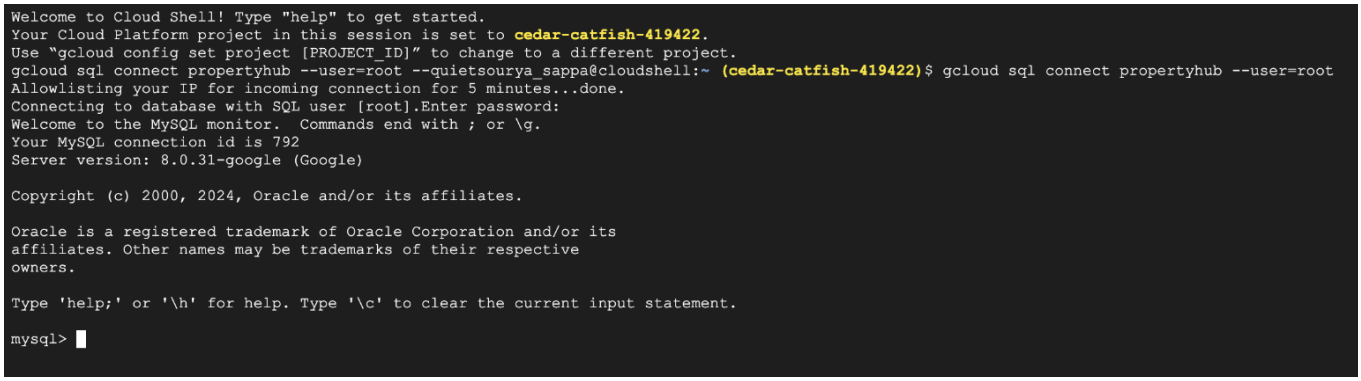
Part 1

Connecting into GCP



DDL Commands

```
CREATE TABLE Users (
        userId INT,
        userName VARCHAR(50),
        password VARCHAR(50),
        favorites VARCHAR(255),
        PRIMARY KEY(userId)
);

CREATE TABLE Properties (
        propertyId INT PRIMARY KEY,
        price INT,
        squareFootage INT,
        numBedrooms INT,
        numBathrooms INT,
        propertyType VARCHAR(50),
        yearBuilt INT
);
```

```
CREATE TABLE Seller (
	propertyId INT,
	priceSold INT,
	datePosted INT,
	dateSold INT,
	sellerId INT,
	PRIMARY KEY (sellerId),
	FOREIGN KEY (propertyId) REFERENCES Properties(propertyId)
);

CREATE TABLE Customer (
	customerId INT,
	userId INT,
	propertyId INT,
	PRIMARY KEY (customerId),
	FOREIGN KEY (userId) REFERENCES Users(userId),
	FOREIGN KEY (propertyId) REFERENCES Properties(propertyId)
);

CREATE TABLE Address (
	latitude DOUBLE(15,7),
	longitude DOUBLE(15, 7),
	city VARCHAR(50),
	state CHAR(2),
	zip INT,
	propertyId INT,
	PRIMARY KEY (latitude),
	FOREIGN KEY (propertyId) REFERENCES Properties(propertyId)
);

CREATE TABLE Search(
	searchQuery VARCHAR(255),
	filters VARCHAR(255),
	searchResults VARCHAR(255)
);

CREATE TABLE Sales(
	saleId INT,
	propertyId INT,
	priceSold DECIMAL(10,2),
	PRIMARY KEY (saleId),
	FOREIGN KEY (propertyId) REFERENCES Properties(propertyId)
);
```

```
mysql> show tables;
+------------------+
| Tables_in_team061 |
+------------------+
| Address          |
| Customer         |
| Properties       |
| Sales            |
| Search           |
| Seller           |
| Users            |
+------------------+
```

```
mysql> select count(*) Address;
+----------+
| Address |
+----------+
|        1 |
+----------+
```

```
mysql> select count(*) Properties;
+------------+
| Properties |
+------------+
|          1 |
+------------+
```

```
mysql> Select count(*) From Seller;
+----------+
| count(*) |
+----------+
|      578 |
+----------+
1 row in set (0.03 sec)
```

For some reason it keeps stopping at 578 rows and we are not too sure why. We asked TA's but they also could not figure out why.

Queries

```
SELECT *
FROM Seller
JOIN Properties
WHERE sellerId IN (
    SELECT sellerId
    FROM Seller
    WHERE priceSold > (
        SELECT AVG(priceSold)
        FROM Seller
        WHERE priceSold IS NOT NULL
    )
);
```

```
+------------+-----------+------------+----------+----------+
| propertyId | priceSold | datePosted | dateSold | sellerId |
+------------+-----------+------------+----------+----------+
|    6436197 |    489000 |       NULL |     NULL |   574131 |
|    9540949 |    351900 |       NULL |     NULL |  2194495 |
|    7275689 |    659333 |       NULL |     NULL | 11580392 |
|    6896679 |    699000 |       NULL |     NULL | 18849183 |
|    5048020 |    625000 |       NULL |     NULL | 22818161 |
|    3663811 |    389900 |       NULL |     NULL | 24308400 |
|    8337914 |    369900 |       NULL |     NULL | 42732452 |
|    2868561 |    499000 |       NULL |     NULL | 47205709 |
|    6003821 |    700000 |       NULL |     NULL | 57306624 |
|     265098 |    369900 |       NULL |     NULL | 68616470 |
|    5934316 |    599900 |       NULL |     NULL | 69720516 |
|    7318179 |    384900 |       NULL |     NULL | 74567104 |
|    1937266 |    965000 |       NULL |     NULL | 80109126 |
|    8089447 |    965000 |       NULL |     NULL | 87496907 |
|    1798237 |    355000 |       NULL |     NULL | 89256609 |
+------------+-----------+------------+----------+----------+
15 rows in set (0.00 sec)
```

In any real estate project, understanding the behavior and performance of sellers in the market is crucial. This query has an important role in providing insights into seller performance and what properties they deal with. By identifying sellers who sell properties above the average price, we can identify high performing sellers. Additionally, showing properties that are sold above the average price can provide information about market trends such as desirable property features. Results from the query can also be used to optimize sales strategies. Knowing what features result in a property being sold at a higher price can improve sales performance through pricing strategies and marketing efforts. The average function is used to find the average price of properties sold and aggregates data and returns the value. A union set operator is also used if we want to include properties sold by a specific seller regardless of sale price.

```
SELECT state, AVG(latitude) AS avg_latitude, AVG(longitude) AS
avg_longitude
FROM
    Address
JOIN Properties
GROUP BY
    state
```

```
+-------+----------------+-----------------+
| state | avg_latitude   | avg_longitude   |
+-------+----------------+-----------------+
| st    |  0.00000000000 |   0.00000000000 |
| AL    | 32.91377566092 | -86.82966566197 |
| NULL  | 31.23221000000 | -85.39264700000 |
| MO    | 38.37980601594 | -92.08084449457 |
+-------+----------------+-----------------+
4 rows in set (0.00 sec)
```

This query calculates the average latitude and longitude for each state by joining the address table and the property table with the common key propertyId. The group by clause groups the results by the state column and the average function is used to calculate the average latitude and longitude in each state. This query is useful for our project for multiple reasons. Understanding the average latitude and longitude in each state helps with market trends. Regions with high property values or potential for growth can be identified. Marketers can also now target specific regions with relevant advertising.

```
SELECT a.city, AVG(p.price) AS avg_price FROM Address a INNER JOIN
Properties p ON a.propertyid = p.propertyid GROUP BY a.city ORDER BY
AVG(p.price) DESC LIMIT 15;
```

```
+------------------+--------------+
| city             | avg_price    |
+------------------+--------------+
| Jacksons Gap     | 3500000.0000 |
| Foristell        | 2685000.0000 |
| Town and Country | 2234000.0000 |
| Mountain Brook   | 1500000.0000 |
| Dauphin Island   | 1375000.0000 |
| Chatom           | 1039933.3333 |
| Porto Cima       |  896000.0000 |
| Greenwood        |  875000.0000 |
| West Plains      |  835000.0000 |
| Grandview        |  830000.0000 |
| Ladue            |  825000.0000 |
| Rushville        |  800000.0000 |
| Orange Beach     |  761780.0000 |
| Wetumpka         |  749900.0000 |
| Gulf Shores      |  726842.8571 |
+------------------+--------------+
15 rows in set (0.00 sec)
```

This query calculates the average price per city by joining the address and properties tables on the propertyId column. Then it orders the result by the average price in descending order. The query utilizes the average function along with the group by clause to perform aggregation by grouping the data by city. This query is also important for several reasons. Pricing trends across different cities can be analyzed. By ranking cities according to their average property price, the market can be divided into luxury markets, mid-range, and affordable markets. Lucrative markets can also be identified for development projects.

SELECT p.numBedrooms, COUNT(*) AS num_properties, MIN(p.price) AS min_price, MAX(p.price) AS max_price, AVG(p.price) AS avg_price, AVG(s.priceSold) as

avg_price_sold FROM Properties p Join Seller s on p.propertyId = s.propertyId GROUP BY p.numBedrooms ORDER BY p.numBedrooms;

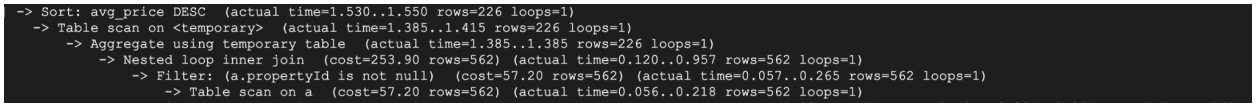
```
+-------------+----------------+-----------+-----------+--------------+----------------+
| numBedrooms | num_properties | min_price | max_price | avg_price    | avg_price_sold |
+-------------+----------------+-----------+-----------+--------------+----------------+
|           0 |              1 |         0 |         0 |       0.0000 |         0.0000 |
|           1 |              8 |    129000 |    435000 |  219525.0000 |    219525.0000 |
|           2 |             98 |     17000 |    800000 |  219705.9082 |    219705.9082 |
|           3 |            282 |     33000 |   2685000 |  300137.8652 |    300137.8652 |
|           4 |            146 |     74900 |   3500000 |  438331.2123 |    438331.2123 |
|           5 |             35 |    150000 |   2250000 |  596679.2286 |    596679.2286 |
|           6 |              4 |    349900 |    749000 |  473475.0000 |    473475.0000 |
|           8 |              2 |    150000 |    599000 |  374500.0000 |    374500.0000 |
|          10 |              1 |    649000 |    649000 |  649000.0000 |    649000.0000 |
|          20 |              1 |   1400000 |   1400000 | 1400000.0000 |   1400000.0000 |
+-------------+----------------+-----------+-----------+--------------+----------------+
10 rows in set (0.00 sec)
```

We select the number of bedrooms (`numBedrooms`) from the `Properties` table Using aggregation via `GROUP BY`. We calculate the count of properties (`num_properties`), the minimum price (`min_price`), the maximum price (`max_price`), and the average price (`avg_price`) for each number of bedrooms. The results are ordered by the number of bedrooms in ascending order. This query is useful as different buyers have different needs and preferences for the number of bedrooms in a property. This allows the buyer to select their most desired property and price differences among the number of bedrooms can be analyzed.

Indexing Analysis

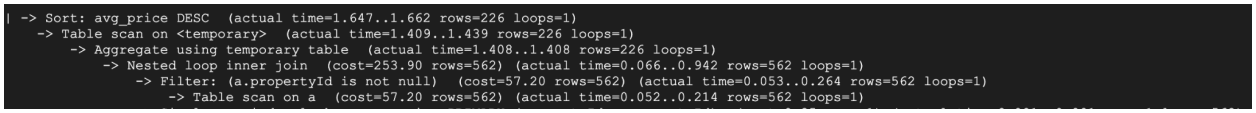
Query 1: SELECT a.city, AVG(p.price) AS avg_price FROM Address a INNER JOIN Properties p ON a.propertyid = p.propertyid GROUP BY a.city ORDER BY AVG(p.price) DESC LIMIT 15;

1. No index



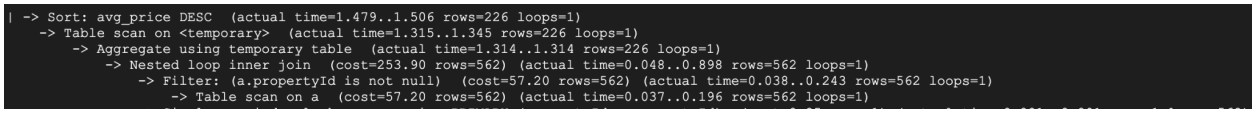
Cost = 253.90

2. Index on address.city



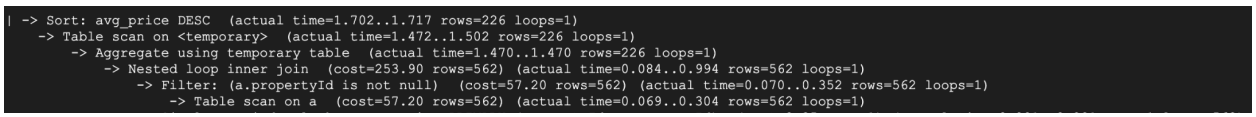
Cost = 253.90

3. Index on properties.price



Cost = 253.90

4. Index on Address.propertyId



Cost = 253.90

Tradeoffs

We implemented different indexes for this query including the propertyId, price and city. After using the Describe Analyze feature, it was found that all the different combinations of indexes produced the same cost as the query in which no index was used. The main reason for the cost optimization to not work is the low cardinality of the index values used. Another reason for the cost optimization to not work rooted from the problem of having null values in the indexed column. It was found that all the indexes that we could have chosen had a significantly low cardinality compared to the number of rows in the table. For a specific scenario, there are only 200 distinct cities in our database which end up yielding a low cardinality. This resulted in the optimizer not using the index and scanned through the whole table.

Final Index Design

Based on the outputs from Describe Analyze on different indexes, it was found that no change occurred regardless of the index due to several underlying reasons. Therefore, we decided to not use any index in our final design.

Query 2: SELECT * FROM Seller JOIN Properties WHERE sellerId IN (SELECT sellerId FROM Seller WHERE priceSold > (SELECT AVG(priceSold) FROM Seller WHERE priceSold IS NOT NULL));

1. No index

```
| -> Inner hash join (no condition)  (cost=11223.74 rows=111350) (actual time=4.983..14.560 rows=113866 loops=1)
    -> Table scan on Properties  (cost=0.31 rows=578) (actual time=0.049..0.399 rows=578 loops=1)
    -> Hash
        -> Nested loop inner join  (cost=87.44 rows=193) (actual time=3.909..4.894 rows=197 loops=1)
            -> Filter: (Seller.priceSold > (select #3))  (cost=20.01 rows=193) (actual time=3.824..4.047 rows=197 loops=1)
                -> Table scan on Seller  (cost=20.01 rows=578) (actual time=0.098..0.246 rows=578 loops=1)
                -> Select #3 (subquery in condition; run only once)
                    -> Aggregate: avg(Seller.priceSold)  (cost=110.57 rows=1) (actual time=0.318..0.320 rows=1 loops=1)
                        -> Filter: (Seller.priceSold is not null)  (cost=58.55 rows=520) (actual time=0.038..0.258 rows=578 loops=1)
                            -> Table scan on Seller  (cost=58.55 rows=578) (actual time=0.037..0.216 rows=578 loops=1)
```

Cost = 11223.74

2. Index on Seller.sellerId

```
| -> Inner hash join (no condition)  (cost=11223.74 rows=111350) (actual time=0.973..10.542 rows=113866 loops=1)
    -> Table scan on Properties  (cost=0.31 rows=578) (actual time=0.068..0.445 rows=578 loops=1)
    -> Hash
        -> Nested loop inner join  (cost=87.44 rows=193) (actual time=0.307..0.859 rows=197 loops=1)
            -> Filter: (Seller.priceSold > (select #3))  (cost=20.01 rows=193) (actual time=0.296..0.537 rows=197 loops=1)
                -> Table scan on Seller  (cost=20.01 rows=578) (actual time=0.044..0.192 rows=578 loops=1)
                -> Select #3 (subquery in condition; run only once)
                    -> Aggregate: avg(Seller.priceSold)  (cost=110.57 rows=1) (actual time=0.222..0.222 rows=1 loops=1)
                        -> Filter: (Seller.priceSold is not null)  (cost=58.55 rows=520) (actual time=0.016..0.167 rows=578 loops=1)
                            -> Table scan on Seller  (cost=58.55 rows=578) (actual time=0.015..0.129 rows=578 loops=1)
```

Cost = 11223.74

3. Index on Seller.priceSold

```
| -> Inner hash join (no condition)  (cost=11496.54 rows=113866) (actual time=0.500..10.272 rows=113866 loops=1)
    -> Table scan on Properties  (cost=0.30 rows=578) (actual time=0.050..0.393 rows=578 loops=1)
    -> Hash
        -> Nested loop inner join  (cost=108.66 rows=197) (actual time=0.020..0.419 rows=197 loops=1)
            -> Filter: (Seller.priceSold > (select #3))  (cost=39.71 rows=197) (actual time=0.009..0.132 rows=197 loops=1)
                -> Covering index range scan on Seller using seller_pricesold_idx over (341693 <= priceSold)  (cost=39.71 rows=197) (actual time=0.007..0.096 rows=197 loops=1)
                -> Select #3 (subquery in condition; run only once)
                    -> Aggregate: avg(Seller.priceSold)  (cost=116.35 rows=1) (actual time=0.222..0.223 rows=1 loops=1)
                        -> Filter: (Seller.priceSold is not null)  (cost=58.55 rows=578) (actual time=0.030..0.166 rows=578 loops=1)
                            -> Covering index scan on Seller using seller_pricesold_idx  (cost=58.55 rows=578) (actual time=0.028..0.127 rows=578 loops=1)
```

Cost = 11496.54

4. Index on Seller.propertyId

```
-> Inner hash join (no condition)  (cost=11223.74 rows=111350) (actual time=0.925..9.901 rows=113866 loops=1)
    -> Table scan on Properties  (cost=0.31 rows=578) (actual time=0.042..0.378 rows=578 loops=1)
    -> Hash
        -> Nested loop inner join  (cost=87.44 rows=193) (actual time=0.400..0.851 rows=197 loops=1)
            -> Filter: (Seller.priceSold > (select #3))  (cost=20.01 rows=193) (actual time=0.387..0.605 rows=197 loops=1)
                -> Table scan on Seller  (cost=20.01 rows=578) (actual time=0.098..0.221 rows=578 loops=1)
                -> Select #3 (subquery in condition; run only once)
                    -> Aggregate: avg(Seller.priceSold)  (cost=110.57 rows=1) (actual time=0.277..0.277 rows=1 loops=1)
                        -> Filter: (Seller.priceSold is not null)  (cost=58.55 rows=520) (actual time=0.016..0.168 rows=578 loops=1)
                            -> Table scan on Seller  (cost=58.55 rows=578) (actual time=0.016..0.129 rows=578 loops=1)
```

Cost = 11223.74

Tradeoffs

We implemented different indexes for this query including the sellerId, priceSold and propertyId. After using the Describe Analyze feature, it was found that using sellerId and propertyId as indexes did not change the cost. However, using priceSold attribute as the index cause the overall cost to increase by a small factor, which is not ideal. This particular query is a bit more complex with 2 nested subqueries and joins. Maintaining the index with this query might have caused an overhead and led to an increased cost. Another reason for the cost optimization to

not work rooted from the problem of having null values in the indexed column.This resulted in the optimizer not using the index and scanned through the whole table.

Final Index Design

Based on the outputs from Describe Analyze on different indexes, it was found that no change occurred in the case of selecting sellerId and propertyId as indexes. In addition, choosing priceSold as an index caused a degradation in the cost. Therefore, we decided to not use any index in our final design.

Query 3: SELECT state, AVG(latitude) AS avg_latitude, AVG(longitude) AS avg_longitude FROM Address JOIN Properties GROUP BY state

1. No index

```
| -> Table scan on <temporary>  (actual time=116.678..116.679 rows=4 loops=1)
    -> Aggregate using temporary table  (actual time=116.675..116.675 rows=4 loops=1)
        -> Inner hash join (no condition)  (cost=32542.12 rows=324836) (actual time=0.308..28.804 rows=324836 loops=1)
            -> Covering index scan on Properties using PRIMARY  (cost=0.11 rows=578) (actual time=0.024..0.288 rows=578 loops=1)
            -> Hash
                -> Table scan on Address  (cost=57.20 rows=562) (actual time=0.061..0.210 rows=562 loops=1)
 |
```

Cost = 32542.12

2. Index on Address.state

```
| -> Table scan on <temporary>  (actual time=116.307..116.308 rows=4 loops=1)
    -> Aggregate using temporary table  (actual time=116.305..116.305 rows=4 loops=1)
        -> Inner hash join (no condition)  (cost=32542.12 rows=324836) (actual time=0.374..27.824 rows=324836 loops=1)
            -> Covering index scan on Properties using PRIMARY  (cost=0.11 rows=578) (actual time=0.026..0.251 rows=578 loops=1)
            -> Hash
                -> Table scan on Address  (cost=57.20 rows=562) (actual time=0.060..0.214 rows=562 loops=1)
 |
```

Cost = 32542.12

3. Index on Address.latitude

```
| -> Table scan on <temporary>  (actual time=119.176..119.178 rows=4 loops=1)
    -> Aggregate using temporary table  (actual time=119.173..119.173 rows=4 loops=1)
        -> Inner hash join (no condition)  (cost=32542.12 rows=324836) (actual time=0.443..30.343 rows=324836 loops=1)
            -> Covering index scan on Properties using PRIMARY  (cost=0.11 rows=578) (actual time=0.031..0.320 rows=578 loops=1)
            -> Hash
                -> Table scan on Address  (cost=57.20 rows=562) (actual time=0.058..0.282 rows=562 loops=1)
 |
```

Cost = 32542.12

4. Index on Address.longitude

```
| -> Table scan on <temporary>  (actual time=118.526..118.527 rows=4 loops=1)
    -> Aggregate using temporary table  (actual time=118.523..118.523 rows=4 loops=1)
        -> Inner hash join (no condition)  (cost=32542.12 rows=324836) (actual time=0.298..28.870 rows=324836 loops=1)
            -> Covering index scan on Properties using PRIMARY  (cost=0.11 rows=578) (actual time=0.024..0.284 rows=578 loops=1)
            -> Hash
                -> Table scan on Address  (cost=57.20 rows=562) (actual time=0.057..0.202 rows=562 loops=1)
 |
```

Cost = 32542.12

Tradeoffs

We implemented different indexes for this query including the state, latitude and longitude. After using the Describe Analyze feature, it was found that all the different combinations of indexes produced the same cost as the query in which no index was used. The main reason for the cost optimization to not work is the low cardinality of the index values used. It was found that all the

indexes that we could have chosen had a significantly low cardinality compared to the number of rows in the table. For a specific scenario, there are only 4 distinct states in our database which end up yielding a low cardinality. In addition, there were also a lot of rows containing null values. This resulted in the optimizer not using the index and scanned through the whole table.

Final Index Design

Based on the outputs from Describe Analyze on different indexes, it was found that no change occurred regardless of the index due to several underlying reasons. Therefore, we decided to not use any index in our final design.

Query 4: SELECT p.numBedrooms, COUNT(*) AS num_properties, MIN(p.price) AS min_price, MAX(p.price) AS max_price, AVG(p.price) AS avg_price, AVG(s.priceSold) as avg_price_sold FROM Properties p Join Seller s on p.propertyId = s.propertyId GROUP BY p.numBedrooms ORDER BY p.numBedrooms;

1. No index

```
| -> Sort: p.numBedrooms  (actual time=1.374..1.375 rows=10 loops=1)
    -> Table scan on <temporary>  (actual time=1.355..1.357 rows=10 loops=1)
        -> Aggregate using temporary table  (actual time=1.354..1.354 rows=10 loops=1)
            -> Nested loop inner join  (cost=260.85 rows=578) (actual time=0.066..1.044 rows=578 loops=1)
                -> Filter: (s.propertyId is not null)  (cost=58.55 rows=578) (actual time=0.054..0.243 rows=578 loops=1)
                    -> Table scan on s  (cost=58.55 rows=578) (actual time=0.053..0.198 rows=578 loops=1)
```

Cost = 260.85

2. Index on Properties.numBedrooms

```
| -> Sort: p.numBedrooms  (actual time=1.314..1.315 rows=10 loops=1)
    -> Table scan on <temporary>  (actual time=1.296..1.297 rows=10 loops=1)
        -> Aggregate using temporary table  (actual time=1.294..1.294 rows=10 loops=1)
            -> Nested loop inner join  (cost=260.85 rows=578) (actual time=0.068..0.915 rows=578 loops=1)
                -> Filter: (s.propertyId is not null)  (cost=58.55 rows=578) (actual time=0.055..0.244 rows=578 loops=1)
                    -> Table scan on s  (cost=58.55 rows=578) (actual time=0.054..0.200 rows=578 loops=1)
```

Cost = 260.85

3. Index on Seller.propertyId

```
| -> Sort: p.numBedrooms  (actual time=1.374..1.375 rows=10 loops=1)
    -> Table scan on <temporary>  (actual time=1.355..1.357 rows=10 loops=1)
        -> Aggregate using temporary table  (actual time=1.354..1.354 rows=10 loops=1)
            -> Nested loop inner join  (cost=260.85 rows=578) (actual time=0.066..1.044 rows=578 loops=1)
                -> Filter: (s.propertyId is not null)  (cost=58.55 rows=578) (actual time=0.054..0.243 rows=578 loops=1)
                    -> Table scan on s  (cost=58.55 rows=578) (actual time=0.053..0.198 rows=578 loops=1)
```

Cost = 260.85

4. Index on Properties.price

```
| -> Sort: p.numBedrooms  (actual time=1.239..1.240 rows=10 loops=1)
    -> Table scan on <temporary>  (actual time=1.224..1.225 rows=10 loops=1)
        -> Aggregate using temporary table  (actual time=1.223..1.223 rows=10 loops=1)
            -> Nested loop inner join  (cost=260.85 rows=578) (actual time=0.046..0.910 rows=578 loops=1)
                -> Filter: (s.propertyId is not null)  (cost=58.55 rows=578) (actual time=0.033..0.239 rows=578 loops=1)
                    -> Table scan on s  (cost=58.55 rows=578) (actual time=0.032..0.175 rows=578 loops=1)
```

Cost = 260.85

Tradeoffs

We implemented different indexes for this query including the propertyId, price and numBedrooms. After using the Describe Analyze feature, it was found that all the different combinations of indexes produced the same cost as the query in which no index was used. The main reason for the cost optimization to not work is the low cardinality of the index values used. It was found that all the indexes that we could have chosen had a significantly low cardinality

compared to the number of rows in the table. Specifically, the numBedrooms attribute has a cardinality of 10, which is a very low value. This resulted in the optimizer not using the index and scanned through the whole table.

Final Index Design

Based on the outputs from Describe Analyze on different indexes, it was found that no change occurred regardless of the index due to several underlying reasons. Therefore, we decided to not use any index in our final design.