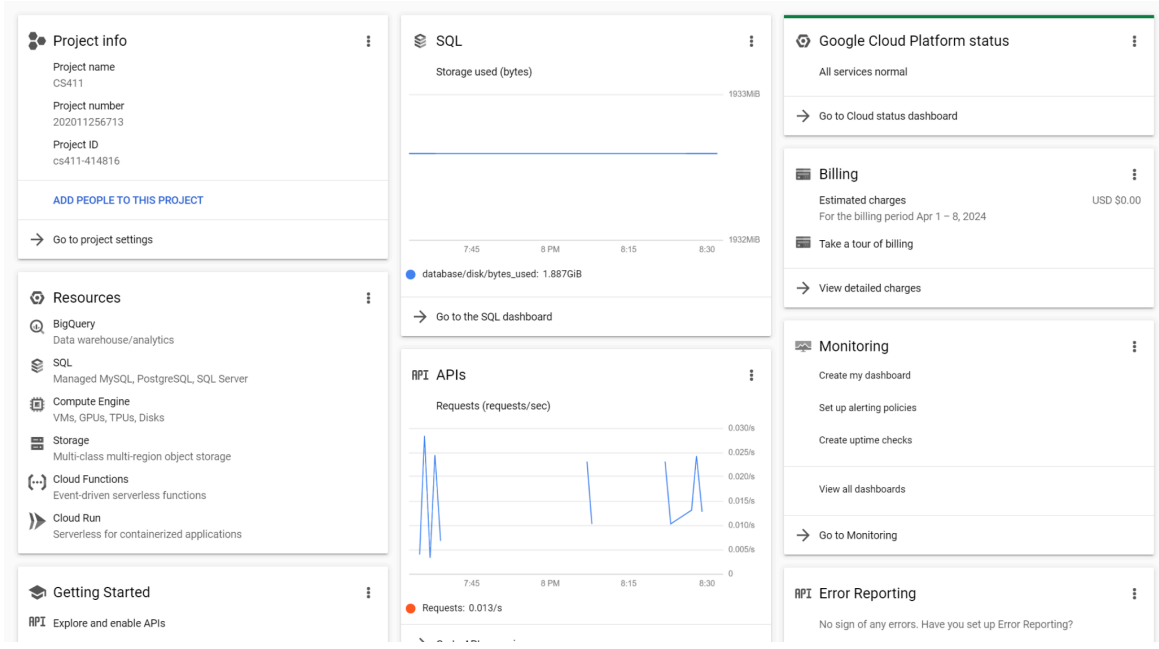
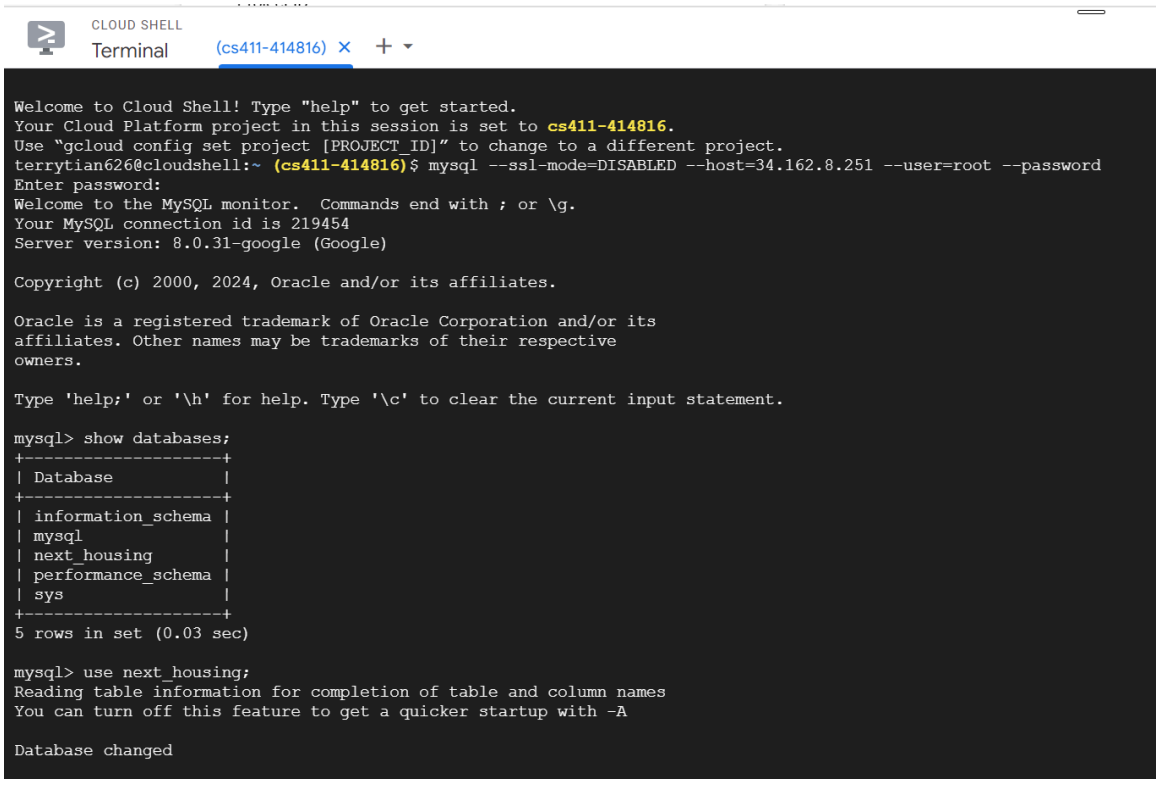
Stage 3

GCP:

We have implemented our database on GCP. The screenshot below shows the connection to the database:





DDL:

```
CREATE TABLE `Favorites` (
  `FavoriteID` int NOT NULL,
  `UserID` int NOT NULL,
  `ListingID` int NOT NULL,
  `PriceAtFavTime` decimal(10,0) NOT NULL,
  `FavTime` timestamp NOT NULL,
  PRIMARY KEY (`FavoriteID`),
  CONSTRAINT `FAV_LID` FOREIGN KEY (`ListingID`) REFERENCES `Listing` (`ListingID`) ON DELETE
CASCADE,
  CONSTRAINT `FAV_UID` FOREIGN KEY (`UserID`) REFERENCES `User` (`UserID`) ON DELETE
CASCADE
);

CREATE TABLE `FloorPlan` (
  `FloorPlanID` int NOT NULL,
  `PropertyID` BIGINT NOT NULL,
  `Bedrooms` int,
  `Bathrooms` int,
  `SquareFeet` int,
  `Price` decimal(10,2),
  `Currency` varchar(50),
  `Fee` decimal(10,0),
  `HasPhoto` tinyint,
  `PetsAllowed` varchar(255),
  PRIMARY KEY (`FloorPlanID`, `PropertyID`),
  CONSTRAINT `PID` FOREIGN KEY (`PropertyID`) REFERENCES `Property` (`PropertyID`) ON DELETE
CASCADE
);

CREATE TABLE `Listing` (
  `ListingID` int NOT NULL,
  `PropertyID` BIGINT NOT NULL,
  `AvailableDate` date,
  `Description` varchar(500),
  PRIMARY KEY (`ListingID`),
  CONSTRAINT `Listing_PID` FOREIGN KEY (`PropertyID`) REFERENCES `Property` (`PropertyID`)
);

CREATE TABLE `Property` (
  `PropertyID` int NOT NULL,
  `Address` varchar(255),
  `Amenities` text,
  `ContactNumber` varchar(255),
```

```
  `Latitude` decimal(10,4),
  `Longitude` decimal(10,4),
  `Source` varchar(255),
  `State` varchar(10),
  `CityName` varchar(50),
  `Category` varchar(50),
  `Title` varchar(255),
  `Description` text,
  `Time` TIMESTAMP,
  PRIMARY KEY (`PropertyID`)
);

CREATE TABLE `Rating` (
  `RatingID` int NOT NULL,
  `PropertyID` int NOT NULL,
  `Score` decimal(10,0) NOT NULL,
  `Description` text,
  PRIMARY KEY (`RatingID`),
  CONSTRAINT `RAT_PID` FOREIGN KEY (`PropertyID`) REFERENCES `Property` (`PropertyID`)
);

CREATE TABLE `User` (
  `UserID` int NOT NULL,
  `Username` varchar(50)NOT NULL,
  `Password` varchar(64)NOT NULL,
  `Email` varchar(100)NOT NULL,
  PRIMARY KEY (`UserID`)
);
```

```
mysql> SELECT COUNT(*) FROM Favorites;
+----------+
| COUNT(*) |
+----------+
|     1000 |
+----------+
1 row in set (0.06 sec)

mysql> SELECT COUNT(*) FROM FloorPlan;
+----------+
| COUNT(*) |
+----------+
|    94600 |
+----------+
1 row in set (0.46 sec)

mysql> SELECT COUNT(*) FROM Listing;
+----------+
| COUNT(*) |
+----------+
|     3000 |
+----------+
1 row in set (0.06 sec)

mysql> SELECT COUNT(*) FROM Property;
+----------+
| COUNT(*) |
+----------+
|    94601 |
+----------+
1 row in set (1.41 sec)

mysql> SELECT COUNT(*) FROM Rating;
+----------+
| COUNT(*) |
+----------+
|     1000 |
+----------+
1 row in set (0.24 sec)

mysql> SELECT COUNT(*) FROM User;
+----------+
| COUNT(*) |
+----------+
|     1000 |
+----------+
1 row in set (0.04 sec)
```

Advanced SQL Queries

Query 1: Get information of top 15 listings of properties that have an average score greater than 4.0

```
SELECT L.ListingID, L.PropertyID, L.AvailableDate, FP.Bedrooms, FP.Bathrooms, AR.AverageRating
FROM Listing L
JOIN Property P ON L.PropertyID = P.PropertyID
JOIN FloorPlan FP ON P.PropertyID = FP.PropertyID
JOIN (SELECT R.PropertyID, AVG(R.Score) AS AverageRating
   FROM Rating R
   GROUP BY R.PropertyID
   HAVING AVG(R.Score) > 4.0
) AR ON P.PropertyID = AR.PropertyID
WHERE L.AvailableDate > CURRENT_DATE
ORDER BY AR.AverageRating DESC
LIMIT 15;
```

```
mysql> SELECT L.ListingID, L.PropertyID, L.AvailableDate, FP.Bedrooms, FP.Bathrooms, AR.AverageRating
    -> FROM Listing L
    -> JOIN Property P ON L.PropertyID = P.PropertyID
    -> JOIN FloorPlan FP ON P.PropertyID = FP.PropertyID
    -> JOIN (
    ->     SELECT R.PropertyID, AVG(R.Score) AS AverageRating
    ->     FROM Rating R
    ->     GROUP BY R.PropertyID
    ->     HAVING AVG(R.Score) > 4.0
    -> ) AR ON P.PropertyID = AR.PropertyID
    -> WHERE L.AvailableDate > CURRENT_DATE
    -> ORDER BY AR.AverageRating DESC
    -> LIMIT 15;
+-----------+------------+---------------+----------+-----------+---------------+
| ListingID | PropertyID | AvailableDate | Bedrooms | Bathrooms | AverageRating |
+-----------+------------+---------------+----------+-----------+---------------+
|      1777 | 5121057044 | 2025-07-08    |        2 |         2 |        5.0000 |
|      2661 | 5121092991 | 2025-05-05    |        1 |         1 |        5.0000 |
|       293 | 5121348312 | 2025-11-28    |        2 |         1 |        5.0000 |
|      2367 | 5121350430 | 2025-08-30    |        1 |         1 |        5.0000 |
|      1406 | 5121418626 | 2025-01-29    |        2 |         1 |        5.0000 |
|      2587 | 5121426212 | 2025-09-11    |        2 |         2 |        5.0000 |
|      2286 | 5121470971 | 2025-04-09    |        2 |         2 |        5.0000 |
|      1999 | 5121566037 | 2024-05-04    |        3 |         2 |        5.0000 |
|       943 | 5121568693 | 2024-09-18    |        2 |         2 |        5.0000 |
|      2993 | 5121733680 | 2024-08-18    |        1 |         1 |        5.0000 |
|      1074 | 5121787044 | 2025-10-05    |        2 |         2 |        5.0000 |
|      1882 | 5121868919 | 2025-11-08    |        3 |         2 |        5.0000 |
|      1101 | 5121872081 | 2025-01-14    |        1 |         1 |        5.0000 |
|      1511 | 5121875768 | 2025-05-16    |        1 |         1 |        5.0000 |
|      2998 | 5121887669 | 2024-04-26    |        2 |         2 |        5.0000 |
+-----------+------------+---------------+----------+-----------+---------------+
15 rows in set (0.39 sec)
```

Query 2: Get available listing match a user's preference

Finds available listings that are either studios or 1-bedroom 1-bathroom (1b1b) units, priced under $1400, and orders the results by rating.

```
SELECT L.ListingID, P.State, FP.Price, FP.Bedrooms, FP.Bathrooms
FROM Listing L
JOIN Property P ON L.PropertyID = P.PropertyID
```

```
JOIN FloorPlan FP ON P.PropertyID = FP.PropertyI
WHERE L.AvailableDate > CURRENT_DATE
   AND ((FP.Bedrooms = 1 AND FP.Bathrooms = 1) OR FP.Bedrooms = 0)
   AND FP.Price < 1400
GROUP BY L.ListingID, P.State, FP.Price, FP.Bedrooms, FP.Bathrooms
ORDER BY  FP.Price DESC LIMIT 15;
```

```
mysql> SELECT L.ListingID, P.State, FP.Price, FP.Bedrooms, FP.Bathrooms
    -> FROM Listing L
    -> JOIN Property P ON L.PropertyID = P.PropertyID
    -> JOIN FloorPlan FP ON P.PropertyID = FP.PropertyID
    -> WHERE L.AvailableDate > CURRENT_DATE
    ->     AND ((FP.Bedrooms = 1 AND FP.Bathrooms = 1) OR FP.Bedrooms = 0)
    ->     AND FP.Price < 1400
    -> GROUP BY L.ListingID, P.State, FP.Price, FP.Bedrooms, FP.Bathrooms
    -> ORDER BY  FP.Price DESC LIMIT 15;
+-----------+-------+---------+----------+-----------+
| ListingID | State | Price   | Bedrooms | Bathrooms |
+-----------+-------+---------+----------+-----------+
|      2608 | FL    | 1392.00 |        1 |         1 |
|       360 | CA    | 1391.00 |        1 |         1 |
|      1646 | IL    | 1390.00 |        1 |         1 |
|      2504 | GA    | 1388.00 |        1 |         1 |
|       623 | GA    | 1388.00 |        1 |         1 |
|       120 | RI    | 1385.00 |        1 |         1 |
|      2376 | TN    | 1378.00 |        1 |         1 |
|       860 | PA    | 1375.00 |        1 |         1 |
|       541 | RI    | 1375.00 |        1 |         1 |
|      1951 | NC    | 1373.00 |        1 |         1 |
|      1316 | TX    | 1373.00 |        1 |         1 |
|        67 | NC    | 1370.00 |        1 |         1 |
|      1942 | FL    | 1355.00 |        1 |         1 |
|      2447 | CA    | 1355.00 |        1 |         1 |
|      1117 | FL    | 1355.00 |        1 |         1 |
+-----------+-------+---------+----------+-----------+
15 rows in set (0.78 sec)
```

Query 3: Map: Display properties on the map with their number of available listings
(Get properties and its location with available listings that have at least 2 bedrooms and are priced between $800 to $1400)

```
SELECT P.PropertyID, P.Latitude, P.Longitude,
                COUNT(L.ListingID) AS TotalAvailableListings
FROM Property P
JOIN Listing L ON P.PropertyID = L.PropertyID
JOIN FloorPlan FP ON P.PropertyID = FP.PropertyID
WHERE L.AvailableDate > CURRENT_DATE
       AND FP.Price BETWEEN 800 AND 1400
       AND FP.Bedrooms >= 2
GROUP BY P.PropertyID, P.Address, P.Latitude, P.Longitude
```

LIMIT 15;

```
mysql> SELECT P.PropertyID, P.Latitude, P.Longitude,
    ->
Display all 881 possibilities? (y or n)
    ->     COUNT(L.ListingID) AS TotalAvailableListings
    -> FROM Property P
    -> JOIN Listing L ON P.PropertyID = L.PropertyID
    -> JOIN FloorPlan FP ON P.PropertyID = FP.PropertyID
    -> WHERE L.AvailableDate > CURRENT_DATE
    ->     AND FP.Price BETWEEN 800 AND 1400
    ->     AND FP.Bedrooms >= 2
    -> GROUP BY P.PropertyID, P.Address, P.Latitude, P.Longitude
    -> LIMIT 15;
+------------+----------+-----------+-----------------------+
| PropertyID | Latitude | Longitude | TotalAvailableListings |
+------------+----------+-----------+-----------------------+
| 5122166454 |  29.5126 |  -98.3661 |                     2 |
| 5121956986 |  35.0072 |  -80.9453 |                     1 |
| 5121517051 |  42.4513 |  -87.8575 |                     1 |
| 5122255890 |  33.9743 |  -84.2384 |                     1 |
| 5122258448 |  29.6546 |  -98.6156 |                     2 |
| 5121301353 |  43.1921 |  -89.2195 |                     1 |
| 5121749303 |  40.1306 |  -75.3475 |                     1 |
| 5121965384 |  29.8979 |  -93.9549 |                     1 |
| 5122084698 |  39.3916 |  -76.5350 |                     2 |
| 5121962784 |  35.1056 |  -81.1831 |                     1 |
| 5121426839 |  35.2016 |  -80.8124 |                     1 |
| 5121072309 |  33.9013 |  -84.3058 |                     2 |
| 5121651118 |  41.7552 |  -88.2415 |                     1 |
| 5121870610 |  42.7030 |  -84.5463 |                     1 |
| 5121874714 |  26.2480 |  -80.2094 |                     1 |
+------------+----------+-----------+-----------------------+
15 rows in set (0.94 sec)
```

Query 4: Find top 15 most popular properties, measured by the number of favorites, that have a pool amenity.

```
SELECT P.PropertyID, P.Source, COUNT(F.FavoriteID) AS FavoritesCount, AVG(R.Score) AS AverageRating
FROM Property P
JOIN Listing L ON P.PropertyID = L.PropertyID
JOIN Favorites F ON L.ListingID = F.ListingID
JOIN Rating R ON P.PropertyID = R.PropertyID
WHERE L.AvailableDate > CURRENT_DATE
    AND P.Amenities LIKE '%pool%'
GROUP BY P.PropertyID, P.Source
HAVING AVG(R.Score) IS NOT NULL
ORDER BY FavoritesCount DESC, AverageRating DESC
LIMIT 15;
```

```
mysql> SELECT P.PropertyID, P.Source, COUNT(F.FavoriteID) AS FavoritesCount, AVG(R.
Score) AS AverageRating
    -> FROM Property P
    -> JOIN Listing L ON P.PropertyID = L.PropertyID
    -> JOIN Favorites F ON L.ListingID = F.ListingID
    -> JOIN Rating R ON P.PropertyID = R.PropertyID
    -> WHERE L.AvailableDate > CURRENT_DATE
    ->       AND P.Amenities LIKE '%pool%'
    -> GROUP BY P.PropertyID, P.Source
    -> HAVING AVG(R.Score) IS NOT NULL
    -> ORDER BY FavoritesCount DESC, AverageRating DESC
    -> LIMIT 15;

+------------+--------------+----------------+---------------+
| PropertyID | Source       | FavoritesCount | AverageRating |
+------------+--------------+----------------+---------------+
| 5121733680 | RentDigs.com |              2 |        5.0000 |
| 5121920055 | RentDigs.com |              2 |        4.0000 |
| 5121470971 | RentDigs.com |              1 |        5.0000 |
| 5121057044 | RentDigs.com |              1 |        5.0000 |
| 5121092991 | RentDigs.com |              1 |        5.0000 |
| 5122222394 | RentDigs.com |              1 |        5.0000 |
| 5121958392 | RentDigs.com |              1 |        4.0000 |
| 5121516092 | RentDigs.com |              1 |        4.0000 |
| 5121232059 | RentDigs.com |              1 |        4.0000 |
| 5121822142 | RentDigs.com |              1 |        4.0000 |
| 5121384072 | RentDigs.com |              1 |        4.0000 |
| 5121580960 | RentDigs.com |              1 |        4.0000 |
| 5121307265 | RentDigs.com |              1 |        4.0000 |
| 5121823264 | RentDigs.com |              1 |        4.0000 |
| 5122176344 | RentDigs.com |              1 |        3.0000 |
+------------+--------------+----------------+---------------+
15 rows in set (1.04 sec)
```

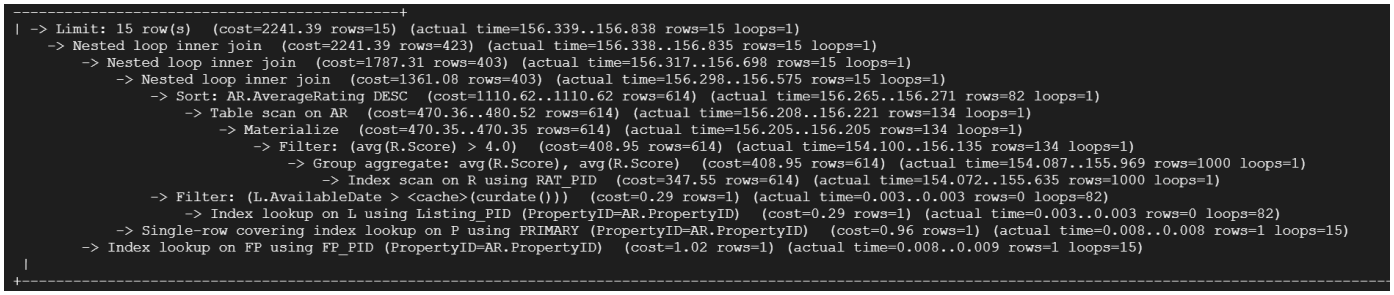
Part 2 Indexing:

Query 1:
Initial:

```
EXPLAIN ANALYZE SELECT L.ListingID, L.PropertyID, L.AvailableDate, FP.Bedrooms, FP.Bathrooms,
AR.AverageRating
FROM Listing L
JOIN Property P ON L.PropertyID = P.PropertyID
JOIN FloorPlan FP ON P.PropertyID = FP.PropertyID
JOIN (SELECT R.PropertyID, AVG(R.Score) AS AverageRating
   FROM Rating R
   GROUP BY R.PropertyID
   HAVING AVG(R.Score) > 4.0
) AR ON P.PropertyID = AR.PropertyID
WHERE L.AvailableDate > CURRENT_DATE
ORDER BY AR.AverageRating DESC
```

LIMIT 15;

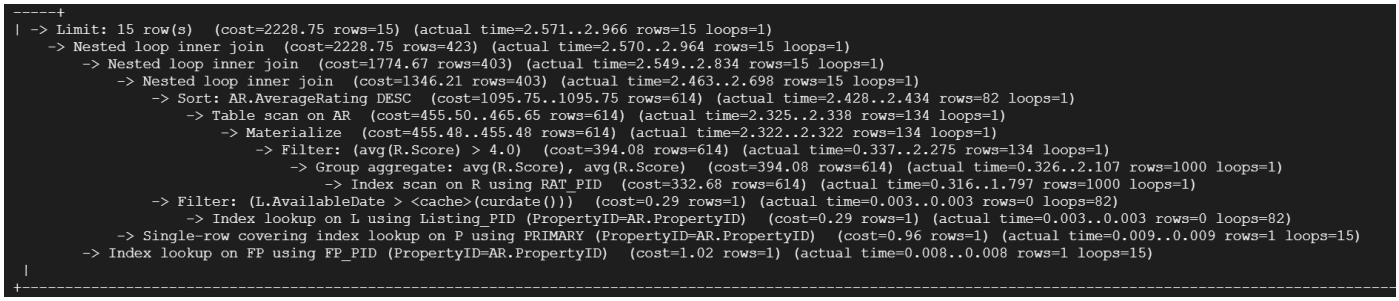
Before:



index 1:

CREATE INDEX idx_rating_score ON Rating(Score);

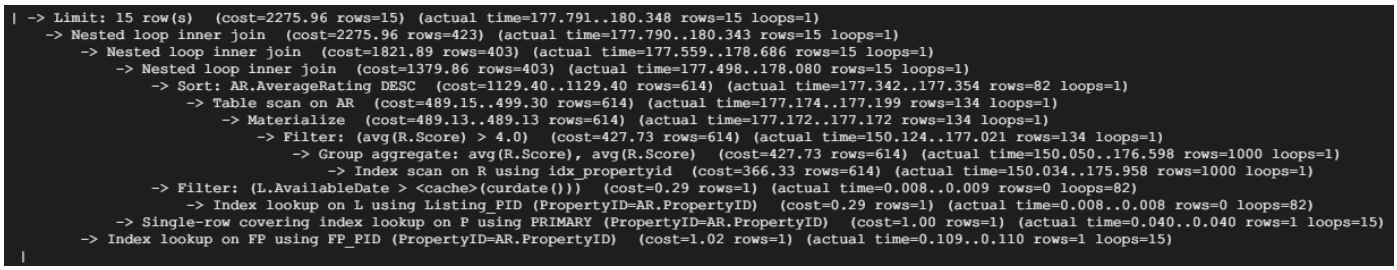
After:



Index 2:

CREATE INDEX idx_listing_availabledate ON Listing(AvailableDate);

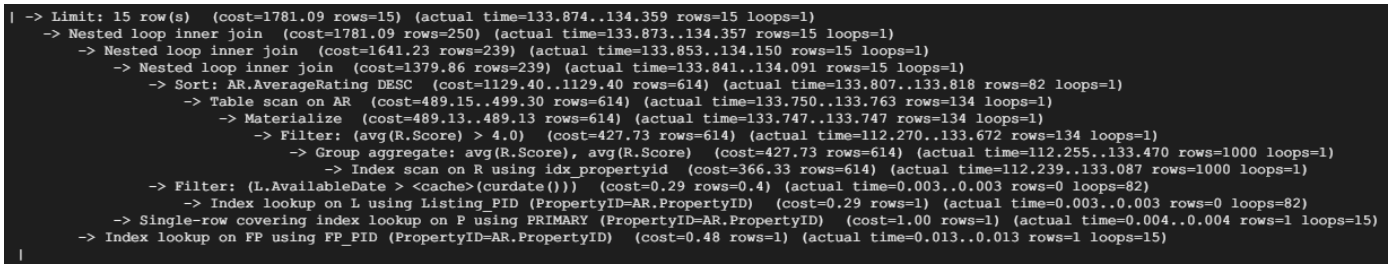
After:



Index 3:

CREATE INDEX idx_floorplan_bedrooms_bathrooms ON FloorPlan(Bedrooms, Bathrooms);

After:



Cost Analysis:

Index 1: The addition of the index ‘idx_rating_score‘ on the Score column in the Rating table shows a slight reduction in the estimated query cost. Before the index was added, the total cost of the query was 2241.39, and after the index was added, the cost decreased to 2228.75. The reduction shows that the index has improved the efficiency of filtering operations, particularly those involving the Score field used in computing the average rating.

Index 2: The addition of the index ”idx_listing_availabledate” on the AvailableDate column in the Listing table resulted in an increase in the estimated query cost from 2241.39 to 2275.96. This indicates that the new index did not reduce the cost of operations involving the AvailableDate filtering but slightly increased the overall query cost. It appears the index may have introduced some overhead, potentially due to the manner in which the index interacts with other components of the query plan.

Index 3: The addition of the composite index “idx_floorplan_bedrooms_bathrooms” on the Bedrooms and Bathrooms columns in the FloorPlan table resulted in a significant reduction in the estimated query cost, from 2241.39 to 1781.09. This suggests that the index effectively optimized data access patterns for queries involving these two columns. The index likely enhances the performance of operations that sort or filter based on the Bedrooms and Bathrooms attributes.

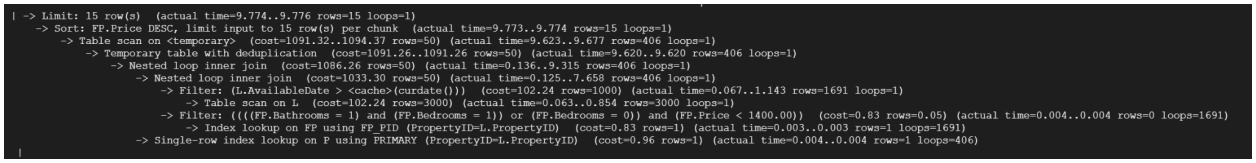
Based on the above result, The third index idx_floorplan_bedrooms_bathrooms, is the best choice for the final design based on its significant reduction in cost, which decreased from 2241.39 to 1781.09. This index directly addresses the specific needs of the query by optimizing access to the FloorPlan table where the attributes Bedrooms and Bathrooms are critical for filtering and joining operations.

Query 2:
Initial:

```
EXPLAIN ANALYZE
SELECT L.ListingID, P.State, FP.Price, FP.Bedrooms, FP.Bathrooms
FROM Listing L
JOIN Property P ON L.PropertyID = P.PropertyID
JOIN FloorPlan FP ON P.PropertyID = FP.PropertyID
WHERE L.AvailableDate > CURRENT_DATE
```

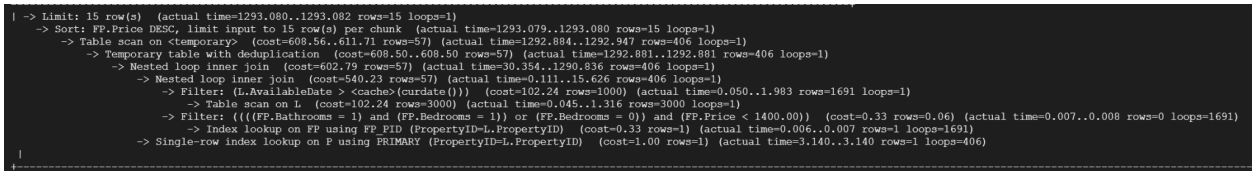
```
    AND ((FP.Bedrooms = 1 AND FP.Bathrooms = 1) OR FP.Bedrooms = 0)
    AND FP.Price < 1400
GROUP BY L.ListingID, P.State, FP.Price, FP.Bedrooms, FP.Bathrooms
ORDER BY  FP.Price DESC LIMIT 15;
```

Before:



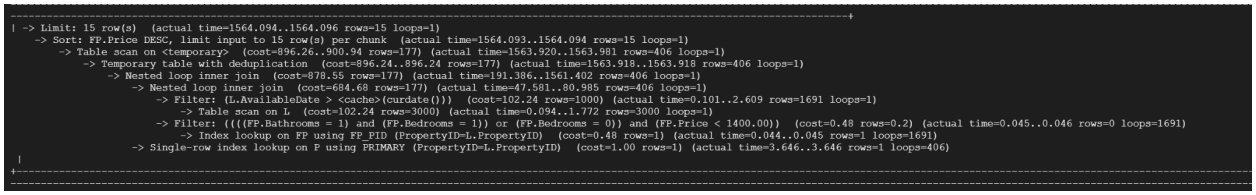
Index 1:

CREATE INDEX idx_floorplan_price ON FloorPlan(Price);



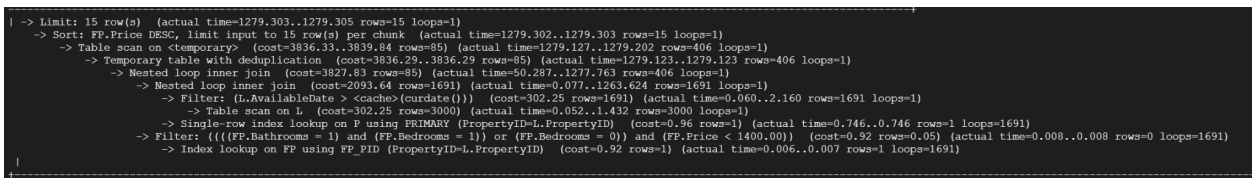
Index 2:

CREATE INDEX idx_floorplan_bedrooms_bathrooms ON FloorPlan(Bedrooms, Bathrooms);



Index 3:

CREATE INDEX idx_listing_availabledate ON Listing(AvailableDate);



Cost Analysis

Initial:
Cost: 1091.32 to 1094.37

Analysis and Final Selection:

Index 1 (idx_floorplan_price):
Cost: Reduced to approximately 608.56 to 611.71

Adding an index on Price significantly reduced the total cost. It indicates index on price can be effective in optimizing queries that filter and sort by price. Since the Price field is directly used in the WHERE and ORDER BY clauses, this index optimized the query performance on cost.

Index 2 (idx_floorplan_bedrooms_bathrooms):
Cost: Reduced to approximately 896.26 to 900.94
The composite index offered performance improvement for filtering operations based on `Bedrooms` and `Bathrooms`, though not as significant as the `Price` index. The reduction in total cost indicates that the composite index was effective in handling specific filtering conditions.

Index 3 (idx_listing_availabledate):
Cost: Significantly increased to approximately 3836.33 to 3839.84
Index on AvailableDate led to a significant cost increase. It indicates that this index did not improve and may even have reduced performance due to the additional overhead of maintaining the index. This might be because even though AvailableDate is used in the WHERE clause, its filtering effect on the result set is not as significant as other conditions, or the data distribution makes the index less effective.
In summary, considering the impacts of all indexes on query performance, the idx_floorplan_price was selected as the final index design due to its substantial performance boost in optimizing query operations, especially for price-related filtering and sorting. The 'Price' field, being a key factor in filtering and ordering, directly affects the core aspects of query efficiency, offering the most significant performance gain.

Although the idx_floorplan_bedrooms_bathrooms index did provide some degree of performance improvement, its impact was less pronounced compared to the 'Price' index. Therefore, the 'Price' index should be prioritized as part of the optimization strategy. This decision underscores the importance of selecting indexes based on the specific needs and operational characteristics of the query to ensure targeted optimization of query performance.

Query 3:

Initial:

```
EXPLAIN ANALYZE
SELECT P.PropertyID, P.Latitude, P.Longitude,
                COUNT(L.ListingID) AS TotalAvailableListings
FROM Property P
JOIN Listing L ON P.PropertyID = L.PropertyID
JOIN FloorPlan FP ON P.PropertyID = FP.PropertyID
WHERE L.AvailableDate > CURRENT_DATE
      AND FP.Price BETWEEN 800 AND 1400
      AND FP.Bedrooms >= 2
GROUP BY P.PropertyID, P.Address, P.Latitude, P.Longitude
LIMIT 15;
```

Before:

```
| -> Limit: 15 row(s)  (actual time=682.221..682.225 rows=15 loops=1)
    -> Table scan on <temporary>  (actual time=682.219..682.222 rows=15 loops=1)
        -> Aggregate using temporary table  (actual time=682.217..682.217 rows=390 loops=1)
            -> Nested loop inner join  (cost=2167.89 rows=50) (actual time=44.468..681.030 rows=425 loops=1)
                -> Nested loop inner join  (cost=1134.75 rows=1000) (actual time=44.438..671.353 rows=1687 loops=1)
                    -> Filter: (L.AvailableDate > <cache>(curdate()))  (cost=102.24 rows=1000) (actual time=44.407..45.879 rows=1687 loops=1)
                        -> Table scan on L  (cost=102.24 rows=3000) (actual time=44.397..45.426 rows=3000 loops=1)
                    -> Single-row index lookup on P using PRIMARY (PropertyID=L.PropertyID)  (cost=0.93 rows=1) (actual time=0.370..0.371 rows=1 loops=1687)
                -> Filter: ((FP.Price between 800 and 1400) and (FP.Bedrooms >= 2))  (cost=0.93 rows=0.05) (actual time=0.005..0.005 rows=0 loops=1687)
                    -> Index lookup on FP using FP_PID (PropertyID=L.PropertyID)  (cost=0.93 rows=1) (actual time=0.004..0.005 rows=1 loops=1687)
|
```

Index 1:

CREATE INDEX idx_listing_availabledate ON Listing(AvailableDate);

Reason: We chose AvailableDate as the index because it is a critical filter in the WHERE clause, and indexes on such columns may provide faster data retrieval.

After:

```
| -> Limit: 15 row(s)  (actual time=929.555..929.559 rows=15 loops=1)
    -> Table scan on <temporary>  (actual time=929.553..929.556 rows=15 loops=1)
        -> Aggregate using temporary table  (actual time=929.551..929.551 rows=390 loops=1)
            -> Nested loop inner join  (cost=3797.72 rows=84) (actual time=163.545..928.372 rows=425 loops=1)
                -> Nested loop inner join  (cost=2044.89 rows=1687) (actual time=82.781..798.338 rows=1687 loops=1)
                    -> Filter: (L.AvailableDate > <cache>(curdate()))  (cost=302.25 rows=1687) (actual time=0.047..1.525 rows=1687 loops=1)
                        -> Table scan on L  (cost=302.25 rows=3000) (actual time=0.043..1.010 rows=3000 loops=1)
                    -> Single-row index lookup on P using PRIMARY (PropertyID=L.PropertyID)  (cost=0.93 rows=1) (actual time=0.472..0.472 rows=1 loops=1687)
                -> Filter: ((FP.Price between 800 and 1400) and (FP.Bedrooms >= 2))  (cost=0.93 rows=0.05) (actual time=0.077..0.077 rows=0 loops=1687)
                    -> Index lookup on FP using FP_PID (PropertyID=L.PropertyID)  (cost=0.93 rows=1) (actual time=0.076..0.076 rows=1 loops=1687)
|
```

Index 2:

CREATE INDEX idx_floorplan_price ON FloorPlan(Price);

Reason: The ‘Price’ field is used in (FP.Price BETWEEN 800 AND 1400) in the WHERE clause of the query. Creating indexes can help the database quickly filter out records that are not within the specified range, thereby processing only those data that meet the criteria.

After:

```
| -> Limit: 15 row(s)  (actual time=917.955..917.959 rows=15 loops=1)
    -> Table scan on <temporary>  (actual time=917.953..917.956 rows=15 loops=1)
        -> Aggregate using temporary table  (actual time=917.950..917.950 rows=390 loops=1)
            -> Nested loop inner join  (cost=819.70 rows=175) (actual time=175.661..915.998 rows=425 loops=1)
                -> Nested loop inner join  (cost=633.28 rows=175) (actual time=66.295..116.163 rows=425 loops=1)
                    -> Filter: (L.AvailableDate > <cache>(curdate()))  (cost=104.77 rows=1000) (actual time=48.732..52.280 rows=1687 loops=1)
                        -> Table scan on L  (cost=104.77 rows=3000) (actual time=48.713..51.724 rows=3000 loops=1)
                    -> Filter: ((FP.Price between 800 and 1400) and (FP.Bedrooms >= 2))  (cost=0.42 rows=0.2) (actual time=0.037..0.038 rows=0 loops=1687)
                        -> Index lookup on FP using FP_PID (PropertyID=L.PropertyID)  (cost=0.42 rows=1) (actual time=0.036..0.037 rows=1 loops=1687)
                -> Single-row index lookup on P using PRIMARY (PropertyID=L.PropertyID)  (cost=0.97 rows=1) (actual time=1.882..1.882 rows=1 loops=425)
|
```

Index 3:

CREATE INDEX idx_floorplan_bedrooms ON FloorPlan(Bedrooms);

Reason: Indexing this field can allow the database to effectively filter out records that meet the condition on Bedrooms.. Especially when there are many listings with different numbers of bedrooms, it can avoid full table scans and improve query efficiency.

After:

```
| -> Limit: 15 row(s)  (actual time=1006.157..1006.163 rows=15 loops=1)
    -> Table scan on <temporary>  (actual time=1006.155..1006.160 rows=15 loops=1)
        -> Aggregate using temporary table  (actual time=1006.152..1006.152 rows=390 loops=1)
            -> Nested loop inner join  (cost=620.48 rows=58) (actual time=83.681..1003.521 rows=425 loops=1)
                -> Nested loop inner join  (cost=557.56 rows=58) (actual time=41.903..148.298 rows=425 loops=1)
                    -> Filter: (L.AvailableDate > <cache>(curdate()))  (cost=102.24 rows=1000) (actual time=20.462..61.399 rows=1687 loops=1)
                        -> Table scan on L  (cost=102.24 rows=3000) (actual time=20.451..60.645 rows=3000 loops=1)
                    -> Filter: ((FP.Price between 800 and 1400) and (FP.Bedrooms >= 2))  (cost=0.35 rows=0.06) (actual time=0.051..0.051 rows=0 loops=1687)
                        -> Index lookup on FP using FP_PID (PropertyID=L.PropertyID)  (cost=0.35 rows=1) (actual time=0.049..0.050 rows=1 loops=1687)
                -> Single-row index lookup on P using PRIMARY (PropertyID=L.PropertyID)  (cost=0.98 rows=1) (actual time=2.012..2.012 rows=1 loops=425)
|
```

Cost Analysis:

Index 1: Before creating the indexes, the original nested loop join cost was 2167.89. After adding the index on Listing(AvailableDate), the cost has increased significantly from 2167.89 to 3797.72. It indicates that the new index on AvailableDate is not as effective as anticipated.

Index 2: After adding the index on 'Price', the nest loop join cost is 819.70. we see a considerable decrease in the estimated cost for the nested loop join compared to the original. This suggests that the index on Price is having some beneficial impact in terms of the optimizer's cost estimation.

Index 3: When adding the index on 'Bedrooms', the nested loop inner join cost becomes 620.48. This cost is the lowest among all the plans. It implies that the index on Bedrooms has positively influenced the cost for the join operation.

Based on the above cost variance, the introduction of the FloorPlan.Bedrooms index results in the lowest cost for the nested loop join operations. We will select it as the final index of the query3 since the Bedrooms index is the most cost-efficient among the other 2 indexes. The cost improvement suggests that there would be fewer row operations due to the index. We can infer that the index on Bedrooms allows the database to more efficiently locate the rows that meet the join condition and the WHERE clause filter of FP.Bedrooms >= 2.

Query 4:
Initial:

```
EXPLAIN ANALYZE SELECT P.PropertyID, P.Source, COUNT(F.FavoriteID) AS FavoritesCount, AVG(R.Score) AS
AverageRating
FROM Property P
JOIN Listing L ON P.PropertyID = L.PropertyID
JOIN Favorites F ON L.ListingID = F.ListingID
JOIN Rating R ON P.PropertyID = R.PropertyID
WHERE L.AvailableDate > CURRENT_DATE
    AND P.Amenities LIKE '%pool%'
GROUP BY P.PropertyID, P.Source
HAVING AVG(R.Score) IS NOT NULL
ORDER BY FavoritesCount DESC, AverageRating DESC
LIMIT 15;
```

Before:

```
| -> Limit: 15 row(s)  (actual time=768.923..768.925 rows=15 loops=1)
    -> Sort: FavoritesCount DESC, AverageRating DESC  (actual time=768.922..768.923 rows=15 loops=1)
        -> Filter: (avg(R.Score) is not null)  (actual time=768.875..768.885 rows=23 loops=1)
            -> Table scan on <temporary>  (actual time=768.868..768.873 rows=23 loops=1)
                -> Aggregate using temporary table  (actual time=768.865..768.865 rows=23 loops=1)
                    -> Nested loop inner join  (cost=929.81 rows=31) (actual time=179.189..768.553 rows=25 loops=1)
                        -> Nested loop inner join  (cost=920.04 rows=26) (actual time=161.187..767.944 rows=96 loops=1)
                            -> Nested loop inner join  (cost=670.79 rows=239) (actual time=148.801..275.673 rows=176 loops=1)
                                -> Table scan on R  (cost=420.32 rows=614) (actual time=117.511..240.020 rows=1000 loops=1)
                                -> Filter: (L.AvailableDate > <cache>(curdate()))  (cost=0.29 rows=0.4) (actual time=0.035..0.035 rows=0 loops=1000)
                                    -> Index lookup on L using Listing_PID (PropertyID=R.PropertyID)  (cost=0.29 rows=1) (actual time=0.035..0.035 rows=0 loops=1000)
                            -> Filter: (P.Amenities like '%pool%')  (cost=0.95 rows=0.1) (actual time=2.797..2.797 rows=1 loops=176)
                                -> Single-row index lookup on P using PRIMARY (PropertyID=R.PropertyID)  (cost=0.95 rows=1) (actual time=2.794..2.794 rows=1 loops=176)
                        -> Covering index lookup on F using FAV_LID (ListingID=L.ListingID)  (cost=0.25 rows=1) (actual time=0.006..0.006 rows=0 loops=96)
|
```

Index 1:

Index on Listing.AvailableDate:
CREATE INDEX idx_listing_availabledate ON Listing(AvailableDate);

Reason: Given the query filters listings based on the AvailableDate being greater than the current date, indexing this attribute may speed up the retrieval process for listings.

After:

```
| -> Limit: 15 row(s)  (actual time=765.389..765.403 rows=15 loops=1)
    -> Sort: FavoritesCount DESC, AverageRating DESC  (actual time=765.388..765.389 rows=15 loops=1)
        -> Filter: (avg(R.Score) is not null)  (actual time=765.347..765.356 rows=23 loops=1)
            -> Table scan on <temporary>  (actual time=765.340..765.344 rows=23 loops=1)
                -> Aggregate using temporary table  (actual time=765.336..765.336 rows=23 loops=1)
                    -> Nested loop inner join  (cost=1025.38 rows=53) (actual time=262.504..765.069 rows=25 loops=1)
                        -> Nested loop inner join  (cost=1008.85 rows=45) (actual time=96.569..764.621 rows=96 loops=1)
                            -> Nested loop inner join  (cost=981.02 rows=68) (actual time=82.629..761.116 rows=527 loops=1)
                                -> Table scan on R  (cost=332.68 rows=614) (actual time=0.098..1.420 rows=1000 loops=1)
                                -> Filter: (P.Amenities like '%pool%')  (cost=0.96 rows=0.1) (actual time=0.759..0.759 rows=1 loops=1000)
                                    -> Single-row index lookup on P using PRIMARY (PropertyID=R.PropertyID)  (cost=0.96 rows=1) (actual time=0.758..0.758 rows=1 loops=1000)
                            -> Filter: (L.AvailableDate > <cache>(curdate()))  (cost=0.29 rows=1) (actual time=0.006..0.006 rows=0 loops=527)
                                -> Index lookup on L using Listing_PID (PropertyID=R.PropertyID)  (cost=0.29 rows=1) (actual time=0.006..0.006 rows=0 loops=527)
                        -> Covering index lookup on F using FAV_LID (ListingID=L.ListingID)  (cost=0.25 rows=1) (actual time=0.004..0.004 rows=0 loops=96)
|
```

Index 2:

Index on Property.Amenities:
CREATE INDEX idx_property_amenities ON Property(Amenities(255));

Reason: Since the query includes a LIKE filter on the Amenities column to find properties with a pool, a full-text index can significantly enhance the performance of text-based searches.

After:

```
| -> Limit: 15 row(s)  (actual time=940.398..940.400 rows=15 loops=1)
    -> Sort: FavoritesCount DESC, AverageRating DESC  (actual time=940.397..940.398 rows=15 loops=1)
        -> Filter: (avg(R.Score) is not null)  (actual time=940.344..940.353 rows=23 loops=1)
            -> Table scan on <temporary>  (actual time=940.338..940.342 rows=23 loops=1)
                -> Aggregate using temporary table  (actual time=940.336..940.336 rows=23 loops=1)
                    -> Nested loop inner join  (cost=1184.80 rows=31) (actual time=498.258..940.123 rows=25 loops=1)
                        -> Nested loop inner join  (cost=1155.15 rows=26) (actual time=337.211..938.666 rows=96 loops=1)
                            -> Nested loop inner join  (cost=1075.15 rows=68) (actual time=252.205..844.788 rows=527 loops=1)
                                -> Table scan on R  (cost=430.50 rows=614) (actual time=124.824..193.191 rows=1000 loops=1)
                                -> Filter: (P.Amenities like '%pool%')  (cost=0.95 rows=0.1) (actual time=0.651..0.651 rows=1 loops=1000)
                                    -> Single-row index lookup on P using PRIMARY (PropertyID=R.PropertyID)  (cost=0.95 rows=1) (actual time=0.649..0.649 rows=1 loops=1000)
                            -> Filter: (L.AvailableDate > <cache>(curdate()))  (cost=1.06 rows=0.4) (actual time=0.178..0.178 rows=0 loops=527)
                                -> Index lookup on L using Listing_PID (PropertyID=R.PropertyID)  (cost=1.06 rows=1) (actual time=0.177..0.177 rows=0 loops=527)
                        -> Covering index lookup on F using FAV_LID (ListingID=L.ListingID)  (cost=1.00 rows=1) (actual time=0.015..0.015 rows=0 loops=96)
|
```

Index 3:

Composite Index on Rating(PropertyID, Score):
CREATE INDEX idx_rating_propertyid_score ON Rating(PropertyID, Score);

Reason: The query calculates the average rating for properties; therefore, indexing both PropertyID and Score could optimize the aggregation process on the Rating table.

After:

```
| -> Limit: 15 row(s)  (actual time=653.592..653.594 rows=15 loops=1)
    -> Sort: FavoritesCount DESC, AverageRating DESC  (actual time=653.591..653.592 rows=15 loops=1)
        -> Filter: (avg(R.Score) is not null)  (actual time=653.552..653.561 rows=23 loops=1)
            -> Table scan on <temporary>  (actual time=653.544..653.548 rows=23 loops=1)
                -> Aggregate using temporary table  (actual time=653.539..653.539 rows=23 loops=1)
                    -> Nested loop inner join  (cost=1073.96 rows=31) (actual time=144.666..653.175 rows=25 loops=1)
                        -> Nested loop inner join  (cost=1044.31 rows=26) (actual time=70.707..650.615 rows=96 loops=1)
                            -> Nested loop inner join  (cost=994.12 rows=68) (actual time=38.304..556.057 rows=527 loops=1)
                                -> Covering index scan on R using idx_rating_propertyid_score  (cost=350.68 rows=614) (actual time=0.047..0.510 rows=1000 loops=1)
                                -> Filter: (P.Amenities like '%pool%')  (cost=0.95 rows=0.1) (actual time=0.555..0.555 rows=1 loops=1000)
                                    -> Single-row index lookup on P using PRIMARY (PropertyID=R.PropertyID)  (cost=0.95 rows=1) (actual time=0.554..0.554 rows=1 loops=1000)
                            -> Filter: (L.AvailableDate > <cache>(curdate()))  (cost=0.62 rows=0.4) (actual time=0.179..0.179 rows=0 loops=527)
                                -> Index lookup on L using Listing_PID (PropertyID=R.PropertyID)  (cost=0.62 rows=1) (actual time=0.178..0.179 rows=0 loops=527)
                        -> Covering index lookup on F using FAV_LID (ListingID=L.ListingID)  (cost=1.00 rows=1) (actual time=0.026..0.026 rows=0 loops=96)
|
```

Cost Analysis:

1. Index on Listing.AvailableDate:
Before adding an index on Listing.AvailableDate, the query cost was 929.81, which was the total computational work to execute the query with nested joins and table scans. After adding the index, the cost slightly increased to 1025.38, suggesting that accessing the index added more work than anticipated. This indicates that while indexing may speed up access to filtered rows, it also introduces additional cost, most likely from managing the index structure alongside the main query execution.

2. Index on Property.Amenities:
Before adding an index on Property.Amenities, the query's cost was 929.81, indicating the work needed to perform nested joins and filter operations. After implementing the index, the cost increased to 1184.80, indicating that while the index aimed to optimize access to properties with specific amenities, it also introduced additional overhead, most likely due to the complex nature of text-based filtering in the Amenities column. (When we perform a LIKE '%pool%' operation, the database needs to scan through the indexed text data, which can be inefficient, leading to increased processing time.)The increase in execution time from 768.923ms to 940.398ms further indicates that the index did not improve performance as expected, possibly due to the inefficiency of handling large text fields or the specific query structure not benefiting from the index as anticipated.

3. After adding the composite index on Rating(PropertyID, Score), the cost to execute the query increased from 929.81 to 1073.96. This suggests that while the index was expected to improve query efficiency, the actual cost increased, possibly due to changes in how the database handles joins and accesses data with the new index.

Based on the analysis, the final index design selected was the composite index on Rating(PropertyID, Score). Despite the increase in cost from 929.81 to 1073.96, this decision was made because it directly impacts the query's efficiency in handling the aggregation and filtering based on ratings, which is a key aspect of the query's functionality. This index was chosen with the expectation that it would improve data retrieval times for operations involving Rating, optimizing the query performance for cases where ratings play a critical role in the data filtering process.