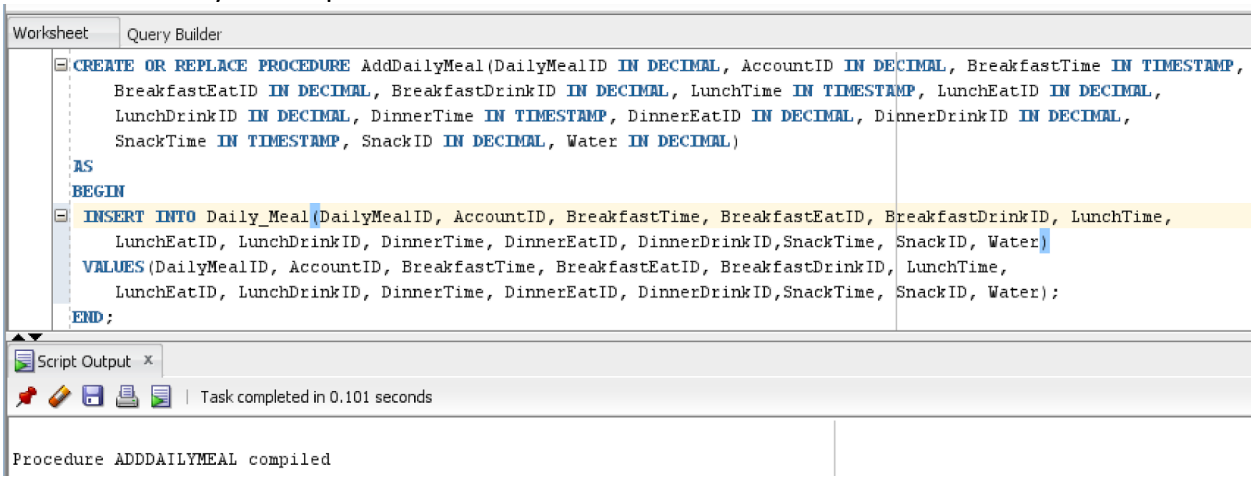
I add my information to become the first user. I nested the stored procedure call between transaction control statements to ensure the transaction is committed.

The second important usage of the database is when users input their daily diet and record by "Monitor" application in its database.

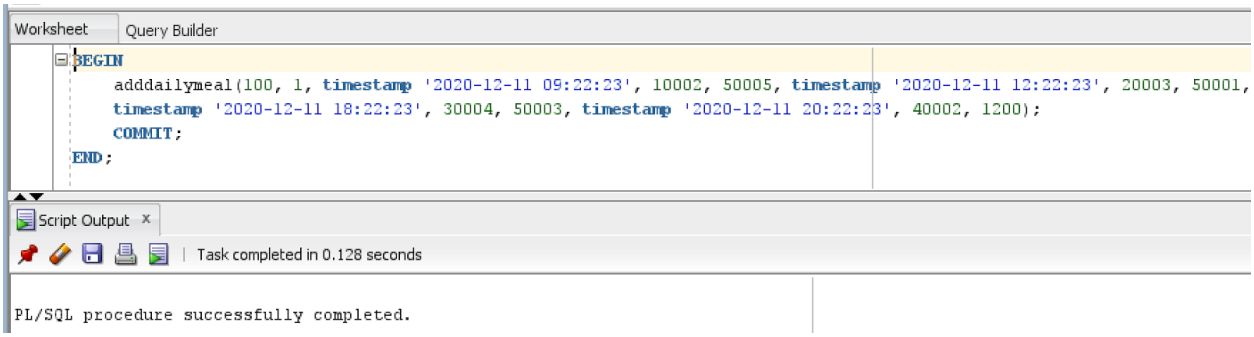
User Input Use Case

1. The users input their daily meals/snacks/water they ate and drank or they take a picture of food or receipt in the database.
2. The users input the approximate time they took meals/snacks/water in the database.
3. (Optional) The users record and update their weight in the database.
4. (Optional) The users input today's exercise time and types in the database.

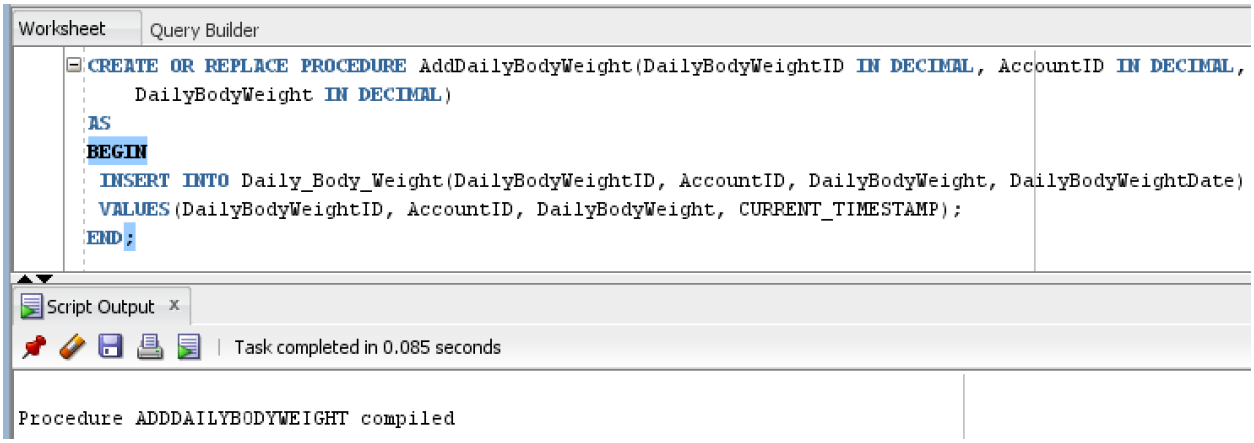
For this use case, I will implement a transaction that adds the user's daily meal. Here is a screenshot of my stored procedure definition.

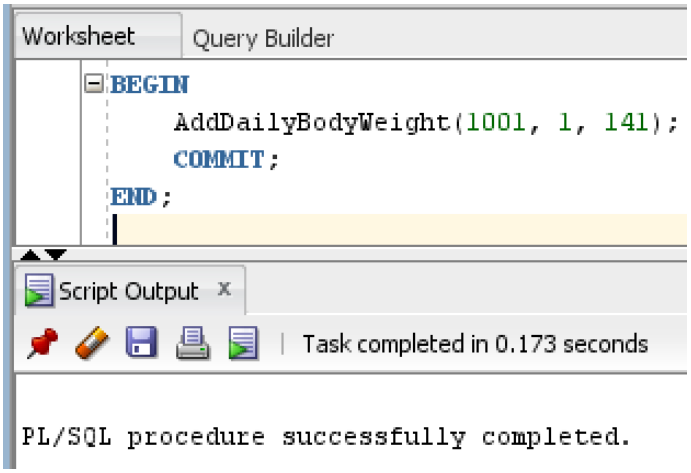


I name the stored procedure “AddDailyMeal” and give it parameters that correspond to the Daily Meal tables. Here is a screenshot of my stored procedure execution.

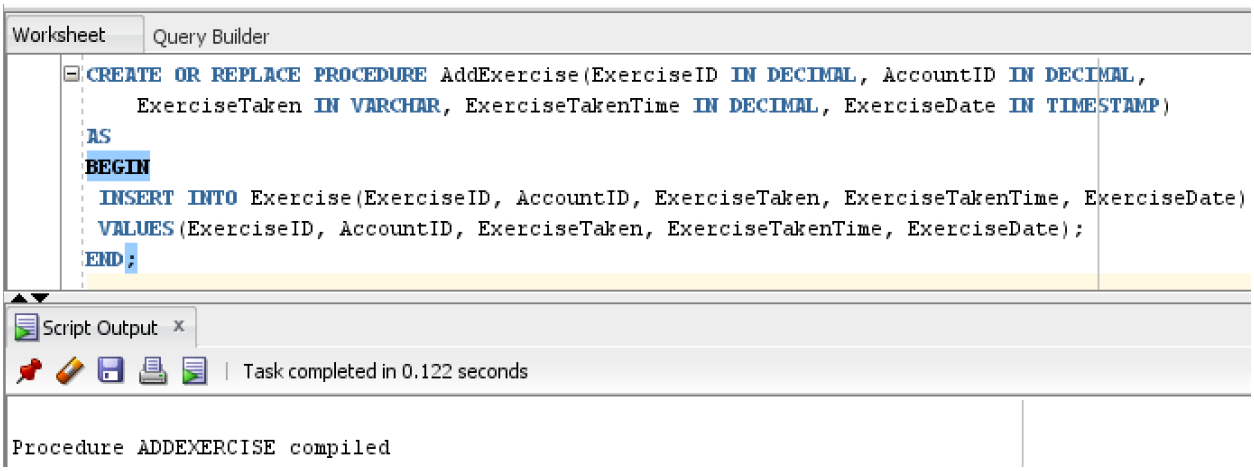


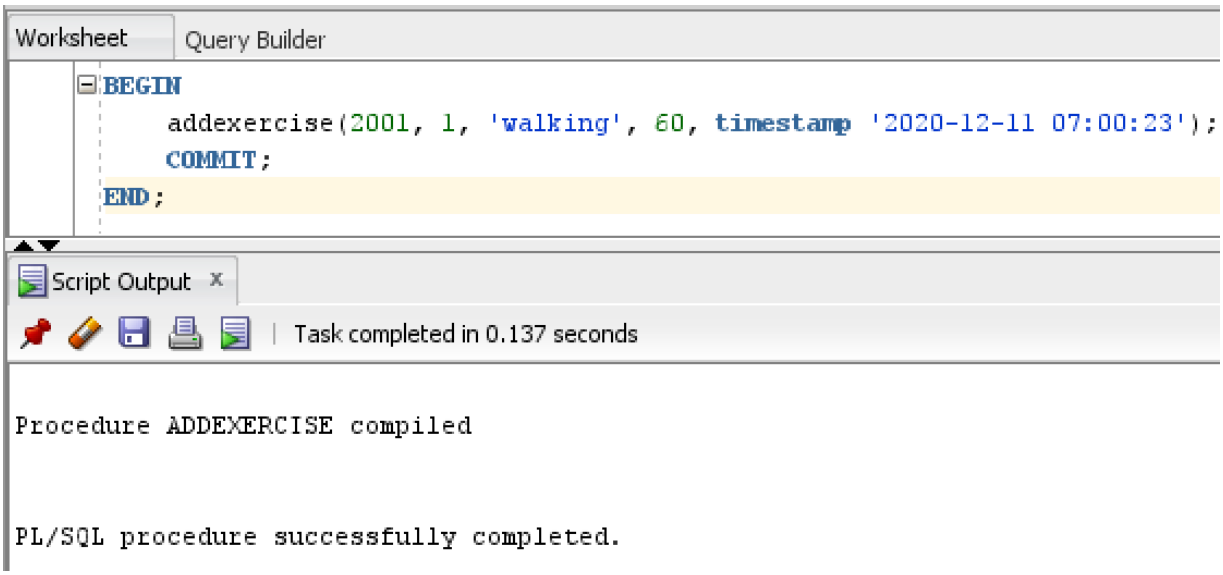
Also, in the second use case, I will implement a transaction that adds the user’s daily bodyweight and name the stored procedure “AddDailyBodyWeight”. Then, I start to insert data to insert my database using tables stored procedure execution. Here are the screenshots of my stored procedure execution.



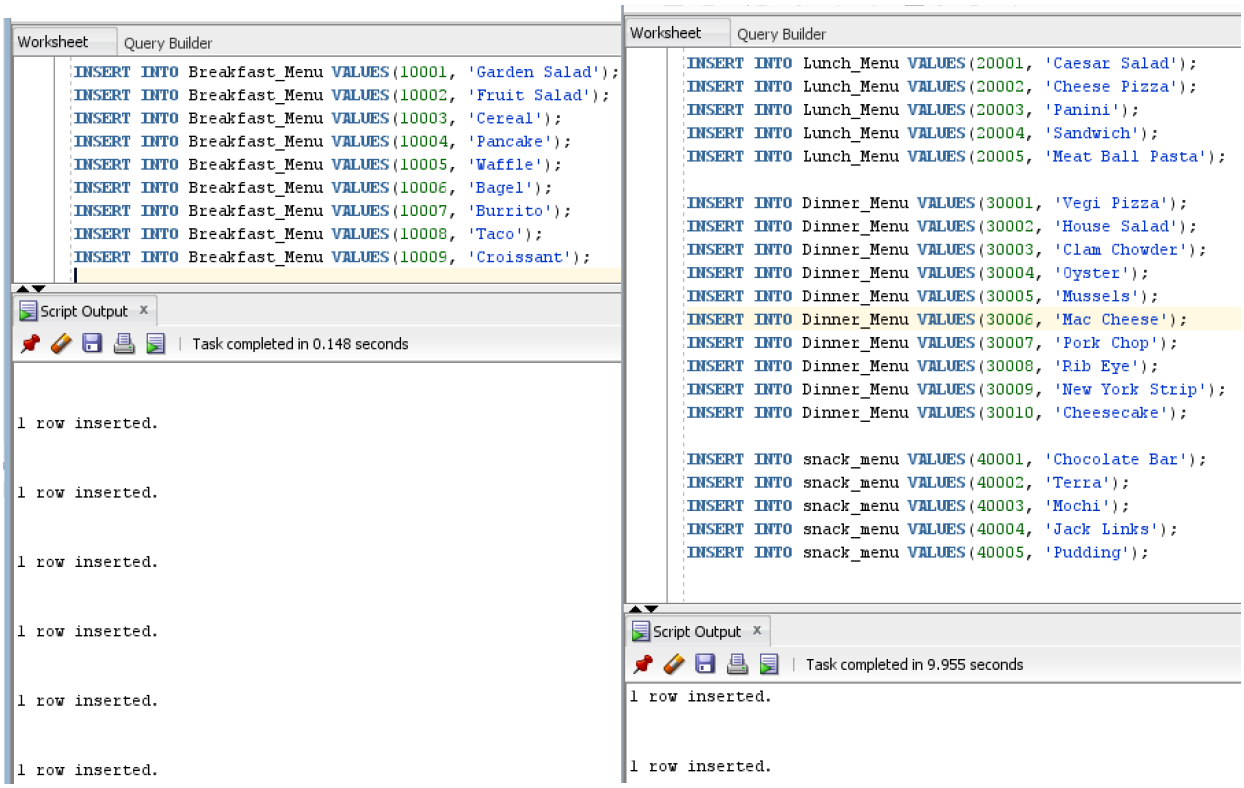


In addition, in the second use case, I also implement a transaction that adds the user's daily exercise and name the stored procedure "AddExercise". Then, I start to insert data to insert my database using tables stored procedure execution. Here are the screenshots of my stored procedure execution.





For the remaining data, I directly use the INSERT clause to fill the corresponding data into the database, as shown in the screenshot below. In this way, my database can better simulate the user's actual use of the scene, which is helpful for the subsequent queries by queries.



History Table

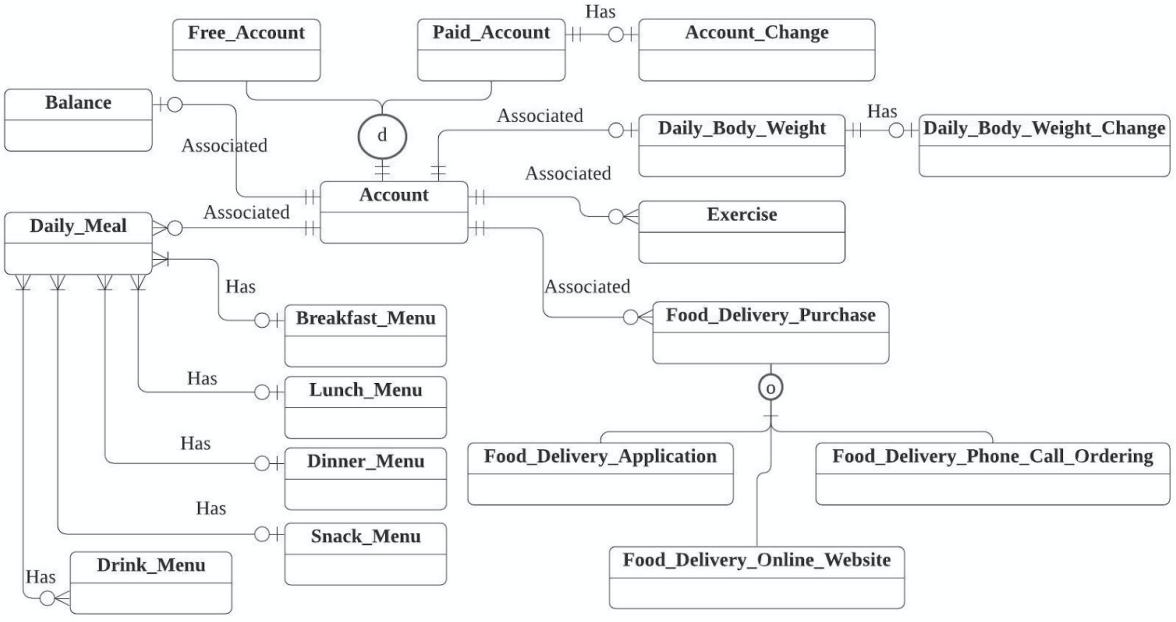
In reviewing my DBMS physical ERD, one piece of data that would benefit from a historical record of a person's balance in the PaidAccount table as well as a historical record of user's body weight in the Daily body weight table. Such a history would help me calculate statistics about account balances that are accurate over time and keep a record of user's weight changes. Thus, I added two new structural database rules:

13. Each paid account may have many account balance changes; each account balance change is for one paid account.
14. Each daily body weight may have many body weight changes; each body weight change is for one daily body weight.

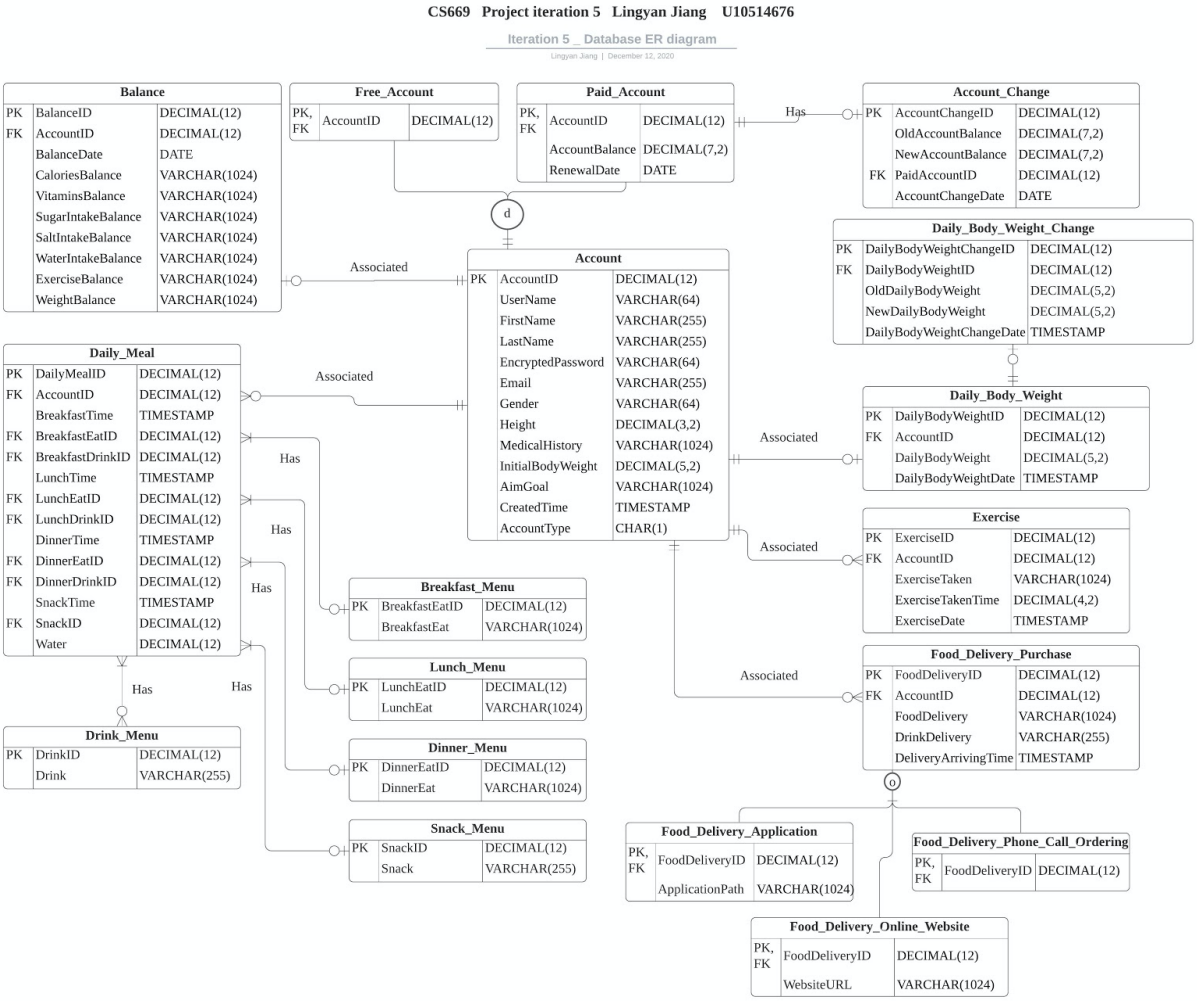
My updated structural database rule and conceptual ERD are below.

Structural Database Rules (iteration 5)
1. Each daily meal is associated with an account, each account may be associated with many daily meals.
2. Each daily body weight is associated with an account, each account may be associated with one daily body weight.

3. Each exercise is associated with an account, each account may be associated with many exercises.
4. Each food delivery purchase is associated with an account, each account may be associated with many food delivery purchases.
5. Each balance is associated with an account, each account may be associated with one balance.
6. A new account is a free account or a paid account.
7. A food delivery purchase is from the food delivery application, online website, phone call ordering, both, or none of the above.
8. Each daily meal may have zero or one breakfast, each breakfast is associated with one or many daily meals.
9. Each daily meal may have zero or one lunch, each lunch is associated with one or many daily meals.
10. Each daily meal may have zero or one dinner, each dinner is associated with one or many daily meals.
11. Each daily meal may have zero or one snack, each snack is associated with one or many daily meals.
12. Each daily meal may have zero or many drinks, one or many drinks are associated with one daily meal.
13. Each paid account may have many account balance changes; each account balance change is for one paid account.
14. Each daily body weight may have many body weight changes; each body weight change is for one daily body weight.



I added the Account_Change entity and Daily_Body_Weight_Change which are related to Paid_Account and Daily_Body_Weight respectively. My updated DBMS physical ERD is below.

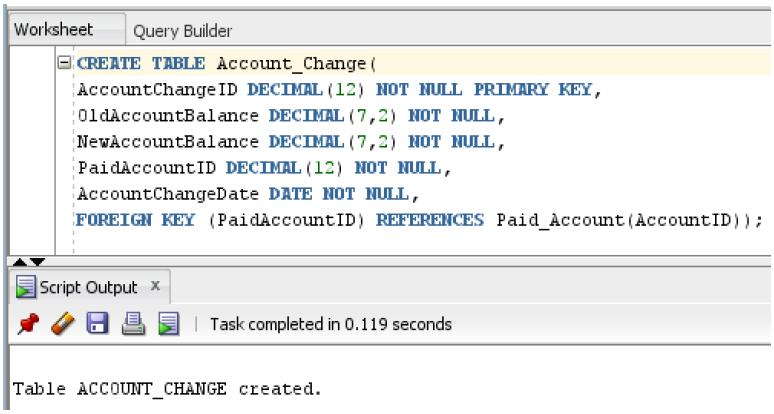


The Account_Change entity is present and linked to PaidAccount. And, Daily_Body_Weight_Change entity is present and linked to Daily_Body_Weight. Below are the attributes I added and why.

Table	Attribute	Datatype	Reasoning
Account _Change	AccountChangeID	DECIMAL(12)	This is the primary key of the Account_Change table. It is a DECIMAL(12) to allow for many values.
	OldAccontBalance	DECIMAL(7,2)	This is the balance of the account before the change. The datatype mirrors the

			AccountBalance datatype in the Paid_Account table.
	NewAccountBalance	DECIMAL(7,2)	This is the balance of the account after the change. The datatype mirrors the AccountBalance datatype in the Paid_Account table.
	PaidAccountID	DECIMAL(12)	This is a foreign key to the Paid_Account table, a reference to the account that had the change in balance.
	AccountChangeDate	DATE	This is the date the balance change occurred, with a DATE datatype.
Daily_Body_Weight_Change	DailyBodyWeightChangeID	DECIMAL(12)	This is the primary key of the Daily_Body_Weight_Change table. It is a DECIMAL(12) to allow for many values.
	DailyBodyWeightID	DECIMAL(12)	This is a foreign key to the Daily_Body_Weight table, a reference to the user that had the change in daily body weight record.
	OldDailyBodyWeight	DECIMAL(5,2)	This is the body weight record of the user before the change. The datatype mirrors the DailyBodyWeight datatype in the Daily_Body_Weight table.
	NewDailyBodyWeight	DECIMAL(5,2)	This is the body weight record of the user after the change. The datatype mirrors the DailyBodyWeight datatype in the Daily_Body_Weight table.
	DailyBodyWeightChangeDate	TIMESTAMP	This is the date and time that the daily body weight record change occurred, with a TIMESTAMP datatype.

Here is a screenshot of my table creation, which has all of the same attributes and datatypes as indicated in the DBMS physical ERD.



```
Worksheet  Query Builder
CREATE TABLE Daily_Body_Weight_Change (
DailyBodyWeightChangeID DECIMAL(12) NOT NULL PRIMARY KEY,
DailyBodyWeightID DECIMAL(12) NOT NULL,
OldDailyBodyWeight DECIMAL(5,2) NOT NULL,
NewDailyBodyWeight DECIMAL(5,2) NOT NULL,
DailyBodyWeightChangeDate TIMESTAMP NOT NULL,
FOREIGN KEY (DailyBodyWeightID) REFERENCES Daily_Body_Weight(DailyBodyWeightID));
Script Output
Task completed in 0.058 seconds
Table DAILY_BODY_WEIGHT_CHANGE created.
```

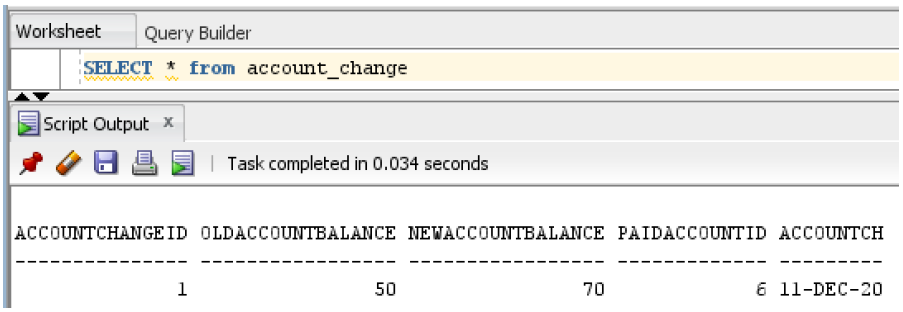
Here is a screenshot of my trigger creation which will maintain the Account_Change table.

```
Worksheet  Query Builder
CREATE OR REPLACE TRIGGER AccountChangeTrigger
BEFORE UPDATE OF AccountBalance ON Paid_Account
FOR EACH ROW
BEGIN
    INSERT INTO Account_Change(AccountChangeID, OldAccountBalance, NewAccountBalance, PaidAccountID, AccountChangeDate)
    VALUES(NVL((SELECT MAX(AccountChangeID)+1 FROM Account_Change), 1),
       :OLD.AccountBalance,
       :NEW.AccountBalance,
       (SELECT AccountID FROM paid_account),
       trunc(sysdate));
END;
Script Output
Task completed in 0.065 seconds
Trigger ACCOUNTCHANGETRIGGER compiled
```

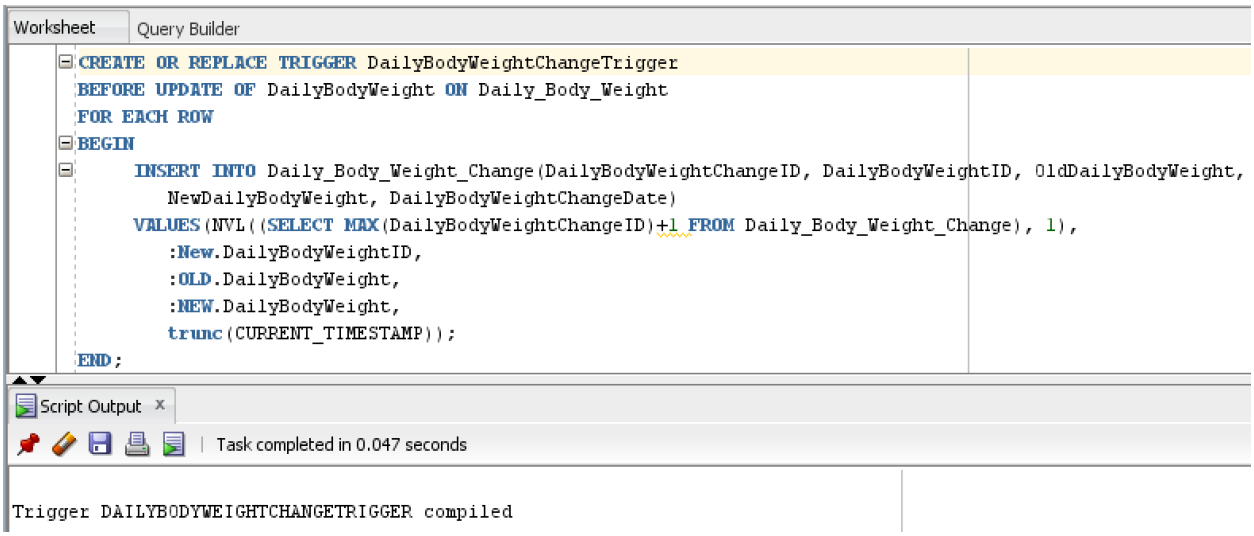
Next, I update the balance a couple of times, once to $50, and again to $70.

```
Worksheet  Query Builder
UPDATE Paid_Account
SET accountbalance = 70
Where AccountID = 6;
Script Output
Task completed in 0.181 seconds
1 row updated.
```

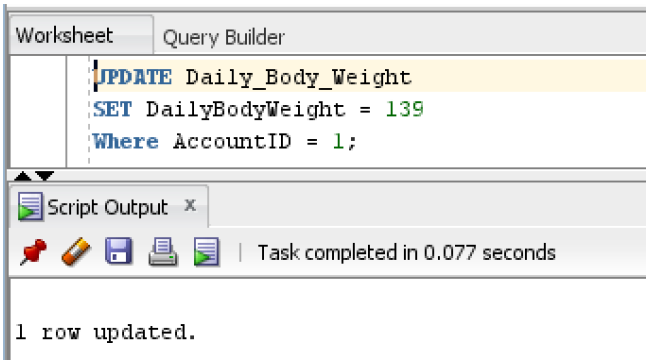
Last, I verify that the Account_Change table has a record of these balance changes in the screenshot below. You'll notice that there is one change history that the account balance has become $70 for that user. The old and new balances are now tracked with a trigger and a history table.



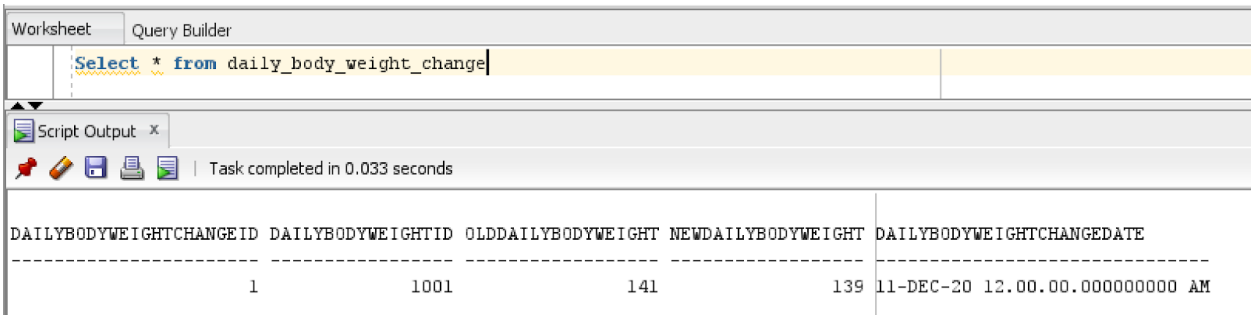
Another trigger is for daily body weight change. Here is a screenshot of my trigger creation which will maintain the Daily_Body_Weight_Change table.



Next, I update the daily body weight record a couple of times, once to 140lbs, and again to 139lbs.



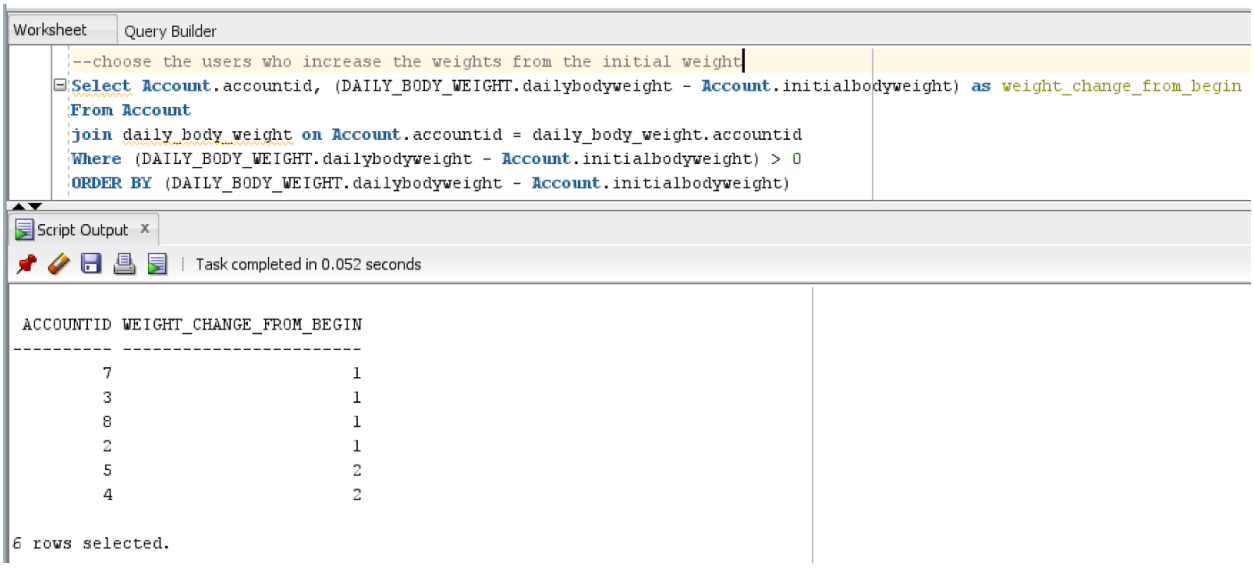
Last, I verify that the Daily_Body_Weight_Change table has a record of these daily body weight record changes in the screenshot below. You'll notice that there is one change history that the weight record has become 139lbs for that user. The old and new records are now tracked with a trigger and a history table.



Questions and Queries

In this process, I created three questions that may need to be used in the future operation of the “Monitor” application. I will explain each question separately, explain the meaning of each question, and show the screenshot of the corresponding query code.

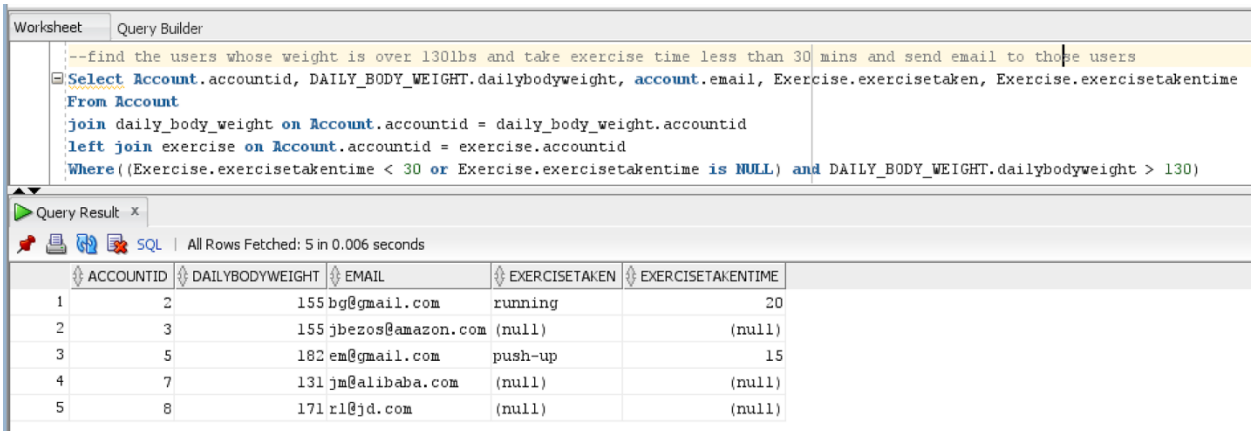
The first question is how many pounds is the difference between the user's newly updated weight record and the weight record recorded when the account was initially created and sorted from small to large to see who has the largest gap among users and who has the smallest. This question is directly related to how the "Monitor" application will give users fitness and diet suggestions in the later stages. Additionally, in the Balance table, users can directly see the difference with the initial body weight. So, this question is very important for the "Monitor" application. Here is a screenshot of the query I use.



To get the results, I first join the Account to the Daily_Body_Weight table, set the constraint that DailyBodyWeight - InitialBodyWeight > 0 to find who has increased the weight while using the

“Monitor” application. Then, I use the ORDER BY clause to sort the difference of weight from small to large.

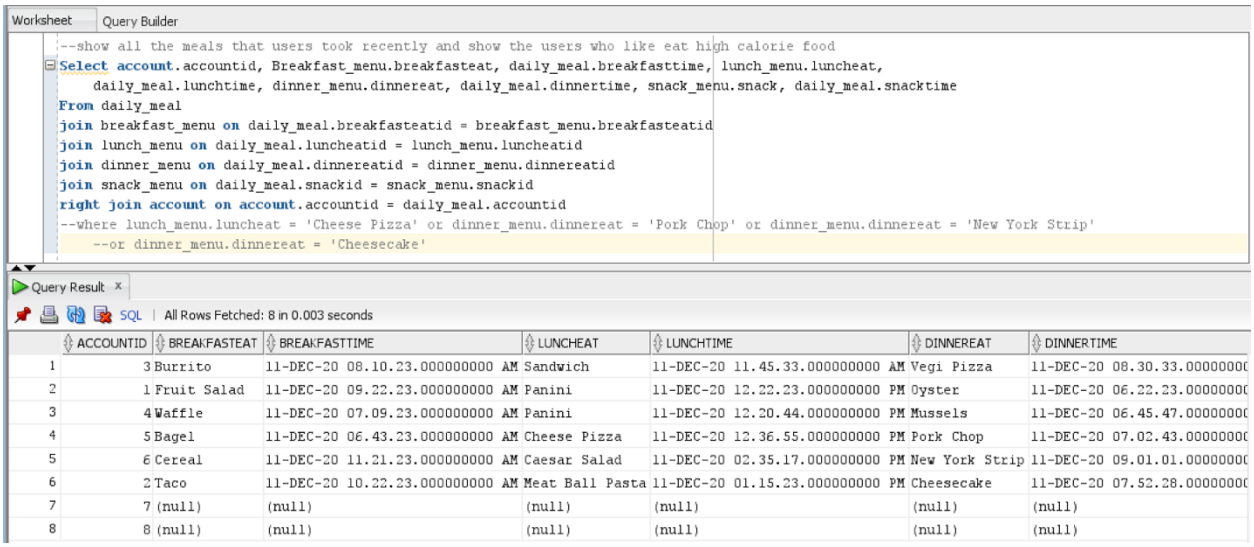
For the second question, I want to find the user whose body weight is over 130lbs and take exercise less than 30 mins. The reason why I want to get the list of those users is that I may email or app notification them later to notice them do more exercise. The “Monitor” application is an application to give users more fitness and diet suggestions. As an operator, I need to focus on the users who don’t like to exercise and help them to exercise by sending notifications or emails. That’s the reason I believe this query will be used in the future and the query is below.



	ACCOUNTID	DAILYBODYWEIGHT	EMAIL	EXERCISETAKEN	EXERCISETAKENTIME
1	2	155	bg@gmail.com	running	20
2	3	155	jbezos@amazon.com	(null)	(null)
3	5	182	em@gmail.com	push-up	15
4	7	131	jm@alibaba.com	(null)	(null)
5	8	171	rl@jd.com	(null)	(null)

In this query, I first join the Account table to the Daily_Body_Weight table and then left join the Exercise. Just in case I won’t miss the user who never inserts this/her exercise record or hasn’t done any exercise yet. Secondly, I use the WHERE clause to find the user whose body weight is over 130lbs and exercise time is less than 30 mins.

The third question is the most important one and related to the algorithm of the “Monitor” application. In the application, we will use the algorithm to analyze the user’s daily calories, vitamins, sugar, and salt content, and water through the information that users input about their diet and weight, as well as other food delivery app/website data, to recommend the proper diet and fitness methods. The list of each users’ meals, drinks will help the algorithm to calculate the daily calories, vitamins, sugar, and salt content. This query also can find out the user’s eating habits and send them a notification if he/she always has a high-calorie intake. Here is a screenshot of the query I use.

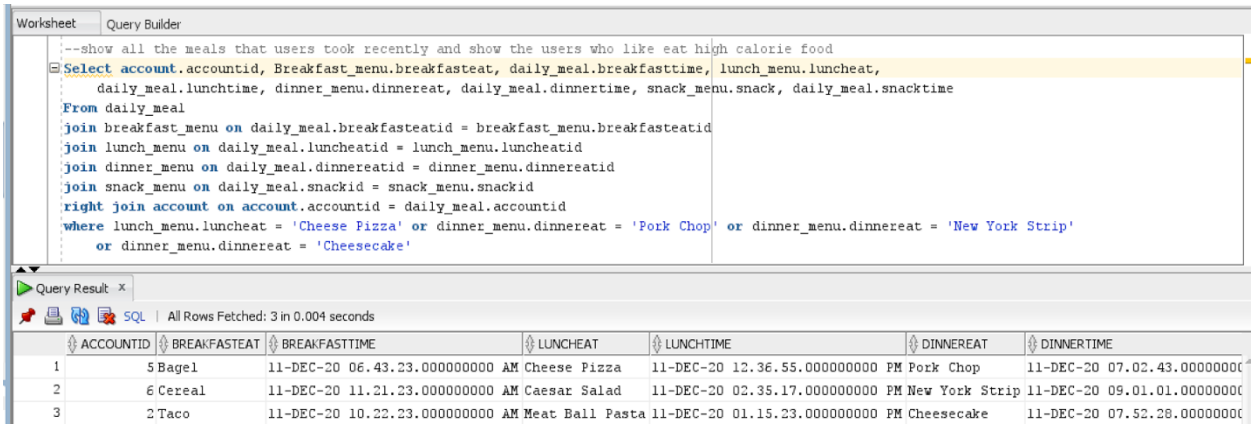


```
--show all the meals that users took recently and show the users who like eat high calorie food
Select account.accountid, Breakfast_menu.breakfasteat, daily_meal.breakfasttime, lunch_menu.luncheat,
    daily_meal.lunchtime, dinner_menu.dinnereat, daily_meal.dinnertime, snack_menu.snack, daily_meal.snacktime
From daily_meal
join breakfast_menu on daily_meal.breakfasteatid = breakfast_menu.breakfasteatid
join lunch_menu on daily_meal.luncheatid = lunch_menu.luncheatid
join dinner_menu on daily_meal.dinnereatid = dinner_menu.dinnereatid
join snack_menu on daily_meal.snackid = snack_menu.snackid
right join account on account.accountid = daily_meal.accountid
--where lunch_menu.luncheat = 'Cheese Pizza' or dinner_menu.dinnereat = 'Pork Chop' or dinner_menu.dinnereat = 'New York Strip'
    --or dinner_menu.dinnereat = 'Cheesecake'
```

Query Result — All Rows Fetched: 8 in 0.003 seconds

	ACCOUNTID	BREAKFASTEAT	BREAKFASTTIME	LUNCHEAT	LUNCHTIME	DINNEREAT	DINNERTIME
1	3	Burrito	11-DEC-20 08.10.23.000000000 AM	Sandwich	11-DEC-20 11.45.33.000000000 AM	Vegi Pizza	11-DEC-20 08.30.33.00000000
2	1	Fruit Salad	11-DEC-20 09.22.23.000000000 AM	Panini	11-DEC-20 12.22.23.000000000 PM	Oyster	11-DEC-20 06.22.23.00000000
3	4	Waffle	11-DEC-20 07.09.23.000000000 AM	Panini	11-DEC-20 12.20.44.000000000 PM	Mussels	11-DEC-20 06.45.47.00000000
4	5	Bagel	11-DEC-20 06.43.23.000000000 AM	Cheese Pizza	11-DEC-20 12.36.55.000000000 PM	Pork Chop	11-DEC-20 07.02.43.00000000
5	6	Cereal	11-DEC-20 11.21.23.000000000 AM	Caesar Salad	11-DEC-20 02.35.17.000000000 PM	New York Strip	11-DEC-20 09.01.01.00000000
6	2	Taco	11-DEC-20 10.22.23.000000000 AM	Meat Ball Pasta	11-DEC-20 01.15.23.000000000 PM	Cheesecake	11-DEC-20 07.52.28.00000000
7	7	(null)	(null)	(null)	(null)	(null)	(null)
8	8	(null)	(null)	(null)	(null)	(null)	(null)

In this query, I join the Daily_Meal table with the other five tables to show all the meals and drinks that users took recently and the users who like to eat high-calorie food. The following screenshot is showing the users who need a notification to help them change their high-calorie diet.



```
--show all the meals that users took recently and show the users who like eat high calorie food
Select account.accountid, Breakfast_menu.breakfasteat, daily_meal.breakfasttime, lunch_menu.luncheat,
    daily_meal.lunchtime, dinner_menu.dinnereat, daily_meal.dinnertime, snack_menu.snack, daily_meal.snacktime
From daily_meal
join breakfast_menu on daily_meal.breakfasteatid = breakfast_menu.breakfasteatid
join lunch_menu on daily_meal.luncheatid = lunch_menu.luncheatid
join dinner_menu on daily_meal.dinnereatid = dinner_menu.dinnereatid
join snack_menu on daily_meal.snackid = snack_menu.snackid
right join account on account.accountid = daily_meal.accountid
where lunch_menu.luncheat = 'Cheese Pizza' or dinner_menu.dinnereat = 'Pork Chop' or dinner_menu.dinnereat = 'New York Strip'
    or dinner_menu.dinnereat = 'Cheesecake'
```

Query Result — All Rows Fetched: 3 in 0.004 seconds

	ACCOUNTID	BREAKFASTEAT	BREAKFASTTIME	LUNCHEAT	LUNCHTIME	DINNEREAT	DINNERTIME
1	5	Bagel	11-DEC-20 06.43.23.000000000 AM	Cheese Pizza	11-DEC-20 12.36.55.000000000 PM	Pork Chop	11-DEC-20 07.02.43.00000000
2	6	Cereal	11-DEC-20 11.21.23.000000000 AM	Caesar Salad	11-DEC-20 02.35.17.000000000 PM	New York Strip	11-DEC-20 09.01.01.00000000
3	2	Taco	11-DEC-20 10.22.23.000000000 AM	Meat Ball Pasta	11-DEC-20 01.15.23.000000000 PM	Cheesecake	11-DEC-20 07.52.28.00000000

Summary and Reflection

In iteration5, I used store procedures to further improve my database and inserted real data to implement the operation of the database. Finally, I added a history table to further improve the database's demand for historical data queries. In the end, I asked three questions about my database, all of which are related to the real operating scenarios of the future database. And, I wrote three queries. I believe these three sentences can also be used in real operation scenarios.

Use Cases Conclusion

I. The first important usage of the database is when a user signs up for an account and installs the application.

Account Signup/Installation Use Case

1. The users visit the app store and install the “Monitor” application.
2. The “Monitor” application asks them to create either a free or paid account when it’s first run.
3. The application also asks users to grant privacy and storage authorization to make it more convenient to obtain users’ information.
4. The users select the type of account and enter their information and the account is created in the database.
5. The application asks users to install browser plugins so that their food delivery purchases can be automatically tracked when they make them.

II. Second important usage of the database is when users input their daily diet and record by “Monitor” application in its database.

User Input Use Case

1. The users input their daily meals/snacks/water they ate and drank or they take a picture of food or receipt in the database.
2. The users input the approximate time they took meals/snacks/water in the database.
3. (Optional) The users record and update their weight in the database.
4. (Optional) The users input today’s exercise time and types in the database.

III. Third important usage of the database is when a food delivery purchase is made and automatically recorded in the “Monitor” application database via browser extensions.

Automatic Purchase Tracking Use Case (new)

1. The user visits a food delivery application or website and makes a purchase.
2. The TrackMyBuys browser plugin detects that the purchase is made, and records the relevant information in the database such as whether the purchase is from a food delivery application or an online website or a phone call order, the purchase date, price, product, store, etc.

IV. Another important usage is when a person decides to look up their past weight and diet history changes and look for the exercise suggestions.

History Lookup and Suggestion Use Case

1. The user signs into the “Monitor” application.
2. The person selects the option to look up past diet history.
3. The “Monitor” application shows the diagram of recent calorie, vitamin, sugar intake, salt intake, water intake, weight changes from the database, and also gives the user the option to search.
4. The user searches based upon a date range, which causes a database search.
5. The application also displays all exercise suggestions from the database.

Full DBMS Conclusion

Table	Attribute	Datatype	Reasoning
Account	UserName	VARCHAR(64)	Every account has a username associated with it, which will be used to login into the "Monitor" application. I allow usernames to be up to 64 characters.
	FirstName	VARCHAR(255)	This is the first name of the account holder, up to 255 characters of the first name.
	LastName	VARCHAR(255)	This is the last name of the account holder, up to 255 characters of the last name.
	EncryptedPassword	VARCHAR(64)	Every account has a password. It will be stored in encrypted text format in the database. 64 characters should be a safe limit to store encrypted text.
	Email	VARCHAR(255)	This is the email address of the account holder. 255 characters should be a safe upper bound.
	Gender	VARCHAR(64)	This is the gender of the account holder. 64 characters should be safe for gender type.
	Height	DECIMAL(3,2)	This is the body height of the account holder. Since we will store centimeters, 3 digits and 2 decimal points will be safe for height type.
	Medical History	VARCHAR(1024)	This is the medical history of the account holder. Since some users may have a different medical history, 1024 characters will be safer.
	InitialBodyWeight	DECIMAL(5,2)	This is the initial body weight of the account holder. Since we will store kilograms, 5 digits and 2 decimal points will be safe for body weight.
	AimGoal	VARCHAR(1024)	This is the aim/goal for using the "Monitor" application, such as keeping fit, losing weight, or keeping a healthier diet. 1024 characters will be safer.
	CreatedTime	DATETIME	This is the date-time when the account holder creates the account. DATETIME can save the date and time so that the application can keep a record of the user's initial body weight and be helpful for calculating the weight changes in the future.
	AccountType	CHAR(1)	As identified in a prior iteration, there are two types of accounts – free and paid. This attribute is the subtype discriminator indicating which it is.

Paid_Account	AccountBalance	DECIMAL(7,2)	This is the unpaid balance, if any, for the paid account. I allow for up to 7 digits and 2 decimal points, though it will likely never get this high.
	RenewalDate	DATE	This is the date on which the account needs to be renewed with a new payment.
Daily_Meal	BreakfastTime	DATETIME	This is the date-time that the account holder takes breakfast approximately. DATETIME can save the date and time so that the application can keep a record of everyday changes.
	BreakfastEat	VARCHAR(1024)	This is the food that the account holder eats at breakfast. Since the data is the record of breakfast, 1024 characters will be safer.
	BreakfastDrink	VARCHAR(255)	This is the drink that the account holder has at breakfast. 255 characters should be a safe upper bound.
	LunchTime	DATETIME	This is the date-time that the account holder takes lunch approximately. DATETIME can save the date and time so that the application can keep a record of everyday changes.
	LunchEat	VARCHAR(1024)	This is the food that the account holder eats at lunch. Since the data is the record of lunch, 1024 characters will be safer.
	LunchDrink	VARCHAR(255)	This is the drink that the account holder has at lunch. 255 characters should be a safe upper bound.
	DinnerTime	DATETIME	This is the date-time that the account holder takes dinner approximately. DATETIME can save the date and time so that the application can keep a record of everyday changes.
	DinnerEat	VARCHAR(1024)	This is the food that the account holder eats at dinner. Since the data is the record of dinner, 1024 characters will be safer.
	DinnerDrink	VARCHAR(255)	This is the drink that the account holder has at dinner. 255 characters should be a safe upper bound.
	Snack	VARCHAR(255)	This is the snack including food and drinks the account holder takes outside of dinner time. 255 characters should be a safe upper bound.
	SnackTime	DATETIME	This is the date-time that the account holder takes a snack approximately. DATETIME can save the date and time so that the application can keep a record of everyday changes.

	Water	VARCHAR(255)	This is the amount of water the account holder drinks outside of meal times. 255 characters should be a safe upper bound.
Daily_Body_Weight	DailyBodyWeight	DECIMAL(5,2)	This is the daily bodyweight of the account holder. Since we will store kilograms, 5 digits and 2 decimal points will be safe for weight.
	DailyBodyWeightDate	DATETIME	This is the date-time of the account holder's daily body weight. DATETIME can save the date and time so that the application can keep a record of the user's everyday body weight changes.
Exercise	ExerciseTaken	VARCHAR(1024)	This is the exercise that the account holder has taken. Users may perform different exercises every day, so I allow for 1024 characters so that people can type in something long if they need to.
	ExerciseTakenTime	DECIMAL(4,2)	This is the length of time that the account holder exercised. I allow for up to 4 digits and standard 2 decimal points.
	ExerciseDate	DATETIME	This is the date-time when the account holder does exercise. DATETIME can save the date and time so that the application can keep a record of the user's exercise.
Food_Delivery_Purchase	FoodDelivery	VARCHAR(1024)	This is the food of online food delivery. Since the data is the record of the food delivery, 1024 characters will be safer.
	DrinkDelivery	VARCHAR(255)	This is the drink of online food delivery. 255 characters should be a safe upper bound.
	DeliveryArrivingTime	DATETIME	This is the arrival time of online food delivery. DATETIME can save the date and time so that the application can keep a record of delivery arrival time.
Food_Delivery_Application	ApplicationPath	VARCHAR(1024)	This is the application path of the food delivery application record so that the food delivery record can directly be stored in the "Monitor" application database instead of being typed in by the account holder.
Food_Delivery_Online_Website	WebsiteURL	VARCHAR(1024)	This is the website URL path of the food delivery record so that the food delivery record can directly be stored in the "Monitor" application database instead of being typed in by the account holder.

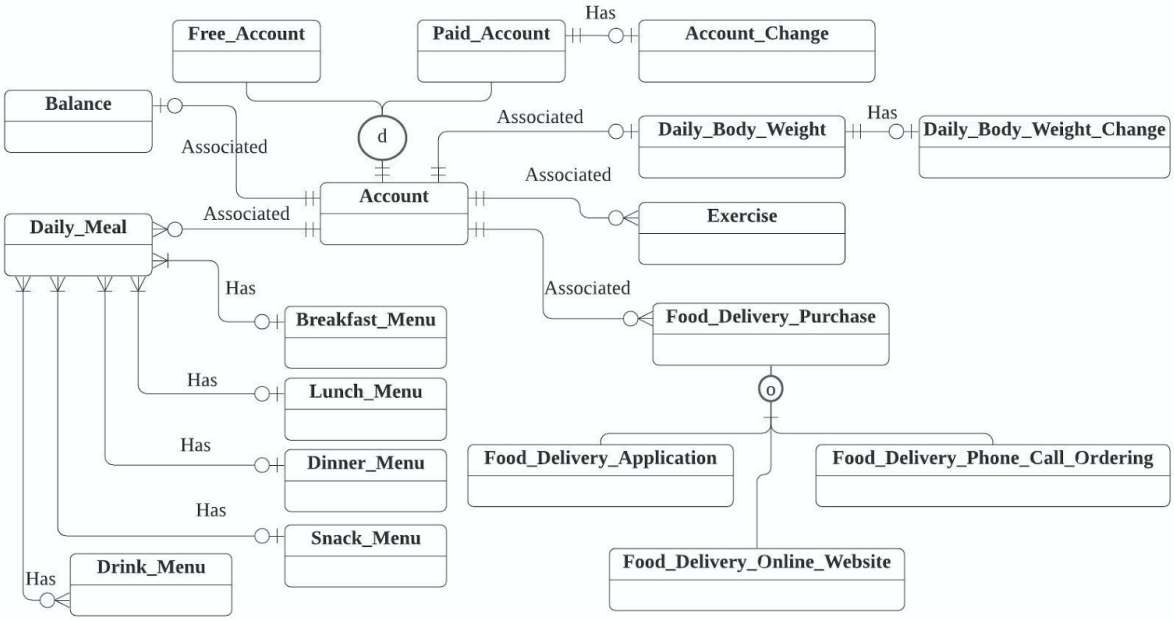
Balance	BalanceDate	DATE	This is the date on which the account holder has different balance status and suggestions for every day.
	CaloriesBalance	VARCHAR(1024)	This is the status of the user's calorie balance after algorithm analysis. I allow for 1024 characters just in case there may be some long suggestions and balance for the user.
	VitaminsBalance	VARCHAR(1024)	This is the status of the user's vitamin balance after algorithm analysis. I allow for 1024 characters just in case there may be some long suggestions and balance for the user.
	SugarIntakeBalance	VARCHAR(1024)	This is the status of the user's sugar intake balance after algorithm analysis. I allow for 1024 characters just in case there may be some long suggestions and balance for the user.
	SaltIntakeBalance	VARCHAR(1024)	This is the status of the user's salt intake balance after algorithm analysis. I allow for 1024 characters just in case there may be some long suggestions and balance for the user.
	WaterIntake Balance	VARCHAR(1024)	This is the status of the user's water intake balance after algorithm analysis. I allow for 1024 characters just in case there may be some long suggestions and balance for the user.
	ExerciseBalance	VARCHAR(1024)	This is the status of the user's exercise balance after algorithm analysis. I allow for 1024 characters just in case there may be some long suggestions and balance for the user.
	WeightBalance	VARCHAR(1024)	This is the status of the user's weight balance after algorithm analysis. I allow for 1024 characters just in case there may be some long suggestions and balance for the user.
Account_Change	AccountChangeID	DECIMAL(12)	This is the primary key of the Account_Change table. It is a DECIMAL(12) to allow for many values.
	OldAccontBalance	DECIMAL(7,2)	This is the balance of the account before the change. The datatype mirrors the AccountBalance datatype in the Paid_Account table.
	NewAccount Balance	DECIMAL(7,2)	This is the balance of the account after the change. The datatype mirrors the AccountBalance datatype in the Paid_Account table.

	PaidAccountID	DECIMAL(12)	This is a foreign key to the Paid_Account table, a reference to the account that had the change in balance.
	AccountChangeDate	DATE	This is the date the balance change occurred, with a DATE datatype.
Daily_Body_Weight_Change	DailyBodyWeightChangeID	DECIMAL(12)	This is the primary key of the Daily_Body_Weight_Change table. It is a DECIMAL(12) to allow for many values.
	DailyBodyWeightID	DECIMAL(12)	This is a foreign key to the Daily_Body_Weight table, a reference to the user that had the change in daily body weight record.
	OldDailyBodyWeight	DECIMAL(5,2)	This is the body weight record of the user before the change. The datatype mirrors the DailyBodyWeight datatype in the Daily_Body_Weight table.
	NewDailyBodyWeight	DECIMAL(5,2)	This is the body weight record of the user after the change. The datatype mirrors the DailyBodyWeight datatype in the Daily_Body_Weight table.
	DailyBodyWeightChangeDate	TIMESTAMP	This is the date and time that the daily body weight record change occurred, with a TIMESTAMP datatype.

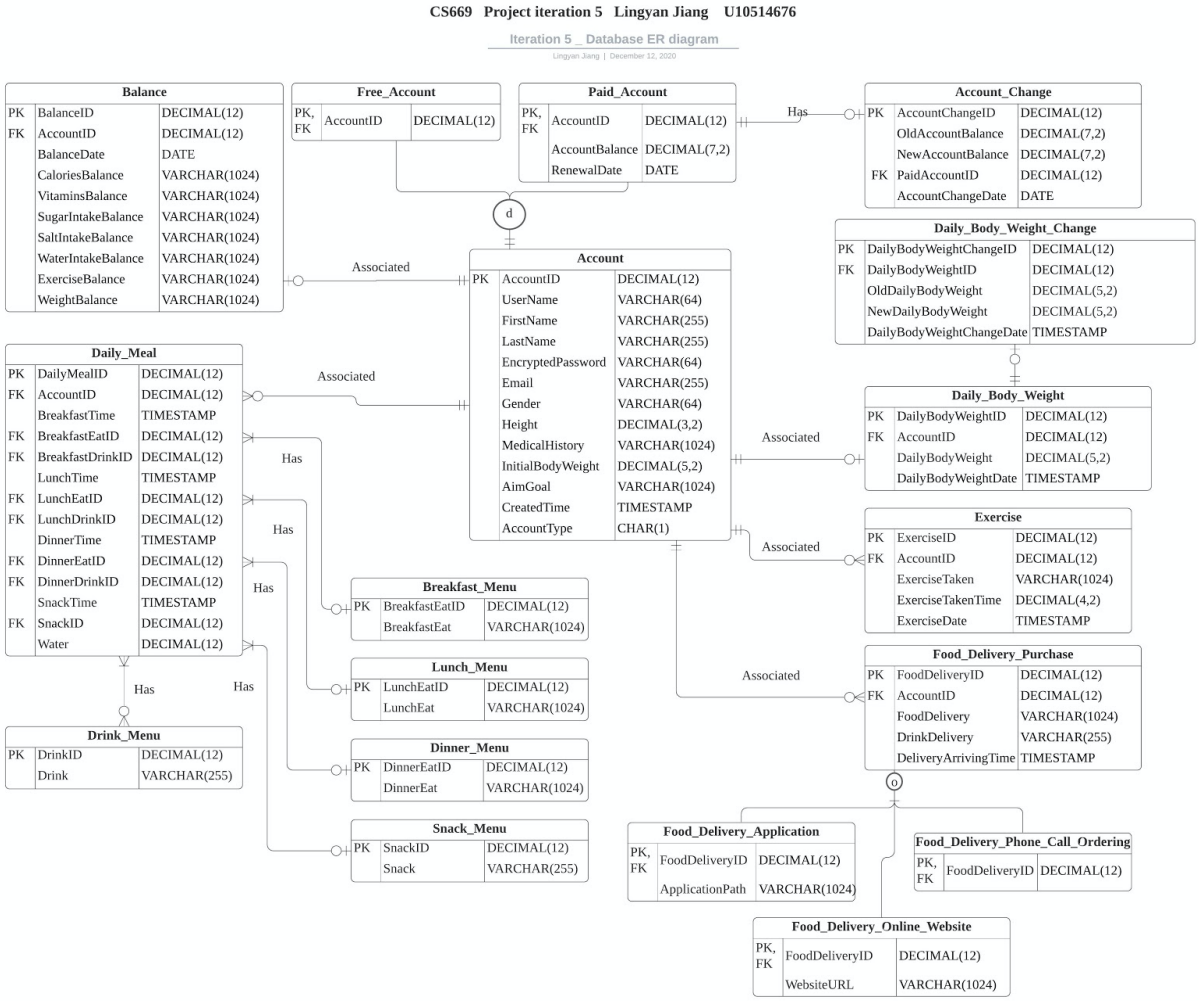
Structural Database Rules Conclusion

Structural Database Rules (iteration 5)
1. Each daily meal is associated with an account, each account may be associated with many daily meals.
2. Each daily body weight is associated with an account, each account may be associated with one daily body weight.
3. Each exercise is associated with an account, each account may be associated with many exercises.
4. Each food delivery purchase is associated with an account, each account may be associated with many food delivery purchases.
5. Each balance is associated with an account, each account may be associated with one balance.
6. A new account is a free account or a paid account.
7. A food delivery purchase is from the food delivery application, online website, phone call ordering, both, or none of the above.
8. Each daily meal may have zero or one breakfast, each breakfast is associated with one or many daily meals.
9. Each daily meal may have zero or one lunch, each lunch is associated with one or many daily meals.
10. Each daily meal may have zero or one dinner, each dinner is associated with one or many daily meals.
11. Each daily meal may have zero or one snack, each snack is associated with one or many daily meals.
12. Each daily meal may have zero or many drinks, one or many drinks are associated with one daily meal.
13. Each paid account may have many account balance changes; each account balance change is for one paid account.
14. Each daily body weight may have many body weight changes; each body weight change is for one daily body weight.

Conceptual ERD Conclusion



DBMS physical ERD Conclusion



Term Project Reflection

My database is for a mobile app named "Monitor-Daily Food Record" application which the user's daily diet and gives reasonable fitness exercises and diet suggestions. Typically, many people don't know whether their daily diets are healthy or not, whether their calorie intake exceeds the recommended limit, or what exercises they can do to expend the calories they have consumed. This app will solve this problem by analyzing the user's daily calories, vitamins, sugar, and salt content, and water through the information that users input about their diet and weight, as well as other food delivery app/website data, so as to recommend the proper diet and fitness methods.

The "Monitor" application database must support a person entering, searching, and even analyzing their delivery ordering purchases across all approaches. The structural database rules and conceptual ERD for my database design contain the important entities of Account, Free_Account, Paid_Account, Account_Change, Daily_Body_Weight, Daily_Body_Weight_Change, Exercise, Food_Delivery_Purchase, Food_Delivery_Application, Food_Delivery_Phone_Call_Ordering, Food_Delivery_Online_Website, Daily_Meal, Balance, Breakfast_Menu, Lunch_Menu, Dinner_Menu, Snack_Menu, and Drink_Menu, as well as relationships between them.
The design also contains a hierarchy of Account/Paid_Account and Account/Free_Account to reflect the fact that people can signup for a free account or a paid account for the "Monitor" application. The design also contains a hierarchy of Food_Delivery_Purchase/Food_Delivery_Application, Food_Delivery_Purchase/Food_Delivery_Online_Website, and Food_Delivery_Purchase/Food_Delivery_Phone_Call_Ordering to reflect the three primary ways people purchase food delivery. The DBMS physical ERD contains the same entities and relationships, uses the best practice of synthetic keys, and contains the important attributes needed by the database to support the application.

The SQL script contains all table creations that follow the specification from the DBMS physical ERD exactly. Important indexes have been created to help speed up access to my database and are also available in an index script. Stored procedures have been created and executed transactionally to populate some of my database with data. Some questions useful to the "Monitor" application have been identified and implemented with SQL queries. A history of balances has been created as well as a query that tracks changes to balances and the user's body weight.

During those four months, I learned a lot about database design, management, creation from this final project. It is amazing to see a real database in action that is created and designed by myself. I can see there is still more to develop in my database, but feel it's a solid foundation to move forward with. I hope that the application design and subsequent algorithms can be combined to launch this "Monitor" application on Google Play.

Appendix

```
DROP TABLE Free_Account;
DROP TABLE Paid_Account;
DROP TABLE Balance;
DROP TABLE Daily_Meal;
DROP TABLE Drink_Menu;
DROP TABLE Breakfast_Menu;
DROP TABLE Lunch_Menu;
DROP TABLE Dinner_Menu;
DROP TABLE Snack_Menu;
DROP TABLE Daily_Body_Weight;
DROP TABLE Exercise;
DROP TABLE Food_Delivery_Application;
DROP TABLE Food_Delivery_Online_Website;
DROP TABLE Food_Delivery_Phone_Call_Ordering;
DROP TABLE Food_Delivery_Purchase;
DROP TABLE Account;


CREATE TABLE Account (
AccountID DECIMAL(12) NOT NULL PRIMARY KEY,
UserName VARCHAR(64) NOT NULL,
FirstName VARCHAR(255) NOT NULL,
LastName VARCHAR(255) NOT NULL,
EncryptedPassword VARCHAR(64) NOT NULL,
Email VARCHAR(255) NOT NULL,
Gender VARCHAR(64) NOT NULL,
Height DECIMAL(3,2) NOT NULL,
MedicalHistory VARCHAR(1024) NOT NULL,
InitialBodyWeight DECIMAL(5,2) NOT NULL,
AimGoal VARCHAR(1024) NOT NULL,
CreatedTime TIMESTAMP NOT NULL,
AccountType CHAR(1) NOT NULL);


CREATE TABLE Free_Account (
AccountID DECIMAL(12) NOT NULL PRIMARY KEY,
FOREIGN KEY (AccountID) REFERENCES Account(AccountID));

CREATE TABLE Paid_Account (
AccountID DECIMAL(12) NOT NULL PRIMARY KEY,
AccountBalance DECIMAL(7,2) NOT NULL,
```

```
RenewalDate DATE NOT NULL,
FOREIGN KEY (AccountID) REFERENCES Account(AccountID));

CREATE TABLE Breakfast_Menu (
BreakfastEatID DECIMAL(12) NOT NULL PRIMARY KEY,
BreakfastEat VARCHAR(1024) NOT NULL);


CREATE TABLE Lunch_Menu (
LunchEatID DECIMAL(12) NOT NULL PRIMARY KEY,
LunchEat VARCHAR(1024) NOT NULL);


CREATE TABLE Dinner_Menu (
DinnerEatID DECIMAL(12) NOT NULL PRIMARY KEY,
DinnerEat VARCHAR(1024) NOT NULL);


CREATE TABLE Snack_Menu (
SnackID DECIMAL(12) NOT NULL PRIMARY KEY,
Snack VARCHAR(255) NOT NULL);


CREATE TABLE Drink_Menu (
DrinkID DECIMAL(12) NOT NULL PRIMARY KEY,
Drink VARCHAR(255) NOT NULL);


CREATE TABLE Daily_Meal (
DailyMealID DECIMAL(12) NOT NULL PRIMARY KEY,
AccountID DECIMAL(12) NOT NULL,
BreakfastTime TIMESTAMP NOT NULL,
BreakfastEatID DECIMAL(12) NOT NULL,
BreakfastDrinkID DECIMAL(12) NOT NULL,
LunchTime TIMESTAMP NOT NULL,
LunchEatID DECIMAL(12) NOT NULL,
LunchDrinkID DECIMAL(12) NOT NULL,
DinnerTime TIMESTAMP NOT NULL,
DinnerEatID DECIMAL(12) NOT NULL,
DinnerDrinkID DECIMAL(12) NOT NULL,
SnackTime TIMESTAMP NOT NULL,
SnackID DECIMAL(12) NOT NULL,
Water DECIMAL(12) NOT NULL,
FOREIGN KEY (AccountID) REFERENCES Account(AccountID),
```

```
FOREIGN KEY (BreakfastEatID) REFERENCES Breakfast_Menu(BreakfastEatID),
FOREIGN KEY (BreakfastDrinkID) REFERENCES Drink_Menu(DrinkID),
FOREIGN KEY (LunchEatID) REFERENCES Lunch_Menu(LunchEatID),
FOREIGN KEY (LunchDrinkID) REFERENCES Drink_Menu(DrinkID),
FOREIGN KEY (DinnerEatID) REFERENCES Dinner_Menu(DinnerEatID),
FOREIGN KEY (DinnerDrinkID) REFERENCES Drink_Menu(DrinkID),
FOREIGN KEY (SnackID) REFERENCES Snack_Menu(SnackID));


CREATE TABLE Daily_Body_Weight (
DailyBodyWeightID DECIMAL(12) NOT NULL PRIMARY KEY,
AccountID DECIMAL(12) NOT NULL,
DailyBodyWeight DECIMAL(5,2) NOT NULL,
DailyBodyWeightDate TIMESTAMP NOT NULL,
FOREIGN KEY (AccountID) REFERENCES Account(AccountID));

CREATE TABLE Exercise (
ExerciseID DECIMAL(12) NOT NULL PRIMARY KEY,
AccountID DECIMAL(12) NOT NULL,
ExerciseTaken VARCHAR(1024) NOT NULL,
ExerciseTakenTime DECIMAL(4,2) NOT NULL,
ExerciseDate TIMESTAMP NOT NULL,
FOREIGN KEY (AccountID) REFERENCES Account(AccountID));


CREATE TABLE Balance (
BalanceID DECIMAL(12) NOT NULL PRIMARY KEY,
AccountID DECIMAL(12) NOT NULL,
BalanceDate DATE NOT NULL,
CaloriesBalance VARCHAR(1024) NOT NULL,
VitaminsBalance VARCHAR(1024) NOT NULL,
SugarIntakeBalance VARCHAR(1024) NOT NULL,
SaltIntakeBalance VARCHAR(1024) NOT NULL,
WaterIntakeBalance VARCHAR(1024) NOT NULL,
ExerciseBalance VARCHAR(1024) NOT NULL,
WeightBalance VARCHAR(1024) NOT NULL,
FOREIGN KEY (AccountID) REFERENCES Account(AccountID));


CREATE TABLE Food_Delivery_Purchase (
FoodDeliveryID DECIMAL(12) NOT NULL PRIMARY KEY,
AccountID DECIMAL(12) NOT NULL,
FoodDelivery VARCHAR(1024) NOT NULL,
DrinkDelivery VARCHAR(255) NOT NULL,
```

```
DeliveryArrivingTime TIMESTAMP NOT NULL,
FOREIGN KEY (AccountID) REFERENCES Account(AccountID));


CREATE TABLE Food_Delivery_Phone_Call_Ordering (
FoodDeliveryID DECIMAL(12) NOT NULL PRIMARY KEY,
FOREIGN KEY (FoodDeliveryID) REFERENCES Food_Delivery_Purchase(FoodDeliveryID));


CREATE TABLE Food_Delivery_Online_Website (
FoodDeliveryID DECIMAL(12) NOT NULL PRIMARY KEY,
WebsiteURL VARCHAR(1024) NOT NULL,
FOREIGN KEY (FoodDeliveryID) REFERENCES Food_Delivery_Purchase(FoodDeliveryID));


CREATE TABLE Food_Delivery_Application (
FoodDeliveryID DECIMAL(12) NOT NULL PRIMARY KEY,
ApplicationPath VARCHAR(1024)NOT NULL,
FOREIGN KEY (FoodDeliveryID) REFERENCES Food_Delivery_Purchase(FoodDeliveryID));

CREATE INDEX AccountCreatedTimeIdx ON  Account(CreatedTime);


CREATE TABLE Account_Change(
AccountChangeID DECIMAL(12) NOT NULL PRIMARY KEY,
OldAccountBalance DECIMAL(7,2) NOT NULL,
NewAccountBalance DECIMAL(7,2) NOT NULL,
PaidAccountID DECIMAL(12) NOT NULL,
AccountChangeDate DATE NOT NULL,
FOREIGN KEY (PaidAccountID) REFERENCES Paid_Account(AccountID));


CREATE TABLE Daily_Body_Weight_Change (
DailyBodyWeightChangeID DECIMAL(12) NOT NULL PRIMARY KEY,
DailyBodyWeightID DECIMAL(12) NOT NULL,
OldDailyBodyWeight DECIMAL(5,2) NOT NULL,
NewDailyBodyWeight DECIMAL(5,2) NOT NULL,
DailyBodyWeightChangeDate TIMESTAMP NOT NULL,
FOREIGN KEY (DailyBodyWeightID) REFERENCES Daily_Body_Weight(DailyBodyWeightID));


--Store Procedure
CREATE OR REPLACE PROCEDURE AddFreeAccount(AccountID IN DECIMAL, UserName IN
VARCHAR, FirstName IN VARCHAR,
```

```
   LastName IN VARCHAR, EncryptedPassword IN VARCHAR, Email IN VARCHAR, Gender IN
VARCHAR, Height IN DECIMAL,
   MedicalHistory IN VARCHAR, InitialBodyWeight IN DECIMAL, AimGoal IN VARCHAR)
AS
BEGIN
 INSERT INTO Account(AccountID, UserName, FirstName, LastName, EncryptedPassword, Email,
Gender, Height,
   MedicalHistory, InitialBodyWeight, AimGoal, CreatedTime, AccountType)
 VALUES(AccountID, UserName, FirstName, LastName, EncryptedPassword, Email, Gender,
Height,
   MedicalHistory, InitialBodyWeight, AimGoal?CURRENT_TIMESTAMP, 'F');

 INSERT INTO free_account(AccountID)
 VALUES(AccountID);
END;


CREATE OR REPLACE PROCEDURE AddPaidAccount(AccountID IN DECIMAL, UserName IN
VARCHAR, FirstName IN VARCHAR,
   LastName IN VARCHAR, EncryptedPassword IN VARCHAR, Email IN VARCHAR, Gender IN
VARCHAR, Height IN DECIMAL,
   MedicalHistory IN VARCHAR, InitialBodyWeight IN DECIMAL, AimGoal IN VARCHAR,
AccountBalance IN DECIMAL, RenewalDate IN DATE)
AS
BEGIN
 INSERT INTO Account(AccountID, UserName, FirstName, LastName, EncryptedPassword, Email,
Gender, Height,
   MedicalHistory, InitialBodyWeight, AimGoal, CreatedTime, AccountType)
 VALUES(AccountID, UserName, FirstName, LastName, EncryptedPassword, Email, Gender,
Height,
   MedicalHistory, InitialBodyWeight, AimGoal?CURRENT_TIMESTAMP, 'P');

 INSERT INTO Paid_account(AccountID, AccountBalance, RenewalDate)
 VALUES(AccountID, AccountBalance, RenewalDate);
END;


CREATE OR REPLACE PROCEDURE AddDailyMeal(DailyMealID IN DECIMAL, AccountID IN
DECIMAL, BreakfastTime IN TIMESTAMP,
   BreakfastEatID IN DECIMAL, BreakfastDrinkID IN DECIMAL, LunchTime IN TIMESTAMP,
LunchEatID IN DECIMAL,
   LunchDrinkID IN DECIMAL, DinnerTime IN TIMESTAMP, DinnerEatID IN DECIMAL,
DinnerDrinkID IN DECIMAL,
   SnackTime IN TIMESTAMP, SnackID IN DECIMAL, Water IN DECIMAL)
```

```
AS
BEGIN
 INSERT INTO Daily_Meal(DailyMealID, AccountID, BreakfastTime, BreakfastEatID,
BreakfastDrinkID, LunchTime,
   LunchEatID, LunchDrinkID, DinnerTime, DinnerEatID, DinnerDrinkID,SnackTime, SnackID,
Water)
 VALUES(DailyMealID, AccountID, BreakfastTime, BreakfastEatID, BreakfastDrinkID, LunchTime,
   LunchEatID, LunchDrinkID, DinnerTime, DinnerEatID, DinnerDrinkID,SnackTime, SnackID,
Water);
END;


CREATE OR REPLACE PROCEDURE AddDailyBodyWeight(DailyBodyWeightID IN DECIMAL,
AccountID IN DECIMAL,
   DailyBodyWeight IN DECIMAL)
AS
BEGIN
 INSERT INTO Daily_Body_Weight(DailyBodyWeightID, AccountID, DailyBodyWeight,
DailyBodyWeightDate)
 VALUES(DailyBodyWeightID, AccountID, DailyBodyWeight, CURRENT_TIMESTAMP);
END;


CREATE OR REPLACE PROCEDURE AddExercise(ExerciseID IN DECIMAL, AccountID IN DECIMAL,
   ExerciseTaken IN VARCHAR, ExerciseTakenTime IN DECIMAL, ExerciseDate IN TIMESTAMP)
AS
BEGIN
 INSERT INTO Exercise(ExerciseID, AccountID, ExerciseTaken, ExerciseTakenTime, ExerciseDate)
 VALUES(ExerciseID, AccountID, ExerciseTaken, ExerciseTakenTime, ExerciseDate);
END;


--insert free account data
BEGIN
   addfreeaccount(1, 'LJ', 'Lingyan', 'Jiang', '9683', 'lingyanj@bu.edu', 'Female' , 5.4, 'penicillin
allergy', 140, 'keep fit');
   addfreeaccount(2, 'BG', 'Bill', 'Gates', '1028', 'bg@gmail.com', 'Male' , 5.9, 'None', 154, 'Keep
fit');
   addfreeaccount(3, 'JB', 'Jeff', 'Bezos', '0112', 'jbezos@amazon.com', 'Male' , 5.7, 'None', 154,
'Get married again');
   addfreeaccount(4, 'WB', 'Warren', 'Buffett', '0830', 'wb@gmail.com', 'Male' , 5.9, 'None', 190,
'Live forever');
   addfreeaccount(5, 'EM', 'Elon', 'Musk', '0628', 'em@gmail.com', 'Male' , 6.2, 'None', 180, 'To
Space');
```

```
   COMMIT;
END;


--insert paid account data
BEGIN
   addpaidaccount(6, 'PC', 'Priscilla', 'Chan', '0224', 'pc@gmail.com', 'Female' , 5.5, 'None', 126,
'Lose weight', 50, CAST('11-DEC-2020' AS DATE));
   addpaidaccount(7, 'JM', 'Jack', 'Ma', '0910', 'jm@alibaba.com', 'Male' , 5.4, 'None', 130, 'Keep
fit', 100, CAST('11-DEC-2020' AS DATE));
   addpaidaccount(8, 'RL', 'Richard', 'Liu', '0310', 'rl@jd.com', 'Male' , 6.0, 'None', 170, 'Keep Fit',
150, CAST('11-DEC-2020' AS DATE));
   COMMIT;
END;


--insert daily meal data
BEGIN
   adddailymeal(100, 1, timestamp '2020-12-11 09:22:23', 10002, 50005, timestamp '2020-12-
11 12:22:23', 20003, 50001,
   timestamp '2020-12-11 18:22:23', 30004, 50003, timestamp '2020-12-11 20:22:23', 40002,
1200);
   adddailymeal(101, 2, timestamp '2020-12-11 10:22:23', 10008, 50005, timestamp '2020-12-
11 13:15:23', 20005, 50001,
   timestamp '2020-12-11 19:52:28', 30010, 50004, timestamp '2020-12-11 11:15:23', 40001,
1000);
   adddailymeal(102, 3, timestamp '2020-12-11 08:10:23', 10007, 50004, timestamp '2020-12-
11 11:45:33', 20004, 50002,
   timestamp '2020-12-11 20:30:33', 30001, 50006, timestamp '2020-12-11 16:12:23', 40002,
800);
   adddailymeal(103, 4, timestamp '2020-12-11 07:09:23', 10005, 50003, timestamp '2020-12-
11 12:20:44', 20003, 50003,
   timestamp '2020-12-11 18:45:47', 30005, 50001, timestamp '2020-12-11 23:43:23', 40003,
900);
   adddailymeal(104, 5, timestamp '2020-12-11 06:43:23', 10006, 50002, timestamp '2020-12-
11 12:36:55', 20002, 50006,
   timestamp '2020-12-11 19:02:43', 30007, 50007, timestamp '2020-12-11 10:22:23', 40004,
1500);
   adddailymeal(105, 6, timestamp '2020-12-11 11:21:23', 10003, 50001, timestamp '2020-12-
11 14:35:17', 20001, 50003,
   timestamp '2020-12-11 21:01:01', 30009, 50008, timestamp '2020-12-11 19:22:23', 40005,
600);
   COMMIT;
END;
```

```
--insert daily body weight
BEGIN
   AddDailyBodyWeight(1001, 1, 141);
   AddDailyBodyWeight(1002, 2, 155);
   AddDailyBodyWeight(1003, 3, 155);
   AddDailyBodyWeight(1004, 4, 192);
   AddDailyBodyWeight(1005, 5, 182);
   AddDailyBodyWeight(1006, 6, 125);
   AddDailyBodyWeight(1007, 7, 131);
   AddDailyBodyWeight(1008, 8, 171);
   COMMIT;
END;


--insert exercise
BEGIN
   addexercise(2001, 1, 'walking', 60, timestamp '2020-12-11 07:00:23');
   addexercise(2002, 2, 'running', 20, timestamp '2020-12-11 09:00:23');
   addexercise(2003, 4, 'jogging', 30, timestamp '2020-12-11 10:00:23');
   addexercise(2004, 5, 'push-up', 15, timestamp '2020-12-11 14:00:23');
   addexercise(2005, 6, 'dancing', 30, timestamp '2020-12-11 09:00:23');
   COMMIT;
END;


--insert normal data
INSERT INTO Breakfast_Menu VALUES(10001, 'Garden Salad');
INSERT INTO Breakfast_Menu VALUES(10002, 'Fruit Salad');
INSERT INTO Breakfast_Menu VALUES(10003, 'Cereal');
INSERT INTO Breakfast_Menu VALUES(10004, 'Pancake');
INSERT INTO Breakfast_Menu VALUES(10005, 'Waffle');
INSERT INTO Breakfast_Menu VALUES(10006, 'Bagel');
INSERT INTO Breakfast_Menu VALUES(10007, 'Burrito');
INSERT INTO Breakfast_Menu VALUES(10008, 'Taco');
INSERT INTO Breakfast_Menu VALUES(10009, 'Croissant');

INSERT INTO Lunch_Menu VALUES(20001, 'Caesar Salad');
INSERT INTO Lunch_Menu VALUES(20002, 'Cheese Pizza');
INSERT INTO Lunch_Menu VALUES(20003, 'Panini');
INSERT INTO Lunch_Menu VALUES(20004, 'Sandwich');
INSERT INTO Lunch_Menu VALUES(20005, 'Meat Ball Pasta');
```

```
INSERT INTO Dinner_Menu VALUES(30001, 'Vegi Pizza');
INSERT INTO Dinner_Menu VALUES(30002, 'House Salad');
INSERT INTO Dinner_Menu VALUES(30003, 'Clam Chowder');
INSERT INTO Dinner_Menu VALUES(30004, 'Oyster');
INSERT INTO Dinner_Menu VALUES(30005, 'Mussels');
INSERT INTO Dinner_Menu VALUES(30006, 'Mac Cheese');
INSERT INTO Dinner_Menu VALUES(30007, 'Pork Chop');
INSERT INTO Dinner_Menu VALUES(30008, 'Rib Eye');
INSERT INTO Dinner_Menu VALUES(30009, 'New York Strip');
INSERT INTO Dinner_Menu VALUES(30010, 'Cheesecake');

INSERT INTO snack_menu VALUES(40001, 'Chocolate Bar');
INSERT INTO snack_menu VALUES(40002, 'Terra');
INSERT INTO snack_menu VALUES(40003, 'Mochi');
INSERT INTO snack_menu VALUES(40004, 'Jack Links');
INSERT INTO snack_menu VALUES(40005, 'Pudding');

INSERT INTO Drink_Menu VALUES(50001, 'Coffee');
INSERT INTO Drink_Menu VALUES(50002, 'Fresh Juice');
INSERT INTO Drink_Menu VALUES(50003, 'Tea');
INSERT INTO Drink_Menu VALUES(50004, 'Soft Drink');
INSERT INTO Drink_Menu VALUES(50005, 'Milk');
INSERT INTO Drink_Menu VALUES(50006, 'Soda');
INSERT INTO Drink_Menu VALUES(50007, 'Red Wine');
INSERT INTO Drink_Menu VALUES(50008, 'Beer');

--Trigger of History
create or replace NONEDITIONABLE TRIGGER AccountChangeTrigger
BEFORE UPDATE OF AccountBalance ON Paid_Account
FOR EACH ROW
BEGIN
   INSERT INTO Account_Change(AccountChangeID, OldAccountBalance, NewAccountBalance,
PaidAccountID, AccountChangeDate)
   VALUES(NVL((SELECT MAX(AccountChangeID)+1 FROM Account_Change), 1),
     :OLD.AccountBalance,
     :NEW.AccountBalance,
     :New.AccountID,
     trunc(sysdate));
END;

UPDATE Paid_Account
SET accountbalance = 70
Where AccountID = 6;
```

```
CREATE OR REPLACE TRIGGER DailyBodyWeightChangeTrigger
BEFORE UPDATE OF DailyBodyWeight ON Daily_Body_Weight
FOR EACH ROW
BEGIN
   INSERT INTO Daily_Body_Weight_Change(DailyBodyWeightChangeID, DailyBodyWeightID,
OldDailyBodyWeight,
     NewDailyBodyWeight, DailyBodyWeightChangeDate)
   VALUES(NVL((SELECT MAX(DailyBodyWeightChangeID)+1 FROM
Daily_Body_Weight_Change), 1),
     :New.DailyBodyWeightID,
     :OLD.DailyBodyWeight,
     :NEW.DailyBodyWeight,
     trunc(CURRENT_TIMESTAMP));
END;

UPDATE Daily_Body_Weight
SET DailyBodyWeight = 139
Where AccountID = 1;


--Query
--choose the users who increase the weights from the initial weight
Select Account.accountid, (DAILY_BODY_WEIGHT.dailybodyweight - Account.initialbodyweight)
as weight_change_from_begin
From Account
join daily_body_weight on Account.accountid = daily_body_weight.accountid
Where (DAILY_BODY_WEIGHT.dailybodyweight - Account.initialbodyweight) > 0
ORDER BY (DAILY_BODY_WEIGHT.dailybodyweight - Account.initialbodyweight)


--find the users whose weight is over 130lbs and take exercise time less than 30 mins and send
email to those users
Select Account.accountid, DAILY_BODY_WEIGHT.dailybodyweight, account.email,
Exercise.exercisetaken, Exercise.exercisetakentime
From Account
join daily_body_weight on Account.accountid = daily_body_weight.accountid
left join exercise on Account.accountid = exercise.accountid
Where((Exercise.exercisetakentime < 30 or Exercise.exercisetakentime is NULL) and
DAILY_BODY_WEIGHT.dailybodyweight > 130)


--find user daily food and the user who like high-calorie food
```

```
Select account.accountid, Breakfast_menu.breakfasteat, daily_meal.breakfasttime,
lunch_menu.luncheat,
   daily_meal.lunchtime, dinner_menu.dinnereat, daily_meal.dinnertime, snack_menu.snack,
daily_meal.snacktime
From daily_meal
join breakfast_menu on daily_meal.breakfasteatid = breakfast_menu.breakfasteatid
join lunch_menu on daily_meal.luncheatid = lunch_menu.luncheatid
join dinner_menu on daily_meal.dinnereatid = dinner_menu.dinnereatid
join snack_menu on daily_meal.snackid = snack_menu.snackid
right join account on account.accountid = daily_meal.accountid
where lunch_menu.luncheat = 'Cheese Pizza' or dinner_menu.dinnereat = 'Pork Chop' or
dinner_menu.dinnereat = 'New York Strip'
   or dinner_menu.dinnereat = 'Cheesecake'
```