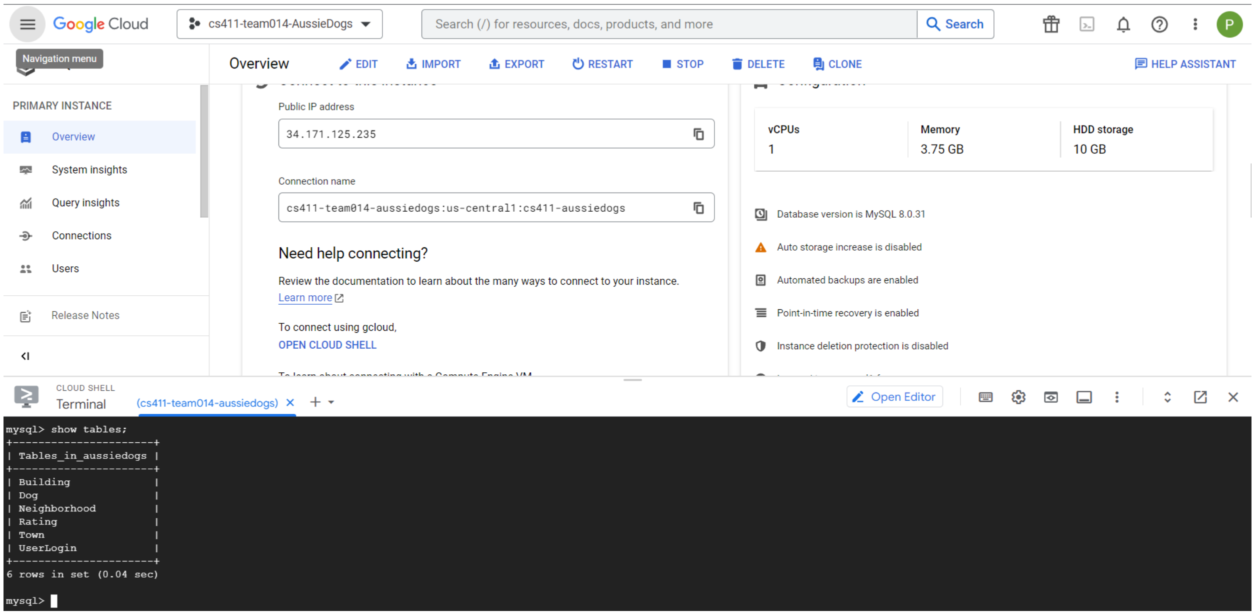
Database Screenshots:



Public IP address

34.171.125.235

Connection name

cs411-team014-aussiedogs:us-central1:cs411-aussiedogs

Need help connecting?

Review the documentation to learn about the many ways to connect to your instance. Learn more

To connect using gcloud,
OPEN CLOUD SHELL

vCPUs	Memory	HDD storage
1	3.75 GB	10 GB

Database version is MySQL 8.0.31

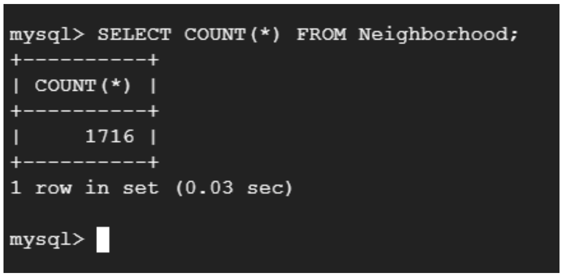
Auto storage increase is disabled

Automated backups are enabled

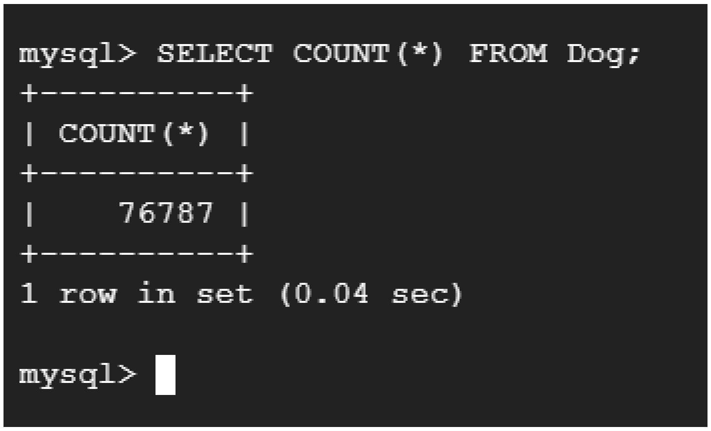
Point-in-time recovery is enabled

Instance deletion protection is disabled

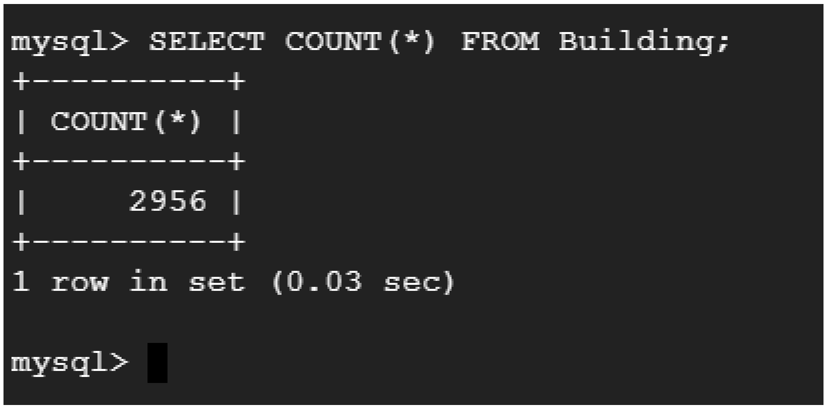
```
mysql> show tables;
+----------------------+
| Tables_in_aussiedogs |
+----------------------+
| Building             |
| Dog                  |
| Neighborhood         |
| Rating               |
| Town                 |
| UserLogin            |
+----------------------+
6 rows in set (0.04 sec)

mysql> 
```



```
mysql> SELECT COUNT(*) FROM Neighborhood;
+----------+
| COUNT(*) |
+----------+
|     1716 |
+----------+
1 row in set (0.03 sec)

mysql> 
```



```
mysql> SELECT COUNT(*) FROM Dog;
+----------+
| COUNT(*) |
+----------+
|    76787 |
+----------+
1 row in set (0.04 sec)

mysql> 
```



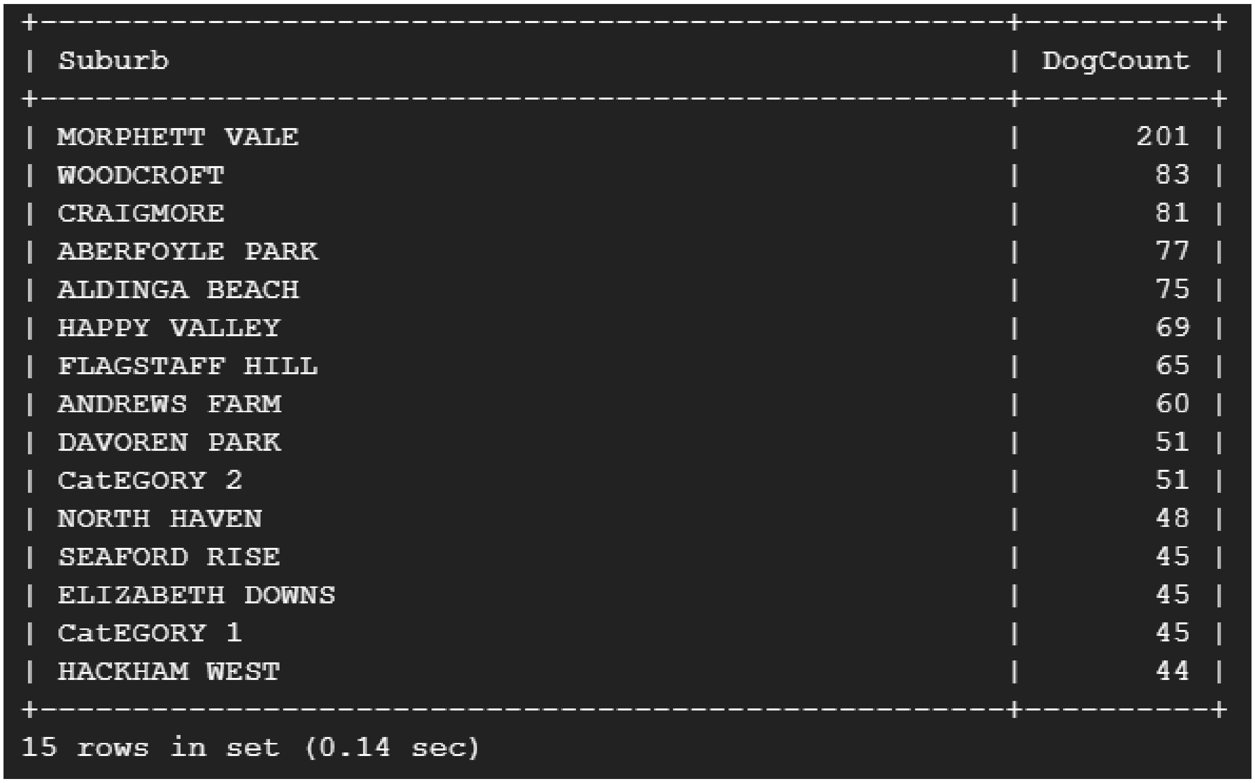
```
mysql> SELECT COUNT(*) FROM Building;
+----------+
| COUNT(*) |
+----------+
|     2956 |
+----------+
1 row in set (0.03 sec)

mysql> 
```

DDL Commands

```
CREATE TABLE Town(TownName VARCHAR(255) NOT NULL PRIMARY KEY, State
VARCHAR(255), Country VARCHAR(255));

CREATE TABLE Rating(TownName VARCHAR(255) NOT NULL, Username VARCHAR(255)
NOT NULL, Type VARCHAR(255), Value INT, PRIMARY KEY(TownName, Username),
FOREIGN KEY(TownName) REFERENCES Town(TownName), FOREIGN KEY(Username)
REFERENCES UserLogin(Username));

CREATE TABLE UserLogin(Username VARCHAR(255) NOT NULL PRIMARY KEY, Email
VARCHAR(255), Phone INT(15), Password VARCHAR(128));

CREATE TABLE Dog(ReferenceID INT NOT NULL, Animal_Name VARCHAR(255), Breed
VARCHAR(255), Suburb VARCHAR(255), BirthYear INT, Gender VARCHAR(6), Latitude REAL,
Longitude REAL, PRIMARY KEY(ReferenceID));


CREATE TABLE Neighborhood(Name VARCHAR(255) NOT NULL, Town VARCHAR(255) NOT
NULL, Latitude REAL, Longitude REAL, PRIMARY KEY(Name, Town), FOREIGN KEY(Town)
REFERENCES Town(TownName) ON DELETE CASCADE);

CREATE TABLE Building(Name VARCHAR(255) NOT NULL, Town VARCHAR(255) NOT
NULL, Latitude REAL, Longitude REAL, Type VARCHAR(255), PRIMARY KEY(Name, Town),
FOREIGN KEY(Town) REFERENCES Town(TownName) ON DELETE CASCADE);
```

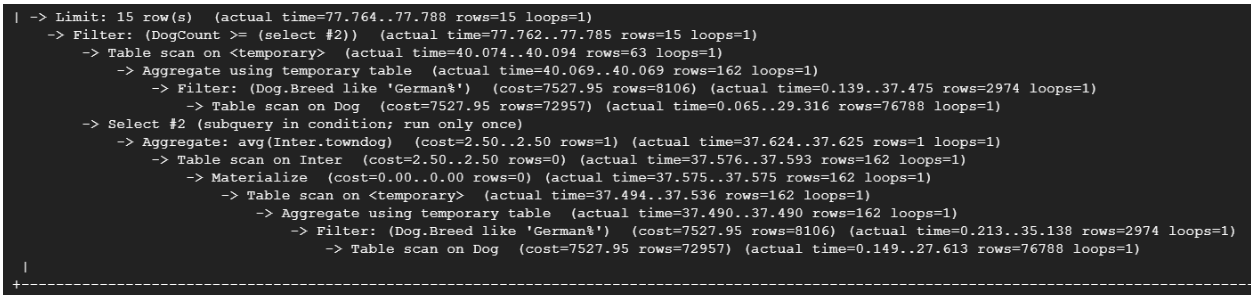
Advance Queries
Find number of dogs in towns with above average German Shepherd

```
SELECT Suburb, Count(*) as DogCount
FROM Dog
Where Breed LIKE 'German%'
GROUP BY Suburb
HAVING DogCount >= (SELECT AVG(towndog)
                                        FROM (SELECT Count(*) AS towndog
                                                 FROM Dog WHERE Breed LIKE 'German%'
                                                GROUP BY Suburb) AS Inter)
LIMIT 15;
```



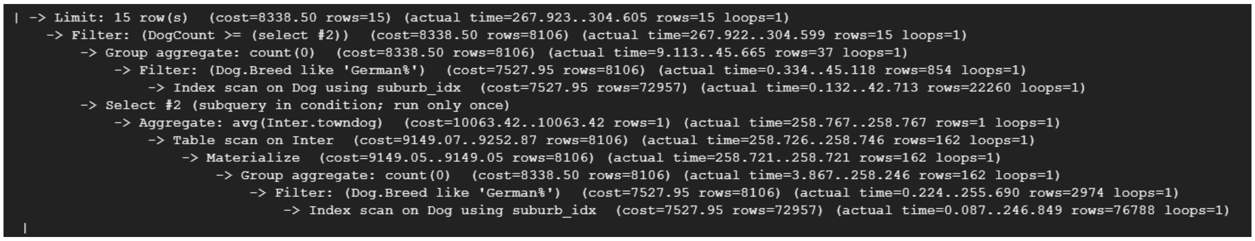
```
+-----------------------------------------------------+----------+
| Suburb                                              | DogCount |
+-----------------------------------------------------+----------+
| MORPHETT VALE                                       |      201 |
| WOODCROFT                                           |       83 |
| CRAIGMORE                                           |       81 |
| ABERFOYLE PARK                                      |       77 |
| ALDINGA BEACH                                       |       75 |
| HAPPY VALLEY                                        |       69 |
| FLAGSTAFF HILL                                      |       65 |
| ANDREWS FARM                                        |       60 |
| DAVOREN PARK                                        |       51 |
| CatEGORY 2                                          |       51 |
| NORTH HAVEN                                         |       48 |
| SEAFORD RISE                                        |       45 |
| ELIZABETH DOWNS                                     |       45 |
| CatEGORY 1                                          |       45 |
| HACKHAM WEST                                        |       44 |
+-----------------------------------------------------+----------+
15 rows in set (0.14 sec)
```

No index



```
| -> Limit: 15 row(s)  (actual time=77.764..77.788 rows=15 loops=1)
    -> Filter: (DogCount >= (select #2))  (actual time=77.762..77.785 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=40.074..40.094 rows=63 loops=1)
            -> Aggregate using temporary table  (actual time=40.069..40.069 rows=162 loops=1)
                -> Filter: (Dog.Breed like 'German%')  (cost=7527.95 rows=8106) (actual time=0.139..37.475 rows=2974 loops=1)
                    -> Table scan on Dog  (cost=7527.95 rows=72957) (actual time=0.065..29.316 rows=76788 loops=1)
        -> Select #2 (subquery in condition; run only once)
            -> Aggregate: avg(Inter.towndog)  (cost=2.50..2.50 rows=1) (actual time=37.624..37.625 rows=1 loops=1)
                -> Table scan on Inter  (cost=2.50..2.50 rows=0) (actual time=37.576..37.593 rows=162 loops=1)
                    -> Materialize  (cost=0.00..0.00 rows=0) (actual time=37.575..37.575 rows=162 loops=1)
                        -> Table scan on <temporary>  (actual time=37.494..37.536 rows=162 loops=1)
                            -> Aggregate using temporary table  (actual time=37.490..37.490 rows=162 loops=1)
                                -> Filter: (Dog.Breed like 'German%')  (cost=7527.95 rows=8106) (actual time=0.213..35.138 rows=2974 loops=1)
                                    -> Table scan on Dog  (cost=7527.95 rows=72957) (actual time=0.149..27.613 rows=76788 loops=1)
 |
+--------------------------------------------------------------------------------------------------------------------------------------
```

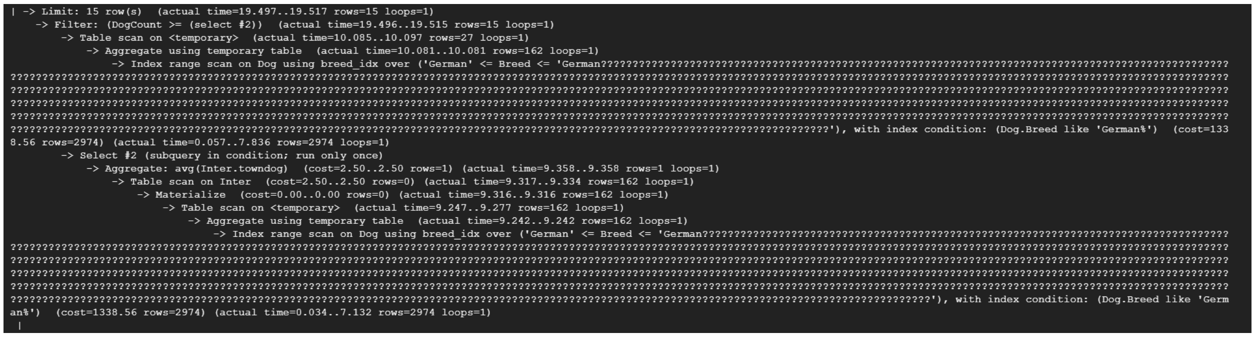
Added index to Dog on Suburb



```
| -> Limit: 15 row(s)  (cost=8338.50 rows=15) (actual time=267.923..304.605 rows=15 loops=1)
    -> Filter: (DogCount >= (select #2))  (cost=8338.50 rows=8106) (actual time=267.922..304.599 rows=15 loops=1)
        -> Group aggregate: count(0)  (cost=8338.50 rows=8106) (actual time=9.113..45.665 rows=37 loops=1)
            -> Filter: (Dog.Breed like 'German%')  (cost=7527.95 rows=8106) (actual time=0.334..45.118 rows=854 loops=1)
                -> Index scan on Dog using suburb_idx  (cost=7527.95 rows=72957) (actual time=0.132..42.713 rows=22260 loops=1)
        -> Select #2 (subquery in condition; run only once)
            -> Aggregate: avg(Inter.towndog)  (cost=10063.42..10063.42 rows=1) (actual time=258.767..258.767 rows=1 loops=1)
                -> Table scan on Inter  (cost=9149.07..9252.87 rows=8106) (actual time=258.726..258.746 rows=162 loops=1)
                    -> Materialize  (cost=9149.05..9149.05 rows=8106) (actual time=258.721..258.721 rows=162 loops=1)
                        -> Group aggregate: count(0)  (cost=8338.50 rows=8106) (actual time=3.867..258.246 rows=162 loops=1)
                            -> Filter: (Dog.Breed like 'German%')  (cost=7527.95 rows=8106) (actual time=0.224..255.690 rows=2974 loops=1)
                                -> Index scan on Dog using suburb_idx  (cost=7527.95 rows=72957) (actual time=0.087..246.849 rows=76788 loops=1)
 |
```

We thought indexing by suburb would help reduce the overall cost because we are selecting for suburb information at the end. However, we can see that although the Table Scans turned into Index scans, there was no reduction in cost/performance at all. Instead, it over doubled the cost during the aggregation steps. We think this is because we are running subqueries without using suburb names, and forcing the table to index based on the suburb name introduces many inefficiencies.

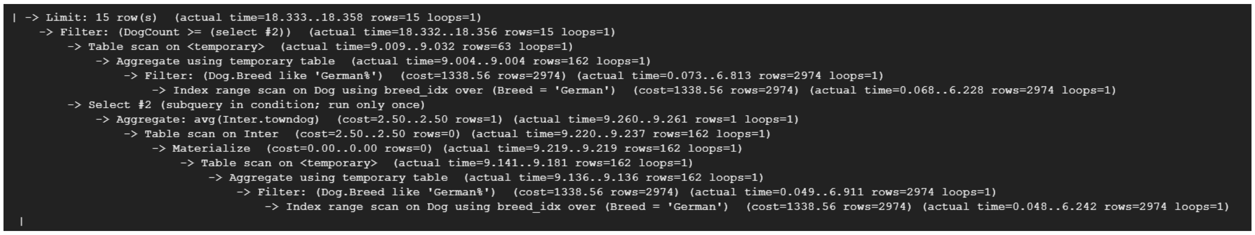
Added index to Dog on Breed



```
| -> Limit: 15 row(s)  (actual time=19.497..19.517 rows=15 loops=1)
    -> Filter: (DogCount >= (select #2))  (actual time=19.496..19.515 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=10.085..10.097 rows=27 loops=1)
            -> Aggregate using temporary table  (actual time=10.081..10.081 rows=162 loops=1)
                -> Index range scan on Dog using breed_idx over ('German' <= Breed <= 'German?????????????????????????????????????????????????????????????????????????????????????????????????????
??????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????
??????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????
??????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????
??????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????
???????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????'), with index condition: (Dog.Breed like 'German%')  (cost=133
8.56 rows=2974) (actual time=0.057..7.836 rows=2974 loops=1)
        -> Select #2 (subquery in condition; run only once)
            -> Aggregate: avg(Inter.towndog)  (cost=2.50..2.50 rows=1) (actual time=9.358..9.358 rows=1 loops=1)
                -> Table scan on Inter  (cost=2.50..2.50 rows=0) (actual time=9.317..9.334 rows=162 loops=1)
                    -> Materialize  (cost=0.00..0.00 rows=0) (actual time=9.316..9.316 rows=162 loops=1)
                        -> Table scan on <temporary>  (actual time=9.247..9.277 rows=162 loops=1)
                            -> Aggregate using temporary table  (actual time=9.242..9.242 rows=162 loops=1)
                                -> Index range scan on Dog using breed_idx over ('German' <= Breed <= 'German?????????????????????????????????????????????????????????????????????????????????????
??????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????
??????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????
??????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????
??????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????
???????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????????'), with index condition: (Dog.Breed like 'Germ
an%')  (cost=1338.56 rows=2974) (actual time=0.034..7.132 rows=2974 loops=1)
 |
```

This index resulted in HUGE performance gains, noticeably during the index range scans, instead of the table scans. Our cost from over 7000 on each scan, to the mid 1000s per scan. The aggregations cost about the same as before, but they represent an insignificant cost in comparison to the scans. We attribute these performance gains due to our limiting condition, which checks that the breed starts with the keyword “German”. By then indexing on the breed, we can cut out a significant amount of unnecessary work at multiple points during the query.

Added index to Dog on Breed(6)



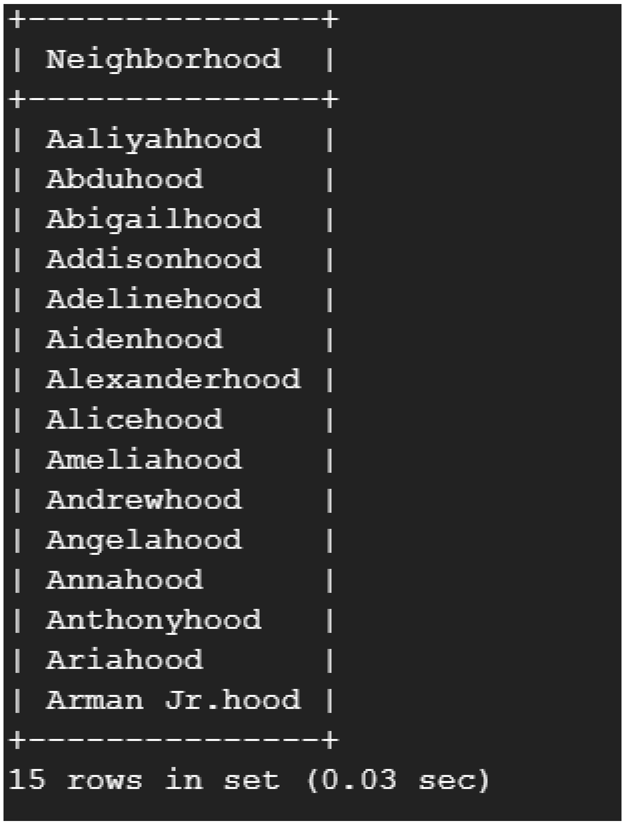
```
| -> Limit: 15 row(s)  (actual time=18.333..18.358 rows=15 loops=1)
    -> Filter: (DogCount >= (select #2))  (actual time=18.332..18.356 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=9.009..9.032 rows=63 loops=1)
            -> Aggregate using temporary table  (actual time=9.004..9.004 rows=162 loops=1)
                -> Filter: (Dog.Breed like 'German%')  (cost=1338.56 rows=2974) (actual time=0.073..6.813 rows=2974 loops=1)
                    -> Index range scan on Dog using breed_idx over (Breed = 'German')  (cost=1338.56 rows=2974) (actual time=0.068..6.228 rows=2974 loops=1)
        -> Select #2 (subquery in condition; run only once)
            -> Aggregate: avg(Inter.towndog)  (cost=2.50..2.50 rows=1) (actual time=9.260..9.261 rows=1 loops=1)
                -> Table scan on Inter  (cost=2.50..2.50 rows=0) (actual time=9.220..9.237 rows=162 loops=1)
                    -> Materialize  (cost=0.00..0.00 rows=0) (actual time=9.219..9.219 rows=162 loops=1)
                        -> Table scan on <temporary>  (actual time=9.141..9.181 rows=162 loops=1)
                            -> Aggregate using temporary table  (actual time=9.136..9.136 rows=162 loops=1)
                                -> Filter: (Dog.Breed like 'German%')  (cost=1338.56 rows=2974) (actual time=0.049..6.911 rows=2974 loops=1)
                                    -> Index range scan on Dog using breed_idx over (Breed = 'German')  (cost=1338.56 rows=2974) (actual time=0.048..6.242 rows=2974 loops=1)
 |
```

Building off of the previous index choice, we thought about limiting the Breed index to just the first 6 characters (since we are searching for breeds starting with the word “German”). However, instead of improving the performance like we thought, there was minimal improvement and even a bit worse performance at each filtering stage. This is likely due to the fact that we are still using an index range scan, including dog breeds of length >= 6, so there is no real improvement to adding an index on just 6 characters of the Breed.

As a result, we will simply be using the index on Breed. There was no noticeable performance boost when limiting the index to just the first 6 characters (and upon further testing the same is true for a variety of values instead of just 6). Furthermore, it is not guaranteed that other queries we perform on this table will rely on only a portion of the name. Instead, we might want to query for a very specific breed of dog (e.g. Labrador Retriever Cross), though being able to efficiently aggregate information over the entire breed will nonetheless be very useful.

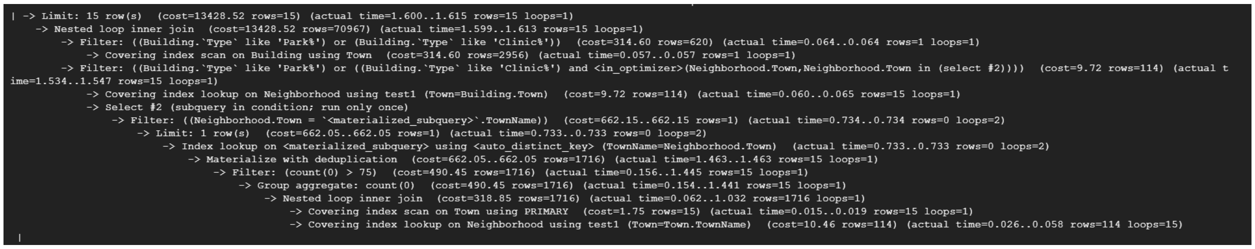
Neighborhoods in large towns with certain types of buildings

```
SELECT Neighborhood.name as Neighborhood
FROM Neighborhood JOIN Building ON Neighborhood.Town = Building.Town
WHERE Building.Type LIKE 'Park%' OR Building.Type LIKE 'Clinic%'
        AND Neighborhood.Town = ANY(SELECT Town.TownName
                                        FROM Town JOIN Neighborhood ON (Town.TownName
                                                                 = Neighborhood.Town)
                                        GROUP BY Town.TownName
                                        HAVING Count(*)>75)
LIMIT 15;
```



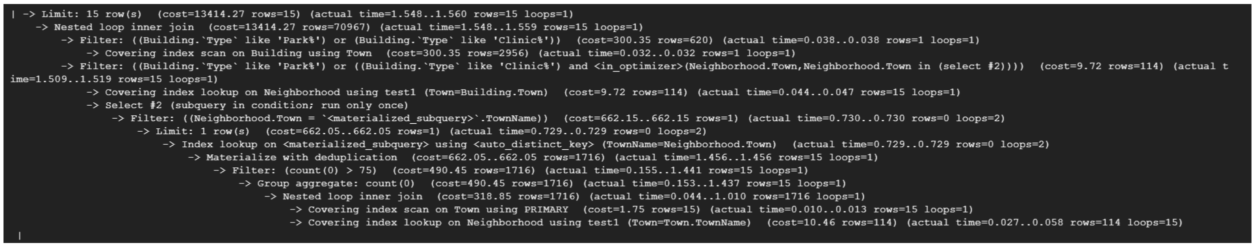
```
+---------------+
| Neighborhood  |
+---------------+
| Aaliyahhood   |
| Abduhood      |
| Abigailhood   |
| Addisonhood   |
| Adelinehood   |
| Aidenhood     |
| Alexanderhood |
| Alicehood     |
| Ameliahood    |
| Andrewhood    |
| Angelahood    |
| Annahood      |
| Anthonyhood   |
| Ariahood      |
| Arman Jr.hood |
+---------------+
15 rows in set (0.03 sec)
```

No index



```
| -> Limit: 15 row(s)  (cost=13428.52 rows=15) (actual time=1.600..1.615 rows=15 loops=1)
    -> Nested loop inner join  (cost=13428.52 rows=70967) (actual time=1.599..1.613 rows=15 loops=1)
        -> Filter: ((Building.`Type` like 'Park%') or (Building.`Type` like 'Clinic%'))  (cost=314.60 rows=620) (actual time=0.064..0.064 rows=1 loops=1)
            -> Covering index scan on Building using Town  (cost=314.60 rows=2956) (actual time=0.057..0.057 rows=1 loops=1)
        -> Filter: ((Building.`Type` like 'Park%') or ((Building.`Type` like 'Clinic%') and <in_optimizer>(Neighborhood.Town,Neighborhood.Town in (select #2))))  (cost=9.72 rows=114) (actual t
ime=1.534..1.547 rows=15 loops=1)
            -> Covering index lookup on Neighborhood using test1 (Town=Building.Town)  (cost=9.72 rows=114) (actual time=0.060..0.065 rows=15 loops=1)
            -> Select #2 (subquery in condition; run only once)
                -> Filter: ((Neighborhood.Town = `<materialized_subquery>`.TownName))  (cost=662.15..662.15 rows=1) (actual time=0.734..0.734 rows=0 loops=2)
                    -> Limit: 1 row(s)  (cost=662.05..662.05 rows=1) (actual time=0.733..0.733 rows=0 loops=2)
                        -> Index lookup on <materialized_subquery> using <auto_distinct_key> (TownName=Neighborhood.Town)  (actual time=0.733..0.733 rows=0 loops=2)
                            -> Materialize with deduplication  (cost=662.05..662.05 rows=1716) (actual time=1.463..1.463 rows=15 loops=1)
                                -> Filter: (count(0) > 75)  (cost=490.45 rows=1716) (actual time=0.156..1.445 rows=15 loops=1)
                                    -> Group aggregate: count(0)  (cost=490.45 rows=1716) (actual time=0.154..1.441 rows=15 loops=1)
                                        -> Nested loop inner join  (cost=318.85 rows=1716) (actual time=0.062..1.032 rows=1716 loops=1)
                                            -> Covering index scan on Town using PRIMARY  (cost=1.75 rows=15) (actual time=0.015..0.019 rows=15 loops=1)
                                            -> Covering index lookup on Neighborhood using test1 (Town=Town.TownName)  (cost=10.46 rows=114) (actual time=0.026..0.058 rows=114 loops=15)
 |
```

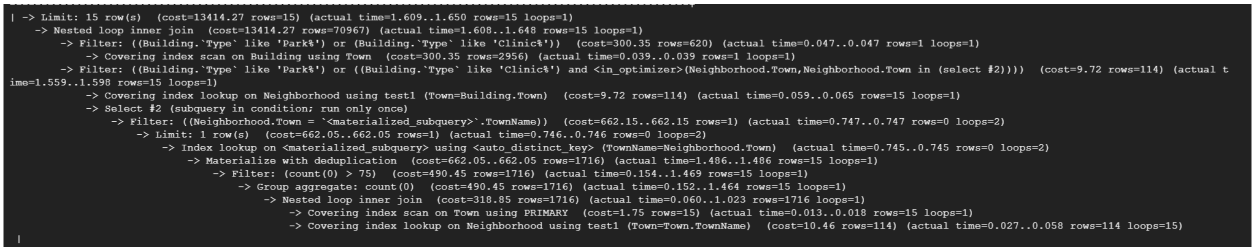
Added index to Town on State



```
| -> Limit: 15 row(s)  (cost=13414.27 rows=15) (actual time=1.548..1.560 rows=15 loops=1)
    -> Nested loop inner join  (cost=13414.27 rows=70967) (actual time=1.548..1.559 rows=15 loops=1)
        -> Filter: ((Building.`Type` like 'Park%') or (Building.`Type` like 'Clinic%'))  (cost=300.35 rows=620) (actual time=0.038..0.038 rows=1 loops=1)
            -> Covering index scan on Building using Town  (cost=300.35 rows=2956) (actual time=0.032..0.032 rows=1 loops=1)
        -> Filter: ((Building.`Type` like 'Park%') or ((Building.`Type` like 'Clinic%') and <in_optimizer>(Neighborhood.Town,Neighborhood.Town in (select #2))))  (cost=9.72 rows=114) (actual t
ime=1.509..1.519 rows=15 loops=1)
            -> Covering index lookup on Neighborhood using test1 (Town=Building.Town)  (cost=9.72 rows=114) (actual time=0.044..0.047 rows=15 loops=1)
            -> Select #2 (subquery in condition; run only once)
                -> Filter: ((Neighborhood.Town = `<materialized_subquery>`.TownName))  (cost=662.15..662.15 rows=1) (actual time=0.730..0.730 rows=0 loops=2)
                    -> Limit: 1 row(s)  (cost=662.05..662.05 rows=1) (actual time=0.729..0.729 rows=0 loops=2)
                        -> Index lookup on <materialized_subquery> using <auto_distinct_key> (TownName=Neighborhood.Town)  (actual time=0.729..0.729 rows=0 loops=2)
                            -> Materialize with deduplication  (cost=662.05..662.05 rows=1716) (actual time=1.456..1.456 rows=15 loops=1)
                                -> Filter: (count(0) > 75)  (cost=490.45 rows=1716) (actual time=0.155..1.441 rows=15 loops=1)
                                    -> Group aggregate: count(0)  (cost=490.45 rows=1716) (actual time=0.153..1.437 rows=15 loops=1)
                                        -> Nested loop inner join  (cost=318.85 rows=1716) (actual time=0.044..1.010 rows=1716 loops=1)
                                            -> Covering index scan on Town using PRIMARY  (cost=1.75 rows=15) (actual time=0.010..0.013 rows=15 loops=1)
                                            -> Covering index lookup on Neighborhood using test1 (Town=Town.TownName)  (cost=10.46 rows=114) (actual time=0.027..0.058 rows=114 loops=15)
 |
```

We thought there might be some value in grouping towns together in some way, since we are joining towns with neighborhoods in the subquery. However, clearly there was pretty much no change in the performance. The costs at each point are pretty much the same.

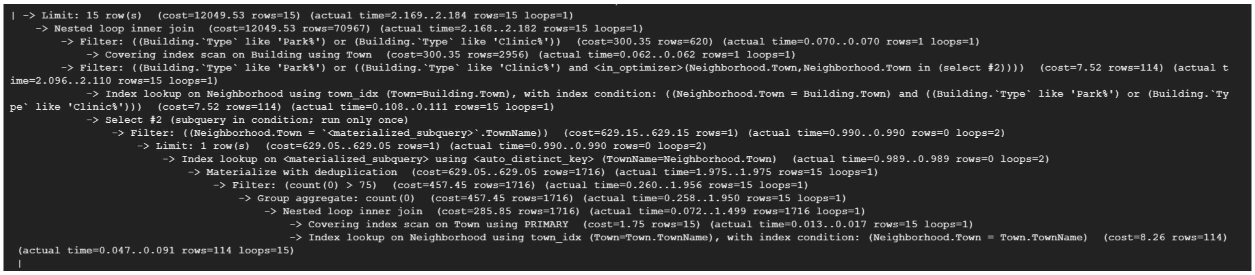
Added index to Neighborhood on (latitude, longitude)



```
| -> Limit: 15 row(s)  (cost=13414.27 rows=15) (actual time=1.609..1.650 rows=15 loops=1)
    -> Nested loop inner join  (cost=13414.27 rows=70967) (actual time=1.608..1.648 rows=15 loops=1)
        -> Filter: ((Building.`Type` like 'Park%') or (Building.`Type` like 'Clinic%'))  (cost=300.35 rows=620) (actual time=0.047..0.047 rows=1 loops=1)
            -> Covering index scan on Building using Town  (cost=300.35 rows=2956) (actual time=0.039..0.039 rows=1 loops=1)
        -> Filter: ((Building.`Type` like 'Park%') or ((Building.`Type` like 'Clinic%') and <in_optimizer>(Neighborhood.Town,Neighborhood.Town in (select #2))))  (cost=9.72 rows=114) (actual t
ime=1.559..1.598 rows=15 loops=1)
            -> Covering index lookup on Neighborhood using test1 (Town=Building.Town)  (cost=9.72 rows=114) (actual time=0.059..0.065 rows=15 loops=1)
            -> Select #2 (subquery in condition; run only once)
                -> Filter: ((Neighborhood.Town = `<materialized_subquery>`.TownName))  (cost=662.15..662.15 rows=1) (actual time=0.747..0.747 rows=0 loops=2)
                    -> Limit: 1 row(s)  (cost=662.05..662.05 rows=1) (actual time=0.746..0.746 rows=0 loops=2)
                        -> Index lookup on <materialized_subquery> using <auto_distinct_key> (TownName=Neighborhood.Town)  (actual time=0.745..0.745 rows=0 loops=2)
                            -> Materialize with deduplication  (cost=662.05..662.05 rows=1716) (actual time=1.486..1.486 rows=15 loops=1)
                                -> Filter: (count(0) > 75)  (cost=490.45 rows=1716) (actual time=0.154..1.469 rows=15 loops=1)
                                    -> Group aggregate: count(0)  (cost=490.45 rows=1716) (actual time=0.152..1.464 rows=15 loops=1)
                                        -> Nested loop inner join  (cost=318.85 rows=1716) (actual time=0.060..1.023 rows=1716 loops=1)
                                            -> Covering index scan on Town using PRIMARY  (cost=1.75 rows=15) (actual time=0.013..0.018 rows=15 loops=1)
                                            -> Covering index lookup on Neighborhood using test1 (Town=Town.TownName)  (cost=10.46 rows=114) (actual time=0.027..0.058 rows=114 loops=15)
 |
```

Once again, since we are joining Neighborhood with other tables multiple times throughout the query, we thought adding another index to group them together might help improve performance. However, since we are not explicitly comparing location data at any point in the query, this did not lead to any significant performance changes.

Added index to Neighborhood on Town(5)



```
| -> Limit: 15 row(s)  (cost=12049.53 rows=15) (actual time=2.169..2.184 rows=15 loops=1)
    -> Nested loop inner join  (cost=12049.53 rows=70967) (actual time=2.168..2.182 rows=15 loops=1)
        -> Filter: ((Building.`Type` like 'Park%') or (Building.`Type` like 'Clinic%'))  (cost=300.35 rows=620) (actual time=0.070..0.070 rows=1 loops=1)
            -> Covering index scan on Building using Town  (cost=300.35 rows=2956) (actual time=0.062..0.062 rows=1 loops=1)
        -> Filter: ((Building.`Type` like 'Park%') or ((Building.`Type` like 'Clinic%') and <in_optimizer>(Neighborhood.Town,Neighborhood.Town in (select #2))))  (cost=7.52 rows=114) (actual t
ime=2.096..2.110 rows=15 loops=1)
            -> Index lookup on Neighborhood using town_idx (Town=Building.Town), with index condition: ((Neighborhood.Town = Building.Town) and ((Building.`Type` like 'Park%') or (Building.`Ty
pe` like 'Clinic%')))  (cost=7.52 rows=114) (actual time=0.108..0.111 rows=15 loops=1)
            -> Select #2 (subquery in condition; run only once)
                -> Filter: ((Neighborhood.Town = `<materialized_subquery>`.TownName))  (cost=629.15..629.15 rows=1) (actual time=0.990..0.990 rows=0 loops=2)
                    -> Limit: 1 row(s)  (cost=629.05..629.05 rows=1) (actual time=0.990..0.990 rows=0 loops=2)
                        -> Index lookup on <materialized_subquery> using <auto_distinct_key> (TownName=Neighborhood.Town)  (actual time=0.989..0.989 rows=0 loops=2)
                            -> Materialize with deduplication  (cost=629.05..629.05 rows=1716) (actual time=1.975..1.975 rows=15 loops=1)
                                -> Filter: (count(0) > 75)  (cost=457.45 rows=1716) (actual time=0.260..1.956 rows=15 loops=1)
                                    -> Group aggregate: count(0)  (cost=457.45 rows=1716) (actual time=0.258..1.950 rows=15 loops=1)
                                        -> Nested loop inner join  (cost=285.85 rows=1716) (actual time=0.072..1.499 rows=1716 loops=1)
                                            -> Covering index scan on Town using PRIMARY  (cost=1.75 rows=15) (actual time=0.013..0.017 rows=15 loops=1)
                                            -> Index lookup on Neighborhood using town_idx (Town=Town.TownName), with index condition: (Neighborhood.Town = Town.TownName)  (cost=8.26 rows=114)
 (actual time=0.047..0.091 rows=114 loops=15)
 |
```

While Town is already a primary key (due to Neighborhood being a weak entity dependent on Town), we thought there might be some value to grouping the Neighborhoods based on just the first few characters of the Town name. This came from the fact that many towns have similar endings (like the -hood suffix). Uniquely identifying based on these last few characters just doesn't seem necessary. We did notice some slight performance gains, with the largest being at the Nested loop inner join (near the top). This dropped by ~1400 cost. The latter aggregations/filters sometimes dropped by a few percentage points as well. **Note: playing around with the value of x for Town(x) did not change the performance boosts by significant amounts.

We will be using the final index of Town(5) on Neighborhood. While the boosts were not massive, there was nonetheless a noticeable improvement to performance. As more and more entries are added to the table, this improvement will likewise become more and more important.