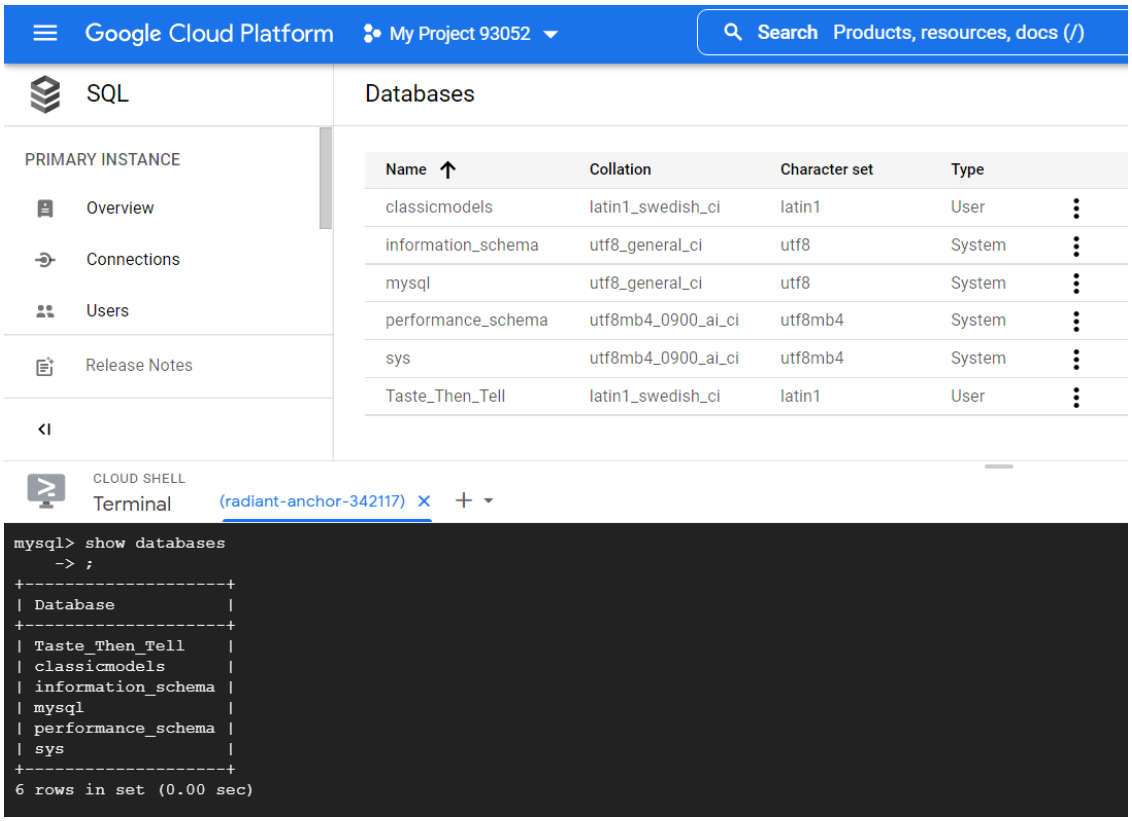
Stage 3: Database Design

Database Implementation

Implement Database Tables (Within “Taste_Then_Tell”)



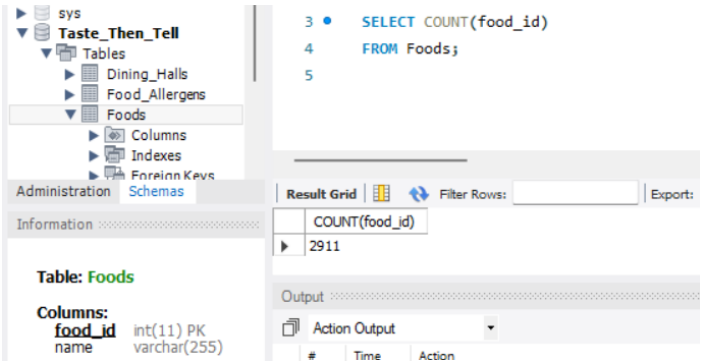
DDL Commands

```
CREATE TABLE foods
  (
     food_id  INT PRIMARY KEY,
     name     VARCHAR(255)
  );

CREATE TABLE universities
  (
     university_id INT PRIMARY KEY,
     location      VARCHAR(255)
     name          VARCHAR(100),

  );

CREATE TABLE dining_halls
  (
     university_id  INT,
     dining_hall_id INT PRIMARY KEY,
     name           VARCHAR(255),
     FOREIGN KEY (university_id) REFERENCES universities(university_id) ON
     DELETE CASCADE
  );

CREATE TABLE students
  (
     user_id       INT PRIMARY KEY,
     university_id INT,
     first_name    VARCHAR(255),
     last_name     VARCHAR(255),
     password      VARCHAR(50),
     FOREIGN KEY (university_id) REFERENCES universities(university_id) ON
     DELETE CASCADE
  );

CREATE TABLE reviews
  (
     review_id      INT PRIMARY KEY,
     user_id        INT,
```

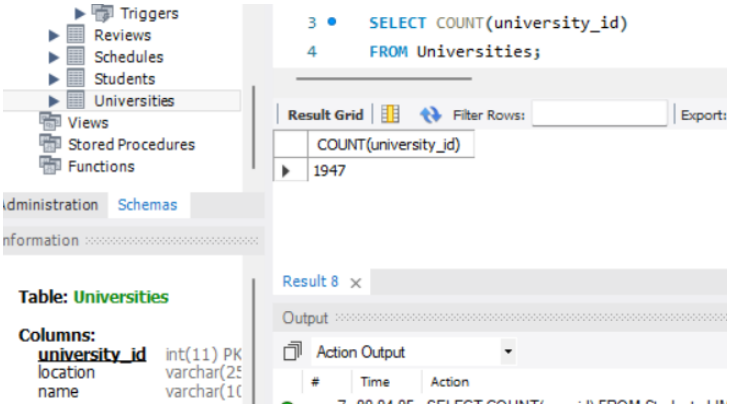
```
    food_id        INT,
    dining_hall_id INT,
    rating         REAL,
    feedback       VARCHAR(255),
    FOREIGN KEY (user_id) REFERENCES students(user_id) ON DELETE CASCADE,
    FOREIGN KEY (food_id) REFERENCES foods(food_id) ON DELETE CASCADE,
    FOREIGN KEY (dining_hall_id) REFERENCES dining_halls(dining_hall_id)
ON DELETE CASCADE
  );

CREATE TABLE schedules
  (
    food_id        INT,
    dining_hall_id INT,
    days           VARCHAR(100),
    meal_type      VARCHAR(100),
    PRIMARY KEY (food_id, dining_hall_id),
    FOREIGN KEY (food_id) REFERENCES foods(food_id) ON DELETE CASCADE,
    FOREIGN KEY (dining_hall_id) REFERENCES dining_halls(dining_hall_id)
ON DELETE CASCADE
  );

CREATE TABLE food_allergens
  (
    food_id        INT,
    allergen       VARCHAR(100),
    PRIMARY KEY (food_id, allergen),
    FOREIGN KEY (food_id) REFERENCES foods(food_id) ON DELETE CASCADE
  );
```

Implemented Tables & Number of Entries:

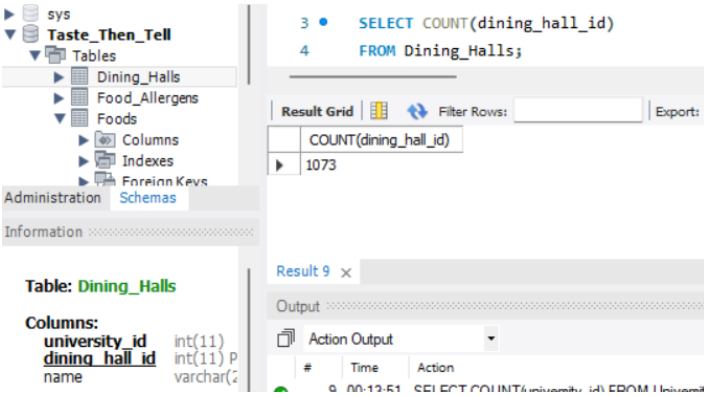
Foods Table: 2911 Entries



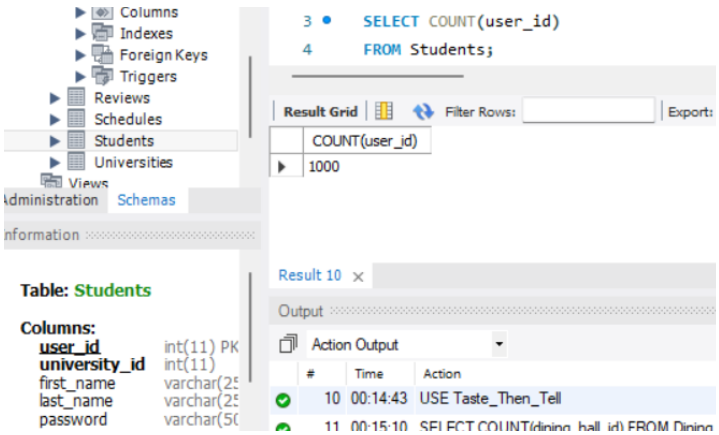
Universities Table: 1947 Entries



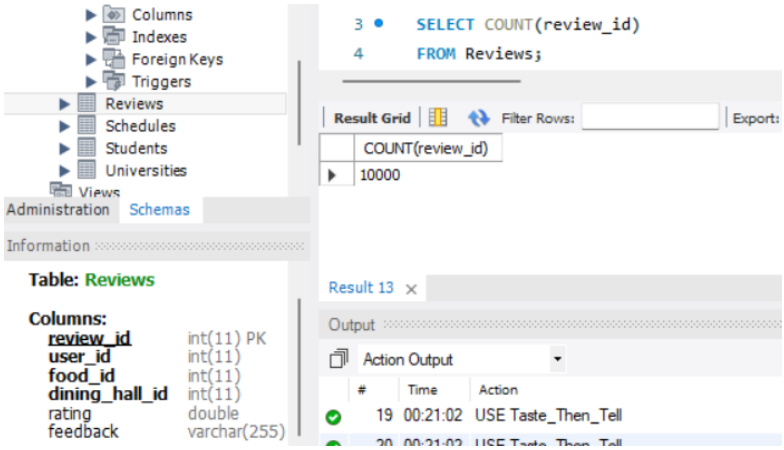
Dining Halls Table: 1073 Entries



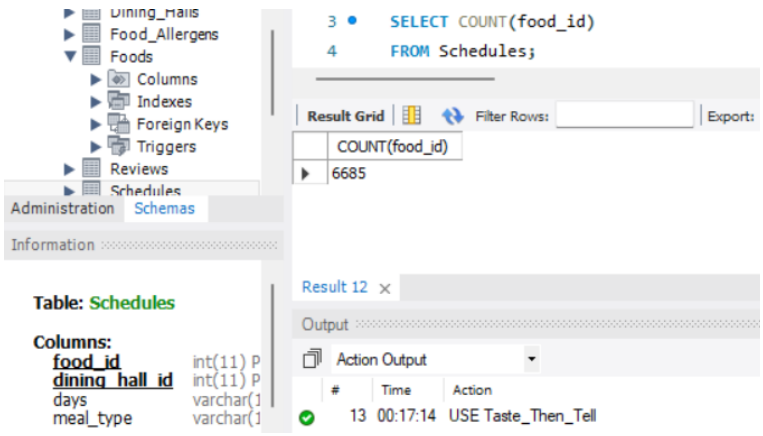
Students Table: 1000 Entries



Reviews Table: 10000 Entries



Schedules Table: 6685 Entries



Advanced Queries & Analysis

Query #1

-- Find the average rating of every dining hall. List the university name, the dining hall name, and the average rating. Order by average rating in descending order.
-- Advanced query 1 w/ Join of multiple relations + Aggregation via GROUP BY

```
SELECT u.name, dh.name, AVG(r.rating) as avgRating
FROM Reviews r NATURAL JOIN Dining_Halls dh JOIN Universities u
USING(university_id)
GROUP BY dh.dining_hall_id
ORDER BY avgRating DESC LIMIT 15;
```

name	name	avgRating
Hanover College	Lidia Matos Hall	4.465110983059095
University of Nebraska at Kearney	Cooper Coleman Hall	4.086244360795601
Pacific Lutheran University	Arlette Koenig Hall	4.008525764865523
Boricua College	Noa Hewitt Hall	3.9373565731828033
Red Rocks Community College	Sloan Cummings Hall	3.879934037763455
Tennessee College of Applied Technolog...	Lainey Thurman Hall	3.8265717865674573
United States Military Academy West Point	Isabella Martinez Hall	3.721968067220062
Midwestern State University	Sharon Hendrix Hall	3.706019007106755
Minnesota State University Moorhead	Carly Rushing Hall	3.6830340058724773
Landmark College	Jaysen Hargrove Hall	3.682456480285972
Alma College	Kymani Peters Hall	3.6781432195749466
SUNY Westchester Community College	Jullian Odom Hall	3.6526825577699427
DePaul University	Bently Darby Hall	3.6384836011430823
Keiser University at Lakeland	Kyrie Langford Hall	3.636742078777103
Concord University	Yarely Whitney Hall	3.6330499647064394

EXPLAIN:

```
-> Limit: 15 row(s)  (actual time=18.199..18.203 rows=15 loops=1)
  -> Sort: <temporary>.avgRating DESC, limit input to 15 row(s) per chunk  (actual time=18.198..18.200 rows=15 loops=1)
    -> Stream results  (actual time=0.136..17.909 rows=1071 loops=1)
      -> Group aggregate: avg(r.rating)  (actual time=0.135..17.788 rows=1071 loops=1)
```

CREATE INDEX rating_index on Reviews(rating)

EXPLAIN:

```
-> Limit: 15 row(s)  (actual time=18.786..18.790 rows=15 loops=1)
  -> Sort: <temporary>.avgRating DESC, limit input to 15 row(s) per chunk  (actual time=18.785..18.787 rows=15 loops=1)
    -> Stream results  (actual time=0.110..18.459 rows=1071 loops=1)
      -> Group aggregate: avg(r.rating)  (actual time=0.110..18.350 rows=1071 loops=1)
```

For our first index that we created for this query, we chose to index on the rating column from our Reviews table, since this is utilized in our SELECT segment. We also chose this because it was not one of the primary keys of the Reviews table, meaning there was no default index for this column already. However, as we can see in the time performance of the summary after creating the index and running the query again, the performance actually worsened slightly. If we were to infer why this index does not benefit our performance much, it would likely be because one reason why indexes are generally useful is because they can improve performance in columns that are accessed frequently and are not already primary keys. However, for our query, although we utilize the ratings column in our SELECT statement, we do not access it very frequently within our query because we only need it once to return the average of the column entries, which is aided by the GROUP BY function, so creating an index for it does not really take advantage of the index's potential benefits.

CREATE INDEX university_name on Universities(name(10))

EXPLAIN:

```
-> Limit: 15 row(s)  (actual time=18.367..18.370 rows=15 loops=1)
  -> Sort: <temporary>.avgRating DESC, limit input to 15 row(s) per chunk  (actual time=18.366..18.368 rows=15 loops=1)
    -> Stream results  (actual time=0.092..18.072 rows=1071 loops=1)
      -> Group aggregate: avg(r.rating)  (actual time=0.092..17.946 rows=1071 loops=1)
```

For our second index, we created one for the first ten characters of entries within the name column of the Universities table. We chose this because we return university names as a part of our SELECT statement, but the names are not a primary key of the Universities table. However, this does not improve the performance of the program. This is likely because within the query itself, the tables that we are joining use the university_id, which is the primary key of the universities table, but other than that the university table entries are not really accessed much other than to return the name of the university at the end, when finally returning the columns in the SELECT statement. Furthermore, the limit on the first ten characters of the University name likely should not make a large difference because again, the University names aren't really being accessed much, and there's no case in this query where using the first ten characters makes a difference from using the entire university name, since there's no comparison happening with the name and we also need to return the entire name entry regardless in the final output of entries.

CREATE INDEX dh_name_idx on Dining_Halls(name(10))

EXPLAIN:

```
-> Limit: 15 row(s)  (actual time=19.345..19.349 rows=15 loops=1)
  -> Sort: <temporary>.avgRating DESC, limit input to 15 row(s) per chunk  (actual time=19.344..19.347 rows=15 loops=1)
    -> Stream results  (actual time=0.077..19.004 rows=1071 loops=1)
      -> Group aggregate: avg(r.rating)  (actual time=0.077..18.895 rows=1071 loops=1)
```

For our last index, we chose to index on the first ten characters of entries within the name column of the Dining_Halls table. This followed very similar reasoning to what we had for creating the second index we tried, as we were going based on the reasoning of using

non-primary keys that are returned in our SELECT statement. However, for similar reasons to the aforementioned index design, this index did not help improve our performance either. The names of Dining_Halls is not accessed other than to return at the very end, which means that it is not being accessed very frequently, which defeats that potential purpose of using an index on it. Also, it is not being used for comparisons either, so limiting it to the first ten characters does not do much for the logic of the query, as it also needs to return the full name anyway for the final output.

Query #2

-- Count the number of food items that each [dining hall] has that don't contain [allergen].
-- Advanced query 2 w/ Subquery + Aggregation via GROUP BY

```
SELECT name, COUNT(DISTINCT food_id) as nonAllergenCount
FROM Schedules s NATURAL JOIN Food_Allergens f NATURAL JOIN Dining_Halls d
WHERE s.food_id NOT IN (
        SELECT DISTINCT food_id
        FROM Food_Allergens f1
        WHERE f1.allergen LIKE "%wheat%")
GROUP BY d.dining_hall_id
ORDER BY nonAllergenCount DESC LIMIT 15;
```

name	nonAllergenCount
Ikenberry Dining Center (IKE Dining Center)	472
Lincoln Avenue Dining Hall (LAR Dining Hall)	410
Pennsylvania Avenue Dining Hall (PAR Dining Hall)	408
Illinois Street Dining Center (ISR Dining Center)	349
Florida Avenue Dining Hall (FAR Dining Hall)	283
Ccrb Mdining To Go	23
Rocco Mccord Hall	23
Forest Thorne Hall	23
Kadence Novak Hall	23
Forest Thorne Hall	21
Bridget Cuevas Hall	21
Arabella Morgan Hall	21
Craig Landers Hall	21
Lisa Gagnon Hall	21
German Arellano Hall	21

EXPLAIN:

```
-> Limit: 15 row(s)  (actual time=129.269..129.271 rows=15 loops=1)
  -> Sort: <temporary>.nonAllergenCount DESC, limit input to 15 row(s) per chunk  (actual time=129.268..129.269 rows=15 loops=1)
    -> Stream results  (actual time=3.405..128.959 rows=1071 loops=1)
      -> Group aggregate: count(distinct s.food_id)  (actual time=3.405..128.826 rows=1071 loops=1)
```

CREATE INDEX allergen_idx on Food_Allergens(allergen(20))

EXPLAIN:

```
-> Limit: 15 row(s)  (actual time=131.407..131.418 rows=15 loops=1)
  -> Sort: <temporary>.nonAllergenCount DESC, limit input to 15 row(s) per chunk  (actual time=131.406..131.416 rows=15 loops=1)
    -> Stream results  (actual time=3.558..131.068 rows=1071 loops=1)
      -> Group aggregate: count(distinct s.food_id)  (actual time=3.557..130.949 rows=1071 loops=1)
```

For this indexing method, we chose to index on the allergen column in the Food_Allergens table. This is because we use allergen in the subquery to filter by a specific allergen. However, we found that this index had no positive effect on performance. Instead, it caused the query to

be 2 seconds slower. We know that indexing on allergen has no effect on the joining of the tables because that is using food_id as the join parameter. We did expect the WHERE clause of the subquery to be slightly faster because the parameter was indexed but because the outer query still has to check every entry in the joined table it makes sense that the query time wasn't improved. We limited it to the first 20 characters of the allergen entries because within the first 20 characters alone, you should be able to tell whether allergens are the same or not since there are very few (if any) allergens that contain another allergen as a substring, but none of them would be 20 characters or longer, but regardless, this did not make a difference.

CREATE INDEX dh_name_idx on Dining_Halls(name(20))

EXPLAIN:

```
-> Limit: 15 row(s)  (actual time=222.425..222.429 rows=15 loops=1)
    -> Sort: <temporary>.nonAllergenCount DESC, limit input to 15 row(s) per chunk  (actual time=222.424..222.426 rows=15 loops=1)
        -> Stream results  (actual time=6.856..221.853 rows=1071 loops=1)
            -> Group aggregate: count(distinct s.food_id)  (actual time=6.855..221.698 rows=1071 loops=1)
```

For this, we chose to index on the name column in Dining_Halls since the dining hall name is used in our SELECT argument and it is not a primary key within the Dining_Halls table. Unfortunately, this index did not benefit performance and in fact, slightly increased processing time. This is likely because each Dining Hall must be "visited" by the program the same number of times with or without the index. The dining hall name is not used as any sort of filter and instead is just needed once at the end for the SELECT argument. We limited it to the first 20 characters but that does not affect the logic of the index or the performance of the query because we never use it for comparisons, nor do we return only the first set amount of characters of names for any reason in our final output.

CREATE INDEX meal_type_index on Schedules(meal_type(20))

EXPLAIN:

```
-> Limit: 15 row(s)  (actual time=131.824..131.827 rows=15 loops=1)
    -> Sort: <temporary>.nonAllergenCount DESC, limit input to 15 row(s) per chunk  (actual time=131.823..131.825 rows=15 loops=1)
        -> Stream results  (actual time=3.674..131.507 rows=1071 loops=1)
            -> Group aggregate: count(distinct s.food_id)  (actual time=3.674..131.398 rows=1071 loops=1)
```

For this index, we chose to index on the meal_type column in the Schedules table. Unsurprisingly, this index had virtually no effect on the runtime performance of the complex query. The Schedules table is necessary to get the "food_id" of each food so that it can be compared against the Allergen table. Since "meal_type" doesn't even appear in the query, the index shouldn't make any difference. As for why we didn't index on the "food_id" column instead, that is because the "food_id" column is already indexed as it is part of the primary key for the Schedules table and so we similarly expect that to have no effect. We limited the index to the first 20 characters of the meal type because within the first 20 characters you should know if the selected meal type is breakfast, lunch, dinner, etc. However, this does not make a difference here because we don't use the meal types in our query operations, nor do we have a need to frequently access them in a manner where the first 20 characters make a difference in performance.