Database Implementation

*the database we created is called “database43” and contains our 9 tables”

```
Welcome to Cloud Shell! Type "help" to get started.
Your Cloud Platform project in this session is set to team43-343623.
Use “gcloud config set project [PROJECT_ID]” to change to a different project.
gcloud sql connect proj-411 --user=root --quietashna_arya@cloudshell:~ (team43-343623)$ gcloud sql connect proj-411 --user=root
Allowlisting your IP for incoming connection for 5 minutes...done.
Connecting to database with SQL user [root].Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 11861
Server version: 8.0.26-google (Google)

Copyright (c) 2000, 2022, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| database43         |
| information_schema |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
5 rows in set (0.00 sec)

mysql>
```

Implementing the Database Locally/on GCP

```
mysql> use database43
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> show tables
    -> \c
mysql> show tables;
+----------------------+
| Tables_in_database43 |
+----------------------+
| Achieves             |
| ConsumeItem          |
| Customers            |
| Exercises            |
| FoodItems            |
| HealthGoals          |
| Performs             |
| RestaurantOrder      |
| Restaurants          |
+----------------------+
9 rows in set (0.01 sec)

mysql>
```

(Showing at least 1000 rows in the Tables)

```
mysql> select count(Goal) from HealthGoals;
+-------------+
| count(Goal) |
+-------------+
|        1000 |
+-------------+
1 row in set (0.02 sec)

mysql> select count(CustomerID) from Performs;
+--------------------+
| count(CustomerID)  |
+--------------------+
|               1000 |
+--------------------+
1 row in set (0.01 sec)

mysql> select count(CustomerID) from RestaurantOrder;
+--------------------+
| count(CustomerID)  |
+--------------------+
|               1000 |
+--------------------+
1 row in set (0.01 sec)

mysql> select count(RestaurantID) from Restaurants;
+----------------------+
| count(RestaurantID)  |
+----------------------+
|                 1000 |
+----------------------+
1 row in set (0.01 sec)
```

```
mysql> select count(CustomerID) from Customers;
+-------------------+
| count(CustomerID) |
+-------------------+
|              1000 |
+-------------------+
1 row in set (0.01 sec)

mysql> select count(CustomerID) from Achieves;
+-------------------+
| count(CustomerID) |
+-------------------+
|              1000 |
+-------------------+
1 row in set (0.00 sec)

mysql> select count(CustomerID) from ConsumeItem;
+-------------------+
| count(CustomerID) |
+-------------------+
|              1000 |
+-------------------+
1 row in set (0.01 sec)

mysql> select count(ExerciseName) from Exercises;
+---------------------+
| count(ExerciseName) |
+---------------------+
|                1000 |
+---------------------+
1 row in set (0.01 sec)

mysql> select count(ItemName) from FoodItems;
+-----------------+
| count(ItemName) |
+-----------------+
|            1000 |
+-----------------+
```

DDL Commands

*only 1 sample “INSERT INTO” lines for each table is provided

Customer Table:

```
CREATE DATABASE IF NOT EXISTS database43;
use database43;

DROP TABLE IF EXISTS Customers;
CREATE TABLE Customers(
   CustomerID INTEGER  NOT NULL PRIMARY KEY
  ,Name       VARCHAR(255) NOT NULL
  ,Age        INTEGER  NOT NULL
  ,Weight     INTEGER  NOT NULL
  ,Height     INTEGER  NOT NULL
);
INSERT INTO Customers(CustomerID,Name,Age,Weight,Height) VALUES (0,'Alvin
Taylor',22,81,66);
```

Restaurant Table:

```
CREATE DATABASE IF NOT EXISTS database43;
use database43;

DROP TABLE IF EXISTS Restaurants;
CREATE TABLE Restaurants(
   RestaurantID   INTEGER  NOT NULL PRIMARY KEY
  ,RestaurantName VARCHAR(255) NOT NULL
  ,Address        VARCHAR(255) NOT NULL
);
INSERT INTO Restaurants(RestaurantID,RestaurantName,Address) VALUES (0,'Taco
Company','0 S. Green Street');
```

FoodItem Table:

```
CREATE DATABASE IF NOT EXISTS database43;
use database43;

DROP TABLE IF EXISTS FoodItems;
CREATE TABLE FoodItems(
   ItemName VARCHAR(255) NOT NULL PRIMARY KEY
```

```
  ,Calories INTEGER  NOT NULL
  ,Carbs    INTEGER  NOT NULL
  ,Protein  INTEGER  NOT NULL
  ,Fat      INTEGER  NOT NULL
);
INSERT INTO FoodItems(ItemName,Calories,Carbs,Protein,Fat) VALUES ('red tofu
(baked)',273,70,34,11);
```

Exercises Table:

```
CREATE DATABASE IF NOT EXISTS database43;
use database43;

DROP TABLE IF EXISTS Exercises;
CREATE TABLE Exercises(
   ExerciseName   VARCHAR(255) NOT NULL PRIMARY KEY
  ,CaloriesBurned INTEGER  NOT NULL
);
INSERT INTO Exercises(ExerciseName,CaloriesBurned) VALUES ('Sitting Leg Tuck',112);
```

HealthGoals Table:

```
CREATE DATABASE IF NOT EXISTS database43;
use database43;

DROP TABLE IF EXISTS HealthGoals;
CREATE TABLE HealthGoals(
   Goal          VARCHAR(255) NOT NULL PRIMARY KEY
  ,CalorieCeiling INTEGER  NOT NULL
  ,CalorieFloor   INTEGER  NOT NULL
  ,TargetWeight   INTEGER  NOT NULL
  ,TargetCarbs    INTEGER  NOT NULL
  ,TargetProtein  INTEGER  NOT NULL
  ,TargetFat      INTEGER  NOT NULL
);
INSERT INTO
HealthGoals(Goal,CalorieCeiling,CalorieFloor,TargetWeight,TargetCarbs,TargetProtein,TargetFa
t) VALUES ('Goal #0',730,473,137,63,65,7);
```

Achieves Table:

```
CREATE DATABASE IF NOT EXISTS database43;
```

```
use database43;

DROP TABLE IF EXISTS Achieves;
CREATE TABLE Achieves(
   CustomerID INTEGER  NOT NULL
  ,Goal       VARCHAR(255) NOT NULL
  ,PRIMARY KEY(CustomerID,Goal)
  ,FOREIGN KEY(CustomerID) REFERENCES Customers(CustomerID)
  ,FOREIGN KEY(Goal) REFERENCES HealthGoals(Goal)
);
INSERT INTO Achieves(CustomerID,Goal) VALUES (0,'Goal #0');
```

RestaurantOrder Table:

```
CREATE DATABASE IF NOT EXISTS database43;
use database43;

DROP TABLE IF EXISTS RestaurantOrder;
CREATE TABLE RestaurantOrder(
   CustomerID   INTEGER  NOT NULL
  ,RestaurantID INTEGER  NOT NULL
  ,PRIMARY KEY(CustomerID,RestaurantID)
  ,FOREIGN KEY(CustomerID) REFERENCES Customers(CustomerID)
  ,FOREIGN KEY(RestaurantID) REFERENCES Restaurants(RestaurantID)
);
INSERT INTO RestaurantOrder(CustomerID,RestaurantID) VALUES (0,0);
```

ConsumeItem Table:

```
CREATE DATABASE IF NOT EXISTS database43;
use database43;

DROP TABLE IF EXISTS ConsumeItem;
CREATE TABLE ConsumeItem(
   CustomerID INTEGER  NOT NULL
  ,ItemName   VARCHAR(255) NOT NULL
  ,PRIMARY KEY(CustomerID,ItemName)
  ,FOREIGN KEY(CustomerID) REFERENCES Customers(CustomerID)
  ,FOREIGN KEY(ItemName) REFERENCES FoodItems(ItemName)
);
INSERT INTO ConsumeItem(CustomerID,ItemName) VALUES (0,'red tofu (baked)');
```

Performs Table:

```
CREATE DATABASE IF NOT EXISTS database43;
use database43;

DROP TABLE IF EXISTS Performs;
CREATE TABLE Performs(
   CustomerID   INTEGER  NOT NULL
  ,ExerciseName VARCHAR(255) NOT NULL
  ,PRIMARY KEY(CustomerID,ExerciseName)
  ,FOREIGN KEY(CustomerID) REFERENCES Customers(CustomerID)
  ,FOREIGN KEY(ExerciseName) REFERENCES Exercises(ExerciseName)
);
INSERT INTO Performs(CustomerID,ExerciseName) VALUES (0,'Sitting Leg Tuck');
```

Queries

QUERY 1

/*finds exercises that have been performed that burn <100 or >400 calories*/
SELECT e.ExerciseName, p.CustomerID FROM Exercises e JOIN Performs p USING(ExerciseName) WHERE e.CaloriesBurned < 100 UNION SELECT e.ExerciseName, p.CustomerID FROM Exercises e JOIN Performs p USING(ExerciseName) WHERE e.CaloriesBurned > 400 ORDER BY CustomerID ASC;

```
mysql> SELECT e.ExerciseName, p.CustomerID
Performs p USING(ExerciseName) WHERE e.Cal
+-----------------------+------------+
| ExerciseName          | CustomerID |
+-----------------------+------------+
| Lifting Leg Tuck      |          2 |
| Lifting Back Tuck     |         12 |
| Pressing Back Tuck    |         17 |
| Sitting Chest Tuck    |         20 |
| Jumping Chest Tuck    |         23 |
| Lifting Shoulder Tuck |         32 |
| Jumping Shoulder Tuck |         33 |
| Rowing Shoulder Tuck  |         34 |
| Pressing Shoulder Tuck|         37 |
| Sitting Delt Tuck     |         40 |
| Squatting Delt Tuck   |         45 |
| Running Delt Tuck     |         48 |
| Lunging Delt Tuck     |         49 |
| Sitting Ab Tuck       |         50 |
| Squatting Ab Tuck     |         55 |
+-----------------------+------------+
15 rows in set (0.01 sec)
```

QUERY 2

/*Finds all food items that contain either tofu or chicken**/**
SELECT c.CustomerID, c.Name, i.ItemName FROM Customers c JOIN ConsumeItem i USING(CustomerID) WHERE i.ItemName LIKE '%tofu%' UNION SELECT c1.CustomerID, c1.Name, i1.ItemName FROM Customers c1 JOIN ConsumeItem i1 USING(CustomerID) WHERE i1.ItemName LIKE '%chicken%' ORDER BY ItemName DESC;

```
+------------+------------------+-------------------------+
| CustomerID | Name             | ItemName                |
+------------+------------------+-------------------------+
|        801 | Seth Bird        | yellow tofu (tossed)    |
|        901 | Astra Baker      | yellow tofu (torched)   |
|        201 | Zeph Norris      | yellow tofu (seared)    |
|        601 | Judith Mckinney  | yellow tofu (sauteed)   |
|        701 | Liberty Evans    | yellow tofu (raw)       |
|        401 | Holmes Cortez    | yellow tofu (fried)     |
|        501 | Merrill Graves   | yellow tofu (broiled)   |
|        101 | Kylee Bird       | yellow tofu (boiled)    |
|        301 | Kimberley Mathews| yellow tofu (blanched)  |
|          1 | Hedda Atkins     | yellow tofu (baked)     |
|        821 | Carla Pate       | yellow chicken (tossed) |
|        921 | Maris Lindsey    | yellow chicken (torched)|
|        221 | Nichole Moon     | yellow chicken (seared) |
|        621 | Julian Brock     | yellow chicken (sauteed)|
|        721 | Colorado Cabrera | yellow chicken (raw)    |
+------------+------------------+-------------------------+
15 rows in set (0.00 sec)
```

Indexing

QUERY 1:

Before:

```
| -> Sort: CustomerID  (cost=2.50 rows=0) (actual time=0.114..0.130 rows=306 loops=1)
    -> Table scan on <union temporary>  (cost=2.50 rows=0) (actual time=0.001..0.021 rows=306 loops=1)
        -> Union materialize with deduplication  (cost=502.47..504.97 rows=667) (actual time=2.178..2.212 rows=306 loops=1)
            -> Nested loop inner join  (cost=217.90 rows=333) (actual time=0.064..0.750 rows=106 loops=1)
                -> Filter: (e.CaloriesBurned < 100)  (cost=101.25 rows=333) (actual time=0.044..0.316 rows=106 loops=1)
                    -> Table scan on e  (cost=101.25 rows=1000) (actual time=0.039..0.251 rows=1000 loops=1)
                -> Filter: (e.ExerciseName = p.ExerciseName)  (cost=0.25 rows=1) (actual time=0.003..0.004 rows=1 loops=106)
                    -> Index lookup on p using ExerciseName (ExerciseName=e.ExerciseName)  (cost=0.25 rows=1) (actual time=0.003..0.003 rows=1 loops=106)
            -> Nested loop inner join  (cost=217.90 rows=333) (actual time=0.024..1.101 rows=200 loops=1)
                -> Filter: (e.CaloriesBurned > 400)  (cost=101.25 rows=333) (actual time=0.018..0.320 rows=200 loops=1)
                    -> Table scan on e  (cost=101.25 rows=1000) (actual time=0.017..0.236 rows=1000 loops=1)
                -> Filter: (e.ExerciseName = p.ExerciseName)  (cost=0.25 rows=1) (actual time=0.003..0.004 rows=1 loops=200)
                    -> Index lookup on p using ExerciseName (ExerciseName=e.ExerciseName)  (cost=0.25 rows=1) (actual time=0.002..0.003 rows=1 loops=200)
|
```

After: create index caloriesBurned_idx on Exercises(CaloriesBurned);

```
| -> Sort: CustomerID  (cost=2.50 rows=0) (actual time=0.115..0.138 rows=306 loops=1)
    -> Table scan on <union temporary>  (cost=2.50 rows=0) (actual time=0.001..0.022 rows=306 loops=1)
        -> Union materialize with deduplication  (cost=202.97..205.46 rows=306) (actual time=1.689..1.730 rows=306 loops=1)
            -> Nested loop inner join  (cost=59.90 rows=106) (actual time=0.066..0.554 rows=106 loops=1)
                -> Filter: (e.CaloriesBurned < 100)  (cost=22.80 rows=106) (actual time=0.045..0.073 rows=106 loops=1)
                    -> Index range scan on e using caloriesBurned_idx  (cost=22.80 rows=106) (actual time=0.042..0.059 rows=106 loops=1)
                -> Filter: (e.ExerciseName = p.ExerciseName)  (cost=0.25 rows=1) (actual time=0.004..0.004 rows=1 loops=106)
                    -> Index lookup on p using ExerciseName (ExerciseName=e.ExerciseName)  (cost=0.25 rows=1) (actual time=0.003..0.004 rows=1 loops=106)
            -> Nested loop inner join  (cost=112.46 rows=200) (actual time=0.030..0.858 rows=200 loops=1)
                -> Filter: (e.CaloriesBurned > 400)  (cost=42.46 rows=200) (actual time=0.024..0.086 rows=200 loops=1)
                    -> Index range scan on e using caloriesBurned_idx  (cost=42.46 rows=200) (actual time=0.024..0.065 rows=200 loops=1)
                -> Filter: (e.ExerciseName = p.ExerciseName)  (cost=0.25 rows=1) (actual time=0.003..0.004 rows=1 loops=200)
                    -> Index lookup on p using ExerciseName (ExerciseName=e.ExerciseName)  (cost=0.25 rows=1) (actual time=0.002..0.003 rows=1 loops=200)
|
```

After: create index excerciseName_idx on Exercises(ExerciseName);

```
| -> Sort: CustomerID  (cost=2.50 rows=0) (actual time=0.188..0.204 rows=306 loops=1)
    -> Table scan on <union temporary>  (cost=2.50 rows=0) (actual time=0.001..0.023 rows=306 loops=1)
        -> Union materialize with deduplication  (cost=502.47..504.97 rows=667) (actual time=5.596..5.629 rows=306 loops=1)
            -> Nested loop inner join  (cost=217.90 rows=333) (actual time=0.160..2.336 rows=106 loops=1)
                -> Filter: (e.CaloriesBurned < 100)  (cost=101.25 rows=333) (actual time=0.027..0.351 rows=106 loops=1)
                    -> Table scan on e  (cost=101.25 rows=1000) (actual time=0.024..0.263 rows=1000 loops=1)
                -> Filter: (e.ExerciseName = p.ExerciseName)  (cost=0.25 rows=1) (actual time=0.018..0.018 rows=1 loops=106)
                    -> Index lookup on p using ExerciseName (ExerciseName=e.ExerciseName)  (cost=0.25 rows=1) (actual time=0.017..0.018 rows=1 loops=106)
            -> Nested loop inner join  (cost=217.90 rows=333) (actual time=0.055..2.851 rows=200 loops=1)
                -> Filter: (e.CaloriesBurned > 400)  (cost=101.25 rows=333) (actual time=0.036..0.362 rows=200 loops=1)
                    -> Table scan on e  (cost=101.25 rows=1000) (actual time=0.034..0.276 rows=1000 loops=1)
                -> Filter: (e.ExerciseName = p.ExerciseName)  (cost=0.25 rows=1) (actual time=0.011..0.012 rows=1 loops=200)
                    -> Index lookup on p using ExerciseName (ExerciseName=e.ExerciseName)  (cost=0.25 rows=1) (actual time=0.011..0.012 rows=1 loops=200)
|
```

After: create index customerID_idx on Performs(CustomerID);

```
| -> Sort: CustomerID  (cost=2.50 rows=0) (actual time=0.111..0.126 rows=306 loops=1)
    -> Table scan on <union temporary>  (cost=2.50 rows=0) (actual time=0.001..0.021 rows=306 loops=1)
        -> Union materialize with deduplication  (cost=502.47..504.97 rows=667) (actual time=2.096..2.129 rows=306 loops=1)
            -> Nested loop inner join  (cost=217.90 rows=333) (actual time=0.055..0.779 rows=106 loops=1)
                -> Filter: (e.CaloriesBurned < 100)  (cost=101.25 rows=333) (actual time=0.034..0.305 rows=106 loops=1)
                    -> Table scan on e  (cost=101.25 rows=1000) (actual time=0.030..0.238 rows=1000 loops=1)
                -> Filter: (e.ExerciseName = p.ExerciseName)  (cost=0.25 rows=1) (actual time=0.004..0.004 rows=1 loops=106)
                    -> Index lookup on p using ExerciseName (ExerciseName=e.ExerciseName)  (cost=0.25 rows=1) (actual time=0.003..0.004 rows=1 loops=106)
            -> Nested loop inner join  (cost=217.90 rows=333) (actual time=0.026..1.051 rows=200 loops=1)
                -> Filter: (e.CaloriesBurned > 400)  (cost=101.25 rows=333) (actual time=0.020..0.293 rows=200 loops=1)
                    -> Table scan on e  (cost=101.25 rows=1000) (actual time=0.019..0.217 rows=1000 loops=1)
                -> Filter: (e.ExerciseName = p.ExerciseName)  (cost=0.25 rows=1) (actual time=0.003..0.004 rows=1 loops=200)
                    -> Index lookup on p using ExerciseName (ExerciseName=e.ExerciseName)  (cost=0.25 rows=1) (actual time=0.002..0.003 rows=1 loops=200)
|
```

For our first query, we tried three different indexing designs on three different attributes → CaloriesBurned, ExerciseName, and CustomerID. Following our analysis, we determined that the indexing design that produced the optimized cost was on CaloriesBurned. Before any indexing, the cost for the nested loop inner join was 217.90 and rows read was 667. After the CaloriesBurned Index, the cost of the same nested loop inner join was dramatically cut down to 59.9 and rows read was 106. It also decreased the cost of the union materialization from 502.47 to 202.97. We chose this field as an index originally because we use CaloriesBurned in the

WHERE clause to filter out the exercises where the calories burned was more than 400 and the other exercises less than 100. The stats showed that this index did in fact overall reduce the cost, probably due to the fact that this attribute was used in the query filtering logic and was not indeed a primary key, therefore benefiting more from indexing. The second index that we tried was on ExerciseName from the Exercise table. We chose this index because we used ExerciseName to join the Exercise table and the Performs table. After seeing the stats with this newly created index (and dropping the previous index), we saw that the performance was actually the same as the query without any custom indexing (default indexing on the primary key). And finally, the third index that we tried was CustomerID from the performs table. Similar to the second index, it did not have a new positive effect on the query execution and runtime as you can see in the stats, the cost numbers are the exact same. This is the case with the last two indices because those specific attributes were not used directly to perform any logic in the query so if we created an index on it, it would not change the performance in any way whatsoever.

QUERY 2:

Before:

```
| -> Table scan on <union temporary>  (cost=0.02..10.82 rows=667) (actual time=0.001..0.026 rows=306 loops=1)
    -> Union materialize with deduplication  (cost=502.49..513.29 rows=667) (actual time=2.411..2.457 rows=306 loops=1)
        -> Nested loop inner join  (cost=217.90 rows=333) (actual time=0.176..0.930 rows=106 loops=1)
            -> Filter: (e.CaloriesBurned < 100)  (cost=101.25 rows=333) (actual time=0.105..0.383 rows=106 loops=1)
                -> Table scan on e  (cost=101.25 rows=1000) (actual time=0.100..0.317 rows=1000 loops=1)
            -> Filter: (e.ExerciseName = p.ExerciseName)  (cost=0.25 rows=1) (actual time=0.004..0.005 rows=1 loops=106)
                -> Index lookup on p using ExerciseName (ExerciseName=e.ExerciseName)  (cost=0.25 rows=1) (actual time=0.003..0.004 rows=1 loops=106)
        -> Nested loop inner join  (cost=217.90 rows=333) (actual time=0.102..1.237 rows=200 loops=1)
            -> Filter: (e.CaloriesBurned > 400)  (cost=101.25 rows=333) (actual time=0.094..0.386 rows=200 loops=1)
                -> Table scan on e  (cost=101.25 rows=1000) (actual time=0.092..0.312 rows=1000 loops=1)
            -> Filter: (e.ExerciseName = p.ExerciseName)  (cost=0.25 rows=1) (actual time=0.003..0.004 rows=1 loops=200)
                -> Index lookup on p using ExerciseName (ExerciseName=e.ExerciseName)  (cost=0.25 rows=1) (actual time=0.003..0.004 rows=1 loops=200)
|
```

After: create index food_idx on on ConsumeItem(ItemName);

```
--+
| -> Sort: ItemName DESC  (cost=2.50 rows=0) (actual time=0.293..0.305 rows=200 loops=1)
    -> Table scan on <union temporary>  (cost=2.50 rows=0) (actual time=0.004..0.020 rows=200 loops=1)
        -> Union materialize with deduplication  (cost=302.50..304.99 rows=222) (actual time=4.913..4.936 rows=200 loops=1)
            -> Nested loop inner join  (cost=140.14 rows=111) (actual time=0.338..2.545 rows=100 loops=1)
                -> Filter: (i.ItemName like '%tofu%')  (cost=101.25 rows=111) (actual time=0.267..0.881 rows=100 loops=1)
                    -> Index scan on i using ItemName  (cost=101.25 rows=1000) (actual time=0.192..0.504 rows=1000 loops=1)
                -> Single-row index lookup on c using PRIMARY (CustomerID=i.CustomerID)  (cost=0.25 rows=1) (actual time=0.016..0.016 rows=1 loops=100)
            -> Nested loop inner join  (cost=140.14 rows=111) (actual time=0.099..1.812 rows=100 loops=1)
                -> Filter: (i1.ItemName like '%chicken%')  (cost=101.25 rows=111) (actual time=0.078..0.599 rows=100 loops=1)
                    -> Index scan on i1 using ItemName  (cost=101.25 rows=1000) (actual time=0.053..0.266 rows=1000 loops=1)
                -> Single-row index lookup on c1 using PRIMARY (CustomerID=i1.CustomerID)  (cost=0.25 rows=1) (actual time=0.012..0.012 rows=1 loops=100)
|
```

After: create index custName_idx on Customers(Name);

```
--+
| -> Sort: ItemName DESC  (cost=2.50 rows=0) (actual time=0.102..0.114 rows=200 loops=1)
    -> Table scan on <union temporary>  (cost=2.50 rows=0) (actual time=0.002..0.018 rows=200 loops=1)
        -> Union materialize with deduplication  (cost=302.50..304.99 rows=222) (actual time=1.691..1.715 rows=200 loops=1)
            -> Nested loop inner join  (cost=140.14 rows=111) (actual time=0.119..0.778 rows=100 loops=1)
                -> Filter: (i.ItemName like '%tofu%')  (cost=101.25 rows=111) (actual time=0.103..0.621 rows=100 loops=1)
                    -> Index scan on i using ItemName  (cost=101.25 rows=1000) (actual time=0.056..0.289 rows=1000 loops=1)
                -> Single-row index lookup on c using PRIMARY (CustomerID=i.CustomerID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=100)
            -> Nested loop inner join  (cost=140.14 rows=111) (actual time=0.057..0.672 rows=100 loops=1)
                -> Filter: (i1.ItemName like '%chicken%')  (cost=101.25 rows=111) (actual time=0.053..0.543 rows=100 loops=1)
                    -> Index scan on i1 using ItemName  (cost=101.25 rows=1000) (actual time=0.031..0.221 rows=1000 loops=1)
                -> Single-row index lookup on c1 using PRIMARY (CustomerID=i1.CustomerID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=100)
|
```

After: create index custID_idx on Customers(CustomerID);

```
--+
| -> Sort: ItemName DESC  (cost=2.50 rows=0) (actual time=0.095..0.116 rows=200 loops=1)
    -> Table scan on <union temporary>  (cost=2.50 rows=0) (actual time=0.002..0.015 rows=200 loops=1)
        -> Union materialize with deduplication  (cost=302.50..304.99 rows=222) (actual time=1.704..1.737 rows=200 loops=1)
            -> Nested loop inner join  (cost=140.14 rows=111) (actual time=0.097..0.776 rows=100 loops=1)
                -> Filter: (i.ItemName like '%tofu%')  (cost=101.25 rows=111) (actual time=0.087..0.598 rows=100 loops=1)
                    -> Index scan on i using ItemName  (cost=101.25 rows=1000) (actual time=0.043..0.241 rows=1000 loops=1)
                -> Single-row index lookup on c using PRIMARY (CustomerID=i.CustomerID)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=100)
            -> Nested loop inner join  (cost=140.14 rows=111) (actual time=0.057..0.697 rows=100 loops=1)
                -> Filter: (i1.ItemName like '%chicken%')  (cost=101.25 rows=111) (actual time=0.053..0.556 rows=100 loops=1)
                    -> Index scan on i1 using ItemName  (cost=101.25 rows=1000) (actual time=0.030..0.213 rows=1000 loops=1)
                -> Single-row index lookup on c1 using PRIMARY (CustomerID=i1.CustomerID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=100)
|
```

For our second query, after trying 3 different indexing designs, we decided that the best two indices were the food items and customerid indices. Before any indexing, the cost for the nested inner loop join was 217.90 and the rows read was 333. After the food item index, it cut down the cost of the same nested inner loop join to 140.14 and the rows read to 111. It also decreased the cost of the union materialization from 502.49 to 302.50. We chose this field as an index originally because we were using the ItemName attribute in the “LIKE” clause to filter out food item names with certain patterns (contain tofu or chicken in this case). After seeing the stats, we saw it did help reduce the cost, as well as reads in this query execution. The second index we tried was on the CustomerID field in the Customers table. We chose this index because we used the CustomerID field to join the Customer and ConsumeItem tables. After viewing the metric with the newly created index, we saw this index had the same effect on query performance as the food item index, with the cost, reads, and time being the same metrics as before. We believe this is because both of these fields we created indices on were primary keys in the two tables of interest. The third index we tried for this query was on the Name field in the Customers table. This did not have any new positive effect on the query execution cost/runtime, as the cost numbers are almost exactly the same as when using solely the customer id or food item index (we ran this index in addition to the customerID at the same time). We believe this is the case because the name field was not directly used to perform any logic/joins in this query, and hence was not expected to improve performance.