Database Design

Part1: Database Implementation

1 Connect Database on GCP

```
(base) chenzhaowang@CHENZHAOdeMacBook-Pro ~ % gcloud compute ssh --zone "us-central1-a" "cs411-066-backend" --project "cs411-pt1-414816"
Linux cs411-066-backend.us-central1-a.c.cs411-pt1-414816.internal 6.1.0-18-cloud-amd64 #1 SMP PREEMPT_DYNAMIC Debian 6.1.76-1 (2024-02-01) x86_64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Sat Apr  6 23:56:50 2024 from 98.215.2.149
chenzhaowang@cs411-066-backend:~$ mysql -h 34.171.144.159 \
    -u root -p
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MySQL connection id is 420126
Server version: 8.0.31-google (Google)

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MySQL [(none)]> show databases
    -> ;
+--------------------+
| Database           |
+--------------------+
| ease_lease         |
| information_schema |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
5 rows in set (0.004 sec)

MySQL [(none)]> use ease_lease
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
MySQL [ease_lease]> show tables
    -> ;
+----------------------+
| Tables_in_ease_lease |
+----------------------+
| Application          |
| Bidding              |
| Landlord             |
| Listing              |
| Neighborhood         |
| Rating               |
| Review               |
| Tenant               |
| User                 |
+----------------------+
9 rows in set (0.003 sec)

MySQL [ease_lease]> select count(*) from Listing;
+----------+
| count(*) |
+----------+
|     4340 |
+----------+
1 row in set (0.059 sec)

MySQL [ease_lease]>
```

2 DDL

User

```
 CREATE TABLE User (
  user_id INT PRIMARY KEY NOT NULL,
  user_name VARCHAR(255) NOT NULL,
  password VARCHAR(255) NOT NULL,
  phone_number BIGINT NOT NULL,
  first_name VARCHAR(255),
  last_name VARCHAR(255)
);
```

Landlord

```
CREATE TABLE Landlord (
  host_id INT PRIMARY KEY NOT NULL,
  host_about TEXT,
  user_id INT NOT NULL,
  FOREIGN KEY (user_id) REFERENCES User(user_id) ON DELETE
CASCADE ON UPDATE CASCADE
);
```

Tenant

```
CREATE TABLE Tenant (
  tenant_id INT AUTO_INCREMENT PRIMARY KEY,
  account_balance INT DEFAULT 0,
  user_id INT NOT NULL,
  FOREIGN KEY (user_id) REFERENCES User(user_id) ON DELETE
CASCADE ON UPDATE CASCADE
);
```

Listing

```
CREATE TABLE Listing (
  listing_id INT AUTO_INCREMENT PRIMARY KEY,
```

```
  room_type VARCHAR(255) NOT NULL,
  description VARCHAR(255),
  price INT NOT NULL,
  from_date DATE,
  to_date DATE,
  landlord_id INT NOT NULL,
  longitude DECIMAL(9,6) NOT NULL,
  latitude DECIMAL(9,6) NOT NULL,
  FOREIGN KEY (landlord_id) REFERENCES Landlord(host_id) ON
DELETE CASCADE ON UPDATE CASCADE,
  FOREIGN KEY (longitude, latitude) REFERENCES
Neighborhood(longitude, latitude) ON DELETE CASCADE ON UPDATE
CASCADE
);
```

Application

```
CREATE TABLE Application (
  application_id INT AUTO_INCREMENT PRIMARY KEY,
  status VARCHAR(255),
  listing_id INT NOT NULL,
  tenant_id INT NOT NULL,
  FOREIGN KEY (listing_id) REFERENCES Listing(listing_id) ON
DELETE CASCADE ON UPDATE CASCADE,
  FOREIGN KEY (tenant_id) REFERENCES Tenant(tenant_id) ON DELETE
CASCADE ON UPDATE CASCADE
);
```

Bidding

```
CREATE TABLE Bidding (
  bidding_id INT AUTO_INCREMENT PRIMARY KEY,
  bid_price INT,
```

```
  bid_time DATETIME,
  listing_id INT NOT NULL,
  tenant_id INT NOT NULL,
  FOREIGN KEY (listing_id) REFERENCES Listing(listing_id) ON
DELETE CASCADE ON UPDATE CASCADE,
  FOREIGN KEY (tenant_id) REFERENCES Tenant(tenant_id) ON DELETE
CASCADE ON UPDATE CASCADE
);
```

Rating

```
CREATE TABLE Rating (
  rating_id INT AUTO_INCREMENT PRIMARY KEY,
  scores_rating DECIMAL(10, 2),
  scores_accuracy DECIMAL(10, 2),
  scores_cleanliness DECIMAL(10, 2),
  scores_checkin DECIMAL(10, 2),
  scores_communication DECIMAL(10, 2),
  scores_location DECIMAL(10, 2),
  scores_value DECIMAL(10, 2),
  listing_id INT,
  FOREIGN KEY (listing_id) REFERENCES Listing(listing_id)
);
```

Neighborhood

```
CREATE TABLE Neighborhood (
  longitude DECIMAL(9,6) NOT NULL,
  latitude DECIMAL(9,6) NOT NULL,
  neighborhood VARCHAR(255),
  PRIMARY KEY (longitude, latitude)
);
```

Review

```
CREATE TABLE Review (
  review_id INT AUTO_INCREMENT PRIMARY KEY,
  reviewer_name VARCHAR(255),
  content TEXT,
  listing_id INT NOT NULL,
  FOREIGN KEY (listing_id) REFERENCES Listing(listing_id) ON
DELETE CASCADE ON UPDATE CASCADE
);
```

3 Data Cleaning and Insertion

The three different tables with more than 1000 rows are shown below:

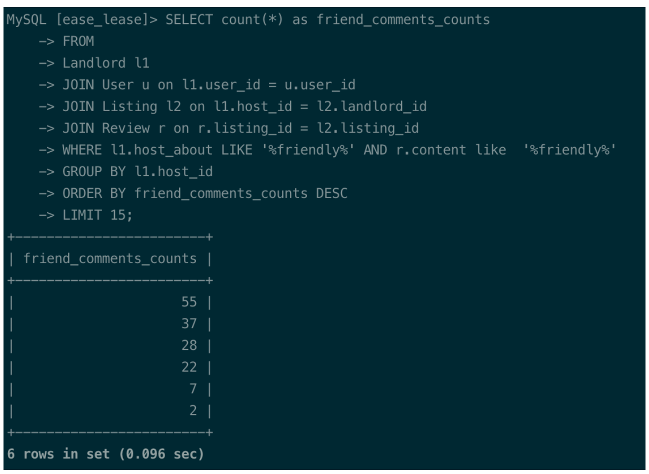
```
MySQL [ease_lease]> SELECT COUNT(*) FROM User;
+----------+
| COUNT(*) |
+----------+
|     3924 |
+----------+
1 row in set (0.112 sec)

MySQL [ease_lease]> SELECT COUNT(*) FROM Listing;
+----------+
| COUNT(*) |
+----------+
|     4340 |
+----------+
1 row in set (0.006 sec)

MySQL [ease_lease]> SELECT COUNT(*) FROM Rating;
+----------+
| COUNT(*) |
+----------+
|     6992 |
+----------+
1 row in set (0.005 sec)
```

Part2: SQL & Indexing Analysis

1 Advanced SQL Queries

SQL query 1

```
SELECT count(*) as friend_comments_counts
FROM
Landlord l1
JOIN User u on l1.user_id = u.user_id
JOIN Listing l2 on l1.host_id = l2.landlord_id
JOIN Review r on r.listing_id = l2.listing_id
WHERE l1.host_about LIKE '%friendly%' AND r.content like '%friendly%'
GROUP BY l1.host_id
ORDER BY friend_comments_counts DESC
LIMIT 15;
```



```
MySQL [ease_lease]> SELECT count(*) as friend_comments_counts
    -> FROM
    -> Landlord l1
    -> JOIN User u on l1.user_id = u.user_id
    -> JOIN Listing l2 on l1.host_id = l2.landlord_id
    -> JOIN Review r on r.listing_id = l2.listing_id
    -> WHERE l1.host_about LIKE '%friendly%' AND r.content like  '%friendly%'
    -> GROUP BY l1.host_id
    -> ORDER BY friend_comments_counts DESC
    -> LIMIT 15;
+------------------------+
| friend_comments_counts |
+------------------------+
|                     55 |
|                     37 |
|                     28 |
|                     22 |
|                      7 |
|                      2 |
+------------------------+
6 rows in set (0.096 sec)
```

This query aims to find the number of comments for the landlords who claim that they are “friendly” while also receiving reviews saying that they are friendly (the real “friendly” hosts). The result shows that out of the 3924 hosts in our database, only 6 of them qualify as really friendly hosts, and the best host receives 55 friendly comments while identifying himself/herself as a friendly host.

SQL query 2 (Apply various filters on listing)

```
(SELECT * FROM Listing WHERE price < 50)
INTERSECT
(SELECT L.listing_id, L.room_type, L.description, L.price, L.from_date, L.to_date, L.landlord_id, L.longitude, L.latitude
FROM Listing AS L JOIN
(SELECT listing_id, AVG(scores_rating) AS average_score FROM Rating GROUP BY listing_id HAVING AVG(scores_rating) > 4.0) AS S
ON L.listing_id = S.listing_id)
```

```
ORDER BY PRICE DESC
LIMIT 15;
```

```
MySQL [ease_lease]> (SELECT * FROM Listing WHERE price < 50)
    -> INTERSECT
    -> (SELECT L.listing_id, L.room_type, L.description, L.price, L.from_date, L.to_date, L.landlord_id, L.longitude, L.latitude FROM Listing AS L JOIN
    -> (SELECT listing_id, AVG(scores_rating) AS average_score FROM Rating GROUP BY listing_id HAVING AVG(scores_rating) > 4.0) AS S ON L.listing_id = S.listing_id)
    -> ORDER BY PRICE DESC
    -> LIMIT 15;
```

listing_id	room_type	description	price	from_date	to_date	landlord_id	longitude	latitude
42234877	Private room	Rental unit in Chicago · 1 bedroom · 2 beds · 2 shared baths	49	2023-12-18	2024-12-16	317530111	-87.702490	41.878790
5804791	Private room	Rental unit in Chicago · ★4.81 · 1 bedroom · 1 bed · 1 shared bath	49	2023-12-18	2024-12-16	250400	-87.698450	41.854400
10100835	Private room	Home in Chicago · ★4.42 · 1 bedroom · 1 bed · 2 shared baths	49	2023-12-18	2024-12-16	11910936	-87.605640	41.782290
32177642	Private room	Home in Chicago · ★4.76 · 5 bedrooms · 1 bed · 3 shared baths	49	2023-12-18	2024-12-16	217927066	-87.765670	41.883970
40408111	Private room	Home in Chicago · ★4.74 · 1 bedroom · 1 bed · 1.5 shared baths	49	2023-12-18	2024-12-16	40832200	-87.618680	41.833550
46107192	Private room	Rental unit in Chicago · ★4.76 · 1 bedroom · 1 bed · 2 shared baths	49	2023-12-18	2024-12-16	102750577	-87.713630	41.947210
13824783	Private room	Cottage in Chicago · ★4.96 · 1 bedroom · 1 bed · 1 private bath	49	2023-12-18	2024-12-16	55020055	-87.684160	41.977650
5807548	Private room	Rental unit in Chicago · ★4.70 · 1 bedroom · 1 bed · 1 shared bath	49	2023-12-18	2024-12-16	250400	-87.696950	41.854420
14141704	Private room	Rental unit in Chicago · ★5.0 · 1 bedroom · 1 bed · 1 private bath	49	2023-12-18	2024-12-16	12955486	-87.662370	41.989450
20036818	Entire home/apt	Rental unit in Chicago · ★4.67 · 1 bedroom · 1 bed · 1 bath	49	2023-12-18	2024-12-16	137501409	-87.720550	41.932380
45368483	Private room	Home in Chicago · ★4.88 · 1 bedroom · 1 bed · 1.5 shared baths	49	2023-12-18	2024-12-16	128265803	-87.714640	41.992640
39154634	Private room	Home in Chicago · ★4.82 · 1 bedroom · 1 bed · 1 private bath	49	2023-12-18	2024-12-16	11910936	-87.606370	41.782770
3590692	Private room	Rental unit in Chicago · ★4.85 · 1 bedroom · 1 bed · 1 shared bath	49	2023-12-18	2024-12-16	12955486	-87.661660	41.991030
37551145	Entire home/apt	Rental unit in Chicago · ★4.70 · 2 bedrooms · 2 beds · 2 baths	49	2023-12-18	2024-12-16	100782278	-87.640000	41.841850
54003597	Entire home/apt	Rental unit in Chicago · ★4.77 · 1 bedroom · 1 bed · 1 bath	49	2023-12-18	2024-12-16	384256562	-87.664980	41.965700

15 rows in set (0.029 sec)

This query aims to find the listing with price less than 50 and average rating score over 4.0. By intersecting, only the listings that meet both criteria are selected. Out of 4340 listings in our database, there are 745 listings that have prices lower than 50 with relatively high ratings.

SQL query 3

```
SELECT
  L.listing_id, L.room_type, L.description, L.price, L.from_date,
L.to_date
FROM
  Listing L
INNER JOIN
  Review R ON L.listing_id = R.listing_id
GROUP BY
   L.listing_id, L.room_type, L.description, L.price,L.from_date,
L.to_date
HAVING
  COUNT(R.review_id) >= 10
ORDER BY
  L.listing_id
LIMIT 15;
```

```
MySQL [ease_lease]> SELECT
    ->   L.listing_id, L.room_type, L.description, L.price, L.from_date, L.to_date
    -> FROM
    ->   Listing L
    -> INNER JOIN
    ->   Review R ON L.listing_id = R.listing_id
    -> GROUP BY
    ->   L.listing_id, L.room_type, L.description, L.price,L.from_date, L.to_date
    -> HAVING
    ->   COUNT(R.review_id) >= 10
    -> ORDER BY
    ->   L.listing_id
    -> LIMIT 15;
+------------+-----------------+--------------------------------------------------------------------------------+-------+------------+------------+
| listing_id | room_type       | description                                                                    | price | from_date  | to_date    |
+------------+-----------------+--------------------------------------------------------------------------------+-------+------------+------------+
|       2384 | Private room    | Condo in Chicago · ★4.99 · 1 bedroom · 1 bed · 1 shared bath                   |    70 | 2023-12-18 | 2024-12-16 |
|       7126 | Entire home/apt | Rental unit in Chicago · ★4.70 · 1 bedroom · 1 bed · 1 bath                    |    90 | 2023-12-18 | 2024-12-16 |
|      10945 | Entire home/apt | Rental unit in Chicago · ★4.63 · 2 bedrooms · 2 beds · 1 bath                  |   106 | 2023-12-18 | 2024-12-16 |
|      12140 | Private room    | Boutique hotel in Chicago · ★4.93 · 1 bedroom · 1 bed · 1 private bath         |   329 | 2023-12-18 | 2024-12-16 |
|      24833 | Entire home/apt | Rental unit in Chicago · ★4.29 · 1 bedroom · 1 bed · 1 bath                    |    59 | 2023-12-18 | 2024-12-16 |
|      25879 | Entire home/apt | Rental unit in Chicago · ★4.32 · 2 bedrooms · 3 beds · 1 bath                  |    74 | 2023-12-18 | 2024-12-16 |
|      28749 | Entire home/apt | Loft in Chicago · ★4.78 · 3 bedrooms · 3 beds · 2 baths                        |   139 | 2023-12-18 | 2024-12-16 |
|      37738 | Private room    | Home in Chicago · ★4.99 · 1 bedroom · 1 bed · 1.5 shared baths                 |     0 | 2023-12-18 | 2024-12-16 |
|      71930 | Private room    | Rental unit in Chicago · ★4.88 · 1 bedroom · 1 bed · 1 shared bath             |    96 | 2023-12-18 | 2024-12-16 |
|      84042 | Private room    | Rental unit in Chicago · ★4.88 · 1 bedroom · 1 bed · 1 shared bath             |     0 | 2023-12-18 | 2024-12-16 |
|      94450 | Entire home/apt | Rental unit in Chicago · ★5.0 · 1 bedroom · 1 bed · 1 bath                     |   111 | 2023-12-18 | 2024-12-16 |
|     145659 | Entire home/apt | Rental unit in Chicago · ★4.78 · 3 bedrooms · 3 beds · 2 baths                 |   198 | 2023-12-18 | 2024-12-16 |
|     145690 | Entire home/apt | Rental unit in Chicago · ★4.78 · 4 bedrooms · 4 beds · 2 baths                 |   298 | 2023-12-18 | 2024-12-16 |
|     189821 | Entire home/apt | Rental unit in Chicago · ★4.95 · 2 bedrooms · 3 beds · 1 bath                  |   200 | 2023-12-18 | 2024-12-16 |
|     207218 | Entire home/apt | Rental unit in Chicago · ★4.90 · 1 bedroom · 1 bed · 1 bath                    |   100 | 2023-12-18 | 2024-12-16 |
+------------+-----------------+--------------------------------------------------------------------------------+-------+------------+------------+
15 rows in set (0.076 sec)
```

This SQL query identifies listings with at least 10 reviews, extracting details like room type, description, price, and availability. By joining the `Listing` and `Review` tables, it selects only those listings that are well-reviewed. The results reveal that out of 4,340 listings in our database, 177 meet this criterion of high customer engagement.

SQL query 4

```
select avg(scores_rating), landlord_id
from Listing li join Rating r on
li.listing_id = r.listing_id
group by landlord_id
order by avg(scores_rating) desc
limit 5
```

```
+--------------------+-------------+
| avg(scores_rating) | landlord_id |
+--------------------+-------------+
|           5.000000 |     8292749 |
|           5.000000 |      504470 |
|           5.000000 |    21178487 |
|           5.000000 |    61600579 |
|           5.000000 |    20624315 |
+--------------------+-------------+
5 rows in set (0.04 sec)
```

This SQL query calculates the average ratings for listings grouped by landlord ID from a real estate database. It joins the Listing table with the Rating table on the listing_id field to aggregate ratings data. The results are grouped by the landlord_id field in the Listing table and are sorted in descending order based on the average scores. The query limits the output to the top 5 landlords with the highest average ratings, providing a focused snapshot of top-performing landlords in terms of tenant satisfaction.

SQL query 5

```
SELECT AVG(r.scores_rating) AS mean_score, li.landlord_id
FROM Listing li
JOIN Rating r ON li.listing_id = r.listing_id
GROUP BY li.landlord_id
HAVING AVG(r.scores_rating) > 3.8
ORDER BY mean_score DESC
LIMIT 5
```

```
mysql> SELECT AVG(r.scores_rating) AS mean_score, li.landlord_id
    -> FROM Listing li
    -> JOIN Rating r ON li.listing_id = r.listing_id
    -> GROUP BY li.landlord_id
    -> HAVING AVG(r.scores_rating) > 3.8
    -> ORDER BY mean_score DESC
    -> LIMIT 5 ;
+------------+-------------+
| mean_score | landlord_id |
+------------+-------------+
|   5.000000 |      504470 |
|   5.000000 |     4995578 |
|   5.000000 |     8292749 |
|   5.000000 |    19471032 |
|   5.000000 |    20130521 |
+------------+-------------+
5 rows in set (0.01 sec)
```

This SQL query calculates and retrieves the average ratings for landlords by joining the `Listing` and `Rating` tables on `listing_id`, focusing on those landlords whose average rating exceeds 3.8. It selects the average rating (`scores_rating`) as `mean_score` and the corresponding `landlord_id` from the listings, groups the results by `landlord_id` for aggregation, and filters using the `HAVING` clause to include only those entries with an average above 3.8. Finally, the results are sorted in descending order by the average rating and limited to displaying only those records, effectively pinpointing landlords with superior ratings in the dataset.

2 Indexing Analysis

We will conduct our indexing analysis on advanced SQL queries 2, 3, 4, and 5.

Analysis for query 2:

```
(SELECT * FROM Listing WHERE price < 50)
INTERSECT
(SELECT L.listing_id, L.room_type, L.description, L.price,
L.from_date, L.to_date, L.landlord_id, L.longitude, L.latitude
FROM Listing AS L JOIN
(SELECT listing_id, AVG(scores_rating) AS average_score FROM
Rating
GROUP BY listing_id HAVING AVG(scores_rating) > 4.0) AS S ON
L.listing_id = S.listing_id);
```

Before adding any index, the total cost is 3742 and execution are as follow:

```
MySQL [ease_lease]> EXPLAIN ANALYZE (SELECT * FROM Listing WHERE price < 50) INTERSECT (SELECT L.listing_id, L.room_type, L.description, L.price, L.from
_date, L.to_date, L.landlord_id, L.longitude, L.latitude FROM Listing AS L JOIN  (SELECT listing_id, AVG(scores_rating) AS average_score FROM Rating GRO
UP BY listing_id HAVING AVG(scores_rating) > 4.0) AS S ON L.listing_id = S.listing_id);
+--------------------------------------------------------------------------------------------------------------------------------------------------------
--+
| EXPLAIN
                                                                                                                                    |
+--------------------------------------------------------------------------------------------------------------------------------------------------------
--+
| -> Table scan on <intersect temporary>  (cost=3742.96..3763.10 rows=1413) (actual time=26.316..26.508 rows=745 loops=1)
    -> Intersect materialize with deduplication  (cost=3742.95..3742.95 rows=1413) (actual time=26.314..26.314 rows=891 loops=1)
        -> Filter: (Listing.price < 50)  (cost=448.15 rows=1413) (actual time=0.091..2.894 rows=891 loops=1)
            -> Table scan on Listing  (cost=448.15 rows=4239) (actual time=0.086..2.598 rows=4340 loops=1)
        -> Nested loop inner join  (cost=3153.51 rows=6813) (actual time=8.489..15.472 rows=3755 loops=1)
            -> Filter: (S.listing_id is not null)  (cost=0.11..768.96 rows=6813) (actual time=8.465..9.283 rows=3755 loops=1)
                -> Table scan on S  (cost=2.50..2.50 rows=0) (actual time=8.463..8.976 rows=3755 loops=1)
                    -> Materialize  (cost=0.00..0.00 rows=0) (actual time=8.462..8.462 rows=3755 loops=1)
                        -> Filter: (avg(Rating.scores_rating) > 4.0)  (actual time=5.941..7.334 rows=3755 loops=1)
                            -> Table scan on <temporary>  (actual time=5.935..6.372 rows=3880 loops=1)
                                -> Aggregate using temporary table  (actual time=5.934..5.934 rows=3880 loops=1)
                                    -> Table scan on Rating  (cost=689.05 rows=6813) (actual time=0.034..2.273 rows=6992 loops=1)
            -> Single-row index lookup on L using PRIMARY (listing_id=S.listing_id)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=3755)
```

1. Single column index on price in the Listing table:

`CREATE INDEX idx_listing_price ON Listing(price);`

```
MySQL [ease_lease]> CREATE INDEX idx_listing_price ON Listing(price);
Query OK, 0 rows affected (0.126 sec)
Records: 0  Duplicates: 0  Warnings: 0

MySQL [ease_lease]> EXPLAIN ANALYZE (SELECT * FROM Listing WHERE price < 50) INTERSECT (SELECT L.listing_id, L.room_type, L.description, L.price, L.from
_date, L.to_date, L.landlord_id, L.longitude, L.latitude FROM Listing AS L JOIN  (SELECT listing_id, AVG(scores_rating) AS average_score FROM Rating GRO
UP BY listing_id HAVING AVG(scores_rating) > 4.0) AS S ON L.listing_id = S.listing_id);
+--------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------+
| EXPLAIN
                                                                                                                                    |
+--------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------+
| -> Table scan on <intersect temporary>  (cost=3643.84..3657.46 rows=891) (actual time=23.331..23.520 rows=745 loops=1)
    -> Intersect materialize with deduplication  (cost=3643.82..3643.82 rows=891) (actual time=23.329..23.329 rows=891 loops=1)
        -> Index range scan on Listing using idx_listing_price over (price < 50), with index condition: (Listing.price < 50)  (cost=401.21 rows=891) (ac
tual time=0.041..1.635 rows=891 loops=1)
        -> Nested loop inner join  (cost=3153.51 rows=6813) (actual time=7.855..14.249 rows=3755 loops=1)
            -> Filter: (S.listing_id is not null)  (cost=0.11..768.96 rows=6813) (actual time=7.839..8.652 rows=3755 loops=1)
                -> Table scan on S  (cost=2.50..2.50 rows=0) (actual time=7.838..8.331 rows=3755 loops=1)
                    -> Materialize  (cost=0.00..0.00 rows=0) (actual time=7.837..7.837 rows=3755 loops=1)
                        -> Filter: (avg(Rating.scores_rating) > 4.0)  (actual time=5.394..6.761 rows=3755 loops=1)
                            -> Table scan on <temporary>  (actual time=5.389..5.794 rows=3880 loops=1)
                                -> Aggregate using temporary table  (actual time=5.388..5.388 rows=3880 loops=1)
                                    -> Table scan on Rating  (cost=689.05 rows=6813) (actual time=0.031..2.125 rows=6992 loops=1)
            -> Single-row index lookup on L using PRIMARY (listing_id=S.listing_id)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=3755)
```

With the index on price, the table scan for the WHERE clause in `(SELECT * FROM Listing WHERE price < 50)` changes to index range scan, which decreases the cost from 448 to 401. Then the overall cost is also decreased to 3643.

2. Composite index on listing_id and scores_rating in the Rating table.

```
CREATE INDEX idx_rating_listingid_scores ON Rating(listing_id,scores_rating);
```

```
MySQL [ease_lease]> CREATE INDEX idx_rating_listingid_scores ON Rating(listing_id,scores_rating);
Query OK, 0 rows affected (0.143 sec)
Records: 0  Duplicates: 0  Warnings: 0

MySQL [ease_lease]> EXPLAIN ANALYZE (SELECT * FROM Listing WHERE price < 50) INTERSECT (SELECT L.listing_id, L.room_type, L.description, L.price, L.from_date
, L.to_date, L.landlord_id, L.longitude, L.latitude FROM Listing AS L JOIN  (SELECT listing_id, AVG(scores_rating) AS average_score FROM Rating GROUP BY list
ing_id HAVING AVG(scores_rating) > 4.0) AS S ON L.listing_id = S.listing_id);
+------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------
+
| EXPLAIN


                                                  |
+------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------
+
| -> Table scan on <intersect temporary>  (cost=3742.96..3763.10 rows=1413) (actual time=50.274..50.566 rows=745 loops=1)
    -> Intersect materialize with deduplication  (cost=3742.95..3742.95 rows=1413) (actual time=50.271..50.271 rows=891 loops=1)
        -> Filter: (Listing.price < 50)  (cost=448.15 rows=1413) (actual time=0.817..7.023 rows=891 loops=1)
            -> Table scan on Listing  (cost=448.15 rows=4239) (actual time=0.806..6.254 rows=4340 loops=1)
        -> Nested loop inner join  (cost=3153.51 rows=6813) (actual time=15.658..28.337 rows=3755 loops=1)
            -> Filter: (S.listing_id is not null)  (cost=2051.46..768.96 rows=6813) (actual time=15.630..16.950 rows=3755 loops=1)
                -> Table scan on S  (cost=2051.66..2139.31 rows=6813) (actual time=15.627..16.456 rows=3755 loops=1)
                    -> Materialize  (cost=2051.65..2051.65 rows=6813) (actual time=15.624..15.624 rows=3755 loops=1)
                        -> Filter: (avg(Rating.scores_rating) > 4.0)  (cost=1370.35 rows=6813) (actual time=0.074..9.315 rows=3755 loops=1)
                            -> Group aggregate: avg(Rating.scores_rating), avg(Rating.scores_rating)  (cost=1370.35 rows=6813) (actual time=0.069..7.898 rows
=3880 loops=1)
                                -> Covering index scan on Rating using idx_rating_listingid_scores  (cost=689.05 rows=6813) (actual time=0.054..3.520 rows=69
92 loops=1)
            -> Single-row index lookup on L using PRIMARY (listing_id=S.listing_id)  (cost=0.25 rows=1) (actual time=0.003..0.003 rows=1 loops=3755)
```

We create a composite index on listing_id and scores_rating in the rating table for the GROUP BY and HAVING operation. Covering index scan on Rating fails to decrease the cost. And the overall cost remains the same (3742).

Comparing the two indexing tables, the index on price in the Listing table is able to improve the efficiency of the table scan and consequently decrease the cost for the nested loop inner join. Additionally, this query is joining two tables on listing_id, which is the primary key for the Listing table and the foreign key for the Rating table. Since either PK or FK has its default index, we are not able to add any other index to directly improve the efficiency of the JOIN operation. Therefore, we decided to choose the index on price for this query.

Analysis for query 3

```
SELECT
  L.listing_id, L.room_type, L.description, L.price, L.from_date,
L.to_date
FROM
  Listing L
INNER JOIN
```

```sql
  Review R ON L.listing_id = R.listing_id
GROUP BY
  L.listing_id, L.room_type, L.description, L.price,L.from_date,
L.to_date
HAVING
  COUNT(R.review_id) >= 10
ORDER BY
  L.listing_id;
```

Before adding any index, the cost of nested loop inner join is 13480 and execution are as follow:

```
MySQL [ease_lease]> EXPLAIN ANALYZE SELECT
    ->   L.listing_id, L.room_type, L.description, L.price, L.from_date, L.to_date
    -> FROM
    ->   Listing L
    -> INNER JOIN
    ->   Review R ON L.listing_id = R.listing_id
    -> GROUP BY
    ->   L.listing_id, L.room_type, L.description, L.price,L.from_date, L.to_date
    -> HAVING
    ->   COUNT(R.review_id) >= 10
    -> ORDER BY
    ->   L.listing_id
    -> ;
+--------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
------------------------------+
| EXPLAIN



                              |
+--------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
------------------------------+
| -> Sort: L.listing_id, L.room_type, L.`description`, L.price, L.from_date, L.to_date  (actual time=81.913..81.931 rows=177 loops=1)
    -> Filter: (count(R.review_id) >= 10)  (actual time=81.102..81.212 rows=177 loops=1)
        -> Table scan on <temporary>  (actual time=81.097..81.186 rows=193 loops=1)
            -> Aggregate using temporary table  (actual time=81.093..81.093 rows=193 loops=1)
                -> Nested loop inner join  (cost=13480.65 rows=28321) (actual time=0.143..17.455 rows=29999 loops=1)
                    -> Covering index scan on R using listing_id  (cost=3568.30 rows=28321) (actual time=0.129..7.275 rows=29999 loops=1)
                    -> Single-row index lookup on L using PRIMARY (listing_id=R.listing_id)  (cost=0.25 rows=1) (actual time=0.000..0.000 rows=1 loops=29999)
```

1. Composite index on the columns used in GROUP BY operation.

```sql
CREATE INDEX idx_listing_groupby ON Listing(listing_id,
room_type, description, price, from_date, to_date);
```

```
MySQL [ease_lease]> CREATE INDEX idx_listing_groupby ON Listing(listing_id, room_type, description, price, from_date, to_date);
Query OK, 0 rows affected (0.184 sec)
Records: 0  Duplicates: 0  Warnings: 0

MySQL [ease_lease]> EXPLAIN ANALYZE SELECT    L.listing_id, L.room_type, L.description, L.price, L.from_date, L.to_date FROM    Listing L INNER JOIN    Revie
w R ON L.listing_id = R.listing_id GROUP BY    L.listing_id, L.room_type, L.description, L.price,L.from_date, L.to_date HAVING    COUNT(R.review_id) >= 10 OR
DER BY    L.listing_id;
+--------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
------------------------------+
| EXPLAIN



                              |
+--------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
------------------------------+
| -> Sort: L.listing_id, L.room_type, L.`description`, L.price, L.from_date, L.to_date  (actual time=96.282..96.298 rows=177 loops=1)
    -> Filter: (count(R.review_id) >= 10)  (actual time=96.016..96.122 rows=177 loops=1)
        -> Table scan on <temporary>  (actual time=96.010..96.099 rows=193 loops=1)
            -> Aggregate using temporary table  (actual time=96.006..96.006 rows=193 loops=1)
                -> Nested loop inner join  (cost=13480.65 rows=28321) (actual time=0.130..19.378 rows=29999 loops=1)
                    -> Covering index scan on R using listing_id  (cost=3568.30 rows=28321) (actual time=0.102..8.576 rows=29999 loops=1)
                    -> Single-row index lookup on L using PRIMARY (listing_id=R.listing_id)  (cost=0.25 rows=1) (actual time=0.000..0.000 rows=1 loops=29999)
```

The cost stays the same. But the actual time increases a lot, which means it introduces complexity and makes it less efficient.

2. Single column index on review_id to improve the COUNT operation in HAVING.

`CREATE INDEX idx_review_id ON Review(review_id);`

```
MySQL [ease_lease]> CREATE INDEX idx_review_id ON Review(review_id);
Query OK, 0 rows affected (0.765 sec)
Records: 0  Duplicates: 0  Warnings: 0

MySQL [ease_lease]> EXPLAIN ANALYZE SELECT    L.listing_id, L.room_type, L.description, L.price, L.from_date, L.to_date FROM    Listing L INNER JOIN    Revie
w R ON L.listing_id = R.listing_id GROUP BY    L.listing_id, L.room_type, L.description, L.price,L.from_date, L.to_date HAVING    COUNT(R.review_id) >= 10 OR
DER BY    L.listing_id;
+-------------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------+
| EXPLAIN

                               |
+-------------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------+
| -> Sort: L.listing_id, L.room_type, L.`description`, L.price, L.from_date, L.to_date  (actual time=84.711..84.728 rows=177 loops=1)
    -> Filter: (count(R.review_id) >= 10)  (actual time=84.447..84.552 rows=177 loops=1)
        -> Table scan on <temporary>  (actual time=84.441..84.528 rows=193 loops=1)
            -> Aggregate using temporary table  (actual time=84.437..84.437 rows=193 loops=1)
                -> Nested loop inner join  (cost=13164.91 rows=28321) (actual time=0.111..16.670 rows=29999 loops=1)
                    -> Covering index scan on R using listing_id  (cost=3252.56 rows=28321) (actual time=0.095..7.169 rows=29999 loops=1)
                    -> Single-row index lookup on L using PRIMARY (listing_id=R.listing_id)  (cost=0.25 rows=1) (actual time=0.000..0.000 rows=1 loops=29999)
```

The cost of covering index scan decreases from 3568 to 3252, which means this index successfully improves the efficiency of COUNT operation. Consequently, the cost of nested loop inner join decreases from 13480 to 13164.

3. Covering Index on all the columns in the SELECT operation

Before adding index:

```
-------------------------------------------------------------------------------------------------------------------------------------+
| -> Sort: L.listing_id, L.room_type, L.`description`, L.price, L.from_date, L.to_date  (actual time=85.737..85.763 rows=177 loops=1)
    -> Filter: (count(R.review_id) >= 10)  (actual time=84.093..84.197 rows=177 loops=1)
        -> Table scan on <temporary>  (actual time=84.088..84.174 rows=193 loops=1)
            -> Aggregate using temporary table  (actual time=84.084..84.084 rows=193 loops=1)
                -> Nested loop inner join  (cost=13360.56 rows=28321) (actual time=0.296..19.678 rows=29999 loops=1)
                    -> Covering index scan on R using listing_id  (cost=3448.21 rows=28321) (actual time=0.276..9.678 rows=29999 loops=1)
                    -> Single-row index lookup on L using PRIMARY (listing_id=R.listing_id)  (cost=0.25 rows=1) (actual time=0.000..0.000 rows=1 loops=29999)
 |
```

Adding Index:

`CREATE INDEX idx_listing_covering ON Listing (listing_id, room_type, description, price, from_date, to_date);`

After Adding Index:

```
| -> Sort: L.listing_id, L.room_type, L.`description`, L.price, L.from_date, L.to_date  (actual time=80.371..80.387 rows=177 loops=1)
    -> Filter: (count(R.review_id) >= 10)  (actual time=80.113..80.218 rows=177 loops=1)
        -> Table scan on <temporary>  (actual time=80.107..80.196 rows=193 loops=1)
            -> Aggregate using temporary table  (actual time=80.102..80.102 rows=193 loops=1)
                -> Nested loop inner join  (cost=13383.63 rows=28321) (actual time=0.049..17.049 rows=29999 loops=1)
                    -> Covering index scan on R using listing_id  (cost=3471.28 rows=28321) (actual time=0.039..7.529 rows=29999 loops=1)
                    -> Single-row index lookup on L using PRIMARY (listing_id=R.listing_id)  (cost=0.25 rows=1) (actual time=0.000..0.000 rows=1 loops=29999)
 |
```

The execution plan shows that the cost and time for the nested loop inner join did not significantly decrease. As for “Aggregate Using Temporary Table”, the temporary table is still being used for aggregating data, indicating that the grouping and having operations are not directly optimized by the indexes added. The overall cost increased a bit. We believe that, the index on the Review table only includes listing_id, which assists in the join operation but doesn't significantly aid in reducing the time to aggregate review_id counts, as this aggregation still involves creating and scanning a temporary table. Also, if the index created on the Listing table doesn't include all columns used in the grouping, select, and sorting operations (listing_id, room_type, description, price, from_date, to_date), the database might still have to access the table data directly, which doesn't alleviate the load as much as a full covering index might. Furthermore, the sorting operation still takes time, and it seems that the indexes have not eliminated the need to sort the results, which can be time-consuming for larger datasets.

Comparing these three indexing tables, we prefer using the single column index on review_id that can improve the performance of scan and consequently decrease the cost of the nested loop inner join.

Analysis for query 4

```
select avg(scores_rating), landlord_id
from Listing li join Rating r on
li.listing_id = r.listing_id
group by landlord_id
order by avg(scores_rating) desc
```

We have tried three different index designs:

1. Composite Index on (listing_id, scores_rating) in the Rating table:

```
CREATE INDEX idx_rating_listing_scores ON Rating (listing_id,
scores_rating);
```

This composite index is created on the Rating table, covering the listing_id and scores_rating columns. It's designed to optimize queries that involve filtering or joining based on the listing_id and also benefit from the sorted order of scores_rating, potentially improving performance for operations such as JOINs and range queries.

2. Index on (listing_id) in the Rating table:

```
CREATE INDEX idx_rating_listing_id ON Rating(listing_id);
```

This index targets only the listing_id column in the Rating table. It's useful for queries that primarily filter or join based on the listing_id column, improving the speed of retrieval for those specific operations. While it may not cover all aspects of the given query, it still provides optimization for the JOIN condition.

3. Composite Index on (landlord_id, scores_rating) in the Rating table:

```
CREATE INDEX idx_rating_landlord_scores ON Rating (landlord_id, scores_rating);
```

This composite index includes the landlord_id and scores_rating columns in the Rating table. It's particularly beneficial for queries involving grouping by landlord_id and sorting or filtering based on scores_rating. This index can speed up both the grouping and ordering operations in the given query by having the necessary columns together in the index structure.

Analysis for Single-column Index:

Before Adding the Index:

```
| -> Sort: avg(scores_rating) DESC  (actual time=12.734..12.855 rows=2202 loops=1)
    -> Table scan on <temporary>  (actual time=11.395..11.657 rows=2202 loops=1)
        -> Aggregate using temporary table  (actual time=11.393..11.393 rows=2202 loops=1)
            -> Nested loop inner join  (cost=3073.60 rows=6813) (actual time=0.067..7.786 rows=6992 loops=1)
                -> Filter: (r.listing_id is not null)  (cost=689.05 rows=6813) (actual time=0.053..2.826 rows=6992 loops=1)
                    -> Table scan on r  (cost=689.05 rows=6813) (actual time=0.053..2.325 rows=6992 loops=1)
                -> Single-row index lookup on li using PRIMARY (listing_id=r.listing_id)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=6992)
 |
```

Execution Overview:

- Nested Loop Inner Join: High cost at 3073.60 due to a full table scan on Rating and primary index lookups on Listing.
- Aggregation and Sorting: Utilized a temporary table, contributing to overall computational load but not detailed separately in costs.

Adding Index:

```
CREATE INDEX idx_rating_listing_id ON Rating(listing_id);
```

After Adding the Index:

```
--------------------------+
| -> Sort: avg(scores_rating) DESC  (actual time=22.675..22.789 rows=2202 loops=1)
    -> Stream results  (cost=3797.67 rows=7443) (actual time=0.672..21.881 rows=2202 loops=1)
        -> Group aggregate: avg(r.scores_rating)  (cost=3797.67 rows=7443) (actual time=0.666..21.100 rows=2202 loops=1)
            -> Nested loop inner join  (cost=3053.33 rows=7443) (actual time=0.609..19.556 rows=6992 loops=1)
                -> Covering index scan on li using landlord_id  (cost=448.15 rows=4239) (actual time=0.574..1.769 rows=4340 loops=1)
                -> Index lookup on r using idx_rating_listing_id (listing_id=li.listing_id)  (cost=0.44 rows=2) (actual time=0.003..0.004 rows=2 loops=4340)
```

Execution Costs:

- Nested Loop Inner Join: Slightly lower join cost at 3053.33, using the new index for Rating, but overall cost remains high.
- Group Aggregate: Increased cost at 3797.67, suggesting higher computational expense despite more efficient row access.
- Covering Index Scan on li: Cost of 448.15 for a scan presumed to optimize record retrieval but adds overhead.

Index Impact:

- The index on Rating.listing_id improved individual lookup efficiency but did not reduce overall execution costs. Key operations like aggregation and sorting did not benefit substantially from the index, reflecting in a higher total cost post-index.

Reasons for Limited Benefit:

- Aggregation and Sorting Overhead: These processes are central to the query and inherently resource-intensive. The new index does not enhance these particular operations directly.
- Execution Plan Complexity: The index introduced more complexity in data management, which could offset the benefits from faster data retrieval.

Analysis for query 5:

```
SELECT AVG(r.scores_rating) AS mean_score, li.landlord_id
FROM Listing li
JOIN Rating r ON li.listing_id = r.listing_id
GROUP BY li.landlord_id
HAVING AVG(r.scores_rating) > 3.8
ORDER BY mean_score DESC
```

We have tried three different index designs:

1. Composite Index on (listing_id, scores_rating) in the Rating table:

```
CREATE INDEX idx_rating_listing_scores ON Rating (listing_id,
scores_rating);
```

This composite index is created on the Rating table, covering the listing_id and scores_rating columns. It's designed to optimize queries that involve filtering, joining, or sorting based on the listing_id, as well as queries that involve accessing the scores_rating column. By combining these two columns into a single index, it can efficiently support queries that require both columns, potentially improving query performance.

2. Index on Rating.scores_rating:

```
CREATE INDEX idx_rating_scoresrating ON Rating(scores_rating);
```

While this index might not be as critical as the first two for the join and group-by operations, it could be useful for quickly filtering rows in the HAVING clause, where only ratings greater than 3.8 are needed. This index could potentially improve the performance of the aggregation calculation by excluding irrelevant rows early in the processing. However, its effectiveness can vary depending on the distribution of scores_rating values and the database engine's query optimizer.

3. Index on (landlord_id) in the Listing table:

```
CREATE INDEX idx_listing_landlord_id ON Listing (landlord_id);
```

This index targets the landlord_id column in the Listing table. It's particularly beneficial for queries that involve grouping or filtering based on the landlord_id column. By indexing this column, it can speed up operations such as GROUP BY and WHERE clause filtering when accessing or filtering rows based on landlord_id.

Analysis for Composite Index:

Before adding index:

```
--------------------+
| -> Sort: mean_score DESC  (actual time=13.843..13.986 rows=2184 loops=1)
    -> Filter: (avg(r.scores_rating) > 3.8)  (actual time=11.871..12.682 rows=2184 loops=1)
        -> Table scan on <temporary>  (actual time=11.866..12.131 rows=2202 loops=1)
            -> Aggregate using temporary table  (actual time=11.864..11.864 rows=2202 loops=1)
                -> Nested loop inner join  (cost=3073.60 rows=6813) (actual time=0.088..7.945 rows=6992 loops=1)
                    -> Filter: (r.listing_id is not null)  (cost=689.05 rows=6813) (actual time=0.063..2.672 rows=6992 loops=1)
                        -> Table scan on r  (cost=689.05 rows=6813) (actual time=0.060..2.140 rows=6992 loops=1)
                    -> Single-row index lookup on li using PRIMARY (listing_id=r.listing_id)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=6992)
 |
```

Adding index:

```
CREATE INDEX idx_rating_listing_scores ON Rating(listing_id, scores_rating);
```

After adding index:

```
| -> Sort: mean_score DESC  (actual time=27.959..28.247 rows=2184 loops=1)
    -> Filter: (avg(r.scores_rating) > 3.8)  (actual time=24.305..25.703 rows=2184 loops=1)
        -> Table scan on <temporary>  (actual time=24.298..24.763 rows=2202 loops=1)
            -> Aggregate using temporary table  (actual time=24.282..24.282 rows=2202 loops=1)
                -> Nested loop inner join  (cost=3073.60 rows=6813) (actual time=0.124..16.716 rows=6992 loops=1)
                    -> Filter: (r.listing_id is not null)  (cost=689.05 rows=6813) (actual time=0.101..4.415 rows=6992 loops=1)
                        -> Covering index scan on r using idx_rating_listing_scores  (cost=689.05 rows=6813) (actual time=0.095..3.663 rows=6992 loops=1)
                    -> Single-row index lookup on li using PRIMARY (listing_id=r.listing_id)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=6992)
 |
```

We added a covering index to our database with the hope of speeding up a specific SQL query that computes average ratings for listings. Despite our efforts, the performance improvement was marginal. Here's a breakdown of why.

The Index Addition:
The index, named `idx_rating_listing_scores`, included both the `listing_id` for joining tables and `scores_rating` for calculating averages. We anticipated this would make the query faster by allowing quicker data retrieval.

After adding the index, we noticed only minor improvements:

- The join condition did not work faster.
- The sorting of average scores and the use of temporary tables for calculations did not improve as expected.

Why Wasn't the Index More Effective?

- Sorting and Aggregating: The main time-consuming steps were sorting the data and aggregating it to find averages. These processes don't benefit much from indexing.
- Nature of Averages: Calculating averages involves looking at all scores, which can't be sped up by an index.
- Database System's Choices: The way the database chooses to run the query might not take full advantage of the index for sorting and filtering steps.