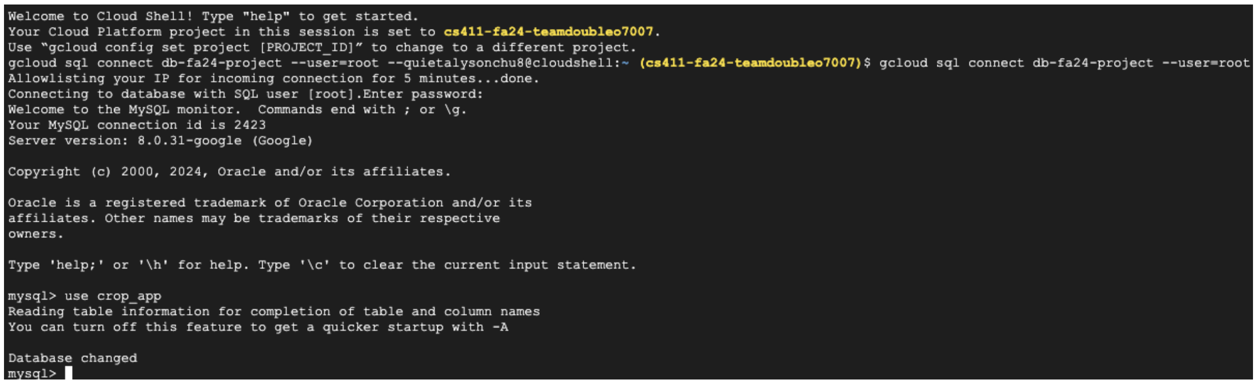
CS 411 Project Stage 3

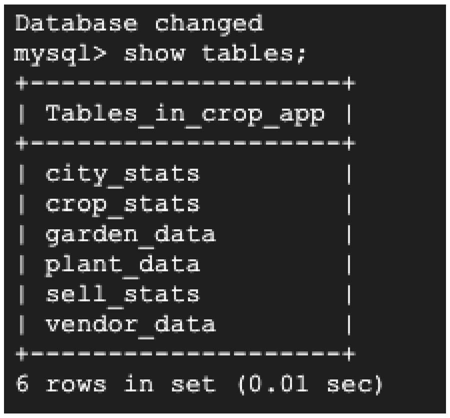
Database Implementation:

GCP Connection:



```
Welcome to Cloud Shell! Type "help" to get started.
Your Cloud Platform project in this session is set to cs411-fa24-teamdoubleo7007.
Use "gcloud config set project [PROJECT_ID]" to change to a different project.
gcloud sql connect db-fa24-project --user=root --quietalysonchu8@cloudshell:~ (cs411-fa24-teamdoubleo7007)$ gcloud sql connect db-fa24-project --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 2423
Server version: 8.0.31-google (Google)

Copyright (c) 2000, 2024, 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> use crop_app
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:

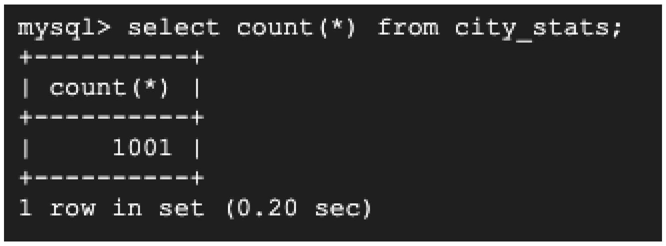


```
Database changed
mysql> show tables;
+--------------------+
| Tables_in_crop_app |
+--------------------+
| city_stats         |
| crop_stats         |
| garden_data        |
| plant_data         |
| sell_stats         |
| vendor_data        |
+--------------------+
6 rows in set (0.01 sec)
```

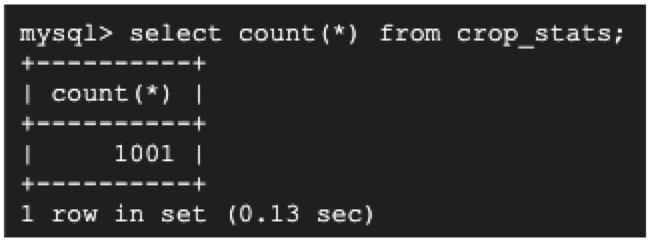
DDL Commands:

```
create table crop_stats (
    crop_name varchar(255) Primary key,
    humidity float,
    temp float,
    soil_ph float,
    nitrogen float,
    phosphorous float,
    potassium float,
    average_rain float
);

create table city_stats(
    state_name varchar(255) Primary key,
    state_name varchar(255),
    temp float,
    soil_ph float,
    nitrogen float,
    phosphorous float,
    potassium float,
    average_rain float,
    humidity float
);

CREATE TABLE sell_stats (
    vendor_name VARCHAR(255),
    crop_name VARCHAR(255),
    price FLOAT,
    PRIMARY KEY (vendor_name, crop_name)
);
```

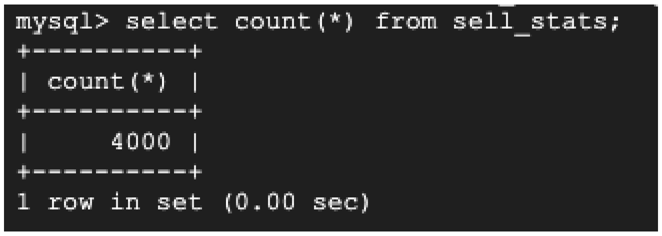
Row Counts:



```
mysql> select count(*) from city_stats;
+----------+
| count(*) |
+----------+
|     1001 |
+----------+
1 row in set (0.20 sec)
```



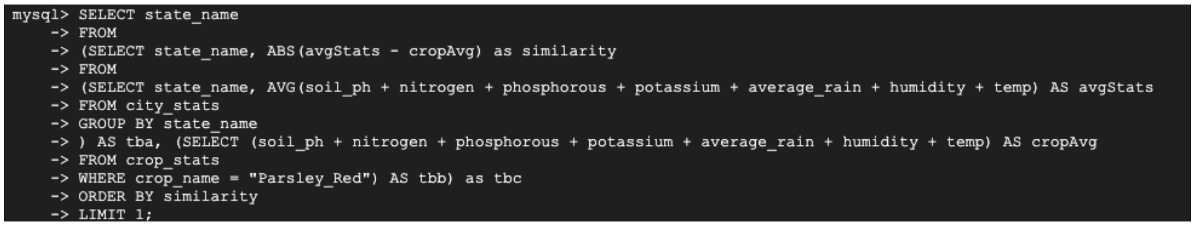
```
mysql> select count(*) from crop_stats;
+----------+
| count(*) |
+----------+
|     1001 |
+----------+
1 row in set (0.13 sec)
```



```
mysql> select count(*) from sell_stats;
+----------+
| count(*) |
+----------+
|     4000 |
+----------+
1 row in set (0.00 sec)
```

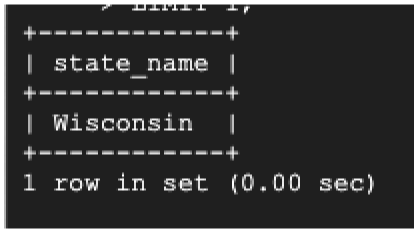
Advanced Query 1 : Select best state given crop (Only results in one row)

Query:



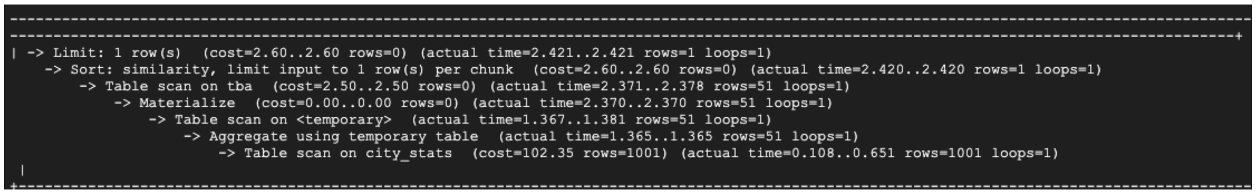
```
mysql> SELECT state_name
    -> FROM
    -> (SELECT state_name, ABS(avgStats - cropAvg) as similarity
    -> FROM
    -> (SELECT state_name, AVG(soil_ph + nitrogen + phosphorous + potassium + average_rain + humidity + temp) AS avgStats
    -> FROM city_stats
    -> GROUP BY state_name
    -> ) AS tba, (SELECT (soil_ph + nitrogen + phosphorous + potassium + average_rain + humidity + temp) AS cropAvg
    -> FROM crop_stats
    -> WHERE crop_name = "Parsley_Red") AS tbb) as tbc
    -> ORDER BY similarity
    -> LIMIT 1;
```

Result:



```
    > LIMIT 1;
+------------+
| state_name |
+------------+
| Wisconsin  |
+------------+
1 row in set (0.00 sec)
```

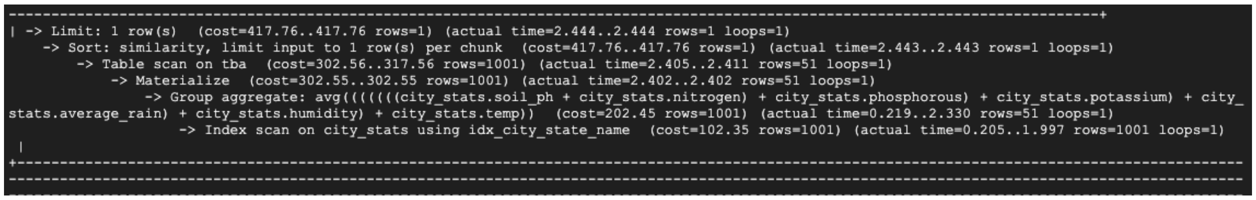
Query Performance Without Indexing: (cost=2.60)



```
----------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------+
| -> Limit: 1 row(s)  (cost=2.60..2.60 rows=0) (actual time=2.421..2.421 rows=1 loops=1)
    -> Sort: similarity, limit input to 1 row(s) per chunk  (cost=2.60..2.60 rows=0) (actual time=2.420..2.420 rows=1 loops=1)
        -> Table scan on tba  (cost=2.50..2.50 rows=0) (actual time=2.371..2.378 rows=51 loops=1)
            -> Materialize  (cost=0.00..0.00 rows=0) (actual time=2.370..2.370 rows=51 loops=1)
                -> Table scan on <temporary>  (actual time=1.367..1.381 rows=51 loops=1)
                    -> Aggregate using temporary table  (actual time=1.365..1.365 rows=51 loops=1)
                        -> Table scan on city_stats  (cost=102.35 rows=1001) (actual time=0.108..0.651 rows=1001 loops=1)
 |
+---------------------------------------------------------------------------------------------------------------------------
```

Index Design 1: (cost=417.76)

CREATE INDEX idx_city_state_name ON city_stats(state_name);



```
-------------------------------------------------------------------------------------------------------------------+
| -> Limit: 1 row(s)  (cost=417.76..417.76 rows=1) (actual time=2.444..2.444 rows=1 loops=1)
    -> Sort: similarity, limit input to 1 row(s) per chunk  (cost=417.76..417.76 rows=1) (actual time=2.443..2.443 rows=1 loops=1)
        -> Table scan on tba  (cost=302.56..317.56 rows=1001) (actual time=2.405..2.411 rows=51 loops=1)
            -> Materialize  (cost=302.55..302.55 rows=1001) (actual time=2.402..2.402 rows=51 loops=1)
                -> Group aggregate: avg((((((city_stats.soil_ph + city_stats.nitrogen) + city_stats.phosphorous) + city_stats.potassium) + city_
stats.average_rain) + city_stats.humidity) + city_stats.temp))  (cost=202.45 rows=1001) (actual time=0.219..2.330 rows=51 loops=1)
                    -> Index scan on city_stats using idx_city_state_name  (cost=102.35 rows=1001) (actual time=0.205..1.997 rows=1001 loops=1)
 |
+---------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------
```

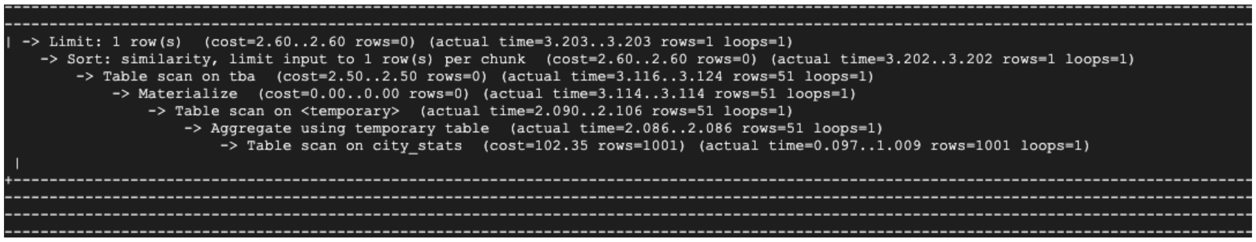
Indexing on state_name had a higher cost of 417.76 than no indexing (2.60). This is surprising since the state name appears multiple times.

Index Design 2: (cost=2.60)

CREATE INDEX idx_city_temp ON city_stats(temp);

CREATE INDEX idx_city_soil_ph ON city_stats(soil_ph);

CREATE INDEX idx_city_nitrogen ON city_stats(nitrogen);



```
----------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------
| -> Limit: 1 row(s)  (cost=2.60..2.60 rows=0) (actual time=3.203..3.203 rows=1 loops=1)
    -> Sort: similarity, limit input to 1 row(s) per chunk  (cost=2.60..2.60 rows=0) (actual time=3.202..3.202 rows=1 loops=1)
        -> Table scan on tba  (cost=2.50..2.50 rows=0) (actual time=3.116..3.124 rows=51 loops=1)
            -> Materialize  (cost=0.00..0.00 rows=0) (actual time=3.114..3.114 rows=51 loops=1)
                -> Table scan on <temporary>  (actual time=2.090..2.106 rows=51 loops=1)
                    -> Aggregate using temporary table  (actual time=2.086..2.086 rows=51 loops=1)
                        -> Table scan on city_stats  (cost=102.35 rows=1001) (actual time=0.097..1.009 rows=1001 loops=1)
 |
+---------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------
```

Indexing on the city temp, soil_ph, and nitrogen had an overall cost of 2.60, similar to the original cost.

Index Design 3:

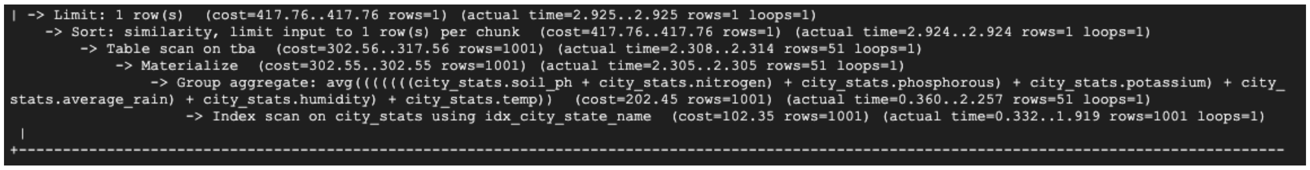
CREATE INDEX idx_city_state_name ON city_stats(state_name);

CREATE INDEX idx_city_temp ON city_stats(temp);

CREATE INDEX idx_city_soil_ph ON city_stats(soil_ph);

CREATE INDEX idx_crop_temp ON crop_stats(temp);

CREATE INDEX idx_crop_soil_ph ON crio_stats(soil_ph);

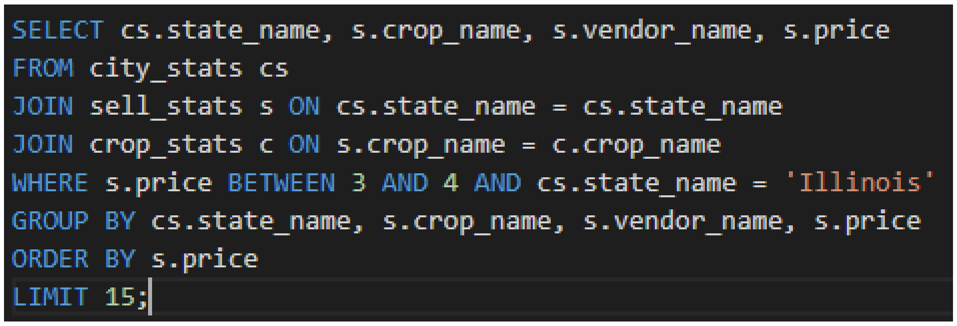


```
| -> Limit: 1 row(s)  (cost=417.76..417.76 rows=1) (actual time=2.925..2.925 rows=1 loops=1)
    -> Sort: similarity, limit input to 1 row(s) per chunk  (cost=417.76..417.76 rows=1) (actual time=2.924..2.924 rows=1 loops=1)
        -> Table scan on tba  (cost=302.56..317.56 rows=1001) (actual time=2.308..2.314 rows=51 loops=1)
            -> Materialize  (cost=302.55..302.55 rows=1001) (actual time=2.305..2.305 rows=51 loops=1)
                -> Group aggregate: avg((((((city_stats.soil_ph + city_stats.nitrogen) + city_stats.phosphorous) + city_stats.potassium) + city_
stats.average_rain) + city_stats.humidity) + city_stats.temp))  (cost=202.45 rows=1001) (actual time=0.360..2.257 rows=51 loops=1)
                    -> Index scan on city_stats using idx_city_state_name  (cost=102.35 rows=1001) (actual time=0.332..1.919 rows=1001 loops=1)
 |
+---------------------------------------------------------------------------------------------------------------------------
```

Indexing on growth attributes from the two tables referenced in the query and the state_name in the city_stats table had a cost of 417.76 as well.
Final Index Design:
No additional indices. It appears that indexing did not change or increased the cost. Indexing on growth stats in the city_stats and crop_stats did not do much because the query does not really search by these fields, and instead sums them into new attributes. It could be that indexing by state_name was not effective because the query does not use it to search or sort. Since the query only sorts based on a newly created attribute within the query, indexing does not really improve performance.

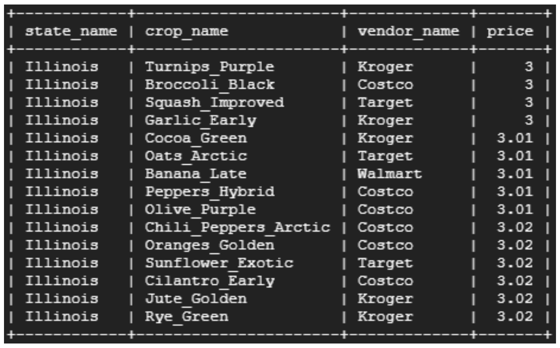
Advanced Query 2: Listing vendors that offer crops within a specific price range in a specific state

Query:



```sql
SELECT cs.state_name, s.crop_name, s.vendor_name, s.price
FROM city_stats cs
JOIN sell_stats s ON cs.state_name = cs.state_name
JOIN crop_stats c ON s.crop_name = c.crop_name
WHERE s.price BETWEEN 3 AND 4 AND cs.state_name = 'Illinois'
GROUP BY cs.state_name, s.crop_name, s.vendor_name, s.price
ORDER BY s.price
LIMIT 15;
```

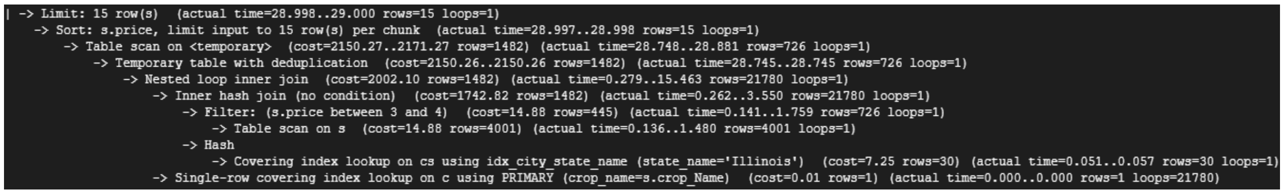
Result:



```
+------------+---------------------+-------------+-------+
| state_name | crop_name           | vendor_name | price |
+------------+---------------------+-------------+-------+
| Illinois   | Turnips_Purple      | Kroger      |     3 |
| Illinois   | Broccoli_Black      | Costco      |     3 |
| Illinois   | Squash_Improved     | Target      |     3 |
| Illinois   | Garlic_Early        | Kroger      |     3 |
| Illinois   | Cocoa_Green         | Kroger      |  3.01 |
| Illinois   | Oats_Arctic         | Target      |  3.01 |
| Illinois   | Banana_Late         | Walmart     |  3.01 |
| Illinois   | Peppers_Hybrid      | Costco      |  3.01 |
| Illinois   | Olive_Purple        | Costco      |  3.01 |
| Illinois   | Chili_Peppers_Arctic | Costco     |  3.02 |
| Illinois   | Oranges_Golden      | Costco      |  3.02 |
| Illinois   | Sunflower_Exotic    | Target      |  3.02 |
| Illinois   | Cilantro_Early      | Costco      |  3.02 |
| Illinois   | Jute_Golden         | Kroger      |  3.02 |
| Illinois   | Rye_Green           | Kroger      |  3.02 |
+------------+---------------------+-------------+-------+
```

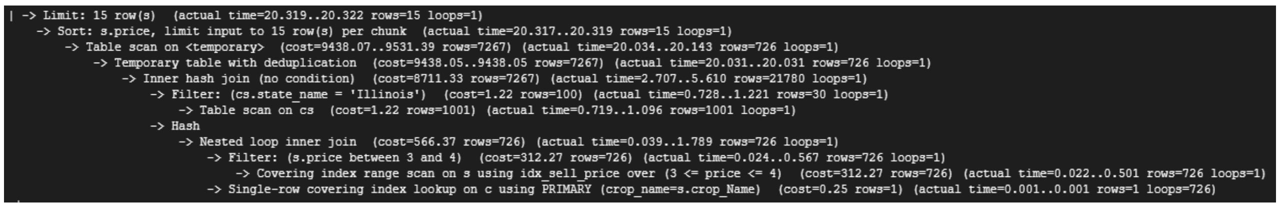
Indexing Designed Tried:
CREATE INDEX idx_city_state_name ON city_stats(state_name);
CREATE INDEX idx_sell_price ON sell_stats(price);
CREATE INDEX idx_crop_sell ON sell_stats(crop_name);
Index analysis report:

There was a significant decrease in the cost for filtering the rows in city_stats after the first index. The cost dropped from 102.35 to 7.25. The cost of the temporary table was also reduced significantly, from 6152.21 to 2150.26



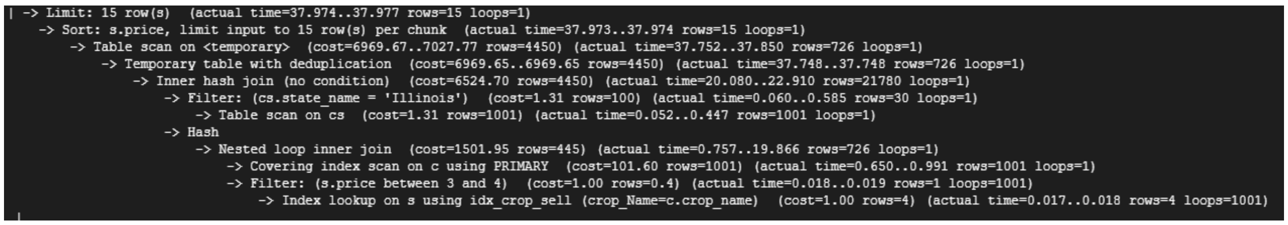
```
| -> Limit: 15 row(s)  (actual time=28.998..29.000 rows=15 loops=1)
    -> Sort: s.price, limit input to 15 row(s) per chunk  (actual time=28.997..28.998 rows=15 loops=1)
        -> Table scan on <temporary>  (cost=2150.27..2171.27 rows=1482) (actual time=28.748..28.881 rows=726 loops=1)
            -> Temporary table with deduplication  (cost=2150.26..2150.26 rows=1482) (actual time=28.745..28.745 rows=726 loops=1)
                -> Nested loop inner join  (cost=2002.10 rows=1482) (actual time=0.279..15.463 rows=21780 loops=1)
                    -> Inner hash join (no condition)  (cost=1742.82 rows=1482) (actual time=0.262..3.550 rows=21780 loops=1)
                        -> Filter: (s.price between 3 and 4)  (cost=14.88 rows=445) (actual time=0.141..1.759 rows=726 loops=1)
                            -> Table scan on s  (cost=14.88 rows=4001) (actual time=0.136..1.480 rows=4001 loops=1)
                        -> Hash
                            -> Covering index lookup on cs using idx_city_state_name (state_name='Illinois')  (cost=7.25 rows=30) (actual time=0.051..0.057 rows=30 loops=1)
                    -> Single-row covering index lookup on c using PRIMARY (crop_name=s.crop_Name)  (cost=0.01 rows=1) (actual time=0.000..0.000 rows=1 loops=21780)
```

After trying the second index in conjunction with the first, there was a big increase in the total cost, jumping from around 2000 to 13000. After trying it individually, we see that the cost is around 9000.



```
| -> Limit: 15 row(s)  (actual time=20.319..20.322 rows=15 loops=1)
    -> Sort: s.price, limit input to 15 row(s) per chunk  (actual time=20.317..20.319 rows=15 loops=1)
        -> Table scan on <temporary>  (cost=9438.07..9531.39 rows=7267) (actual time=20.034..20.143 rows=726 loops=1)
            -> Temporary table with deduplication  (cost=9438.05..9438.05 rows=7267) (actual time=20.031..20.031 rows=726 loops=1)
                -> Inner hash join (no condition)  (cost=8711.33 rows=7267) (actual time=2.707..5.610 rows=21780 loops=1)
                    -> Filter: (cs.state_name = 'Illinois')  (cost=1.22 rows=100) (actual time=0.728..1.221 rows=30 loops=1)
                        -> Table scan on cs  (cost=1.22 rows=1001) (actual time=0.719..1.096 rows=1001 loops=1)
                    -> Hash
                        -> Nested loop inner join  (cost=566.37 rows=726) (actual time=0.039..1.789 rows=726 loops=1)
                            -> Filter: (s.price between 3 and 4)  (cost=312.27 rows=726) (actual time=0.024..0.567 rows=726 loops=1)
                                -> Covering index range scan on s using idx_sell_price over (3 <= price <= 4)  (cost=312.27 rows=726) (actual time=0.022..0.501 rows=726 loops=1)
                            -> Single-row covering index lookup on c using PRIMARY (crop_name=s.crop_Name)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=726)
```

After trying the third index. The cost was around 7000. Trying it in tandem with the other two never decreased the cost from only using the first index.

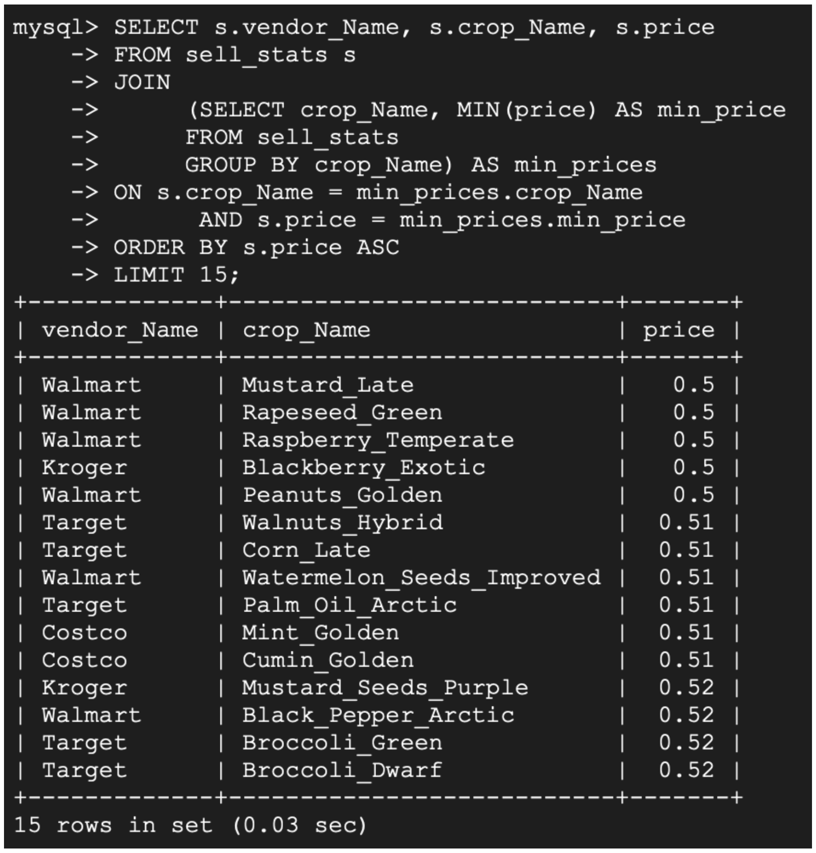


```
| -> Limit: 15 row(s)  (actual time=37.974..37.977 rows=15 loops=1)
    -> Sort: s.price, limit input to 15 row(s) per chunk  (actual time=37.973..37.974 rows=15 loops=1)
        -> Table scan on <temporary>  (cost=6969.67..7027.77 rows=4450) (actual time=37.752..37.850 rows=726 loops=1)
            -> Temporary table with deduplication  (cost=6969.65..6969.65 rows=4450) (actual time=37.748..37.748 rows=726 loops=1)
                -> Inner hash join (no condition)  (cost=6524.70 rows=4450) (actual time=20.080..22.910 rows=21780 loops=1)
                    -> Filter: (cs.state_name = 'Illinois')  (cost=1.31 rows=100) (actual time=0.060..0.585 rows=30 loops=1)
                        -> Table scan on cs  (cost=1.31 rows=1001) (actual time=0.052..0.447 rows=1001 loops=1)
                    -> Hash
                        -> Nested loop inner join  (cost=1501.95 rows=445) (actual time=0.757..19.866 rows=726 loops=1)
                            -> Covering index scan on c using PRIMARY  (cost=101.60 rows=1001) (actual time=0.650..0.991 rows=1001 loops=1)
                            -> Filter: (s.price between 3 and 4)  (cost=1.00 rows=0.4) (actual time=0.018..0.019 rows=1 loops=1001)
                                -> Index lookup on s using idx_crop_sell (crop_Name=c.crop_name)  (cost=1.00 rows=4) (actual time=0.017..0.018 rows=4 loops=1001)
```

Final Index Design:
My final index design was to simply use CREATE INDEX idx_city_state_name ON city_stats(state_name);
Reason for Choice:
Of the various combinations of index choices I tried, it seemed like simply doing the first index was the most efficient, as it decreased the cost the most, and when implemented with other index choices, it would only increase the cost. Thus, I decided to only use the first index as my final index design.

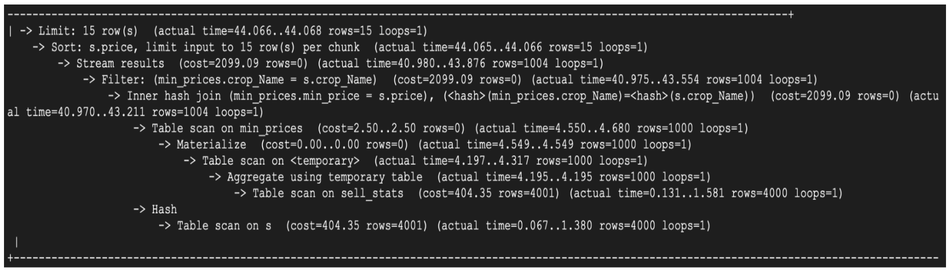
Advanced Query 3 : Select best vendors per crop based on price

Result / Query:



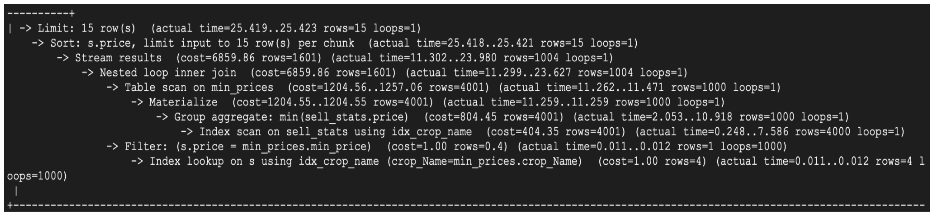
```
mysql> SELECT s.vendor_Name, s.crop_Name, s.price
    -> FROM sell_stats s
    -> JOIN
    ->      (SELECT crop_Name, MIN(price) AS min_price
    ->      FROM sell_stats
    ->      GROUP BY crop_Name) AS min_prices
    -> ON s.crop_Name = min_prices.crop_Name
    ->       AND s.price = min_prices.min_price
    -> ORDER BY s.price ASC
    -> LIMIT 15;
+-------------+---------------------------+-------+
| vendor_Name | crop_Name                 | price |
+-------------+---------------------------+-------+
| Walmart     | Mustard_Late              |   0.5 |
| Walmart     | Rapeseed_Green            |   0.5 |
| Walmart     | Raspberry_Temperate       |   0.5 |
| Kroger      | Blackberry_Exotic         |   0.5 |
| Walmart     | Peanuts_Golden            |   0.5 |
| Target      | Walnuts_Hybrid            |  0.51 |
| Target      | Corn_Late                 |  0.51 |
| Walmart     | Watermelon_Seeds_Improved |  0.51 |
| Target      | Palm_Oil_Arctic           |  0.51 |
| Costco      | Mint_Golden               |  0.51 |
| Costco      | Cumin_Golden              |  0.51 |
| Kroger      | Mustard_Seeds_Purple      |  0.52 |
| Walmart     | Black_Pepper_Arctic       |  0.52 |
| Target      | Broccoli_Green            |  0.52 |
| Target      | Broccoli_Dwarf            |  0.52 |
+-------------+---------------------------+-------+
15 rows in set (0.03 sec)
```

Indexes tried:
No indexing
CREATE INDEX idx_crop_name ON sell_stats(crop_Name);
CREATE INDEX idx_price ON sell_stats(price);
CREATE INDEX idx_crop_price ON sell_stats(crop_Name, price);
No indexing:



```
-----------------------------------------------------------------------------------------------------------------------+
| -> Limit: 15 row(s)  (actual time=44.066..44.068 rows=15 loops=1)
    -> Sort: s.price, limit input to 15 row(s) per chunk  (actual time=44.065..44.066 rows=15 loops=1)
        -> Stream results  (cost=2099.09 rows=0) (actual time=40.980..43.876 rows=1004 loops=1)
            -> Filter: (min_prices.crop_Name = s.crop_Name)  (cost=2099.09 rows=0) (actual time=40.975..43.554 rows=1004 loops=1)
                -> Inner hash join (min_prices.min_price = s.price), (<hash>(min_prices.crop_Name)=<hash>(s.crop_Name))  (cost=2099.09 rows=0) (actu
al time=40.970..43.211 rows=1004 loops=1)
                    -> Table scan on min_prices  (cost=2.50..2.50 rows=0) (actual time=4.550..4.680 rows=1000 loops=1)
                        -> Materialize  (cost=0.00..0.00 rows=0) (actual time=4.549..4.549 rows=1000 loops=1)
                            -> Table scan on <temporary>  (actual time=4.197..4.317 rows=1000 loops=1)
                                -> Aggregate using temporary table  (actual time=4.195..4.195 rows=1000 loops=1)
                                    -> Table scan on sell_stats  (cost=404.35 rows=4001) (actual time=0.131..1.581 rows=4000 loops=1)
                    -> Hash
                        -> Table scan on s  (cost=404.35 rows=4001) (actual time=0.067..1.380 rows=4000 loops=1)
 |
+-----------------------------------------------------------------------------------------------------------------------------------
```

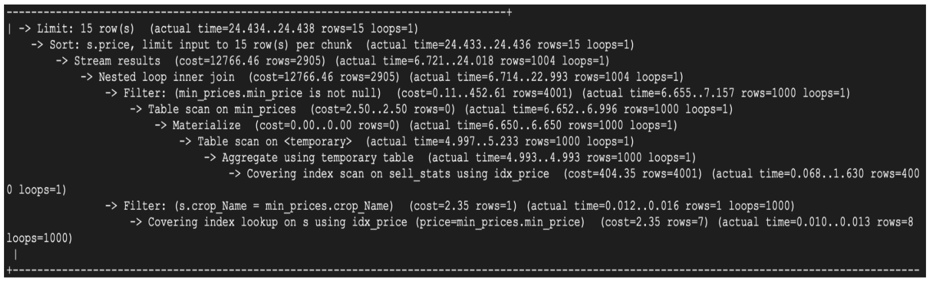
The query made full table scans. It had low performance because no indexing was applied, leading to inefficient joins and filtering. Cost was 2099.09.
Index on crop_name:



```
----------+
| -> Limit: 15 row(s)  (actual time=25.419..25.423 rows=15 loops=1)
    -> Sort: s.price, limit input to 15 row(s) per chunk  (actual time=25.418..25.421 rows=15 loops=1)
        -> Stream results  (cost=6859.86 rows=1601) (actual time=11.302..23.980 rows=1004 loops=1)
            -> Nested loop inner join  (cost=6859.86 rows=1601) (actual time=11.299..23.627 rows=1004 loops=1)
                -> Table scan on min_prices  (cost=1204.56..1257.06 rows=4001) (actual time=11.262..11.471 rows=1000 loops=1)
                    -> Materialize  (cost=1204.55..1204.55 rows=4001) (actual time=11.259..11.259 rows=1000 loops=1)
                        -> Group aggregate: min(sell_stats.price)  (cost=804.45 rows=4001) (actual time=2.053..10.918 rows=1000 loops=1)
                            -> Index scan on sell_stats using idx_crop_name  (cost=404.35 rows=4001) (actual time=0.248..7.586 rows=4000 loops=1)
                -> Filter: (s.price = min_prices.min_price)  (cost=1.00 rows=0.4) (actual time=0.011..0.012 rows=1 loops=1000)
                    -> Index lookup on s using idx_crop_name (crop_Name=min_prices.crop_Name)  (cost=1.00 rows=4) (actual time=0.011..0.012 rows=4 l
oops=1000)
 |
+-----------------------------------------------------------------------------------------------------------------------------------------------------
```

Doesn’t need a full table scan on crop_name anymore, which reduces the cost of the join. Overall cost jumped to 6859.86 though.

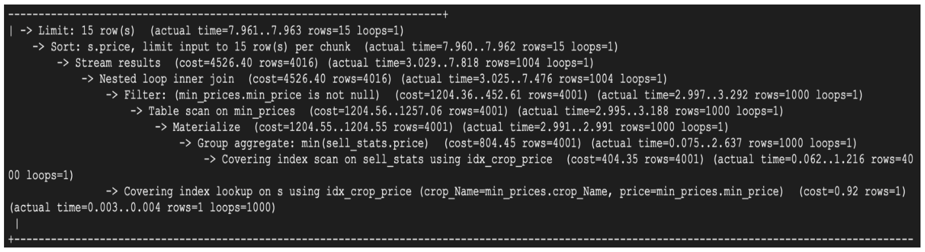
Index on price:



```
------------------------------------------------------------------------------+
| -> Limit: 15 row(s)  (actual time=24.434..24.438 rows=15 loops=1)
    -> Sort: s.price, limit input to 15 row(s) per chunk  (actual time=24.433..24.436 rows=15 loops=1)
        -> Stream results  (cost=12766.46 rows=2905) (actual time=6.721..24.018 rows=1004 loops=1)
            -> Nested loop inner join  (cost=12766.46 rows=2905) (actual time=6.714..22.993 rows=1004 loops=1)
                -> Filter: (min_prices.min_price is not null)  (cost=0.11..452.61 rows=4001) (actual time=6.655..7.157 rows=1000 loops=1)
                    -> Table scan on min_prices  (cost=2.50..2.50 rows=0) (actual time=6.652..6.996 rows=1000 loops=1)
                        -> Materialize  (cost=0.00..0.00 rows=0) (actual time=6.650..6.650 rows=1000 loops=1)
                            -> Table scan on <temporary>  (actual time=4.997..5.233 rows=1000 loops=1)
                                -> Aggregate using temporary table  (actual time=4.993..4.993 rows=1000 loops=1)
                                    -> Covering index scan on sell_stats using idx_price  (cost=404.35 rows=4001) (actual time=0.068..1.630 rows=400
0 loops=1)
                -> Filter: (s.crop_Name = min_prices.crop_Name)  (cost=2.35 rows=1) (actual time=0.012..0.016 rows=1 loops=1000)
                    -> Covering index lookup on s using idx_price (price=min_prices.min_price)  (cost=2.35 rows=7) (actual time=0.010..0.013 rows=8
loops=1000)
 |
+-----------------------------------------------------------------------------------------------------------------------------------------------------
```

Better performance on sorting and filtering by price, which reduced the time needed for those operations. Overall cost was 12766.46 which is nearly double the cost of indexing on crop_name alone.

Index on crop_name and price:



```
---------------------------------------------------------------------+
| -> Limit: 15 row(s)  (actual time=7.961..7.963 rows=15 loops=1)
    -> Sort: s.price, limit input to 15 row(s) per chunk  (actual time=7.960..7.962 rows=15 loops=1)
        -> Stream results  (cost=4526.40 rows=4016) (actual time=3.029..7.818 rows=1004 loops=1)
            -> Nested loop inner join  (cost=4526.40 rows=4016) (actual time=3.025..7.476 rows=1004 loops=1)
                -> Filter: (min_prices.min_price is not null)  (cost=1204.36..452.61 rows=4001) (actual time=2.997..3.292 rows=1000 loops=1)
                    -> Table scan on min_prices  (cost=1204.56..1257.06 rows=4001) (actual time=2.995..3.188 rows=1000 loops=1)
                        -> Materialize  (cost=1204.55..1204.55 rows=4001) (actual time=2.991..2.991 rows=1000 loops=1)
                            -> Group aggregate: min(sell_stats.price)  (cost=804.45 rows=4001) (actual time=0.075..2.637 rows=1000 loops=1)
                                -> Covering index scan on sell_stats using idx_crop_price  (cost=404.35 rows=4001) (actual time=0.062..1.216 rows=40
00 loops=1)
                -> Covering index lookup on s using idx_crop_price (crop_Name=min_prices.crop_Name, price=min_prices.min_price)  (cost=0.92 rows=1)
(actual time=0.003..0.004 rows=1 loops=1000)
 |
+-----------------------------------------------------------------------------------------------------------------------------------------------------
```

Overall cost dropped to 4526.4 which is much better than other indexing options.

Final choice:

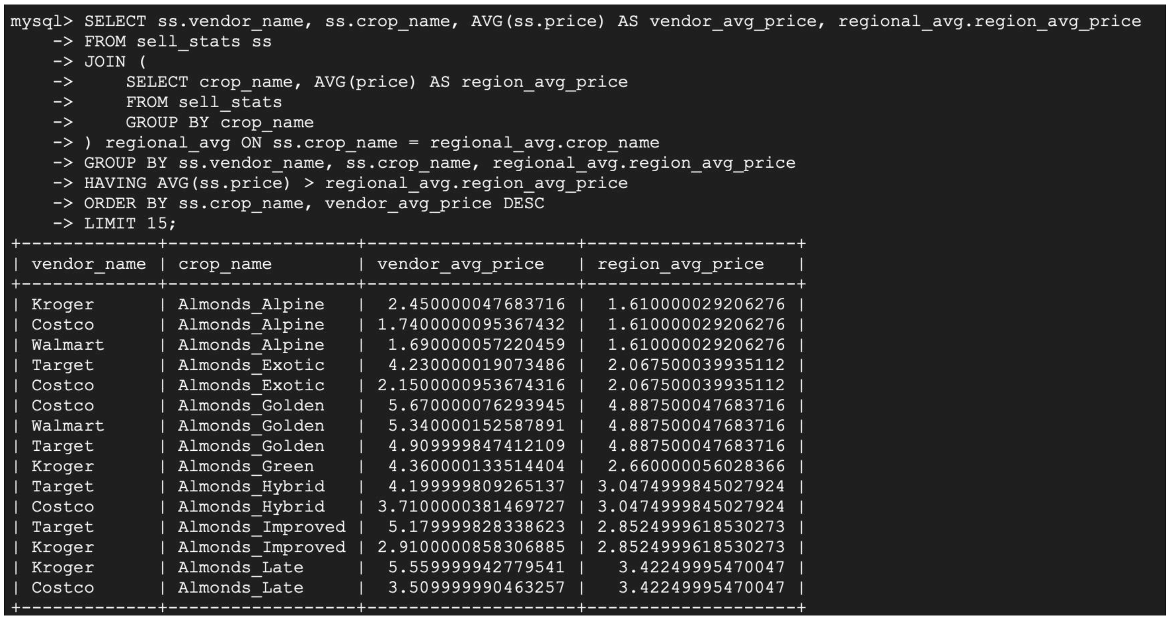
CREATE INDEX idx_crop_price ON sell_stats(crop_Name, price);

Reason:

While no indexing has an even lower cost, it needs to do full table scans for each query which can be really inefficient on large tables. I think indexing on both crop_name and price is the way to go here since it has the lowest cost among all indexing options that don't require a full table search.

Advanced Query 4 : Find vendors who sell crops at an average price higher than the average price of that crop across all vendors

Result / Query:

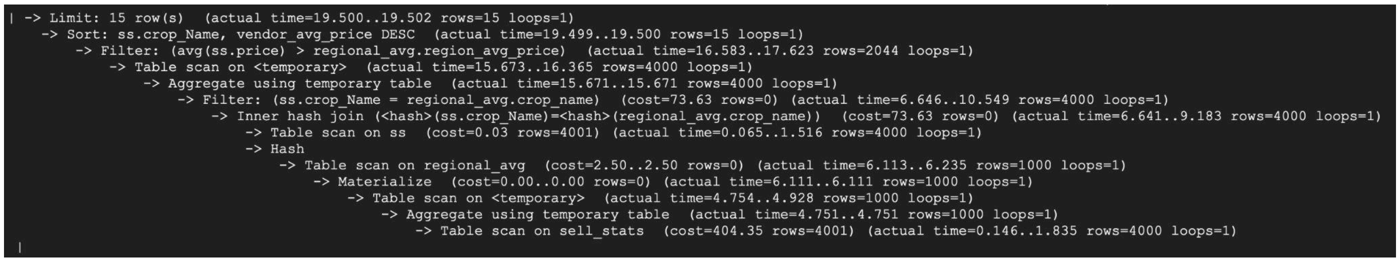


```
mysql> SELECT ss.vendor_name, ss.crop_name, AVG(ss.price) AS vendor_avg_price, regional_avg.region_avg_price
    -> FROM sell_stats ss
    -> JOIN (
    ->     SELECT crop_name, AVG(price) AS region_avg_price
    ->     FROM sell_stats
    ->     GROUP BY crop_name
    -> ) regional_avg ON ss.crop_name = regional_avg.crop_name
    -> GROUP BY ss.vendor_name, ss.crop_name, regional_avg.region_avg_price
    -> HAVING AVG(ss.price) > regional_avg.region_avg_price
    -> ORDER BY ss.crop_name, vendor_avg_price DESC
    -> LIMIT 15;
+-------------+------------------+--------------------+--------------------+
| vendor_name | crop_name        | vendor_avg_price   | region_avg_price   |
+-------------+------------------+--------------------+--------------------+
| Kroger      | Almonds_Alpine   |  2.450000047683716 |  1.610000029206276 |
| Costco      | Almonds_Alpine   | 1.7400000095367432 |  1.610000029206276 |
| Walmart     | Almonds_Alpine   |  1.690000057220459 |  1.610000029206276 |
| Target      | Almonds_Exotic   |  4.230000019073486 |  2.067500039935112 |
| Costco      | Almonds_Exotic   | 2.1500000953674316 |  2.067500039935112 |
| Costco      | Almonds_Golden   |  5.670000076293945 |  4.887500047683716 |
| Walmart     | Almonds_Golden   |  5.340000152587891 |  4.887500047683716 |
| Target      | Almonds_Golden   |  4.909999847412109 |  4.887500047683716 |
| Kroger      | Almonds_Green    |  4.360000133514404 |  2.660000056028366 |
| Target      | Almonds_Hybrid   |  4.199999809265137 | 3.0474999845027924 |
| Costco      | Almonds_Hybrid   | 3.7100000381469727 | 3.0474999845027924 |
| Target      | Almonds_Improved |  5.179999828338623 | 2.8524999618530273 |
| Kroger      | Almonds_Improved | 2.9100000858306885 | 2.8524999618530273 |
| Kroger      | Almonds_Late     |  5.559999942779541 |   3.42249995470047 |
| Costco      | Almonds_Late     |  3.509999990463257 |   3.42249995470047 |
+-------------+------------------+--------------------+--------------------+
```

Indexes tried:

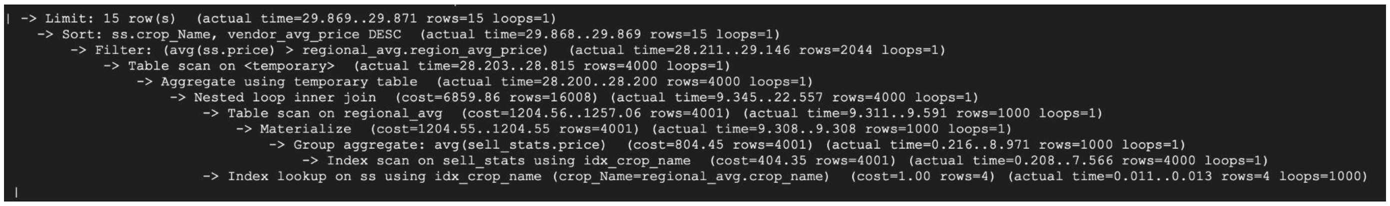
No indexing
CREATE INDEX idx_crop_name ON sell_stats(crop_name);
CREATE INDEX idx_sell_price ON sell_stats(price);
CREATE INDEX idx_vendor_name ON sell_stats(vendor_name);

No indexing:



```
| -> Limit: 15 row(s)  (actual time=19.500..19.502 rows=15 loops=1)
    -> Sort: ss.crop_Name, vendor_avg_price DESC  (actual time=19.499..19.500 rows=15 loops=1)
        -> Filter: (avg(ss.price) > regional_avg.region_avg_price)  (actual time=16.583..17.623 rows=2044 loops=1)
            -> Table scan on <temporary>  (actual time=15.673..16.365 rows=4000 loops=1)
                -> Aggregate using temporary table  (actual time=15.671..15.671 rows=4000 loops=1)
                    -> Filter: (ss.crop_Name = regional_avg.crop_name)  (cost=73.63 rows=0) (actual time=6.646..10.549 rows=4000 loops=1)
                        -> Inner hash join (<hash>(ss.crop_Name)=<hash>(regional_avg.crop_name))  (cost=73.63 rows=0) (actual time=6.641..9.183 rows=4000 loops=1)
                            -> Table scan on ss  (cost=0.03 rows=4001) (actual time=0.065..1.516 rows=4000 loops=1)
                            -> Hash
                                -> Table scan on regional_avg  (cost=2.50..2.50 rows=0) (actual time=6.113..6.235 rows=1000 loops=1)
                                    -> Materialize  (cost=0.00..0.00 rows=0) (actual time=6.111..6.111 rows=1000 loops=1)
                                        -> Table scan on <temporary>  (actual time=4.754..4.928 rows=1000 loops=1)
                                            -> Aggregate using temporary table  (actual time=4.751..4.751 rows=1000 loops=1)
                                                -> Table scan on sell_stats  (cost=404.35 rows=4001) (actual time=0.146..1.835 rows=4000 loops=1)
 |
```

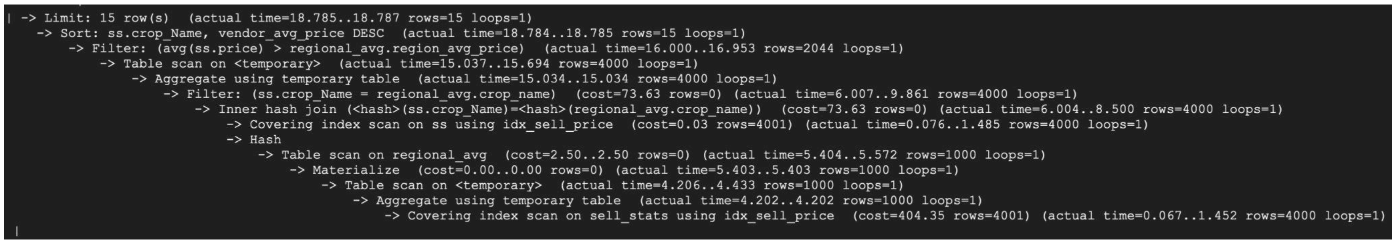
Index on crop_name:



```
| -> Limit: 15 row(s)  (actual time=29.869..29.871 rows=15 loops=1)
    -> Sort: ss.crop_Name, vendor_avg_price DESC  (actual time=29.868..29.869 rows=15 loops=1)
        -> Filter: (avg(ss.price) > regional_avg.region_avg_price)  (actual time=28.211..29.146 rows=2044 loops=1)
            -> Table scan on <temporary>  (actual time=28.203..28.815 rows=4000 loops=1)
                -> Aggregate using temporary table  (actual time=28.200..28.200 rows=4000 loops=1)
                    -> Nested loop inner join  (cost=6859.86 rows=16008) (actual time=9.345..22.557 rows=4000 loops=1)
                        -> Table scan on regional_avg  (cost=1204.56..1257.06 rows=4001) (actual time=9.311..9.591 rows=1000 loops=1)
                            -> Materialize  (cost=1204.55..1204.55 rows=4001) (actual time=9.308..9.308 rows=1000 loops=1)
                                -> Group aggregate: avg(sell_stats.price)  (cost=804.45 rows=4001) (actual time=0.216..8.971 rows=1000 loops=1)
                                    -> Index scan on sell_stats using idx_crop_name  (cost=404.35 rows=4001) (actual time=0.208..7.566 rows=4000 loops=1)
                        -> Index lookup on ss using idx_crop_name (crop_Name=regional_avg.crop_name)  (cost=1.00 rows=4) (actual time=0.011..0.013 rows=4 loops=1000)
 |
```

Indexing on crop_name significantly increased the cost of the query. The cost for the table scan on regional average increased from 2.5..2.5 to 1204.56..1257.06. Materialize also increased to 1204.55..1204.55. This shows that indexing on crop_name is not a good choice for this query.

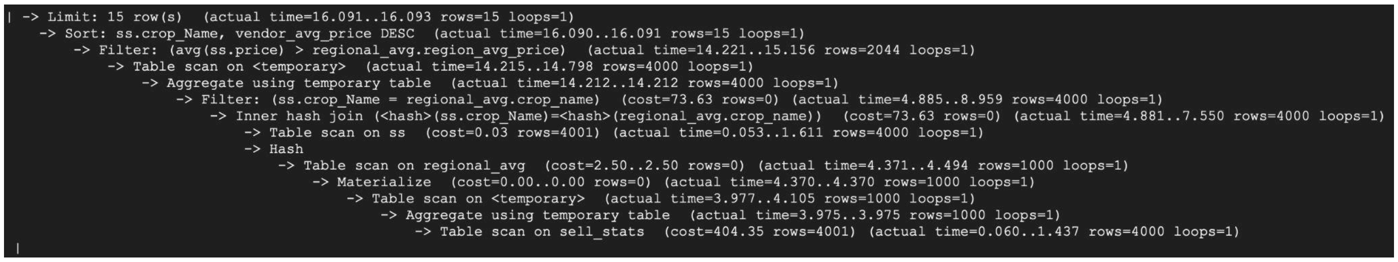
Index on price:



```
| -> Limit: 15 row(s)  (actual time=18.785..18.787 rows=15 loops=1)
    -> Sort: ss.crop_Name, vendor_avg_price DESC  (actual time=18.784..18.785 rows=15 loops=1)
        -> Filter: (avg(ss.price) > regional_avg.region_avg_price)  (actual time=16.000..16.953 rows=2044 loops=1)
            -> Table scan on <temporary>  (actual time=15.037..15.694 rows=4000 loops=1)
                -> Aggregate using temporary table  (actual time=15.034..15.034 rows=4000 loops=1)
                    -> Filter: (ss.crop_Name = regional_avg.crop_name)  (cost=73.63 rows=0) (actual time=6.007..9.861 rows=4000 loops=1)
                        -> Inner hash join (<hash>(ss.crop_Name)=<hash>(regional_avg.crop_name))  (cost=73.63 rows=0) (actual time=6.004..8.500 rows=4000 loops=1)
                            -> Covering index scan on ss using idx_sell_price  (cost=0.03 rows=4001) (actual time=0.076..1.485 rows=4000 loops=1)
                            -> Hash
                                -> Table scan on regional_avg  (cost=2.50..2.50 rows=0) (actual time=5.404..5.572 rows=1000 loops=1)
                                    -> Materialize  (cost=0.00..0.00 rows=0) (actual time=5.403..5.403 rows=1000 loops=1)
                                        -> Table scan on <temporary>  (actual time=4.206..4.433 rows=1000 loops=1)
                                            -> Aggregate using temporary table  (actual time=4.202..4.202 rows=1000 loops=1)
                                                -> Covering index scan on sell_stats using idx_sell_price  (cost=404.35 rows=4001) (actual time=0.067..1.452 rows=4000 loops=1)
 |
```

When indexing on price, the cost stayed the same for everything. This shows that indexing on price does not make a difference in performance and is not needed for this query.

Index on vendor_name:



```
| -> Limit: 15 row(s)  (actual time=16.091..16.093 rows=15 loops=1)
    -> Sort: ss.crop_Name, vendor_avg_price DESC  (actual time=16.090..16.091 rows=15 loops=1)
        -> Filter: (avg(ss.price) > regional_avg.region_avg_price)  (actual time=14.221..15.156 rows=2044 loops=1)
            -> Table scan on <temporary>  (actual time=14.215..14.798 rows=4000 loops=1)
                -> Aggregate using temporary table  (actual time=14.212..14.212 rows=4000 loops=1)
                    -> Filter: (ss.crop_Name = regional_avg.crop_name)  (cost=73.63 rows=0) (actual time=4.885..8.959 rows=4000 loops=1)
                        -> Inner hash join (<hash>(ss.crop_Name)=<hash>(regional_avg.crop_name))  (cost=73.63 rows=0) (actual time=4.881..7.550 rows=4000 loops=1)
                            -> Table scan on ss  (cost=0.03 rows=4001) (actual time=0.053..1.611 rows=4000 loops=1)
                            -> Hash
                                -> Table scan on regional_avg  (cost=2.50..2.50 rows=0) (actual time=4.371..4.494 rows=1000 loops=1)
                                    -> Materialize  (cost=0.00..0.00 rows=0) (actual time=4.370..4.370 rows=1000 loops=1)
                                        -> Table scan on <temporary>  (actual time=3.977..4.105 rows=1000 loops=1)
                                            -> Aggregate using temporary table  (actual time=3.975..3.975 rows=1000 loops=1)
                                                -> Table scan on sell_stats  (cost=404.35 rows=4001) (actual time=0.060..1.437 rows=4000 loops=1)
 |
```

Indexing on vendor_name also does not make a difference as the costs stay the same for each. Therefore the index is not needed for this query to improve performance.

Final Index Design:
After testing different indexing strategies, I found that no indices improved the performance of my query, so no additional indexing was ultimately needed. Initially, I assumed indexing on the price column would help reduce query cost by enabling faster lookups, but this was not true. This lack of improvement is likely because the query aggregates average prices at vendor and regional levels, requiring MySQL to scan all relevant rows rather than using an index. Because adding an index increased storage and maintenance costs without improving query performance, I concluded that additional indexing would not benefit this query.