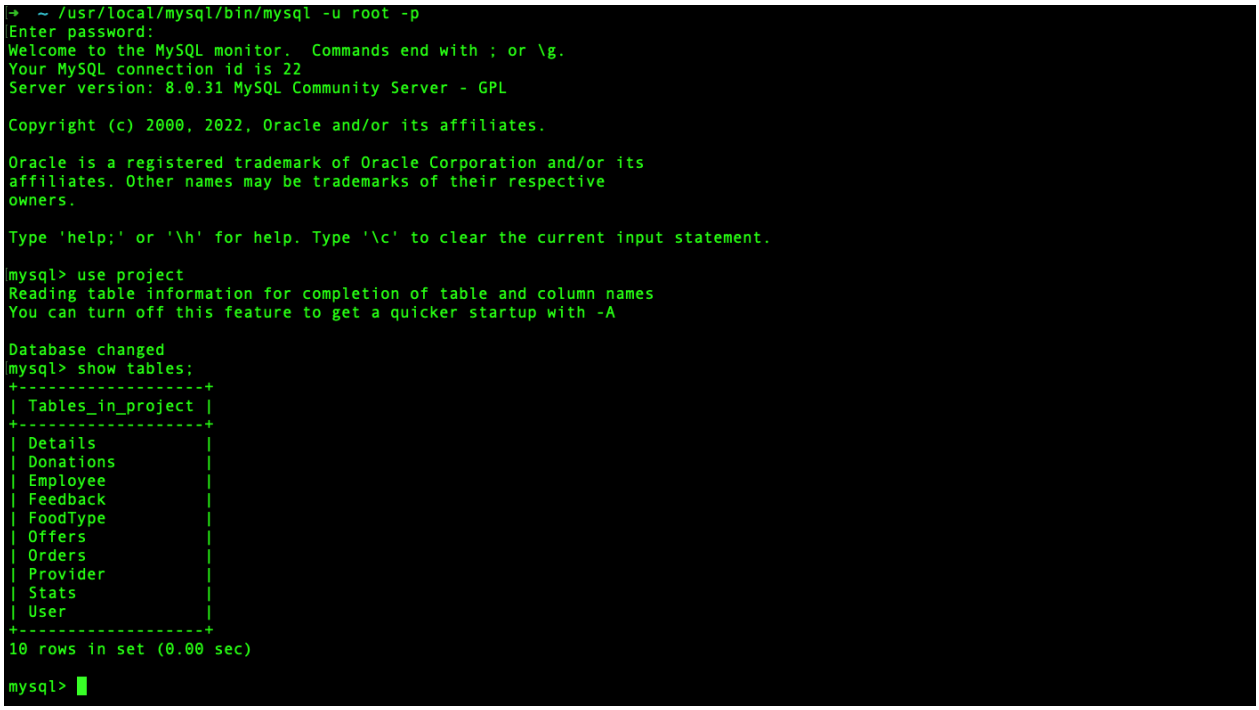
Stage 3:Database Design

Databases Implementation:



DDL Commands:

```
CREATE TABLE FoodType(
  FoodID VARCHAR (255) PRIMARY KEY,
  Kind VARCHAR(255)
);

CREATE TABLE Details(
  ContactID VARCHAR(255) PRIMARY KEY,
  Email VARCHAR(255) ,
  PhoneNumber VARCHAR(255) NOT NULL,
  SocialHandle VARCHAR(255),
  Address VARCHAR(255) NOT NULL,
  ZipCode INTEGER
);


CREATE TABLE User(
  ContactID VARCHAR(255),
  UserID VARCHAR(255) PRIMARY KEY,
  Name VARCHAR(255) NOT NULL,
  Password VARCHAR(255) NOT NULL,
  Kind VARCHAR(255)  NOT NULL,
  FOREIGN KEY User(ContactID) REFERENCES Details(ContactID)
);

CREATE TABLE Provider(
  ProviderID VARCHAR(255) PRIMARY KEY,
  Name VARCHAR(255),
  ContactID VARCHAR(255),
  Password VARCHAR(255) NOT NULL,
  Rating INTEGER,
  FOREIGN KEY Provider(ContactID) references Details(ContactID)
);

CREATE TABLE Donations(
  TransactionID VARCHAR(255) PRIMARY KEY,
  ReceiverID VARCHAR(255),
```

```
  Amount REAL,
  Donor VARCHAR(255),
  FOREIGN KEY Donations(ReceiverID) REFERENCES User(UserID)
);

CREATE TABLE Employee(
EmpID VARCHAR(255),
ProviderID VARCHAR(255),
Name VARCHAR(255),
Designation  VARCHAR(255),
FOREIGN KEY Employee(ProviderID) references Provider(ProviderID),
PRIMARY KEY(EmpID, ProviderID)
);

CREATE TABLE Orders (
  OrderID VARCHAR(255) PRIMARY KEY,
  UserID VARCHAR(255),
  ProviderID VARCHAR(255),
  FoodID VARCHAR(255),
  Quantity INT,
  ODate TIMESTAMP,
  FOREIGN KEY(UserID) REFERENCES User(UserID),
  FOREIGN KEY(FoodID) REFERENCES FoodType(FoodID)
);


CREATE TABLE Stats(
ProviderID VARCHAR(255),
ODate DATETIME,
MealsProduced INTEGER,
MealsSold INTEGER,
ExcessFoodSold INTEGER,
Wastage INTEGER,
FOREIGN KEY Stats(ProviderID) references Provider(ProviderID),
PRIMARY KEY (ProviderID, ODate)
);

CREATE TABLE Offers(
ProviderID VARCHAR(255),
FoodID VARCHAR (255),
Item_Description VARCHAR(255),
ODate DATETIME,
Quantity INTEGER,
```

```
FOREIGN KEY Offers(ProviderID) references Provider(ProviderID),
FOREIGN KEY Offers(FoodID) references FoodType(FoodID),
PRIMARY KEY (ProviderID, FoodID, ODate)
);

CREATE TABLE Feedback(
UserID VARCHAR(255),
ProviderID VARCHAR(255),
Comment VARCHAR(255),
Rating INT,
Date DATETIME,
FOREIGN KEY Feedback(UserID) references User(UserID),
FOREIGN KEY (ProviderID) references Provider(ProviderID)
);
```

COUNT Query on Tables:

```
mysql> select count(*) from Details;
+----------+
| count(*) |
+----------+
|     1323 |
+----------+
1 row in set (0.01 sec)

mysql> select count(*) from Offers;
+----------+
| count(*) |
+----------+
|    29389 |
+----------+
1 row in set (0.00 sec)

mysql> select count(*) from Orders;
+----------+
| count(*) |
+----------+
|    29389 |
+----------+
1 row in set (0.00 sec)
```

Advanced Queries:

Query 1

```
SELECT ofr.ProviderID, ofr.ODate as ODate, TotalOffered as MealsProduced,
TotalSold as MealsSold, 10 as ExcessSold, TotalOffered-TotalSold as Wasted
FROM
(SELECT ProviderID, DATE(ODate) as ODate, SUM(Quantity) as TotalSold FROM
Orders od
GROUP BY ProviderID, Date(ODate)) od JOIN
(SELECT ProviderID, Date(ODate) as ODate, SUM(Quantity) as TotalOffered
FROM Offers
WHERE ODate = "2022-08-01"
GROUP BY ProviderID, Date(ODate)) ofr on (Date(od.ODate)=Date(ofr.ODate)
AND od.ProviderID=ofr.ProviderID)
ORDER BY Date(ofr.ODate)
LIMIT 15
```

ProviderID	ODate	MealsProduced	MealsSold	ExcessSold	Wasted
P146	2022-08-01	20	2	10	18
P77	2022-08-01	10	1	10	9
P286	2022-08-01	10	1	10	9
P57	2022-08-01	10	1	10	9
P54	2022-08-01	20	2	10	18
P84	2022-08-01	30	3	10	27
P254	2022-08-01	10	1	10	9
P128	2022-08-01	10	1	10	9
P193	2022-08-01	10	1	10	9
P139	2022-08-01	10	1	10	9
P24	2022-08-01	10	1	10	9
P98	2022-08-01	10	1	10	9
P185	2022-08-01	10	1	10	9
P223	2022-08-01	20	2	10	18

QUERY 2

```
Select f.Kind, count(*) as availableOptions
From FoodType f natural join Offers o
group by f.Kind
Order by o.ODate;
```

Output:

Kind	availableOptions
VEGAN BREAKFAST	3229
VEGAN DINNER	3199
VEGAN LUNCH	3315
VEG DINNER	3201
VEG BREAKFAST	3267
NONVEG LUNCH	3233
VEG LUNCH	3330
NONVEG BREAKFAST	3159
NONVEG DINNER	3237

Indexing:

Query 1:

Without indexing : total time: 169.217ms

```
EXPLAIN No Index
-> Limit: 15 row(s)  (actual time=169.217..169.222 rows=15 loops=1)
    -> Sort: cast(ofr.ODate as date), limit input to 15 row(s) per chunk
(actual time=169.215..169.218 rows=15 loops=1)
        -> Stream results  (cost=4869.32 rows=0) (actual
time=153.586..169.147 rows=32 loops=1)
            -> Filter: (od.ProviderID = ofr.ProviderID)  (cost=4869.32
rows=0) (actual time=153.576..169.066 rows=32 loops=1)
                -> Inner hash join
(<hash>(od.ProviderID)=<hash>(ofr.ProviderID)), (cast(od.ODate as date) =
cast(ofr.ODate as date))  (cost=4869.32 rows=0) (actual
time=153.573..169.044 rows=32 loops=1)
                    -> Table scan on od  (cost=2.50..2.50 rows=0) (actual
time=131.792..139.434 rows=27679 loops=1)
                        -> Materialize  (cost=0.00..0.00 rows=0) (actual
time=131.790..131.790 rows=27679 loops=1)
                            -> Table scan on <temporary>  (actual
time=110.996..119.784 rows=27679 loops=1)
                                -> Aggregate using temporary table  (actual
time=110.994..110.994 rows=27679 loops=1)
                                    -> Table scan on od  (cost=3149.95
rows=30937) (actual time=0.042..24.595 rows=29389 loops=1)
                    -> Hash
                        -> Table scan on ofr  (cost=2.50..2.50 rows=0)
(actual time=20.795..20.803 rows=32 loops=1)
                            -> Materialize  (cost=0.00..0.00 rows=0)
(actual time=20.793..20.793 rows=32 loops=1)
                                -> Table scan on <temporary>  (actual
time=20.758..20.767 rows=32 loops=1)
                                    -> Aggregate using temporary table
(actual time=20.756..20.756 rows=32 loops=1)
                                        -> Filter: (offers.ODate =
DATE'2022-08-01')  (cost=2921.65 rows=2897) (actual time=1.839..20.604
rows=33 loops=1)
                                            -> Table scan on Offers
(cost=2921.65 rows=28974) (actual time=0.053..16.884 rows=29170 loops=1)
```

1. Index on ProviderID

Total time : 237ms

```
EXPLAIN ProviderID
-> Limit: 15 row(s)  (actual time=237.037..237.044 rows=15 loops=1)
    -> Sort: cast(ofr.ODate as date), limit input to 15 row(s) per chunk
(actual time=237.036..237.040 rows=15 loops=1)
        -> Stream results  (cost=4869.32 rows=0) (actual
time=212.900..236.940 rows=32 loops=1)
            -> Filter: (od.ProviderID = ofr.ProviderID)  (cost=4869.32
rows=0) (actual time=212.869..236.812 rows=32 loops=1)
                -> Inner hash join
(<hash>(od.ProviderID)=<hash>(ofr.ProviderID)), (cast(od.ODate as date) =
cast(ofr.ODate as date))  (cost=4869.32 rows=0) (actual
time=212.864..236.782 rows=32 loops=1)
                    -> Table scan on od  (cost=2.50..2.50 rows=0) (actual
time=188.564..200.729 rows=27679 loops=1)
                        -> Materialize  (cost=0.00..0.00 rows=0) (actual
time=188.560..188.560 rows=27679 loops=1)
                            -> Table scan on <temporary>  (actual
time=162.215..173.192 rows=27679 loops=1)
                                -> Aggregate using temporary table  (actual
time=162.213..162.213 rows=27679 loops=1)
                                    -> Table scan on od  (cost=3149.95
rows=30937) (actual time=0.037..37.046 rows=29389 loops=1)
                    -> Hash
                        -> Table scan on ofr  (cost=2.50..2.50 rows=0)
(actual time=22.229..22.237 rows=32 loops=1)
                            -> Materialize  (cost=0.00..0.00 rows=0)
(actual time=22.228..22.228 rows=32 loops=1)
                                -> Table scan on <temporary>  (actual
time=22.191..22.200 rows=32 loops=1)
                                    -> Aggregate using temporary table
(actual time=22.189..22.189 rows=32 loops=1)
                                        -> Filter: (offers.ODate =
DATE'2022-08-01')  (cost=2921.65 rows=2897) (actual time=1.806..21.982
rows=33 loops=1)
                                            -> Table scan on Offers
(cost=2921.65 rows=28974) (actual time=0.049..18.124 rows=29170 loops=1)
```

Brief Analysis: We did not observe an improvement on using ProviderID as the SQL optimizer find table scan cost cheaper than using the indexed column.

2. Index on ODate:

   Time taken : 139ms

```
EXPLAIN ODate
-> Limit: 15 row(s)  (actual time=139.625..139.630 rows=15 loops=1)
    -> Sort: cast(ofr.ODate as date), limit input to 15 row(s) per chunk
(actual time=139.623..139.625 rows=15 loops=1)
        -> Stream results  (cost=442.72 rows=0) (actual
time=124.550..139.532 rows=32 loops=1)
            -> Filter: (od.ProviderID = ofr.ProviderID)  (cost=442.72
rows=0) (actual time=124.540..139.433 rows=32 loops=1)
                -> Inner hash join
(<hash>(od.ProviderID)=<hash>(ofr.ProviderID)), (cast(od.ODate as date) =
cast(ofr.ODate as date))  (cost=442.72 rows=0) (actual
time=124.538..139.405 rows=32 loops=1)
                    -> Table scan on od  (cost=2.50..2.50 rows=0) (actual
time=123.124..130.512 rows=27679 loops=1)
                        -> Materialize  (cost=0.00..0.00 rows=0) (actual
time=123.121..123.121 rows=27679 loops=1)
                            -> Table scan on <temporary>  (actual
time=105.268..112.946 rows=27679 loops=1)
                                -> Aggregate using temporary table  (actual
time=105.266..105.266 rows=27679 loops=1)
                                    -> Table scan on od  (cost=3149.95
rows=30937) (actual time=0.038..22.985 rows=29389 loops=1)
                    -> Hash
                        -> Table scan on ofr  (cost=2.50..2.50 rows=0)
(actual time=0.426..0.436 rows=32 loops=1)
                            -> Materialize  (cost=0.00..0.00 rows=0)
(actual time=0.425..0.425 rows=32 loops=1)
                                -> Table scan on <temporary>  (actual
time=0.389..0.398 rows=32 loops=1)
                                    -> Aggregate using temporary table
(actual time=0.388..0.388 rows=32 loops=1)
                                        -> Index lookup on Offers using
ind5 (ODate=DATE'2022-08-01')  (cost=11.55 rows=33) (actual
time=0.305..0.319 rows=33 loops=1)
```

Brief Analysis:

For a particular Date, the amount of food wasted is calculated using the above query. In the first method, without indexing, the rows are searched and the records of a particular date are analyzed. However, after indexing, since we are indexing on date, the database is easily able to

search through the database for records of a particular date. Hence, we get faster performance after indexing on the Date field.

3. Index on : Quantity

Total time : 230.905ms

```
EXPLAIN Qty
```

```
-> Limit: 15 row(s)  (actual time=230.905..230.913 rows=15 loops=1)
    -> Sort: cast(ofr.ODate as date), limit input to 15 row(s) per chunk
(actual time=230.904..230.908 rows=15 loops=1)
        -> Stream results  (cost=4869.32 rows=0) (actual
time=207.895..230.805 rows=32 loops=1)
            -> Filter: (od.ProviderID = ofr.ProviderID)  (cost=4869.32
rows=0) (actual time=207.882..230.685 rows=32 loops=1)
                -> Inner hash join
(<hash>(od.ProviderID)=<hash>(ofr.ProviderID)), (cast(od.ODate as date) =
cast(ofr.ODate as date))  (cost=4869.32 rows=0) (actual
time=207.879..230.653 rows=32 loops=1)
                    -> Table scan on od  (cost=2.50..2.50 rows=0) (actual
time=186.615..198.007 rows=27679 loops=1)
                    -> Materialize  (cost=0.00..0.00 rows=0) (actual
time=186.612..186.612 rows=27679 loops=1)
                        -> Table scan on <temporary>  (actual
time=156.767..169.070 rows=27679 loops=1)
                            -> Aggregate using temporary table  (actual
time=156.765..156.765 rows=27679 loops=1)
                                -> Table scan on od  (cost=3149.95
rows=30937) (actual time=0.044..36.237 rows=29389 loops=1)
                    -> Hash
                        -> Table scan on ofr  (cost=2.50..2.50 rows=0)
(actual time=19.854..19.862 rows=32 loops=1)
                            -> Materialize  (cost=0.00..0.00 rows=0)
(actual time=19.853..19.853 rows=32 loops=1)
                                -> Table scan on <temporary>  (actual
time=19.816..19.824 rows=32 loops=1)
                                    -> Aggregate using temporary table
(actual time=19.813..19.813 rows=32 loops=1)
                                        -> Filter: (offers.ODate =
DATE'2022-08-01')  (cost=2921.65 rows=2897) (actual time=1.923..19.700
rows=33 loops=1)
                                            -> Table scan on Offers
(cost=2921.65 rows=28974) (actual time=0.048..16.245 rows=29170 loops=1)
```

Brief Analysis: For analyzing the records that meet the range of quantities we have mentioned, the time taken by the query without indexes takes more time. However, after we apply indexing on the quantity field, the query uses a B+ Tree index which works really well on range queries. Thus, the performance of the query is enhanced when we apply indexing on the quantity field.

Query 2 :

```
Select f.Kind, count(f.FoodID) as availableOptions
From FoodType f natural join Offers o
group by f.Kind
Order by o.ODate;
```

First we show the output of the query without indexing: Total time : 76ms

```
EXPLAIN
-> Limit: 5 row(s)  (actual time=76.145..76.147 rows=5 loops=1)
    -> Sort: o.ODate, limit input to 5 row(s) per chunk  (actual
time=76.143..76.144 rows=5 loops=1)
        -> Table scan on <temporary>  (actual time=76.067..76.072 rows=9
loops=1)
            -> Aggregate using temporary table  (actual time=76.059..76.059
rows=9 loops=1)
                -> Nested loop inner join  (cost=5599.63 rows=30529)
(actual time=0.157..27.885 rows=29170 loops=1)
                    -> Table scan on f  (cost=1.15 rows=9) (actual
time=0.076..0.106 rows=9 loops=1)
                    -> Covering index lookup on o using Offers
(FoodID=f.FoodID)  (cost=320.53 rows=3392) (actual time=0.044..2.583
rows=3241 loops=9)
```

1.Index: FoodID

Next, we tried Indexing on the FoodID attribute. We found the following :
Total time : 65.310s

```
EXPLAIN FoodID
-> Limit: 5 row(s)  (actual time=65.265..65.267 rows=5 loops=1)
    -> Sort: o.ODate, limit input to 5 row(s) per chunk  (actual
time=65.264..65.265 rows=5 loops=1)
        -> Table scan on <temporary>  (actual time=65.226..65.230 rows=9
loops=1)
            -> Aggregate using temporary table  (actual time=65.223..65.223
rows=9 loops=1)
                -> Nested loop inner join  (cost=12990.04 rows=30529)
(actual time=0.103..24.484 rows=29170 loops=1)
                    -> Table scan on f  (cost=1.15 rows=9) (actual
time=0.042..0.070 rows=9 loops=1)
                    -> Covering index lookup on o using ind2
(FoodID=f.FoodID)  (cost=1141.69 rows=3392) (actual time=0.036..2.223
rows=3241 loops=9)
```

Brief analysis: Here ind2 covers the index on FoodID. Indexing on FoodID is not reducing the implementation time significantly as FoodID is only used once for the count operation. As FoodID is not searched multiple times, hence it is justified if this indexing is not reducing the implementation time.

2.Index: ODate

Next, we tried indexing on the Date attribute. We found the following results:
Time : 67.711ms

```
EXPLAIN ODate
-> Limit: 5 row(s)  (actual time=67.711..67.712 rows=5 loops=1)
    -> Sort: o.ODate, limit input to 5 row(s) per chunk  (actual
time=67.710..67.710 rows=5 loops=1)
        -> Table scan on <temporary>  (actual time=67.680..67.683 rows=9
loops=1)
            -> Aggregate using temporary table  (actual time=67.677..67.677
rows=9 loops=1)
                -> Nested loop inner join  (cost=5599.63 rows=30529)
```

```
(actual time=0.082..25.121 rows=29170 loops=1)
                    -> Table scan on f  (cost=1.15 rows=9) (actual
time=0.035..0.055 rows=9 loops=1)
                    -> Covering index lookup on o using Offers
(FoodID=f.FoodID)  (cost=320.53 rows=3392) (actual time=0.032..2.291
rows=3241 loops=9)
```

Brief Analysis: Indexing on the ODate does not provide much improvement because sorting Date is the most time consuming operation which requires going through all records. Thus, indexing is not providing improvement.

3. Index: Kind

Next, we tried indexing on the Kind attribute. We found the following results:

```
EXPLAIN Q2_Kind
-> Limit: 5 row(s)  (actual time=32.408..32.410 rows=5 loops=1)
    -> Sort: o.ODate, limit input to 5 row(s) per chunk  (actual
time=32.407..32.408 rows=5 loops=1)
        -> Stream results  (cost=8652.53 rows=30529) (actual
time=3.708..32.367 rows=9 loops=1)
            -> Group aggregate: count(0)  (cost=8652.53 rows=30529) (actual
time=3.704..32.348 rows=9 loops=1)
                -> Nested loop inner join  (cost=5599.63 rows=30529)
(actual time=0.067..21.534 rows=29170 loops=1)
                    -> Covering index scan on f using ind3  (cost=1.15
rows=9) (actual time=0.026..0.037 rows=9 loops=1)
                    -> Covering index lookup on o using Offers
(FoodID=f.FoodID)  (cost=320.53 rows=3392) (actual time=0.030..1.916
rows=3241 loops=9)
```

Brief Analysis: Indexing on the Kind attribute works best among the three indexes we have tried. This is mainly because the we have applied the “group by” operation on the Kind attribute. This indexing has decreased the implementation time by 2.38 times.