An Attempt to Alleviate Farmers' Financial Struggles

Tyler Landivar
Dept. of Computer Science
University at Buffalo
Buffalo, US
tklandiv@buffalo.edu

Gavin Rufus Arul Samraj
Dept. of Computer Science
University at Buffalo
Buffalo, US
gavinruf@buffalo.edu

Simhavishnu Ramprasad
Dept. of Computer Science
University at Buffalo
Buffalo, US
simhavis@buffalo.edu

NOTE: ER DIAGRAM IS AT THE VERY END!

PHASE 1 (Updated)

I. Project details

Name of project: Farmers Market Database

Team Members: Tyler Landivar (tklandiv)

Gavin Rufus Arul Samraj (gavinruf)

Simhavishnu Ramprasad (simhavis)

II. Problem Statement

In recent years, farmers have growingly struggled to maintain their farms for economic reasons. There are a variety of reasons why they are struggling financially. Some include federal spending, corporate farming and inflation. Federal spending continues to divert away from helping out farmers which of course results in less financial aid for them. Corporate farming (large-scale agriculture done by bigger companies) makes competition very difficult as they are able to sell their products at reduced costs and produce them at higher rates compared to local farmers. Lastly, inflation makes farming equipment and other utility/miscellaneous expenses more devastating. With all these trends, it's no wonder why local farmers are struggling.

This situation is the last thing people should want for the following reasons: local farmers are losing their livelihoods, overreliance on corporate farming means an increase in intake of chemicals such as pesticides, and less overall competition can lead to a monopoly, potentially enabling large companies to dictate prices.

The potential outcomes of this problem are too bad. Our project tries to help prevent this. We want to bring local farmers and consumers together, so they can get a steady income and keep their farms running. The database we are creating will allow people to view farmers markets (a place where farmers sell their goods) in a specified location along with the vendors and the products they sell. That way, they are aware of the fresh goods they can buy and potentially support local farmers as a result.

A database is a more acceptable solution to this approach than an excel file. This is because users can update, insert, delete, retrieve, and maintain information efficiently despite a constantly growing dataset (scalability) unlike excel; authorizations can be implemented to allow certain people to view private information; table joins and other SQL functionalities enable efficient and clean, complex querying that excel just can't do; integrity constraints can be applied to ensure accuracy and consistency of data; and a database is the typical container for data that a website or application relies on for their backend (this project would want a website to provide a friendly UI for everyday people to query the database).

III. Target users

The main target users of this database would be the local farmers and producers. It would help them broaden their reach with their consumers by posting and updating information about their products, their availability, and their location. Since this helps them to have a steady income. Whereas, consumers and local shoppers would be the key users who want to purchase these locally sourced fresh produce from the local farmers and producers. They would be able to find information about the nearest farmers' market, available products and their pricing via a website or application (Front-end) that is directly connected to the database

(Back-end). This enables them to stay informed and plan their trips accordingly or connect with local farmers directly. Fostering this connection increases the potential income of farmers.

IV. Database schema

Assumptions: attributes can't be null unless specified. No default value unless specified. Delete cascade is implemented for the primary key referenced by a foreign key for ALL foreign keys. Update cascade only applied to on_hand relation

Changes made from phase 1: the names of certain attributes in the relations have been changed since they were viewed as keywords. Attributes known as name have been changed to (relation name)_name; e.g. name in markets changed to market_name. Attributes named state has been changed to state_. To help avoid the confusion about changes, the updated database schema is included.

Markets (market_id, market_name, open_hours, close_hours, street, city, state_, season_op)

- Market_id - Unique identifier for the market (INT). Since unique, it's the primary key
- Market_name - the name of the market (VARCHAR(100))
- Open_hours - the time when the market opens (TIME)
- Close_hours - the time when the market closes (TIME)
- Street - street location of the market (VARCHAR(100))
- City - city location of the market (VARCHAR(100))
- State_ - state (abbreviated) location of the market (VARCHAR(2))
- Season_op - the season(s) when the market operates (VARCHAR(100))

Vendors (vendor_id, vendor_name, open_hours, close_hours, street, city, state_, email, phone_number, season_op)

- Vendor_id - Unique identifier for the vendor (INT). Since unique, it's the primary key
- Vendor_name - the name of the vendor (VARCHAR(100))
- Open_hours - the time when the vendor opens (TIME)
- Close_hours - the time when the vendor closes (TIME)
- Street - street location of the vendor (VARCHAR(100))
- City - city location of the vendor (VARCHAR(100))
- State_ - state (abbreviated) location of the vendor (VARCHAR(2))
- Email - the email of the vendor (VARCHAR(100)).
- Phone_number - the phone number of the vendor (VARCHAR(12)).
- Season_op - the season(s) when the vendor operates (VARCHAR(100))

Products (vendor_id, product_name, price)

- Vendor_id- Unique identifier of vendor selling product (INT). Foreign key and primary key since it references the primary key of the identifying entity.
- Product_name - name of the product (VARCHAR(100)). Primary key since it's a discriminant
- Price - price of the product (NUMERIC(5,2)). Primary key since it's a discriminant

Consumers(consumer_id, first_name, last_name, phone_number, email)

- Consumer_id - Unique identifier for the consumer (INT). Since unique, it's the primary key
- First_name - First name of the consumer (VARCHAR(100))
- Last_name - Last name of the consumer (VARCHAR(100))
- Phone_number - The phone number of the consumer (VARCHAR(12))
- Email - The email of the consumer (VARCHAR(100))

Vendor_reviews (consumer_id, vendor_id, rating)

- Consumer_id - Unique identifier for the consumer (INT). Foreign key since references primary key of consumer relation and Primary key due to rules of representation of relationship sets.
- Vendor_id - Unique identifier for the vendor (INT). Foreign key since references primary key of vendor relation and Primary key due to rules of representation of relationship sets.
- Rating - Customer rating [1-5 rating]. (NUMERIC(1,0))

Market_reviews (consumer_id, market_id, rating)

- Consumer_id - Unique identifier for the consumer (INT). Foreign key since references primary key of consumer relation and Primary key due to rules of representation of relationship sets.
- Market_id - Unique identifier for the vendor (INT). Foreign key since references primary key of market relation and Primary key due to rules of representation of relationship sets.

- Rating - Customer rating [1-5 rating]. (NUMERIC(1,0))

Hosts (market_id, vendor_id)

- Market_id - Unique identifier for the market (INT). Foreign key since references primary key of market relation and Primary key due to rules of representation of relationship sets.
- Vendor_id - Unique identifier for the vendor that a market hosts (INT). Foreign key since references primary key of vendor relation and Primary key due to rules of representation of relationship sets.

On_hand (market_id, vendor_id, product_name, price, quantity)

- Market_id - Unique identifier for the market (INT). Foreign key since references primary key of market relation and Primary key due to rules of representation of relationship sets.
- Vendor_id - Unique identifier for the vendor (INT). Foreign key since references primary key of product relation and Primary key due to rules of representation of relationship sets.
- Product_name - name of the product (VARCHAR(100)). Foreign key since references primary key of product relation and Primary key due to rules of representation of relationship sets.
- Price - price of the product (NUMERIC(5,2)). Foreign key since references primary key of product relation and Primary key due to rules of representation of relationship sets.
- Quantity - Total number of products available at the market (SMALLINT)

V. Acquiring the data. Details about dataset

For the project, we are generating our own dataset to be placed into our created database. This is because we couldn't find a consolidated, pre-made database that contained all of the information we wanted like which farmer's markets have which farms/vendors selling their products. However, because we are creating data from scratch, this provides us with a lot of flexibility. We can generate all the information we need and thus not have to worry about database design restrictions that are in place due to the limitations of a pre-made dataset. There's also a negative to this. We will not be generating real information and thus data in our database will not be factually accurate. This is acceptable though as our project's goal is to design a database that will allow individuals involved with this domain to create and manipulate information themselves. We are just showcasing how our database design can effectively help them accomplish this.

Back to the main topic, we generated data for each of our eight relations. These were created using ipynb files that generated random tuples and placed them in their respective csv files. These csv files were then properly imported into the relations we created in the SQL server. There were originally 260 rows for the vendor relation, 150 rows in the users relation, 100 rows for the markets, 300 rows in the market reviews relation, 300 rows in the vendor reviews relation, 200 rows in the host relation, 400 rows in the products relation, and 400 rows for the onhand. All of these relations now have more rows with the slight exception to markets. This is because in real life, there are of course way more vendors than there are farmers markets. Now there are 700 consumers, 773 rows for hosts relation, 100 markets still, 1079 market reviews, 2528 rows in the onhand relation, 5378 rows in the products relation, 850 vendors, and 2825 vendor reviews.

VI. Proving database is in BCNF

- Markets (market_id, market_name, open_hours, close_hours, street, city, state_, season_op)

  F: {market_id -> market_name, open_hours, close_hours, street, city, state_, season_op}

  We are assuming that a market might have the same name as another. Thus, the specified dependency function is the only one that exists which isn't trivial. Since market_id here is a superkey, this relation is in BCNF

- Vendors (vendor_id, vendor_name, open_hours, close_hours, street, city, state_, email, phone_number, season_op)

  F: {vendor_id -> vendor_name, open_hours, close_hours, street, city, state_, email, phone_number, season_op; email-> vendor_id, vendor_name, open_hours, close_hours, street, city, state_, phone_number, season_op}

  We are assuming that the email for each tuple is unique, names of vendors might not be unique, and phone_numbers might not be unique. Thus we are left with the function dependencies above that are non trivial. Since both functional dependencies have a candidate key on the left side, this relation is in BCNF.

- Consumers (consumer_id, first_name, last_name, phone_number, email)

  F: {consumer_id -> first_name, last_name, phone_number, email; email -> consumer_id, first_name, last_name, phone_number}

  We are assuming that the email for each tuple is unique and phone_numbers might not be unique. Thus we are left with the function dependencies above that are non trivial. Since both functional dependencies have a candidate key on the left side, this relation is in BCNF.

- vendor_reviews (consumer_id, vendor_id, rating)

  F: {consumer_id, vendor_id -> rating}

This is the only dependency function that exists which isn't trivial. Since consumer_id, vendor_id here is a superkey, this relation is in BCNF

- Market_reviews (consumer_id, market_id, rating)

  F: {consumer_id, market_id -> rating}

  This is the only dependency function that exists which isn't trivial. Since consumer_id, market_id here is a superkey, this relation is in BCNF

- Hosts (market_id, vendor_id)

  This relation is clearly in BCNF as both attributes are the primary key and thus the functional dependencies are trivial.

- On_hand (market_id, vendor_id, product_name, price, quantity)

  F: {market_id, vendor_id, product_name, price -> quantity}

  This is the only dependency function that exists which isn't trivial. Since market_id, vendor_id, product_name, and price here is a superkey, this relation is in BCNF

- Products (vendor_id, product_name, price)

  This relation is clearly in BCNF as all three attributes are the primary key and thus the functional dependencies are trivial

Phase 2

I. How we handled the larger dataset

We didn't really run into a lot of problems when working with a bigger dataset. There were only a couple of concerns that we had. Since our database has a lot of relationships between the relations, we needed to decide what to do if we were to delete a foreign key. Would we allow a cascade or just prevent the records from being deleted altogether. We felt that the "right" decision was to allow for the deletions to cascade since we believe that if our database were to be used, then there would be only a select people that would have the authorization to do this and they'll use the privilege wisely. Preventing deletion would be problematic. We also had to decide on updates as well. The only foreign key that we decided to apply an update cascade on was for the product primary keys in the on_hand relation since things like product prices will require updates. We felt that other foreign keys like vendor_id and market_id don't need update cascade since they shouldn't really be touched and don't need to be. We of course tested the delete and update cascade to make sure that it works like how we envisioned it; it worked. Lastly, are costs of our queries. We admit that our queries may not be the best our larger dataset, but we did improve a couple in the third section with techniques like indexing (elaborated on in that section).

II. Test queries and outputs

Query 1: The query ran here is: "SELECT v.vendor_name, COUNT(p.product_name) AS product_count FROM vendors v JOIN products p ON v.vendor_id = p.vendor_id GROUP BY v.vendor_name;". It returns a list of vendors with the number of products each offers. Below is a screenshot of the output.

Data Output | Messages | Notifications

	vendor_name character varying (100)	product_count bigint
1	Unity Valley Homestead	10
2	Riverbend Farm	9
3	Happy Hen Homestead	4
4	Whistling Wind Ranch	3
5	Rich Radish Range	9
6	Currency Carrot Country	5
7	Family Traditions Farmstead	9

Total rows: 850 of 850 Query complete 00:00:00.155

Query 2: The query ran here is: "SELECT v.vendor_name, COUNT(DISTINCT h.market_id) AS market_count FROM vendors v JOIN hosts h ON v.vendor_id = h.vendor_id GROUP BY v.vendor_name HAVING COUNT(DISTINCT h.market_id) > 1;". It returns a list of vendors operating in more than one market. Below is a screenshot of the output.

Data Output | Messages | Notifications

	vendor_name character varying (100)	market_count bigint
1	Affluent Asparagus Acres	2
2	Azure Sky Ranch	2
3	Blossom Hill Acres	2
4	Blue Ridge Farmstead	3
5	Blue Sky Meadows	2
6	Briarwood Farmstead	3
7	Bumblebee Bungalow	2

Total rows: 193 of 193 Query complete 00:00:00.127

Query 3: The query ran here is: "SELECT v.vendor_name, AVG(vr.rating) AS average_rating FROM vendors v JOIN vendor_reviews vr ON v.vendor_id = vr.vendor_id GROUP BY v.vendor_name;". It returns a list of the average rating of each vendor. Below is a screenshot of the output.

Data Output | Messages | Notifications

	vendor_name character varying (100)	average_rating numeric
1	Riverbend Farm	2.7500000000000000
2	Happy Hen Homestead	3.0000000000000000
3	Whistling Wind Ranch	3.3333333333333333
4	Rich Radish Range	1.0000000000000000000
5	Currency Carrot Country	3.3333333333333333
6	Family Traditions Farmstead	3.2500000000000000
7	Eggstatic Homestead	3.2500000000000000

Total rows: 815 of 815 Query complete 00:00:00.146

Query 4: The query ran here is: "SELECT m.market_name, COUNT(h.vendor_id) AS vendor_count FROM markets m JOIN hosts h ON m.market_id = h.market_id GROUP BY m.market_name ORDER BY COUNT(h.vendor_id) DESC;". It returns a list of markets with the highest number of vendors. Below is a screenshot of the output.

Data Output | Messages | Notifications

	market_name character varying (100)	vendor_count bigint
1	Whistling Wind Market	29
2	Sycamore Valley Market	28
3	Birchwood Market	26
4	Hidden Springs Market	24
5	Golden Wheat Market	24
6	Starlight Meadows Market	23
7	Silver Lake Market	17

Total rows: 80 of 80 Query complete 00:00:00.114

Query 5: The query ran here is: "SELECT c.first_name, c.last_name, COUNT(vr.vendor_id) AS ratings_given FROM consumers c JOIN vendor_reviews vr ON c.consumer_id = vr.consumer_id GROUP BY c.consumer_id HAVING COUNT(vr.vendor_id) > 3;". It returns a list of consumers who have rated more than three vendors. Below is a screenshot of the output.

Data Output | Messages | Notifications

	first_name character varying (100)	last_name character varying (100)	ratings_given bigint
1	Toshiko	Jensen	8
2	Cameron	Renaldo	8
3	Kylah	Nicola	6
4	Billye	Lessie	7
5	Tayler	Kaleigh	4
6	Seth	Alivia	8
7	Edd	Marilla	4

Total rows: 399 of 399 Query complete 00:00:00.140

Query 6: The query ran here is: "SELECT p.product_name, p.price FROM products p JOIN on_hand oh ON p.vendor_id = oh.vendor_id AND p.product_name = oh.product_name WHERE oh.market_id = 1;". It returns a list of products available in a specific market (in this case market with ID of 1). Below is a screenshot of the output.

Data Output | Messages | Notifications

	product_name character varying (100)	price numeric (5,2)
1	Almonds	1.11
2	Avocados	2.62
3	Bananas	3.98
4	Cashews	2.30
5	Cilantro	2.33
6	Nectarines	1.96
7	Parsley	4.02

Total rows: 26 of 26 Query complete 00:00:00.116

Query 7: The query ran here is: "SELECT m.market_name FROM markets m JOIN hosts h ON m.market_id = h.market_id WHERE h.vendor_id = 2;". It returns a list of markets where a specific vendor (in this case vendor with ID 2) Sells Products. Below is a screenshot of the output.

Data Output | Messages | Notifications

	market_name character varying (100)
1	Sunset Ridge Market
2	Golden Gables Market

Total rows: 2 of 2 Query complete 00:00:00.180

Query 8: We tested inserting a tuple into the market_rating relation here. For insertions into this relation, we define a trigger that enforces a value constraint on the rating attribute so that only the values from 1 to 5 included can be inserted. The first picture shows how we implemented the trigger. The two queries we used to test this are "INSERT INTO market_reviews VALUES (200, 1, 9);" and "INSERT INTO market_reviews VALUES (200, 1, 3);". As we wanted, the first query results in a customized error being returned as shown in the second picture. The next query goes through and we see a proper insertion with the last picture.

```
/* Trigger: Prevent Adding rating number outside 1-5 for Market Rati
CREATE OR REPLACE function prevent_market_rev()
RETURNS trigger
LANGUAGE plpgsql AS $$
BEGIN
        RAISE EXCEPTION 'rating can only be values between 1-5';
        RETURN NULL;
END;
$$;

CREATE OR REPLACE TRIGGER market_rev_check BEFORE INSERT ON market_r
FOR EACH ROW
WHEN (NEW.rating < 1 OR NEW.rating > 5)
EXECUTE FUNCTION prevent_market_rev();
```

Data Output | Messages | Notifications

```
ERROR:  rating can only be values between 1-5
CONTEXT:  PL/pgSQL function prevent_market_rev() 1

SQL state: P0001
```

Total rows: 0 of 0 | Query complete 00:00:00.130

Data Output | Messages | Notifications

	consumer_id [PK] integer	market_id [PK] integer	rating numeric (1)
6	305	1	4
7	379	1	5
8	495	1	3
9	530	1	4
10	551	1	1
11	668	1	3
12	200	1	3

Total rows: 12 of 12 | Query complete 00:00:00.390

Query 9: For this, we tested deletions from the database. Earlier, we defined that all deletions will be cascaded. In order to test if we did this properly, we deleted all vendors from the vendors relation that had an average rating of 1 with the following command: "DELETE FROM vendors WHERE vendor_id IN (SELECT vendor_id FROM vendor_reviews GROUP BY vendor_id HAVING AVG(rating) = 1);". We then checked if entries with the vendor ids that were deleted appeared in another relation (on_hand). Prior to the deletion, we get rows that contain these vendor ids (shown in first picture). After the deletion, we see that the delete properly cascaded as there are no results (shown in second picture).

Data Output | Messages | Notifications

	market_id [PK] integer	vendor_id [PK] integer	product_name [PK] character varying (100)	price [PK] numeric (5,2)	quantity smallint
1	10	336	Cabbage	4.15	20
2	10	336	Pistachios	1.16	8
3	10	336	Cranberries	1.98	14
4	16	505	Peas	4.78	11
5	16	505	Peaches	4.76	18
6	16	505	Collard greens	5.12	6
7	16	505	Pomegranates	3.66	4

Total rows: 43 of 43 | Query complete 00:00:00.155

Data Output | Messages | Notifications

	market_id [PK] integer	vendor_id [PK] integer	product_name [PK] character varying (100)	price [PK] numeric (5,2)	quantity smallint

Total rows: 0 of 0 | Query complete 00:00:00.128

Query 10: Here we created an update procedure that can be used to update the price of products. It takes in vendor id, product name, and new price as inputs (shown in first picture). We defined earlier that this relation will have an update cascade as users of the database will probably want to update the prices at some point, so it should also be reflected in the relation on_hand. We call this procedure to update vendor 0's price for beets to $3.52 using the command: "CALL update_prod_price(0, 'Beets', 3.52);". The second picture shows that the original price is $3.01. We see in the third picture that after calling the procedure, the price for vendor 0's beets changed to $3.52 and even cascaded as the price is updated for the on_hand relation.

```
/* 15. Update Procedure: Update a particular product's price */
CREATE OR REPLACE PROCEDURE update_prod_price(in v_id INT, in p_name VARCHAR(100), in new_price NUMERIC(5,2))
LANGUAGE SQL
AS $$
UPDATE products
SET price = new_price
WHERE vendor_id = v_id AND product_name LIKE p_name;
$$;
```

Data Output Messages Notifications

	vendor_id [PK] integer	product_name [PK] character varying (100)	price [PK] numeric (5,2)
1	0	Beets	3.01
2	0	Cinnamon	4.49
3	0	Lettuce	3.26
4	0	Pineapples	1.49
5	0	Pomegranates	2.99

Total rows: 5 of 5 Query complete 00:00:00.148

Data Output Messages Notifications

	market_id [PK] integer	vendor_id [PK] integer	product_name [PK] character varying (100)	price [PK] numeric (5,2)	quantity smallint
1	20	0	Beets	3.52	2

Total rows: 1 of 1 Query complete 00:00:00.204

III. Query execution analysis

1. A query that we could improve the cost and timing of is:

```
SELECT *
FROM markets m
JOIN hosts h ON m.market_id = h.market_id
WHERE h.vendor_id = 2;
```

For this query, we improved it by confining what we are selecting. Here, we see that all columns are being returned, but that is not best practice. We should only select columns that we care about as it would make the result clean, and execution efficient. Not selecting all columns improves efficiency as there are less columns/data for the DBMS to move around and return during the query's execution. As can be inferred, we improved the query by altering "SELECT *" to "SELECT m.market_name". Improvement is seen as cost was reduced from 44.01 to 43.89 and time from 0.252 ms to 0.174 ms.

Data Output Messages Explain Notifications

Graphical Analysis Statistics

		Exclusive	Inclusive
1.	→ Hash Inner Join (cost=13.69..16.96 rows=2 width=90) (actual... **Hash Cond**: (m.market_id = h.market_id)	0.043 ms	0.252 ms
2.	→ Seq Scan on public.markets as m (cost=0..3 rows=100 w...	0.042 ms	0.042 ms
3.	→ Hash (cost=13.66..13.66 rows=2 width=8) (actual=0.167.... **Buckets**: 1024 **Batches**: 1 **Memory Usage**: 9 kB	0.007 ms	0.167 ms
4.	→ Seq Scan on public.hosts as h (cost=0..13.66 rows=... **Filter**: (h.vendor_id = 2) **Rows Removed by Filter**: 771	0.161 ms	0.161 ms

Total rows: 1 of 1 Query complete 00:00:00.113

Data Output Messages Explain Notifications

Graphical Analysis Statistics

		Exclusive	Inclusive	Rows X
1.	→ Hash Inner Join (cost=13.69..16.96 rows=2 width=20) (actual... **Hash Cond**: (m.market_id = h.market_id)	0.023 ms	0.174 ms	↑
2.	→ Seq Scan on public.markets as m (cost=0..3 rows=100 w...	0.025 ms	0.025 ms	↑
3.	→ Hash (cost=13.66..13.66 rows=2 width=4) (actual=0.126.... **Buckets**: 1024 **Batches**: 1 **Memory Usage**: 9 kB	0.005 ms	0.126 ms	↑
4.	→ Seq Scan on public.hosts as h (cost=0..13.66 rows=... **Filter**: (h.vendor_id = 2) **Rows Removed by Filter**: 771	0.122 ms	0.122 ms	↑

Total rows: 1 of 1 Query complete 00:00:00.236

2. Another query that we could improve the cost and timing of is:

```
SELECT market_id, vendor_id, product_name, quantity
FROM on_hand
WHERE quantity = 0;
```

For this query, we improved it by adding an index on the column "quantity" for the on_hand relation. This makes things more efficient as the DBMS can search based on quantity, rather than searching through every tuple and filtering with the quantity value. We see that the cost decreases from 47.8 to 10.16 and the time to run it was .088 ms total instead of the original 0.665 ms, a pretty notable improvement. To add indexing, we ran the command: CREATE INDEX index_qty ON on_hand (quantity);

)utput Messages Explain Notifications

ical Analysis Statistics

	Node	Timings	
		Exclusive	Inclusive
1.	→ Seq Scan on public.on_hand as on_hand (cost=0..47.8 rows=1... **Filter**: (on_hand.quantity = 0) **Rows Removed by Filter**: 2281	0.665 ms	0.665 ms

rows: 1 of 1 Query complete 00:00:00.144

Output Messages Explain Notifications

ical Analysis Statistics

	Node	Timings	
		Exclusive	Inclusive
1.	→ Bitmap Heap Scan on public.on_hand as on_hand (cost=5.09..... **Recheck Cond**: (on_hand.quantity = 0) **Heap Blocks**: exact=18	0.064 ms	0.088 ms
2.	→ Bitmap Index Scan using index_qty (cost=0..5.07 rows=10... **Index Cond**: (on_hand.quantity = 0)	0.024 ms	0.024 ms

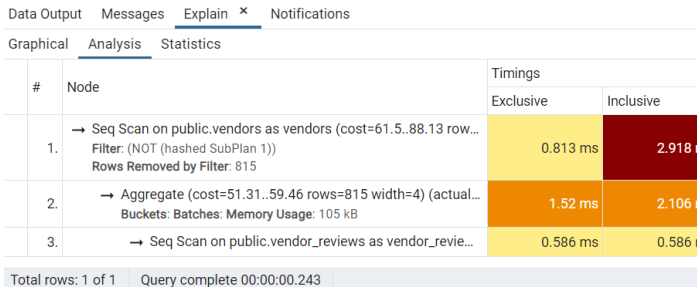
rows: 1 of 1 Query complete 00:00:00.161

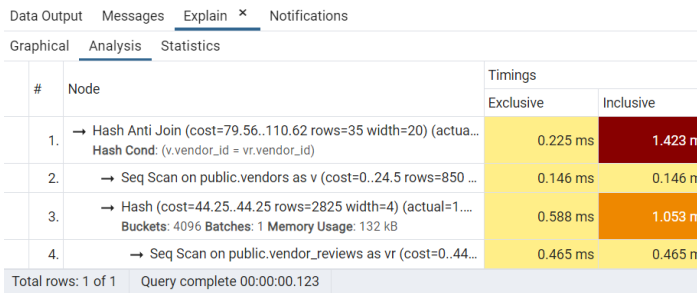
3. A third query that we could improve the cost and timing of is:

```
SELECT vendor_name
FROM vendors
WHERE vendor_id not in(
```

SELECT DISTINCT vendor_id
FROM vendor_reviews);

For this query, we improved it by altering the query to a left join. So the new query is “SELECT v.vendor_name FROM vendors v LEFT JOIN vendor_reviews vr ON v.vendor_id = vr.vendor_id WHERE vr.vendor_id IS NULL;”. This makes things quicker as queries with a join are normally quicker than ones with subqueries. In our case here, the query would first join the table and then run the filter. In our original query, you would access all the elements twice to get the correct tuples (once for the filter and once for the prime table). We see that the time to run it decreases from 2.918 to 1.423, a substantial difference. However, the cost does rise as there are more operations, going from 157.3 to 172.31. This transaction is a bit tricky as there is a tradeoff.

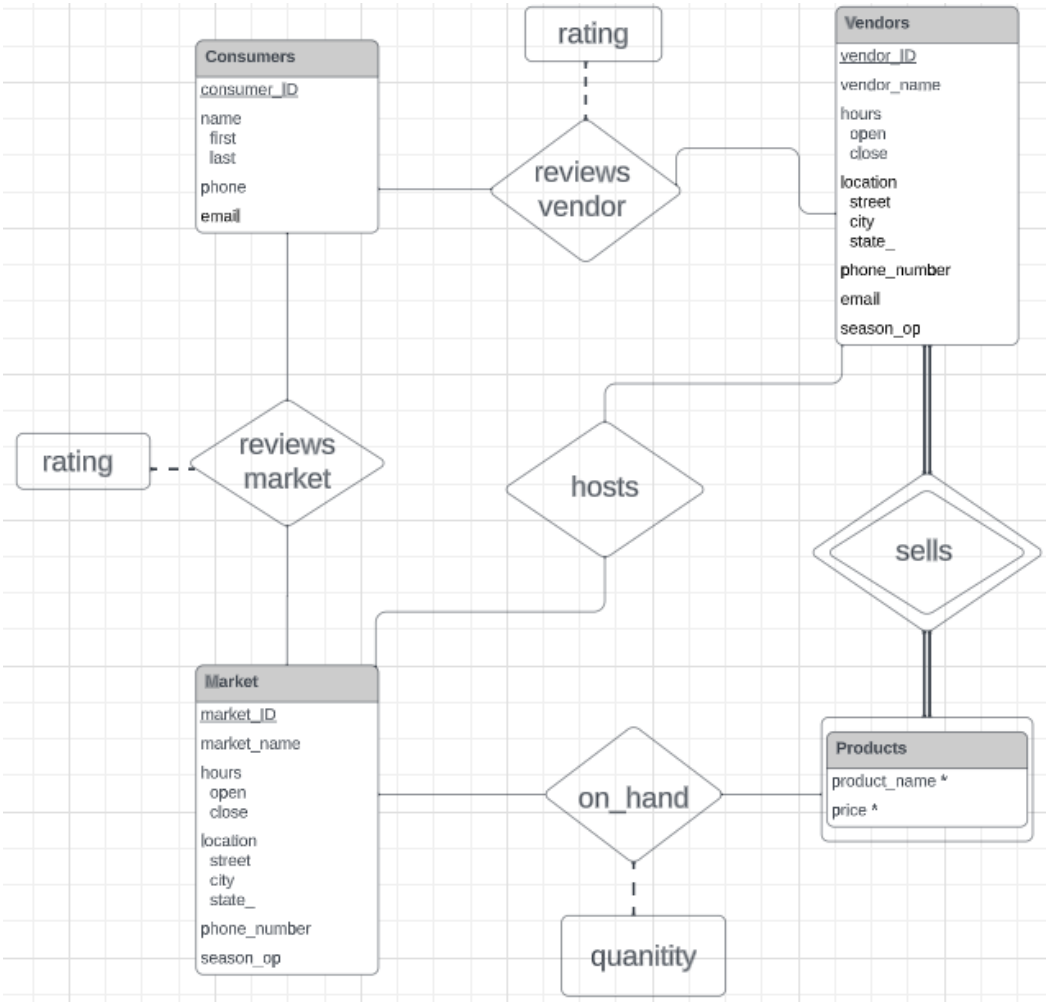




Important note: Instructions to run our webapp for the bonus task is mentioned in our readme.txt

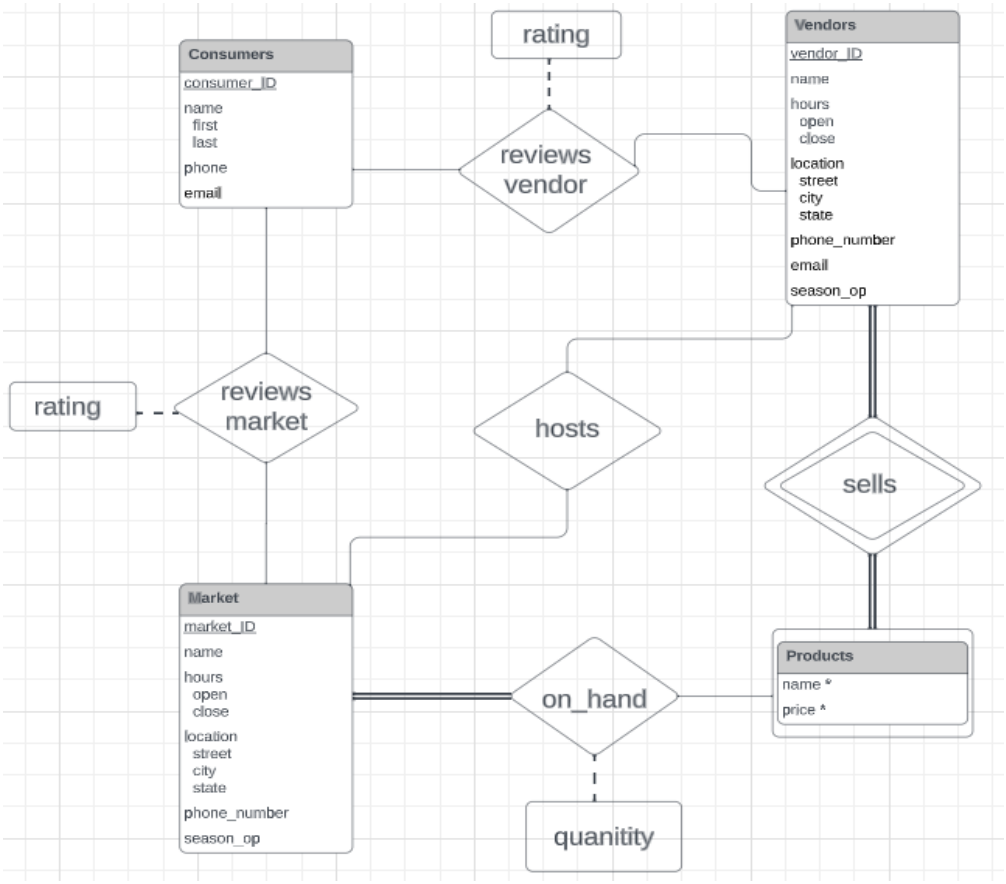
E/R diagram

Updated



Changes: Updated the attribute names to match the new ones mentioned in section 4. Market is no longer fully participating in the on_hand relationship since a market may be under creation and not have any listed or hosted vendors.

Old



Note: The asterisk next to name and price indicates it's a discriminant. Couldn't get dashed underline