CS411 - Team48 - Project Track 1
Stage 3: Database Implementation and Indexing

Data Definition Language (DDL) commands

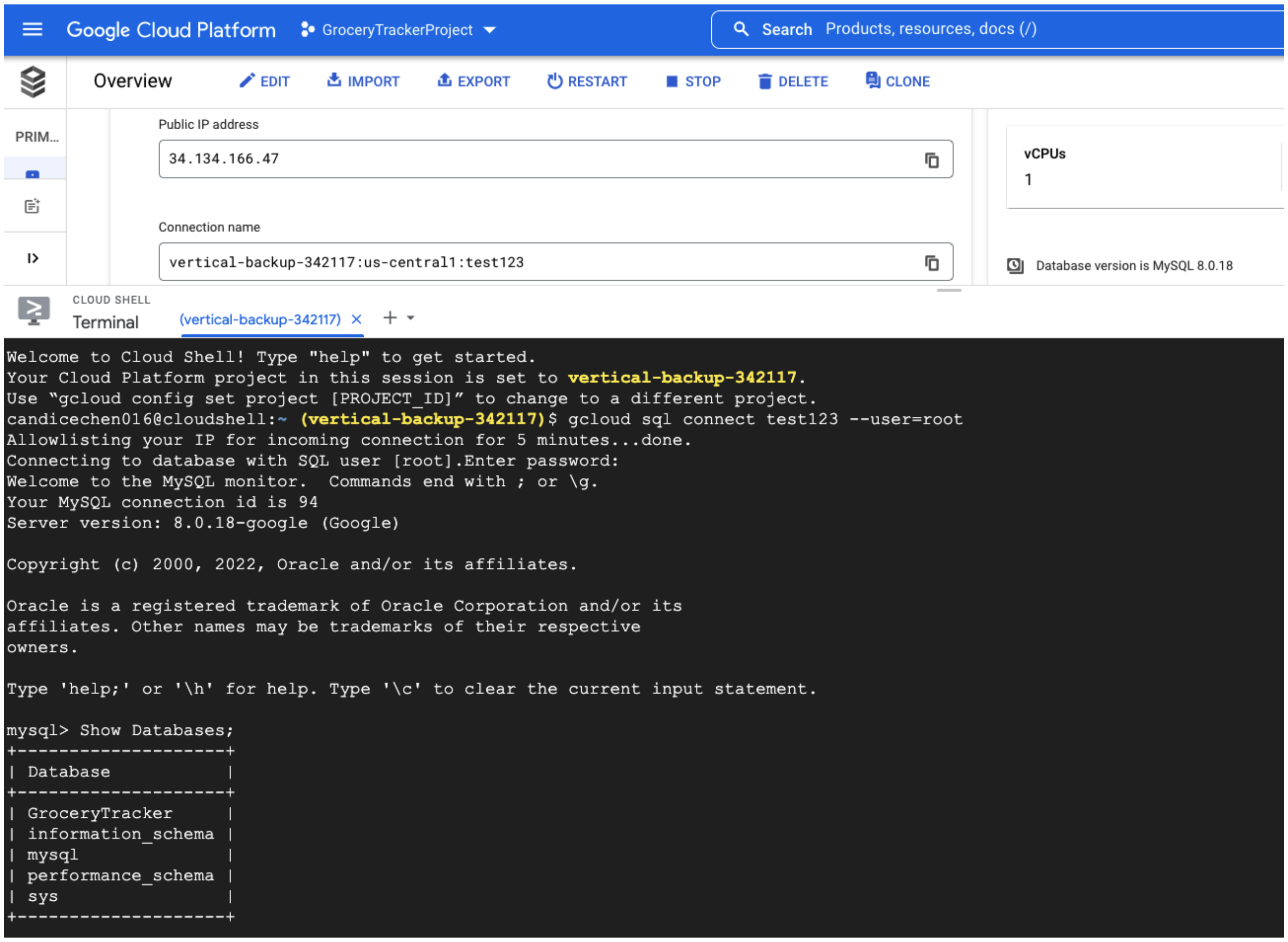
```
CREATE DATABASE GroceryTracker;

CREATE TABLE Customer (
	CustomerID varchar(255) NOT NULL,
	LastName varchar(255),
	FirstName varchar(255),
	Gender char(1),
	Age int,
	Email varchar(255),
	Password varchar(255),
	PRIMARY KEY (CustomerID)
);


CREATE TABLE Product(
	ProductID varchar(255) NOT NULL,
	PLUcode int,
	Name varchar(255),
	Lifespan varchar(255),
	Calories varchar(255),
	Photo varchar(255),
	PRIMARY KEY (ProductID)
	);


CREATE TABLE Purchase(
	PurchaseID varchar(255) NOT NULL,
	StoreID int,
	InventoryID varchar(255),
	PurchaseDate date,
	PRIMARY KEY (PurchaseID),
	FOREIGN KEY (StoreID) REFERENCES Store(StoreID),
	FOREIGN KEY (InventoryID) REFERENCES InventoryList(InventoryID)
);
```

```
CREATE TABLE Store(
	StoreID int NOT NULL,
	StoreName varchar(255) NOT NULL,
	PhoneNumber Varchar(255),
	Address Varchar(255),
	City Varchar(255),
	State Varchar(255),
	PostalCode Varchar(255),
	OpenHours Varchar(255),
	PRIMARY KEY(StoreID)
);


CREATE TABLE Has(
	H_purchaseID Varchar (255) NOT NULL,
	H_productID Varchar (255) NOT NULL,
	Price real,
	Amount real (255),
	Unit varchar (45),
	PRIMARY KEY (H_purchaseID, H_productID),
	FOREIGN KEY (H_purchaseID) REFERENCES Purchase(PurchaseID) on
	delete cascade,
	FOREIGN KEY (H_productID) REFERENCES Product(ProductID) on
	delete cascade
);


CREATE TABLE InventoryList(
	InventoryID Varchar(255) NOT NULL,
	CustomerID Varchar(255) NOT NULL,
	ProductID Varchar(255) NOT NULL,
	ExpirationDate date,
	StorageSpace varchar(255),
	Amount REAL,
	Unit varchar (45),
	PRIMARY KEY (InventoryID),
	FOREIGN KEY (CustomerID) REFERENCES Customer(CustomerID) on
	delete cascade,
	FOREIGN KEY (ProductID) REFERENCES Product(ProductID) on delete
	cascade
);
```

```
CREATE TABLE Shoppinglist(
	ShoppingID varchar(255) NOT NULL,
	CustomerID varchar(255) NOT NULL,
	ListName varchar(255),
	PRIMARY KEY(ShoppingID, CustomerID),
	FOREIGN KEY (CustomerID) REFERENCES Customer(CustomerID) on
	delete cascade,
	);


CREATE TABLE include(
	ShoppingID VARCHAR(255),
	ProductID VARCHAR(255),
	Amount real,
	Unit VARCHAR(255),
	PRIMARY KEY (ShoppingID, ProductID),
	FOREIGN KEY (ShoppingID) REFERENCES Shoppinglist(ShoppingID),
	FOREIGN KEY (ProductID) REFERENCES Product(ProductID)
);


CREATE TABLE In_List (
	StoreID INT,
	ShoppingID VARCHAR(255),
	PRIMARY KEY (StoreID, ShoppingID),
	FOREIGN KEY (StoreID) REFERENCES Store (StoreID),
);
```

Screenshots for Database Connection

We implemented the database on MySQL@GCP.



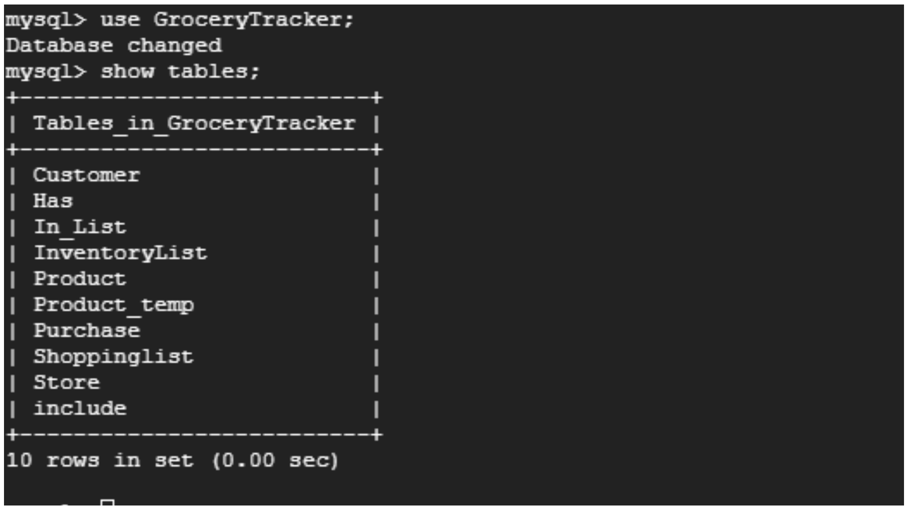
Public IP address
34.134.166.47

Connection name
vertical-backup-342117:us-central1:test123

vCPUs
1

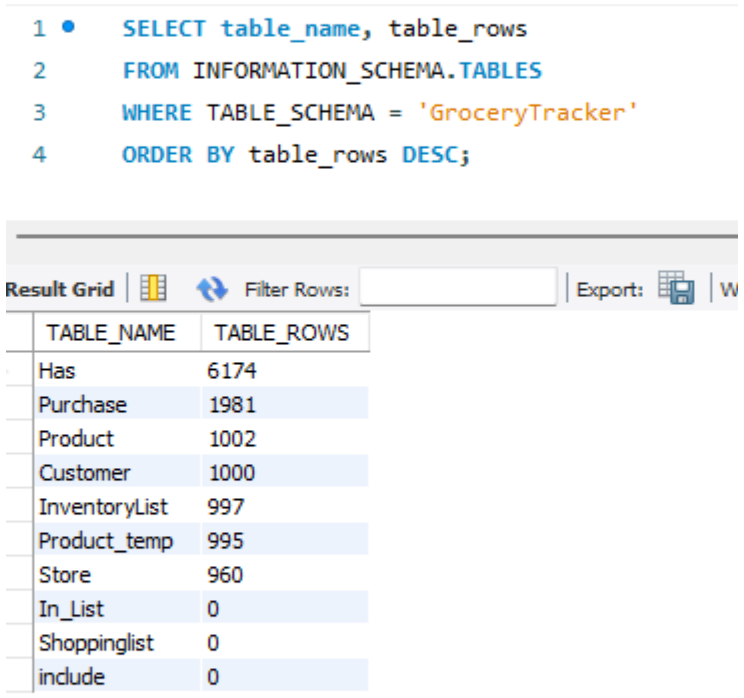
Database version is MySQL 8.0.18

```
Welcome to Cloud Shell! Type "help" to get started.
Your Cloud Platform project in this session is set to vertical-backup-342117.
Use "gcloud config set project [PROJECT_ID]" to change to a different project.
candicechen016@cloudshell:~ (vertical-backup-342117)$ gcloud sql connect test123 --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 94
Server version: 8.0.18-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           |
+--------------------+
| GroceryTracker     |
| information_schema |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
```



```
mysql> use GroceryTracker;
Database changed
mysql> show tables;
+--------------------------+
| Tables_in_GroceryTracker |
+--------------------------+
| Customer                 |
| Has                      |
| In_List                  |
| InventoryList            |
| Product                  |
| Product_temp             |
| Purchase                 |
| Shoppinglist             |
| Store                    |
| include                  |
+--------------------------+
10 rows in set (0.00 sec)
```

Data Screenshots for the number of rows in Each Table



```
SELECT table_name, table_rows
FROM INFORMATION_SCHEMA.TABLES
WHERE TABLE_SCHEMA = 'GroceryTracker'
ORDER BY table_rows DESC;
```

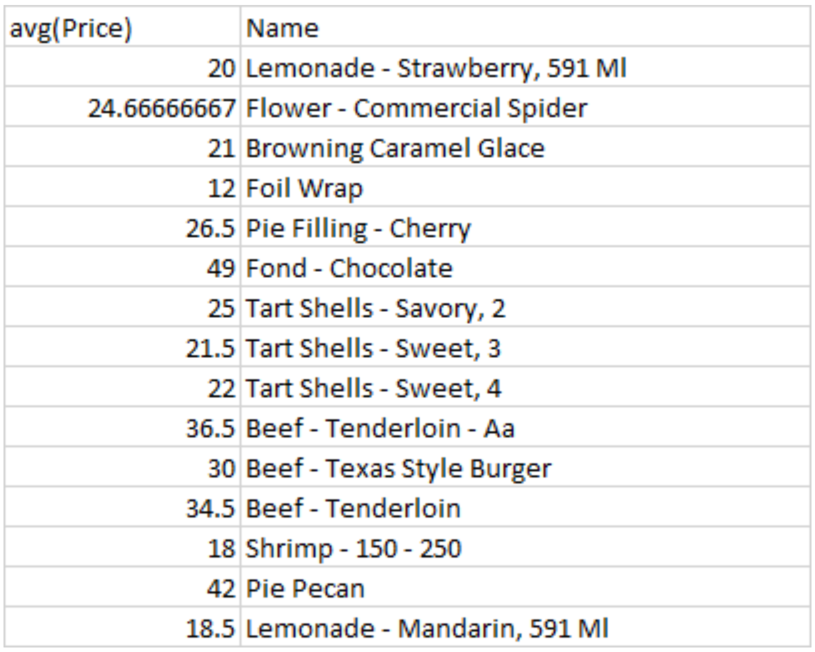
TABLE_NAME	TABLE_ROWS
Has	6174
Purchase	1981
Product	1002
Customer	1000
InventoryList	997
Product_temp	995
Store	960
In_List	0
Shoppinglist	0
include	0

Advanced SQL Query 1:

Return the average price and product name of the top 15 frequently purchased items in stores in the Champaign area. Sort the data by productID in descending order.

```
SELECT avg(Price), pr.Name
FROM Purchase pu natural join Store s join Has h on (pu.PurchaseID =  h.H_purchaseID)
                                              join Product pr on (h.H_productID=pr.ProductID)
WHERE s.StoreID in (select StoreID from Store where City='Champaign')
GROUP BY h.H_productID
ORDER BY count(H_productID) DESC
LIMIT 15
```

Screenshots with the top 15 rows of the query results:



avg(Price)	Name
20	Lemonade - Strawberry, 591 Ml
24.66666667	Flower - Commercial Spider
21	Browning Caramel Glace
12	Foil Wrap
26.5	Pie Filling - Cherry
49	Fond - Chocolate
25	Tart Shells - Savory, 2
21.5	Tart Shells - Sweet, 3
22	Tart Shells - Sweet, 4
36.5	Beef - Tenderloin - Aa
30	Beef - Texas Style Burger
34.5	Beef - Tenderloin
18	Shrimp - 150 - 250
42	Pie Pecan
18.5	Lemonade - Mandarin, 591 Ml

Query 1: Indexing Analysis

- Without additional indexing

```
-> Limit: 15 row(s)  (actual time=4.827..4.830 rows=15 loops=1)
    -> Sort: <temporary>.tmp_field_0 DESC, limit input to 15 row(s) per chunk  (actual
time=4.826..4.828 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=0.001..0.036 rows=307 loops=1)
            -> Aggregate using temporary table  (actual time=4.685..4.750 rows=307 loops=1)
                -> Nested loop inner join  (cost=673.06 rows=911) (actual time=0.315..4.146 rows=370
loops=1)
                    -> Nested loop inner join  (cost=354.28 rows=911) (actual time=0.306..3.369 rows=370
loops=1)
                        -> Nested loop inner join  (cost=189.33 rows=293) (actual time=0.294..2.349 rows=120
loops=1)
                            -> Nested loop inner join  (cost=131.60 rows=96) (actual time=0.284..2.142 rows=47
loops=1)
                                -> Filter: (Store.City = 'Champaign')  (cost=98.00 rows=96) (actual
time=0.035..0.559 rows=47 loops=1)
                                    -> Table scan on Store  (cost=98.00 rows=960) (actual time=0.027..0.417
rows=986 loops=1)
                                -> Single-row index lookup on s using PRIMARY (StoreID=Store.StoreID)
(cost=0.25 rows=1) (actual time=0.033..0.033 rows=1 loops=47)
                            -> Index lookup on pu using StoreID (StoreID=Store.StoreID)  (cost=0.30 rows=3)
(actual time=0.003..0.004 rows=3 loops=47)
                        -> Index lookup on h using PRIMARY (H_purchaseID=pu.PurchaseID)  (cost=0.25
rows=3) (actual time=0.007..0.008 rows=3 loops=120)
                    -> Single-row index lookup on pr using PRIMARY (ProductID=h.H_productID)
(cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=370)
```

- With additional indexing

    a. Index on Store.City:
    Create INDEX idx_storeName ON Store(City);

```
-> Limit: 15 row(s)  (actual time=2.349..2.353 rows=15 loops=1)
    -> Sort: <temporary>.tmp_field_0 DESC, limit input to 15 row(s) per chunk  (actual
time=2.347..2.350 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=0.002..0.038 rows=307 loops=1)
            -> Aggregate using temporary table  (actual time=2.187..2.255 rows=307 loops=1)
                -> Nested loop inner join  (cost=290.46 rows=446) (actual time=0.055..1.860 rows=370
loops=1)
```

```
          -> Nested loop inner join  (cost=134.39 rows=446) (actual time=0.047..1.069 rows=370 loops=1)
            -> Nested loop inner join  (cost=53.63 rows=143) (actual time=0.034..0.350 rows=120 loops=1)
              -> Nested loop inner join  (cost=25.37 rows=47) (actual time=0.023..0.172 rows=47 loops=1)
                -> Index lookup on Store using idx_storeName (City='Champaign')  (cost=8.92 rows=47) (actual time=0.013..0.043 rows=47 loops=1)
                -> Single-row index lookup on s using PRIMARY (StoreID=Store.StoreID)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=47)
              -> Index lookup on pu using StoreID (StoreID=Store.StoreID)  (cost=0.30 rows=3) (actual time=0.003..0.003 rows=3 loops=47)
            -> Index lookup on h using PRIMARY (H_purchaseID=pu.PurchaseID)  (cost=0.25 rows=3) (actual time=0.005..0.005 rows=3 loops=120)
          -> Single-row index lookup on pr using PRIMARY (ProductID=h.H_productID)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=370)
```

b. Index on Product.ProductName:
   Create INDEX idx_productname ON Product(ProductName);

```
-> Limit: 15 row(s)  (actual time=2.995..2.999 rows=15 loops=1)
    -> Sort: <temporary>.tmp_field_0 DESC, limit input to 15 row(s) per chunk  (actual time=2.995..2.997 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=0.001..0.030 rows=307 loops=1)
            -> Aggregate using temporary table  (actual time=2.860..2.919 rows=307 loops=1)
                -> Nested loop inner join  (cost=673.06 rows=911) (actual time=0.067..2.543 rows=370 loops=1)
                    -> Nested loop inner join  (cost=354.28 rows=911) (actual time=0.059..1.844 rows=370 loops=1)
                        -> Nested loop inner join  (cost=189.33 rows=293) (actual time=0.046..0.733 rows=120 loops=1)
                            -> Nested loop inner join  (cost=131.60 rows=96) (actual time=0.035..0.555 rows=47 loops=1)
                                -> Filter: (Store.City = 'Champaign')  (cost=98.00 rows=96) (actual time=0.028..0.457 rows=47 loops=1)
                                    -> Table scan on Store  (cost=98.00 rows=960) (actual time=0.021..0.338 rows=986 loops=1)
                                -> Single-row index lookup on s using PRIMARY (StoreID=Store.StoreID)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=47)
                            -> Index lookup on pu using StoreID (StoreID=Store.StoreID)  (cost=0.30 rows=3) (actual time=0.002..0.003 rows=3 loops=47)
                        -> Index lookup on h using PRIMARY (H_purchaseID=pu.PurchaseID)  (cost=0.25 rows=3) (actual time=0.008..0.009 rows=3 loops=120)
                    -> Single-row index lookup on pr using PRIMARY (ProductID=h.H_productID)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=370)
```

c. Index on Product.ProductName & Store.City:

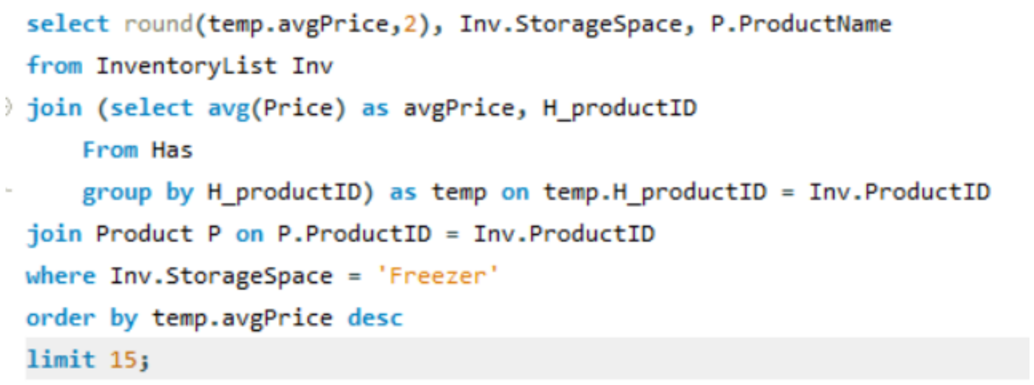
Create INDEX idx_product_store_name ON Store(City);
Create INDEX idx_productname ON Product(ProductName);

```
-> Limit: 15 row(s)  (actual time=2.753..2.756 rows=15 loops=1)
    -> Sort: <temporary>.tmp_field_0 DESC, limit input to 15 row(s) per chunk  (actual
time=2.752..2.754 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=0.002..0.031 rows=307 loops=1)
            -> Aggregate using temporary table  (actual time=2.616..2.675 rows=307 loops=1)
                -> Nested loop inner join  (cost=290.46 rows=446) (actual time=0.106..2.292 rows=370
loops=1)
                    -> Nested loop inner join  (cost=134.39 rows=446) (actual time=0.098..1.481 rows=370
loops=1)
                        -> Nested loop inner join  (cost=53.63 rows=143) (actual time=0.034..0.391
rows=120 loops=1)
                            -> Nested loop inner join  (cost=25.37 rows=47) (actual time=0.024..0.174
rows=47 loops=1)
                                -> Index lookup on Store using idx_storeName (City='Champaign')  (cost=8.92
rows=47) (actual time=0.013..0.043 rows=47 loops=1)
                                -> Single-row index lookup on s using PRIMARY (StoreID=Store.StoreID)
(cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=47)
                            -> Index lookup on pu using StoreID (StoreID=Store.StoreID)  (cost=0.30 rows=3)
(actual time=0.003..0.004 rows=3 loops=47)
                        -> Index lookup on h using PRIMARY (H_purchaseID=pu.PurchaseID)  (cost=0.25
rows=3) (actual time=0.008..0.009 rows=3 loops=120)
                    -> Single-row index lookup on pr using PRIMARY (ProductID=h.H_productID)
(cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=370)
```

- Conclusion
  - We notice that the execution time without indexing is 4.83s. The first index we choose is the Store.Name, since we are returning the stores located in Champaign, we are investigating if the index could accelerate the index lookup on Store table. It turns out that the indexing scheme have a better performance, and the total time is 2.349s, which is much faster than the time when no indexing is selected.
  - The second index is Product.ProductName, since this is another attribute in the Product table. From the analysis, the program doesn't use the indexing strategy, because it uses only the ProdcutID when joining the table. So the indexing scheme did not bring a better effect to the query.
  - The third index scheme is Product.ProdcutName and Store.City. It turns out that only the indexing on Store.City accelerates the running time. Therefore, we will only do the indexing on the Store.City to improve our query.

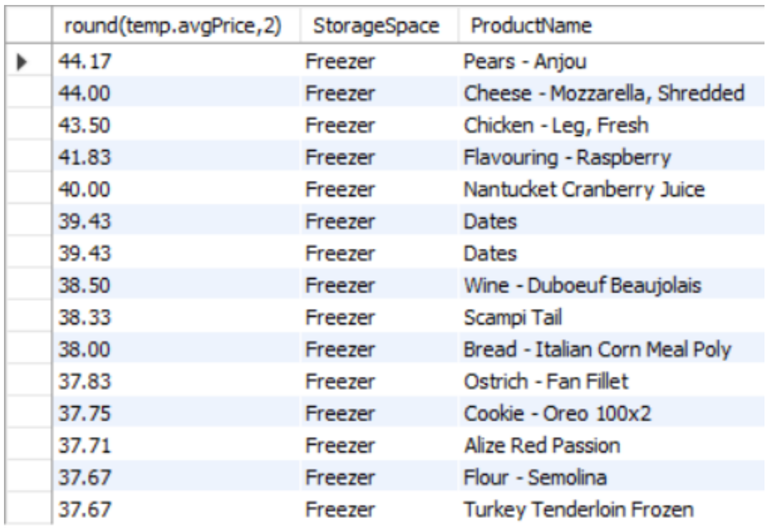
Advanced SQL Query 2:

Return the average price and name of items in a certain storage space. Sort the data by average price in descending order.



```
select round(temp.avgPrice,2), Inv.StorageSpace, P.ProductName
from InventoryList Inv
join (select avg(Price) as avgPrice, H_productID
    From Has
    group by H_productID) as temp on temp.H_productID = Inv.ProductID
join Product P on P.ProductID = Inv.ProductID
where Inv.StorageSpace = 'Freezer'
order by temp.avgPrice desc
limit 15;
```

Screenshots with the top 15 rows of the query results:

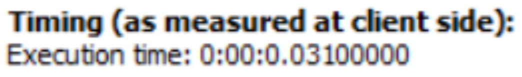


round(temp.avgPrice,2)	StorageSpace	ProductName
44.17	Freezer	Pears - Anjou
44.00	Freezer	Cheese - Mozzarella, Shredded
43.50	Freezer	Chicken - Leg, Fresh
41.83	Freezer	Flavouring - Raspberry
40.00	Freezer	Nantucket Cranberry Juice
39.43	Freezer	Dates
39.43	Freezer	Dates
38.50	Freezer	Wine - Duboeuf Beaujolais
38.33	Freezer	Scampi Tail
38.00	Freezer	Bread - Italian Corn Meal Poly
37.83	Freezer	Ostrich - Fan Fillet
37.75	Freezer	Cookie - Oreo 100x2
37.71	Freezer	Alize Red Passion
37.67	Freezer	Flour - Semolina
37.67	Freezer	Turkey Tenderloin Frozen

Query 2: Indexing Analysis

- Without additional indexing

```
-> Limit: 15 row(s)  (actual time=13.371..13.374 rows=15 loops=1)
    -> Sort: <temporary>.avgPrice DESC, limit input to 15 row(s) per chunk  (actual
time=13.370..13.372 rows=15 loops=1)
        -> Stream results  (actual time=11.534..13.274 rows=330 loops=1)
            -> Nested loop inner join  (actual time=11.529..13.171 rows=330 loops=1)
                -> Nested loop inner join  (cost=136.10 rows=100) (actual time=0.051..1.085
rows=330 loops=1)
                    -> Filter: (Inv.StorageSpace = 'Freezer')  (cost=101.20 rows=100) (actual
time=0.035..0.464 rows=330 loops=1)
                        -> Table scan on Inv  (cost=101.20 rows=997) (actual time=0.032..0.330
rows=997 loops=1)
                    -> Single-row index lookup on P using PRIMARY (ProductID=Inv.ProductID)
(cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=330)
                -> Index lookup on temp using <auto_key0> (H_productID=Inv.ProductID)  (actual
time=0.001..0.001 rows=1 loops=330)
                    -> Materialize  (actual time=0.036..0.036 rows=1 loops=330)
                        -> Group aggregate: avg(Has.Price)  (actual time=0.306..10.495 rows=1000
loops=1)
                            -> Index scan on Has using idx_productID  (cost=641.65 rows=6174) (actual
time=0.297..9.376 rows=6174 loops=1)
```

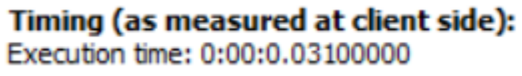

Timing (as measured at client side):
Execution time: 0:00:0.03100000

- With additional indexing

  a. Index on InventoryList.StorageSpace:

  Create index idx_storagespace on InventoryList(StorageSpace);

```
-> Limit: 15 row(s)  (actual time=12.938..12.942 rows=15 loops=1)
    -> Sort: <temporary>.avgPrice DESC, limit input to 15 row(s) per chunk  (actual
time=12.937..12.939 rows=15 loops=1)
        -> Stream results  (actual time=10.965..12.847 rows=330 loops=1)
            -> Nested loop inner join  (actual time=10.957..12.743 rows=330 loops=1)
                -> Nested loop inner join  (cost=153.00 rows=330) (actual time=0.333..1.475 rows=330
loops=1)
                    -> Index lookup on Inv using idx_storagespace (StorageSpace='Freezer')  (cost=37.50
rows=330) (actual time=0.317..0.831 rows=330 loops=1)
                    -> Single-row index lookup on P using PRIMARY (ProductID=Inv.ProductID)
(cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=330)
                -> Index lookup on temp using <auto_key0> (H_productID=Inv.ProductID)  (actual
time=0.001..0.001 rows=1 loops=330)
               -> Materialize  (actual time=0.034..0.034 rows=1 loops=330)
                  -> Group aggregate: avg(Has.Price)  (actual time=0.241..9.676 rows=1000 loops=1)
                     -> Index scan on Has using idx_productID  (cost=641.65 rows=6174) (actual
time=0.230..8.571 rows=6174 loops=1)
```

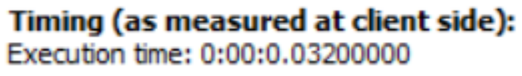


```
Timing (as measured at client side):
Execution time: 0:00:0.03100000
```

b. Index on Has.Price:

Create index idx_price on Has(Price);

```
-> Limit: 15 row(s)  (actual time=12.216..12.219 rows=15 loops=1)
   -> Sort: <temporary>.avgPrice DESC, limit input to 15 row(s) per chunk  (actual
time=12.215..12.217 rows=15 loops=1)
      -> Stream results  (actual time=10.449..12.122 rows=330 loops=1)
         -> Nested loop inner join  (actual time=10.444..12.020 rows=330 loops=1)
            -> Nested loop inner join  (cost=136.10 rows=100) (actual time=0.029..1.029 rows=330
loops=1)
               -> Filter: (Inv.StorageSpace = 'Freezer')  (cost=101.20 rows=100) (actual
time=0.020..0.445 rows=330 loops=1)
                  -> Table scan on Inv  (cost=101.20 rows=997) (actual time=0.017..0.311 rows=997
loops=1)
               -> Single-row index lookup on P using PRIMARY (ProductID=Inv.ProductID)
(cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=330)
            -> Index lookup on temp using <auto_key0> (H_productID=Inv.ProductID)  (actual
time=0.001..0.001 rows=1 loops=330)
               -> Materialize  (actual time=0.033..0.033 rows=1 loops=330)
                  -> Group aggregate: avg(Has.Price)  (actual time=0.203..9.481 rows=1000 loops=1)
                     -> Index scan on Has using idx_productID  (cost=641.65 rows=6174) (actual
time=0.196..8.357 rows=6174 loops=1)
```

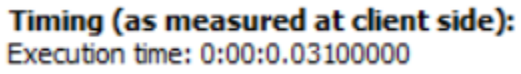


```
Timing (as measured at client side):
Execution time: 0:00:0.03200000
```

c. Index on : Product.ProductName

Create index idx_productname on Product(ProductName);

```
-> Limit: 15 row(s)  (actual time=11.755..11.758 rows=15 loops=1)
   -> Sort: <temporary>.avgPrice DESC, limit input to 15 row(s) per chunk  (actual
time=11.754..11.756 rows=15 loops=1)
      -> Stream results  (actual time=9.921..11.670 rows=330 loops=1)
         -> Nested loop inner join  (actual time=9.916..11.567 rows=330 loops=1)
            -> Nested loop inner join  (cost=136.10 rows=100) (actual time=0.088..1.135 rows=330
loops=1)
            -> Filter: (Inv.StorageSpace = 'Freezer')  (cost=101.20 rows=100) (actual
time=0.080..0.532 rows=330 loops=1)
                -> Table scan on Inv  (cost=101.20 rows=997) (actual time=0.077..0.397 rows=997
loops=1)
            -> Single-row index lookup on P using PRIMARY (ProductID=Inv.ProductID)
(cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=330)
        -> Index lookup on temp using <auto_key0> (H_productID=Inv.ProductID)  (actual
time=0.001..0.001 rows=1 loops=330)
            -> Materialize  (actual time=0.031..0.031 rows=1 loops=330)
                -> Group aggregate: avg(Has.Price)  (actual time=0.224..8.942 rows=1000 loops=1)
                    -> Index scan on Has using idx_productID  (cost=641.65 rows=6174) (actual
time=0.217..7.879 rows=6174 loops=1)
```

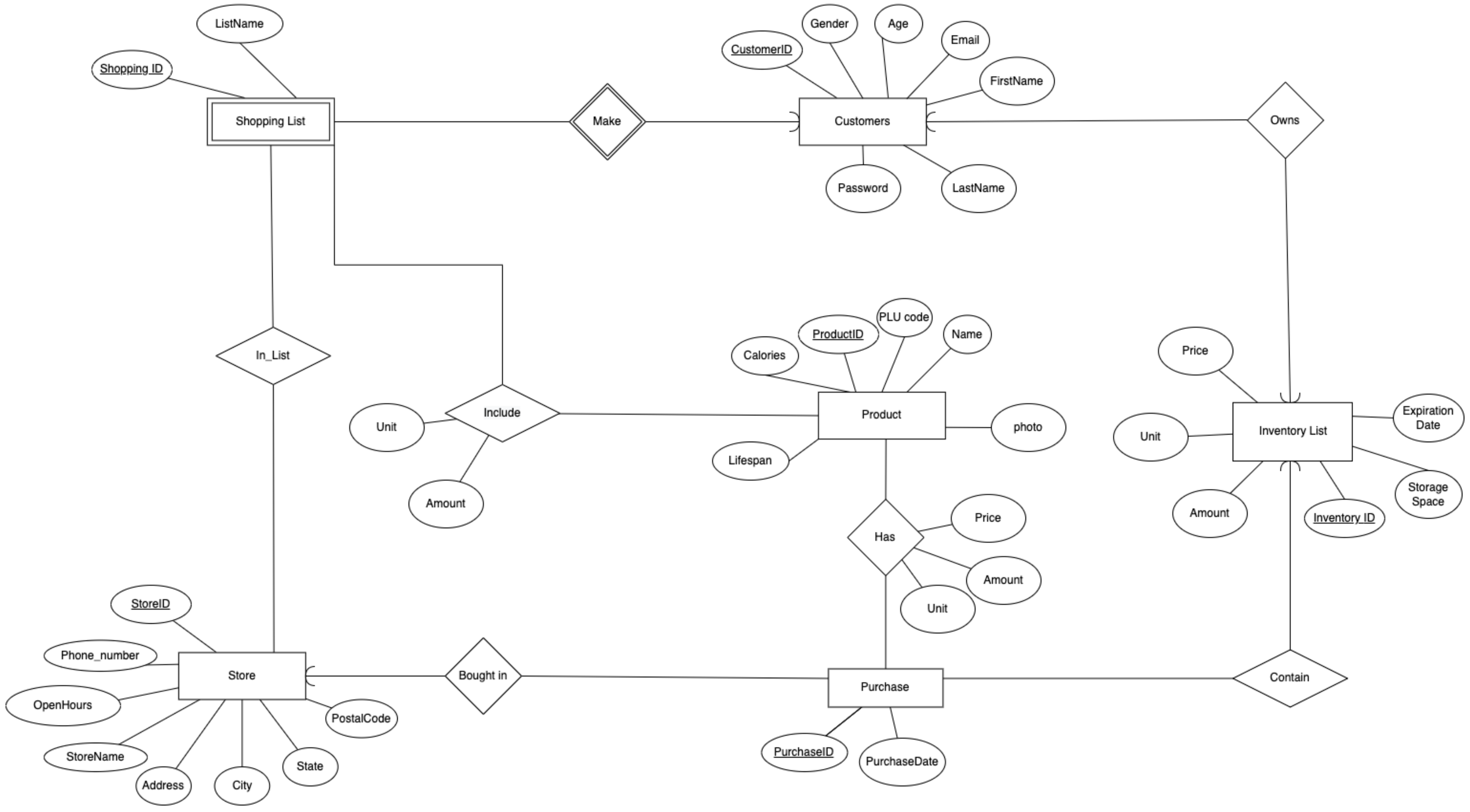


```
Timing (as measured at client side):
Execution time: 0:00:0.03100000
```

- Conclusion

  We chose to use no indexes for this advanced query because there were no significant differences in the performances in each of the indexing designs. The index on Has.Price probably didn't make a difference in this situation because the query groups the subquery by H_productID, which is a foreign key and is an index by default because of how mySQL is designed. The other two indexing techniques with an index on either StorageSpace or ProductName also probably didn't make a difference because there weren't any complex operations involving those variables to begin with.

Appendix - Updated ER Diagram


ListName
Shopping ID
Shopping List
Make
Gender
Age
Email
CustomerID
FirstName
Customers
Owns
Password
LastName
In_List
PLU code
ProductID
Name
Calories
Price
Include
Unit
Product
photo
Unit
Inventory List
Expiration Date
Lifespan
Amount
Storage Space
Amount
Inventory ID
Has
Price
Amount
Unit
StoreID
Phone_number
Store
Bought in
Purchase
Contain
OpenHours
PostalCode
StoreName
State
Address
City
PurchaseID
PurchaseDate