SchemaX-070: Food Donations Matching System Database Design

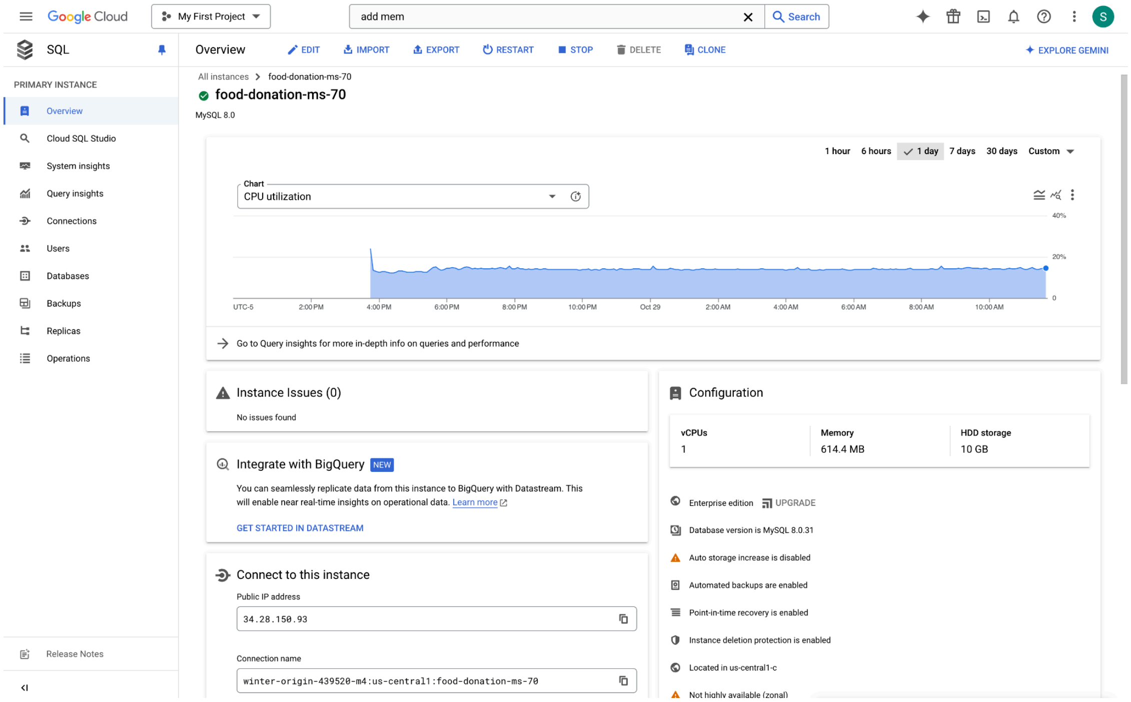
Data Definition Language Commands

```
CREATE TABLE User (
    user_id BIGINT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(1024) NOT NULL,
    location VARCHAR(2048),
    contact_preference VARCHAR(255),
    email VARCHAR(255) NOT NULL,
    phone VARCHAR(255) NOT NULL,
    type VARCHAR(128)
);

CREATE TABLE Donor (
    donor_id BIGINT AUTO_INCREMENT PRIMARY KEY,
    type VARCHAR(255),
    preferred_pickup_time VARCHAR(255),
    user_id BIGINT NOT NULL,
    FOREIGN KEY (user_id) REFERENCES User(user_id)
);

CREATE TABLE Recipient (
    recipient_id BIGINT AUTO_INCREMENT PRIMARY KEY,
    preferences VARCHAR(2048),
    user_id BIGINT NOT NULL,
    notification_enabled BOOLEAN,
    FOREIGN KEY (user_id) REFERENCES User(user_id)
);

CREATE TABLE Item (
    item_id BIGINT AUTO_INCREMENT PRIMARY KEY,
    item_name VARCHAR(255),
    category VARCHAR(255)
);

CREATE TABLE Listing (
    listing_id BIGINT AUTO_INCREMENT PRIMARY KEY,
    listed_by BIGINT NOT NULL,
    location VARCHAR(2048),
    expiration_date DATE,
```

```sql
    type VARCHAR(255),
    pickup_time_range VARCHAR(255),
    status VARCHAR(255),
    FOREIGN KEY (listed_by) REFERENCES Donor(donor_id)
);

CREATE TABLE Booking (
    booking_id BIGINT AUTO_INCREMENT PRIMARY KEY,
    booked_by BIGINT NOT NULL,
    booking_status VARCHAR(255) NOT NULL,
    pickup_datetime Date(255),
    FOREIGN KEY (booked_by) REFERENCES Recipient(recipient_id)
);

CREATE TABLE ListingItem (
    listing_item_id BIGINT AUTO_INCREMENT PRIMARY KEY,
    quantity BIGINT,
    item_id BIGINT NOT NULL,
    listing_id BIGINT NOT NULL,
    booking_id BIGINT,
    expiration_date DATE,
    status VARCHAR(255),
    FOREIGN KEY (item_id) REFERENCES Item(item_id),
    FOREIGN KEY (listing_id) REFERENCES Listing(listing_id),
    FOREIGN KEY (booking_id) REFERENCES Booking(booking_id)
);

CREATE TABLE Review (
    review_id BIGINT AUTO_INCREMENT PRIMARY KEY,
    review TEXT,
    review_date DATE,
    rating INT,
    donor_id BIGINT NOT NULL,
    recipient_id BIGINT NOT NULL,
    FOREIGN KEY (donor_id) REFERENCES Donor(donor_id),
    FOREIGN KEY (recipient_id) REFERENCES Recipient(recipient_id)
);

CREATE TABLE Recommendation (
    matching_id BIGINT AUTO_INCREMENT PRIMARY KEY,
    type VARCHAR(255),
    food_type VARCHAR(255),
    recipient_id BIGINT NOT NULL,
    donor_id BIGINT NOT NULL,
    score DOUBLE,
    FOREIGN KEY (recipient_id) REFERENCES Recipient(recipient_id),
    FOREIGN KEY (donor_id) REFERENCES Donor(donor_id)
);
```

Database Hosting (GCP), Setup and Connection

The database server is hosted on GCP with minimum configurations as suggested in the workshop lecture.



food-donation-ms-70

MySQL 8.0

Go to Query insights for more in-depth info on queries and performance

Instance Issues (0)

No issues found

Integrate with BigQuery NEW

You can seamlessly replicate data from this instance to BigQuery with Datastream. This will enable near real-time insights on operational data. Learn more

Connect to this instance

Public IP address

34.28.150.93

Connection name

winter-origin-439520-m4:us-central1:food-donation-ms-70

Configuration

vCPUs	Memory	HDD storage
1	614.4 MB	10 GB

Enterprise edition

Database version is MySQL 8.0.31

Auto storage increase is disabled

Automated backups are enabled

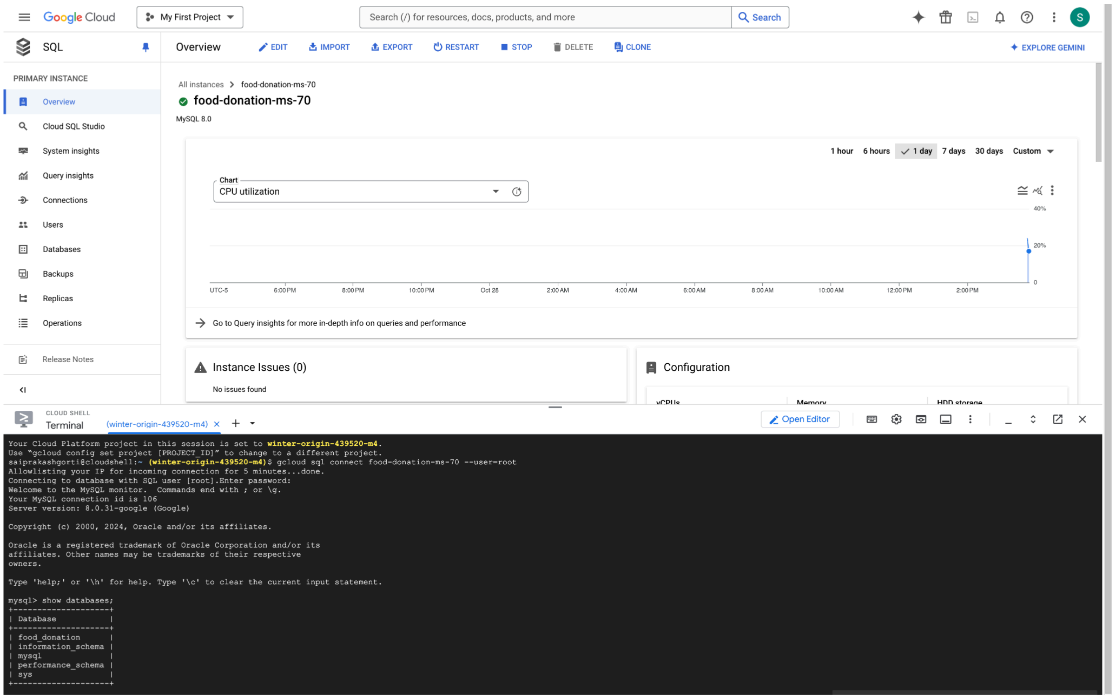
Point-in-time recovery is enabled

Instance deletion protection is enabled

Located in us-central1-c

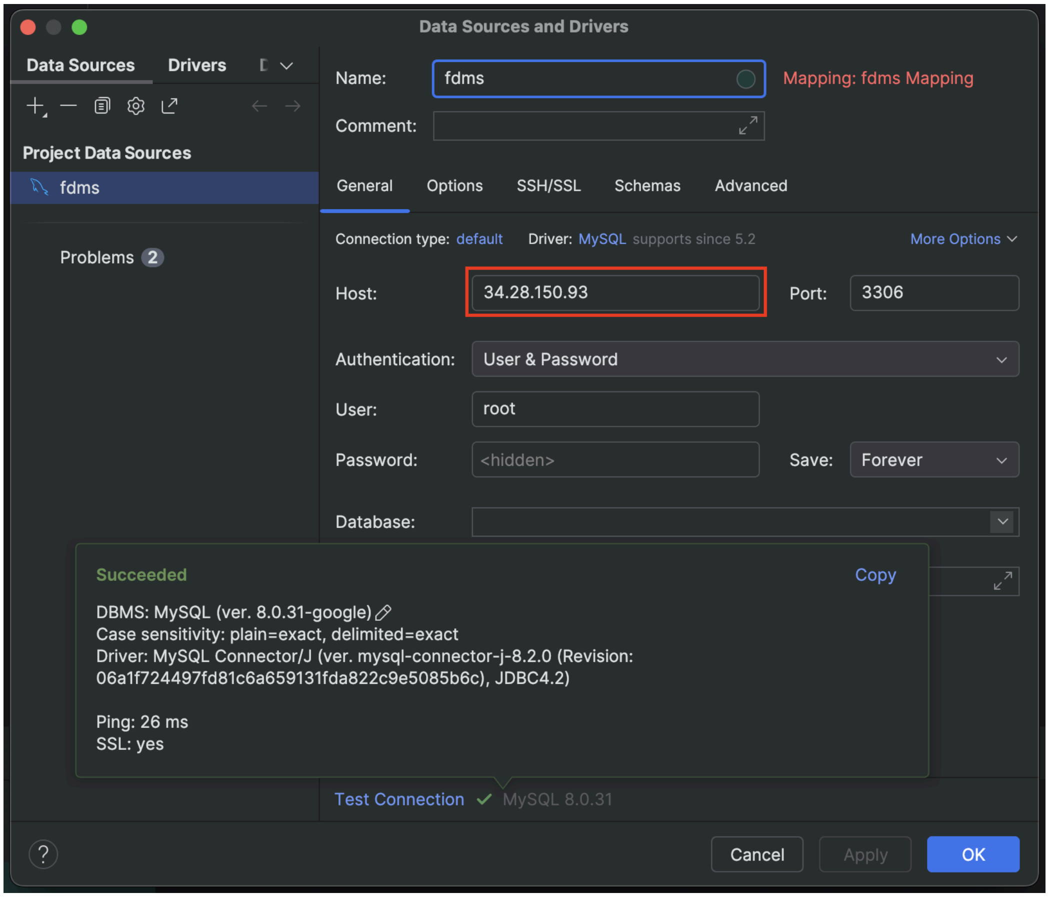
Not highly available (zonal)

Here’s the connection through GCP Cloud Shell



```
Your Cloud Platform project in this session is set to winter-origin-439520-m4.
Use “gcloud config set project [PROJECT_ID]” to change to a different project.
saiprakashgorti@cloudshell:~ (winter-origin-439520-m4)$ gcloud sql connect food-donation-ms-70 --user=root
Allowlisting your IP for incoming connection for 5 minutes...done.
Connecting to database with SQL user [root].Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 106
Server version: 8.0.31-google (Google)

Copyright (c) 2000, 2024, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

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

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| food_donation      |
| information_schema |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
```

Moving forward, we’ll use Datagrip to connect to this server for a better user interface when writing queries. Here’s the connection to the MySQL server hosted on GCP through Datagrip.



Name: fdms

Mapping: fdms Mapping

Comment:

Connection type: default Driver: MySQL supports since 5.2

Host: 34.28.150.93

Port: 3306

Authentication: User & Password

User: root

Password: <hidden>

Save: Forever

Database:

Succeeded

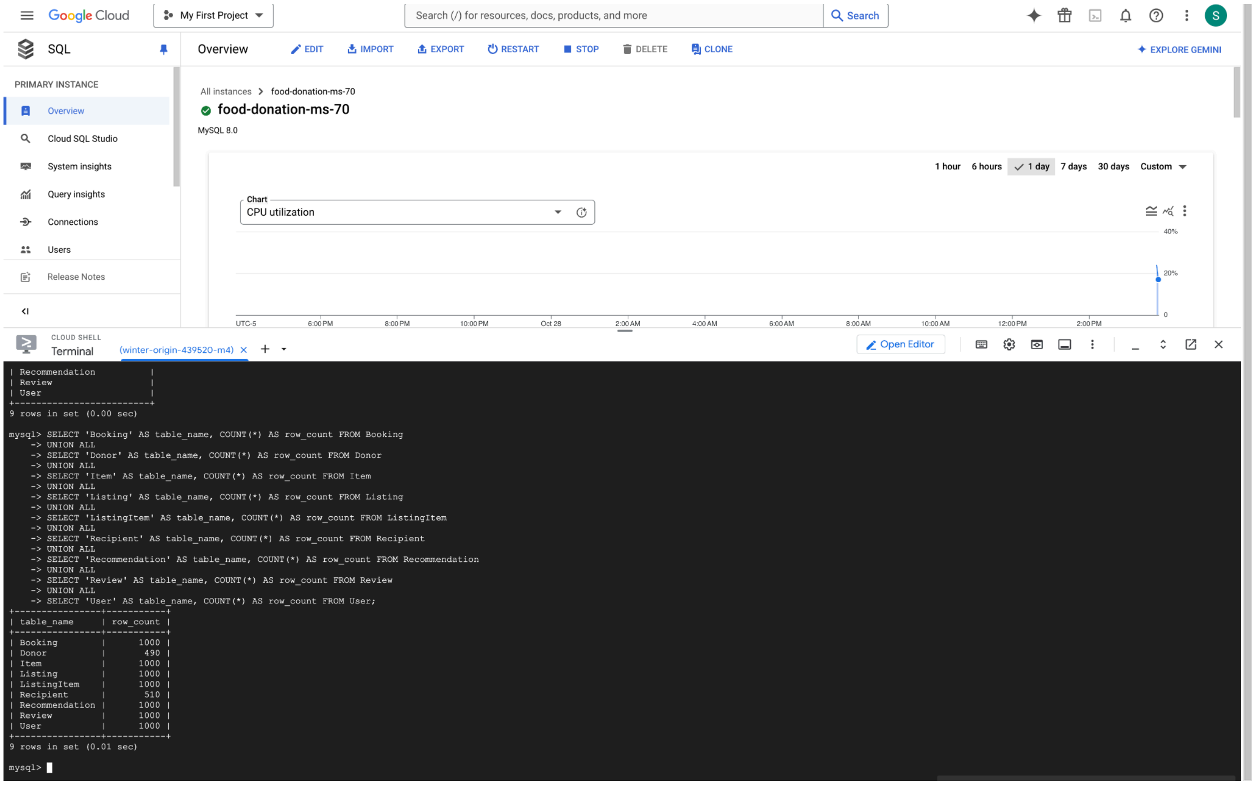
DBMS: MySQL (ver. 8.0.31-google)
Case sensitivity: plain=exact, delimited=exact
Driver: MySQL Connector/J (ver. mysql-connector-j-8.2.0 (Revision: 06a1f724497fd81c6a659131fda822c9e5085b6c), JDBC4.2)

Ping: 26 ms
SSL: yes

Test Connection ✓ MySQL 8.0.31

Data Insertion

The data inserted into the tables is a combination of the datasets mentioned in the previous stage and synthetic data to get optimal results. Here’s the number of rows we inserted into each table.



```
| Recommendation          |
| Review                  |
| User                    |
+-------------------------+
9 rows in set (0.00 sec)

mysql> SELECT 'Booking' AS table_name, COUNT(*) AS row_count FROM Booking
    -> UNION ALL
    -> SELECT 'Donor' AS table_name, COUNT(*) AS row_count FROM Donor
    -> UNION ALL
    -> SELECT 'Item' AS table_name, COUNT(*) AS row_count FROM Item
    -> UNION ALL
    -> SELECT 'Listing' AS table_name, COUNT(*) AS row_count FROM Listing
    -> UNION ALL
    -> SELECT 'ListingItem' AS table_name, COUNT(*) AS row_count FROM ListingItem
    -> UNION ALL
    -> SELECT 'Recipient' AS table_name, COUNT(*) AS row_count FROM Recipient
    -> UNION ALL
    -> SELECT 'Recommendation' AS table_name, COUNT(*) AS row_count FROM Recommendation
    -> UNION ALL
    -> SELECT 'Review' AS table_name, COUNT(*) AS row_count FROM Review
    -> UNION ALL
    -> SELECT 'User' AS table_name, COUNT(*) AS row_count FROM User;
+----------------+-----------+
| table_name     | row_count |
+----------------+-----------+
| Booking        |      1000 |
| Donor          |       490 |
| Item           |      1000 |
| Listing        |      1000 |
| ListingItem    |      1000 |
| Recipient      |       510 |
| Recommendation |      1000 |
| Review         |      1000 |
| User           |      1000 |
+----------------+-----------+
9 rows in set (0.01 sec)

mysql> █
```

Advanced SQL Queries

AQ1: Listing Status Summary for Donor

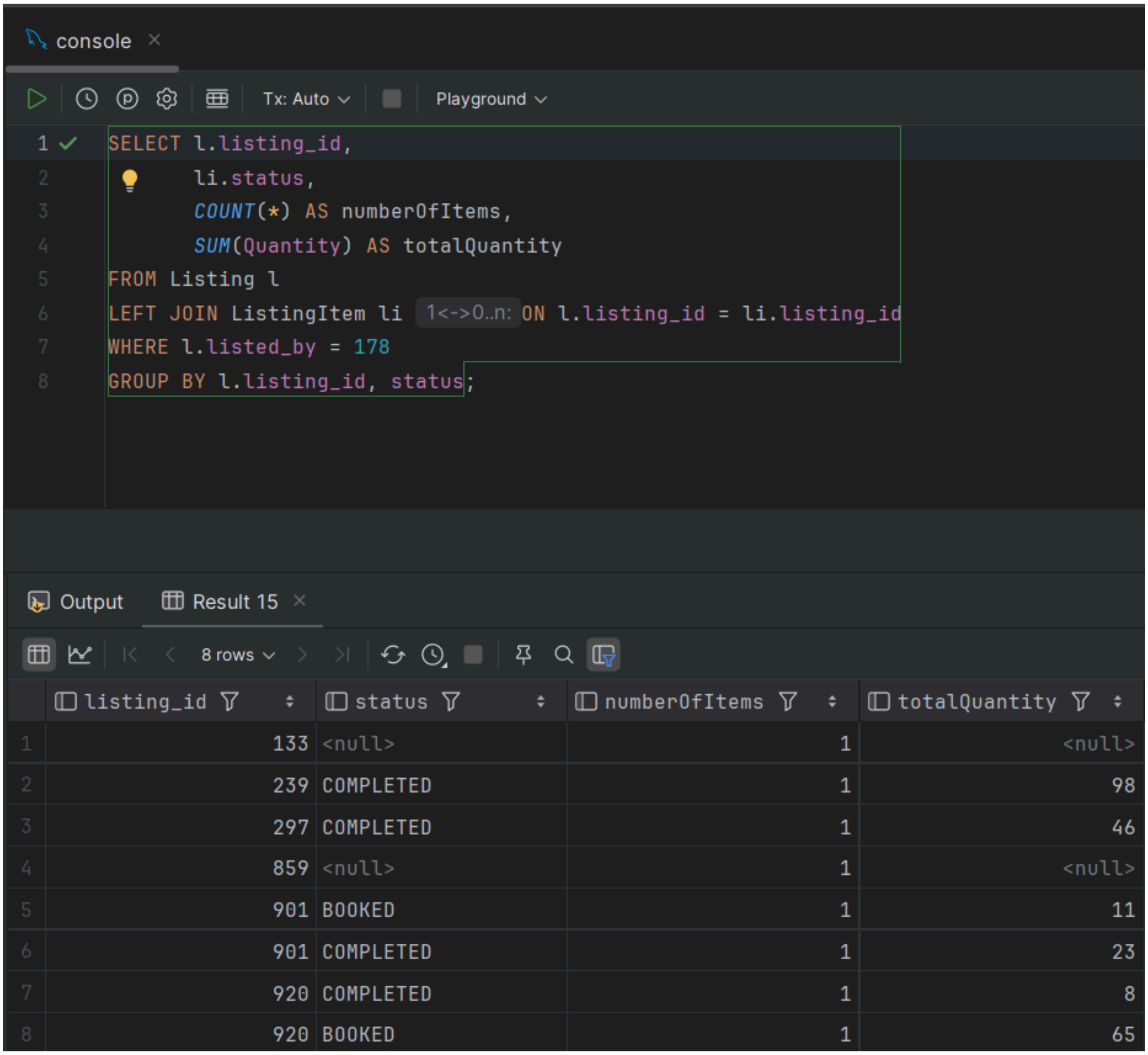
This query is to provide a summary of all the listings and their associated items for a specific donor. The key purpose is to keep track of the status of all listings and the number of items within each listing.

Note: The query will have 2 Complex SQL relations: JOINS, AGGREGATION

```
SELECT l.listing_id,
       li.status,
       COUNT(*) AS numberOfItems,
       SUM(Quantity) AS totalQuantity
FROM Listing l
LEFT JOIN ListingItem li ON l.listing_id = li.listing_id
```

```
WHERE l.listed_by = '<donor_id>'
GROUP BY l.listing_id, status;
```

Below is the same query run on actual data.



```
SELECT l.listing_id,
       li.status,
       COUNT(*) AS numberOfItems,
       SUM(Quantity) AS totalQuantity
FROM Listing l
LEFT JOIN ListingItem li ON l.listing_id = li.listing_id
WHERE l.listed_by = 178
GROUP BY l.listing_id, status;
```

	listing_id	status	numberOfItems	totalQuantity
1	133	<null>	1	<null>
2	239	COMPLETED	1	98
3	297	COMPLETED	1	46
4	859	<null>	1	<null>
5	901	BOOKED	1	11
6	901	COMPLETED	1	23
7	920	COMPLETED	1	8
8	920	BOOKED	1	65

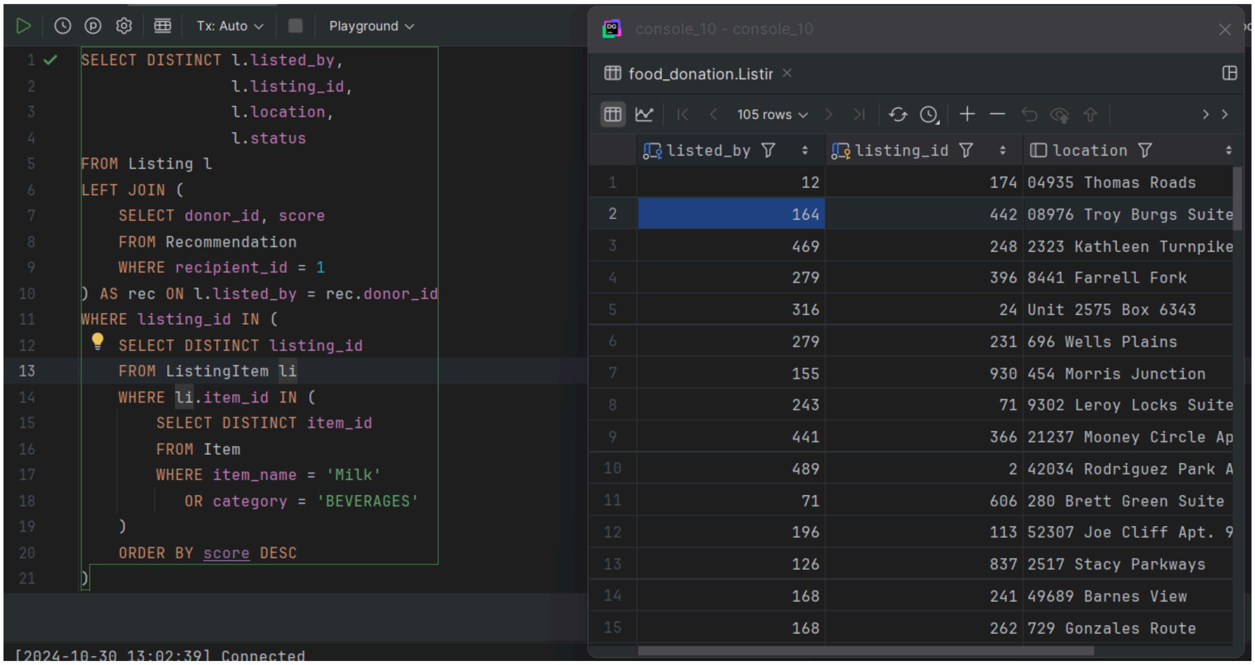
AQ2: Search-based Listing Recommendations

This query combines search functionality with a recommendations table to fetch relevant listings having the best recommendation scores by using nested queries to filter out the items based on search terms (ordered by best score) and then joining listings with recommendations on donor_id.

Note: The query will have 2 Complex SQL relations: JOINS, NESTED SUBQUERY

```sql
SELECT DISTINCT l.listed_by,
                l.listing_id,
                l.location,
                l.status
FROM Listing l
LEFT JOIN (
    SELECT donor_id, score
    FROM Recommendation
    WHERE recipient_id = <RECIPIENT_ID>
) AS rec ON l.listed_by = rec.donor_id
WHERE listing_id IN (
    SELECT DISTINCT listing_id
    FROM ListingItem li
    WHERE li.item_id IN (
        SELECT DISTINCT item_id
        FROM Item
        WHERE item_name = '<ITEM_SEARCH_QUERY>'
            OR category = '<CATEGORY_SEARCH_QUERY>'
    )
    ORDER BY score DESC
);
```

Below is the same query run on actual data. The resulting output contains more than 15 rows; hence, LIMIT was used to show 15 rows.



```sql
SELECT DISTINCT l.listed_by,
                l.listing_id,
                l.location,
                l.status
FROM Listing l
LEFT JOIN (
    SELECT donor_id, score
    FROM Recommendation
    WHERE recipient_id = 1
) AS rec ON l.listed_by = rec.donor_id
WHERE listing_id IN (
    SELECT DISTINCT listing_id
    FROM ListingItem li
    WHERE li.item_id IN (
        SELECT DISTINCT item_id
        FROM Item
        WHERE item_name = 'Milk'
            OR category = 'BEVERAGES'
    )
    ORDER BY score DESC
)
```

```
[2024-10-30 13:02:39] Connected
```

	listed_by	listing_id	location
1	12	174	04935 Thomas Roads
2	164	442	08976 Troy Burgs Suite
3	469	248	2323 Kathleen Turnpike
4	279	396	8441 Farrell Fork
5	316	24	Unit 2575 Box 6343
6	279	231	696 Wells Plains
7	155	930	454 Morris Junction
8	243	71	9302 Leroy Locks Suite
9	441	366	21237 Mooney Circle Ap
10	489	2	42034 Rodriguez Park A
11	71	606	280 Brett Green Suite
12	196	113	52307 Joe Cliff Apt. 9
13	126	837	2517 Stacy Parkways
14	168	241	49689 Barnes View
15	168	262	729 Gonzales Route

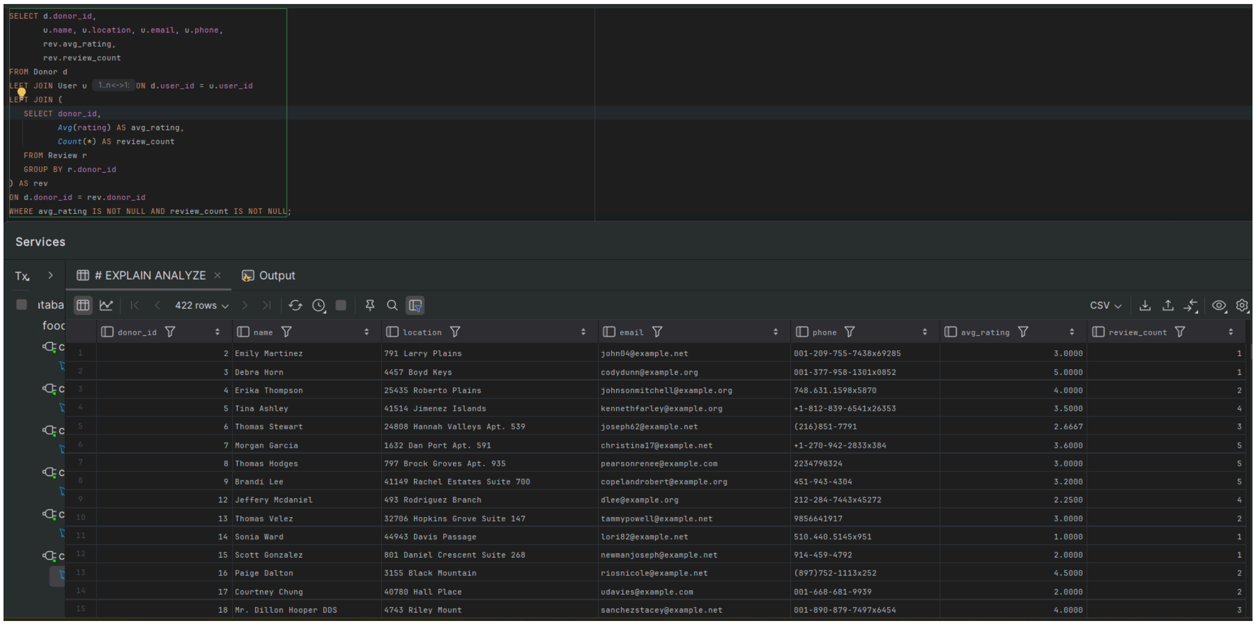
AQ3: Fetch Detailed Donor Profile

The below query fetches the donors details, along with the average rating and overall number of reviews. This will be used on the donor detail view page, where we fetch the user details along with the review summary. We can also add filter to filter for a specific donor_id using where clause as per need.

Note: The query will have 3 Complex SQL relations: JOINS, AGGREGATIONS, SUBQUERY

```sql
SELECT d.donor_id,
       u.name, u.location, u.email, u.phone,
       rev.avg_rating,
       rev.review_count
FROM Donor d
LEFT JOIN User u ON d.user_id = u.user_id
LEFT JOIN (
   SELECT donor_id,
          Avg(rating) AS avg_rating,
          Count(*) AS review_count
   FROM Review r
   GROUP BY r.donor_id
) AS rev
ON d.donor_id = rev.donor_id
WHERE avg_rating IS NOT NULL AND review_count IS NOT NULL
```

Below is the same query run on actual data. The resulting output contains more than 15 rows; hence, LIMIT was used to show 15 rows.



```sql
SELECT d.donor_id,
       u.name, u.location, u.email, u.phone,
       rev.avg_rating,
       rev.review_count
FROM Donor d
LEFT JOIN User u  1..n<->1:  ON d.user_id = u.user_id
LEFT JOIN (
   SELECT donor_id,
          Avg(rating) AS avg_rating,
          Count(*) AS review_count
   FROM Review r
   GROUP BY r.donor_id
) AS rev
ON d.donor_id = rev.donor_id
WHERE avg_rating IS NOT NULL AND review_count IS NOT NULL;
```

	donor_id	name	location	email	phone	avg_rating	review_count
1	2	Emily Martinez	791 Larry Plains	john04@example.net	001-209-755-7438x69285	3.0000	1
2	3	Debra Horn	4457 Boyd Keys	codydunn@example.org	001-377-958-1301x0852	5.0000	1
3	4	Erika Thompson	25435 Roberto Plains	johnsonmitchell@example.org	748.631.1598x5870	4.0000	2
4	5	Tina Ashley	41514 Jimenez Islands	kennethfarley@example.org	+1-812-839-6541x26353	3.5000	4
5	6	Thomas Stewart	24808 Hannah Valleys Apt. 539	joseph62@example.net	(216)851-7791	2.6667	3
6	7	Morgan Garcia	1632 Dan Port Apt. 591	christina17@example.net	+1-270-942-2833x384	3.6000	5
7	8	Thomas Hodges	797 Brock Groves Apt. 935	pearsonrenee@example.com	2234798324	3.0000	5
8	9	Brandi Lee	41149 Rachel Estates Suite 700	copelandrobert@example.org	451-943-4304	3.2000	5
9	12	Jeffery Mcdaniel	493 Rodriguez Branch	dlee@example.org	212-284-7443x45272	2.2500	4
10	13	Thomas Velez	32706 Hopkins Grove Suite 147	tammypowell@example.net	9856641917	3.0000	2
11	14	Sonia Ward	44943 Davis Passage	lori82@example.net	510.440.5145x951	1.0000	1
12	15	Scott Gonzalez	801 Daniel Crescent Suite 268	newmanjoseph@example.net	914-459-4792	2.0000	1
13	16	Paige Dalton	3155 Black Mountain	riosnicole@example.net	(897)752-1113x252	4.5000	2
14	17	Courtney Chung	40780 Hall Place	udavies@example.com	001-668-681-9939	2.0000	2
15	18	Mr. Dillon Hooper DDS	4743 Riley Mount	sanchezstacey@example.net	001-890-879-7497x6454	4.0000	3

AQ4: Fetch Available Listings by Recipient Preferences

This query fetches all the available listings based on the preferences set by users. The main idea is to split all the comma-separated preferences and then search for these in the listings items table. Also, need to ensure that either the product has no expiry date, or it has not expired yet. Another check would be to check if each listing is ACTIVE or not, and if the listing_item is AVAILABLE or not.

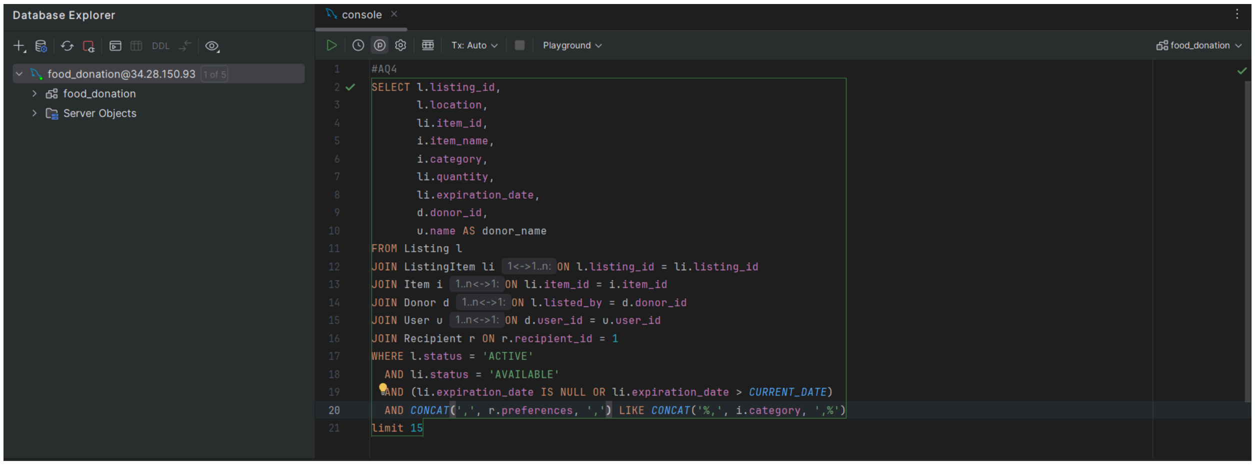
Note: This query requires multiple JOIN, NESTED SUBQUERY.

```sql
SELECT l.listing_id, l.location,
       li.item_id,
       i.item_name,
       i.category,
       li.quantity, li.expiration_date,
       d.donor_id,
       u.name AS donor_name
FROM Listing l
JOIN ListingItem li ON l.listing_id = li.listing_id
JOIN Item i ON li.item_id = i.item_id
JOIN Donor d ON l.listed_by = d.donor_id
JOIN User u ON d.user_id = u.user_id
JOIN Recipient r ON r.recipient_id = <RECIPIENT_ID>
WHERE l.status = 'ACTIVE' AND li.status = 'AVAILABLE'
  AND (li.expiration_date IS NULL OR li.expiration_date > CURRENT_DATE)
  AND CONCAT(',', r.preferences, ',') LIKE CONCAT('%,', i.category, ',%')
```

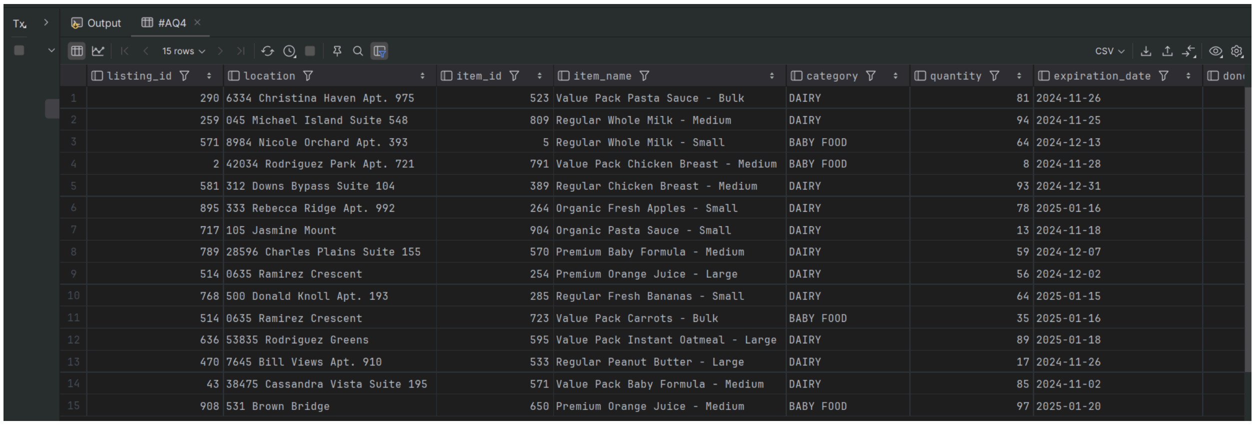
We can extend this by adding a sort-by option and appending the following to the query

```sql
ORDER BY li.expiration_date ASC,
         l.pickup_time_range ASC;
```

Below is the same query run on actual data. The resulting output contains more than 15 rows; hence, LIMIT was used to show 15 rows.



```sql
#AQ4
SELECT l.listing_id,
       l.location,
       li.item_id,
       i.item_name,
       i.category,
       li.quantity,
       li.expiration_date,
       d.donor_id,
       u.name AS donor_name
FROM Listing l
JOIN ListingItem li ON l.listing_id = li.listing_id
JOIN Item i ON li.item_id = i.item_id
JOIN Donor d ON l.listed_by = d.donor_id
JOIN User u ON d.user_id = u.user_id
JOIN Recipient r ON r.recipient_id = 1
WHERE l.status = 'ACTIVE'
  AND li.status = 'AVAILABLE'
  AND (li.expiration_date IS NULL OR li.expiration_date > CURRENT_DATE)
  AND CONCAT(',', r.preferences, ',') LIKE CONCAT('%,', i.category, ',%')
limit 15
```

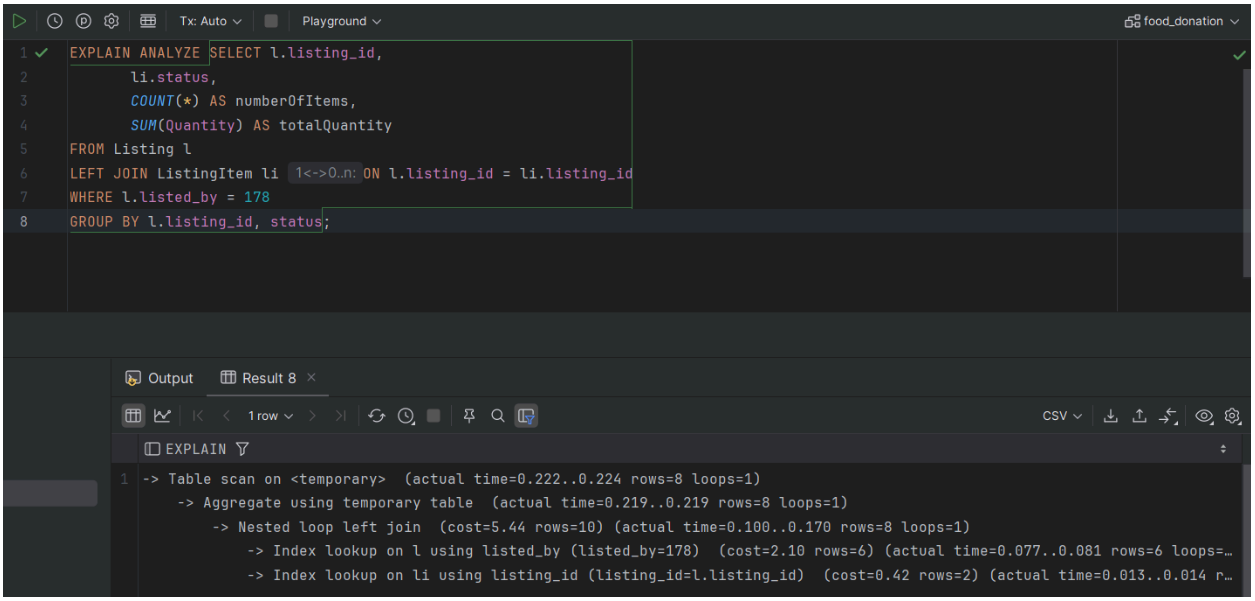


	listing_id	location	item_id	item_name	category	quantity	expiration_date	don
1	290	6334 Christina Haven Apt. 975	523	Value Pack Pasta Sauce - Bulk	DAIRY	81	2024-11-26	
2	259	045 Michael Island Suite 548	809	Regular Whole Milk - Medium	DAIRY	94	2024-11-25	
3	571	8984 Nicole Orchard Apt. 393	5	Regular Whole Milk - Small	BABY FOOD	64	2024-12-13	
4	2	42034 Rodriguez Park Apt. 721	791	Value Pack Chicken Breast - Medium	BABY FOOD	8	2024-11-28	
5	581	312 Downs Bypass Suite 104	389	Regular Chicken Breast - Medium	DAIRY	93	2024-12-31	
6	895	333 Rebecca Ridge Apt. 992	264	Organic Fresh Apples - Small	DAIRY	78	2025-01-16	
7	717	105 Jasmine Mount	904	Organic Pasta Sauce - Small	DAIRY	13	2024-11-18	
8	789	28596 Charles Plains Suite 155	570	Premium Baby Formula - Medium	DAIRY	59	2024-12-07	
9	514	0635 Ramirez Crescent	254	Premium Orange Juice - Large	DAIRY	56	2024-12-02	
10	768	500 Donald Knoll Apt. 193	285	Regular Fresh Bananas - Small	DAIRY	64	2025-01-15	
11	514	0635 Ramirez Crescent	723	Value Pack Carrots - Bulk	BABY FOOD	35	2025-01-16	
12	636	53835 Rodriguez Greens	595	Value Pack Instant Oatmeal - Large	DAIRY	89	2025-01-18	
13	470	7645 Bill Views Apt. 910	533	Regular Peanut Butter - Large	DAIRY	17	2024-11-26	
14	43	38475 Cassandra Vista Suite 195	571	Value Pack Baby Formula - Medium	DAIRY	85	2024-11-02	
15	908	531 Brown Bridge	650	Premium Orange Juice - Medium	BABY FOOD	97	2025-01-20	

Indexing: Design and Analysis

AQ1: Index Design and Analysis

The EXPLAIN ANALYZE command BEFORE Indexing is shown below:

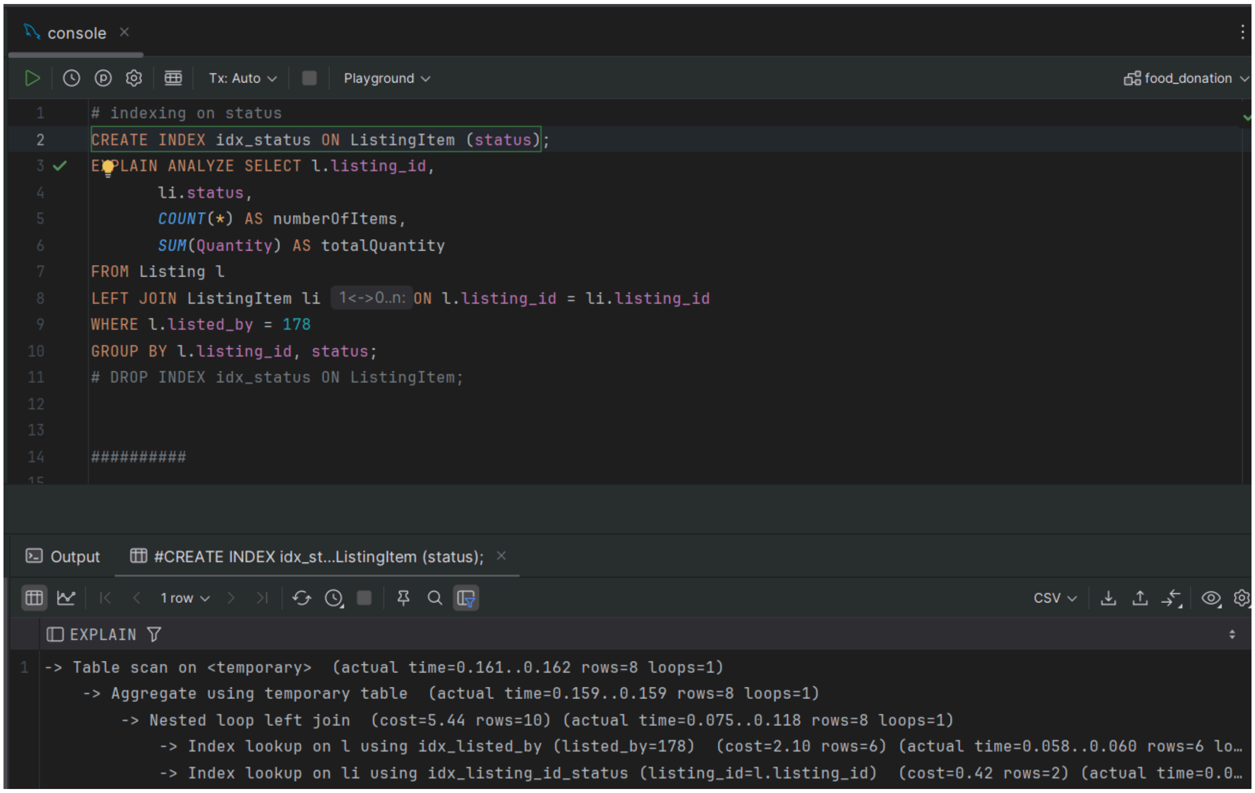


```
EXPLAIN ANALYZE SELECT l.listing_id,
       li.status,
       COUNT(*) AS numberOfItems,
       SUM(Quantity) AS totalQuantity
FROM Listing l
LEFT JOIN ListingItem li 1<->0..n: ON l.listing_id = li.listing_id
WHERE l.listed_by = 178
GROUP BY l.listing_id, status;
```

```
-> Table scan on <temporary>  (actual time=0.222..0.224 rows=8 loops=1)
    -> Aggregate using temporary table  (actual time=0.219..0.219 rows=8 loops=1)
        -> Nested loop left join  (cost=5.44 rows=10) (actual time=0.100..0.170 rows=8 loops=1)
            -> Index lookup on l using listed_by (listed_by=178)  (cost=2.10 rows=6) (actual time=0.077..0.081 rows=6 loops=…
            -> Index lookup on li using listing_id (listing_id=l.listing_id)  (cost=0.42 rows=2) (actual time=0.013..0.014 r…
```

Let’s explore some Index designs for the AQ1:

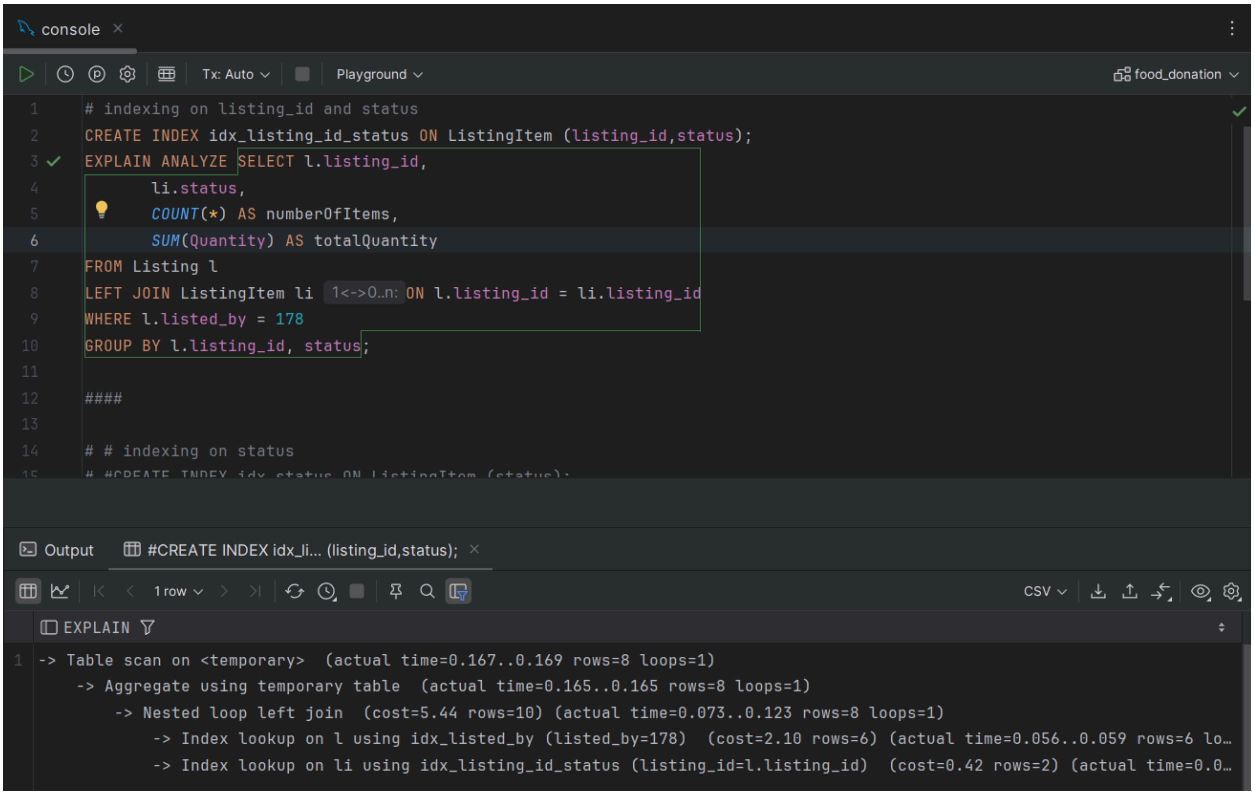
1) Creating index on ListingItem.Status**:** No effect was observed on the overall cost.



```
# indexing on status
CREATE INDEX idx_status ON ListingItem (status);
EXPLAIN ANALYZE SELECT l.listing_id,
       li.status,
       COUNT(*) AS numberOfItems,
       SUM(Quantity) AS totalQuantity
FROM Listing l
LEFT JOIN ListingItem li 1<->0..n: ON l.listing_id = li.listing_id
WHERE l.listed_by = 178
GROUP BY l.listing_id, status;
# DROP INDEX idx_status ON ListingItem;


##########
```

```
-> Table scan on <temporary>  (actual time=0.161..0.162 rows=8 loops=1)
    -> Aggregate using temporary table  (actual time=0.159..0.159 rows=8 loops=1)
        -> Nested loop left join  (cost=5.44 rows=10) (actual time=0.075..0.118 rows=8 loops=1)
            -> Index lookup on l using idx_listed_by (listed_by=178)  (cost=2.10 rows=6) (actual time=0.058..0.060 rows=6 lo…
            -> Index lookup on li using idx_listing_id_status (listing_id=l.listing_id)  (cost=0.42 rows=2) (actual time=0.0…
```

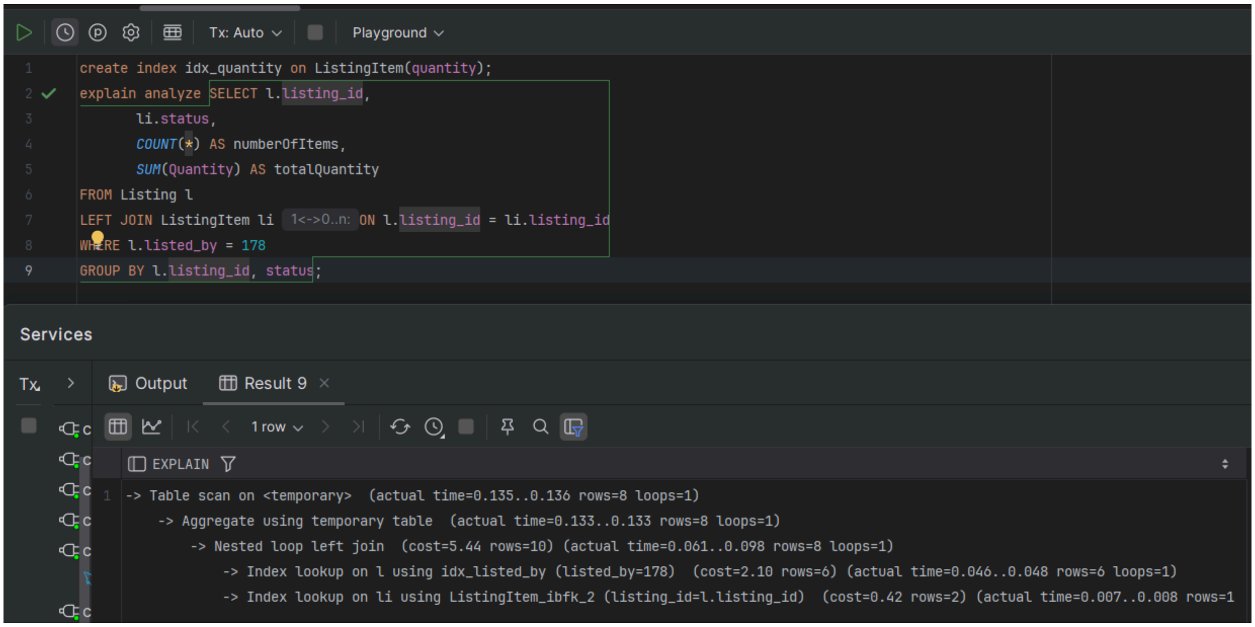
2) <u>Creating composite index on listing_id and Status</u>: No effect observed on overall cost.



```sql
# indexing on listing_id and status
CREATE INDEX idx_listing_id_status ON ListingItem (listing_id,status);
EXPLAIN ANALYZE SELECT l.listing_id,
       li.status,
       COUNT(*) AS numberOfItems,
       SUM(Quantity) AS totalQuantity
FROM Listing l
LEFT JOIN ListingItem li ON l.listing_id = li.listing_id
WHERE l.listed_by = 178
GROUP BY l.listing_id, status;

####

# # indexing on status
# #CREATE INDEX idx_status ON ListingItem (status);
```

```
-> Table scan on <temporary>  (actual time=0.167..0.169 rows=8 loops=1)
    -> Aggregate using temporary table  (actual time=0.165..0.165 rows=8 loops=1)
        -> Nested loop left join  (cost=5.44 rows=10) (actual time=0.073..0.123 rows=8 loops=1)
            -> Index lookup on l using idx_listed_by (listed_by=178)  (cost=2.10 rows=6) (actual time=0.056..0.059 rows=6 lo…
            -> Index lookup on li using idx_listing_id_status (listing_id=l.listing_id)  (cost=0.42 rows=2) (actual time=0.0…
```

3) <u>Creating Index on Quantity</u>: The overall cost remains contant even with Quantity index



```sql
create index idx_quantity on ListingItem(quantity);
explain analyze SELECT l.listing_id,
       li.status,
       COUNT(*) AS numberOfItems,
       SUM(Quantity) AS totalQuantity
FROM Listing l
LEFT JOIN ListingItem li ON l.listing_id = li.listing_id
WHERE l.listed_by = 178
GROUP BY l.listing_id, status;
```

```
-> Table scan on <temporary>  (actual time=0.135..0.136 rows=8 loops=1)
    -> Aggregate using temporary table  (actual time=0.133..0.133 rows=8 loops=1)
        -> Nested loop left join  (cost=5.44 rows=10) (actual time=0.061..0.098 rows=8 loops=1)
            -> Index lookup on l using idx_listed_by (listed_by=178)  (cost=2.10 rows=6) (actual time=0.046..0.048 rows=6 loops=1)
            -> Index lookup on li using ListingItem_ibfk_2 (listing_id=l.listing_id)  (cost=0.42 rows=2) (actual time=0.007..0.008 rows=1
```

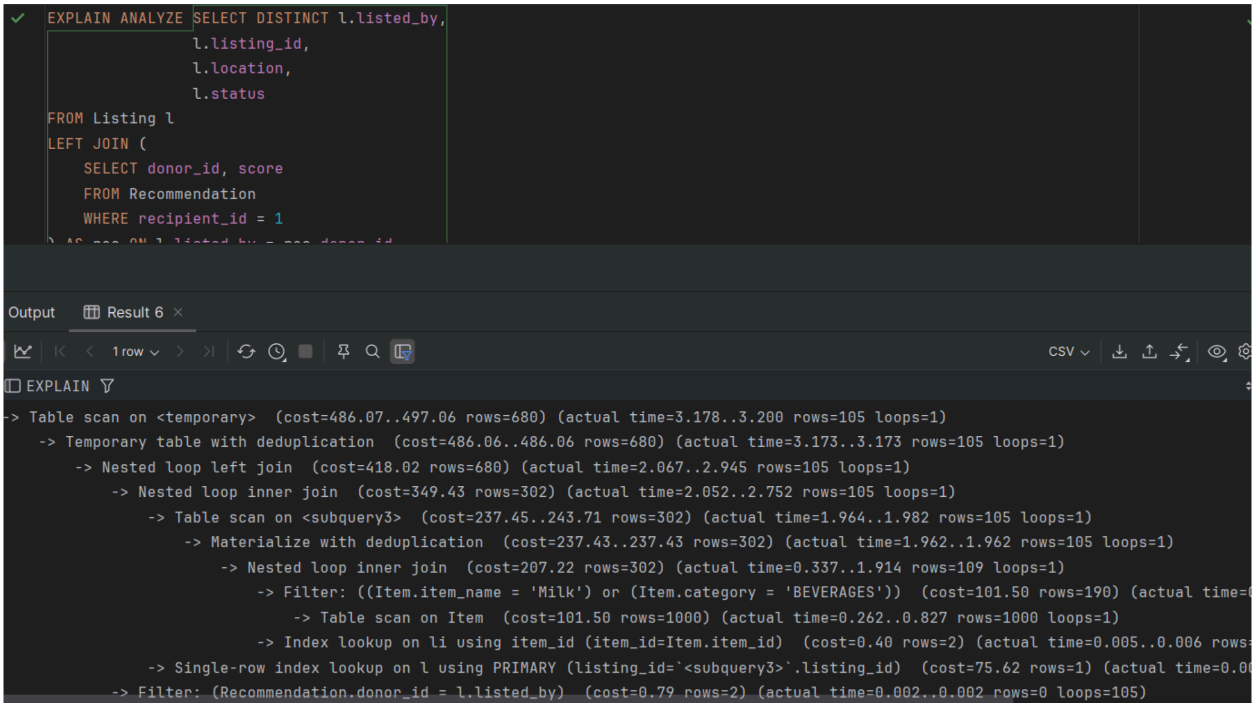
Analysis/Reason for no change in cost:

Despite creating a composite index on **listing_id** and **status**, no improvement was observed in the query's performance. This can be attributed to the query structure, which utilizes a LEFT JOIN and a filter condition (WHERE l.listed_by = '<donor_id>') that does not involve the indexed fields. Therefore, the index on listing_id and status is not fully utilized because the query's filtering condition is based on **listed_by** (**already a foreign key index**), making the composite index less relevant for optimizing this specific query. Additionally, aggregating and grouping operations (**COUNT** and **SUM**) may already be efficiently handled by the database engine without significant dependence on the specified index, further reducing the index's impact. Also, the cardinality of the **status** column is less, which might be the reason behind no change since we are anyway fetching that many rows. Also, indexing on quantity did not bring that much impact. The reason might be that it is a high cardinality column which is used inside aggregation. As we can see, the index was not even used in the query plan.

Note: The above query **only** has a **listing_id** and **status** field on which the Indexing can be applied on. Other columns are already foreign or primary key, hence we can't index on those..

AQ2: Index Design and Analysis

For AQ2, the EXPLAIN ANALYZE command BEFORE Indexing is shown below:

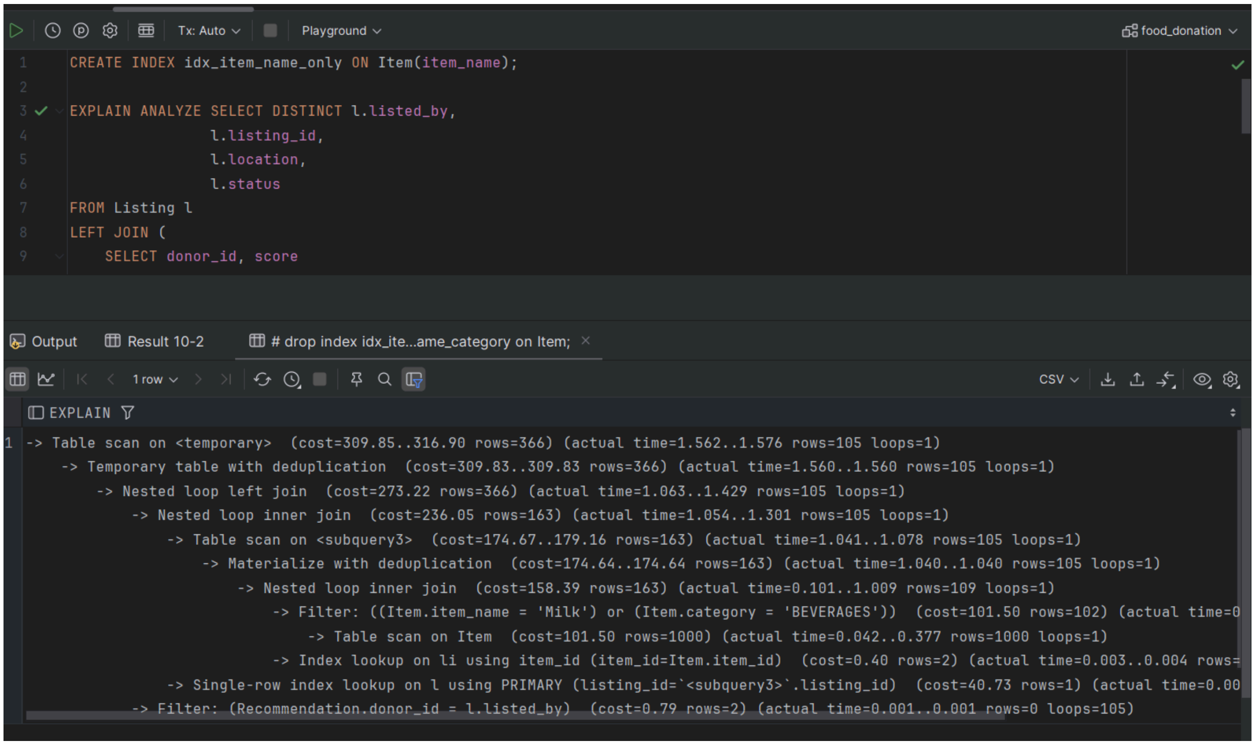


```
EXPLAIN ANALYZE SELECT DISTINCT l.listed_by,
                l.listing_id,
                l.location,
                l.status
FROM Listing l
LEFT JOIN (
    SELECT donor_id, score
    FROM Recommendation
    WHERE recipient_id = 1
) AS rec ON l.listed_by = rec.donor_id
```

```
-> Table scan on <temporary>  (cost=486.07..497.06 rows=680) (actual time=3.178..3.200 rows=105 loops=1)
    -> Temporary table with deduplication  (cost=486.06..486.06 rows=680) (actual time=3.173..3.173 rows=105 loops=1)
        -> Nested loop left join  (cost=418.02 rows=680) (actual time=2.067..2.945 rows=105 loops=1)
            -> Nested loop inner join  (cost=349.43 rows=302) (actual time=2.052..2.752 rows=105 loops=1)
                -> Table scan on <subquery3>  (cost=237.45..243.71 rows=302) (actual time=1.964..1.982 rows=105 loops=1)
                    -> Materialize with deduplication  (cost=237.43..237.43 rows=302) (actual time=1.962..1.962 rows=105 loops=1)
                        -> Nested loop inner join  (cost=207.22 rows=302) (actual time=0.337..1.914 rows=109 loops=1)
                            -> Filter: ((Item.item_name = 'Milk') or (Item.category = 'BEVERAGES'))  (cost=101.50 rows=190) (actual time=(
                                -> Table scan on Item  (cost=101.50 rows=1000) (actual time=0.262..0.827 rows=1000 loops=1)
                            -> Index lookup on li using item_id (item_id=Item.item_id)  (cost=0.40 rows=2) (actual time=0.005..0.006 rows=
                -> Single-row index lookup on l using PRIMARY (listing_id=`<subquery3>`.listing_id)  (cost=75.62 rows=1) (actual time=0.00
            -> Filter: (Recommendation.donor_id = l.listed_by)  (cost=0.79 rows=2) (actual time=0.002..0.002 rows=0 loops=105)
```

Let's explore the index designs for AQ2:

1) Indexing on Item.item_name only:

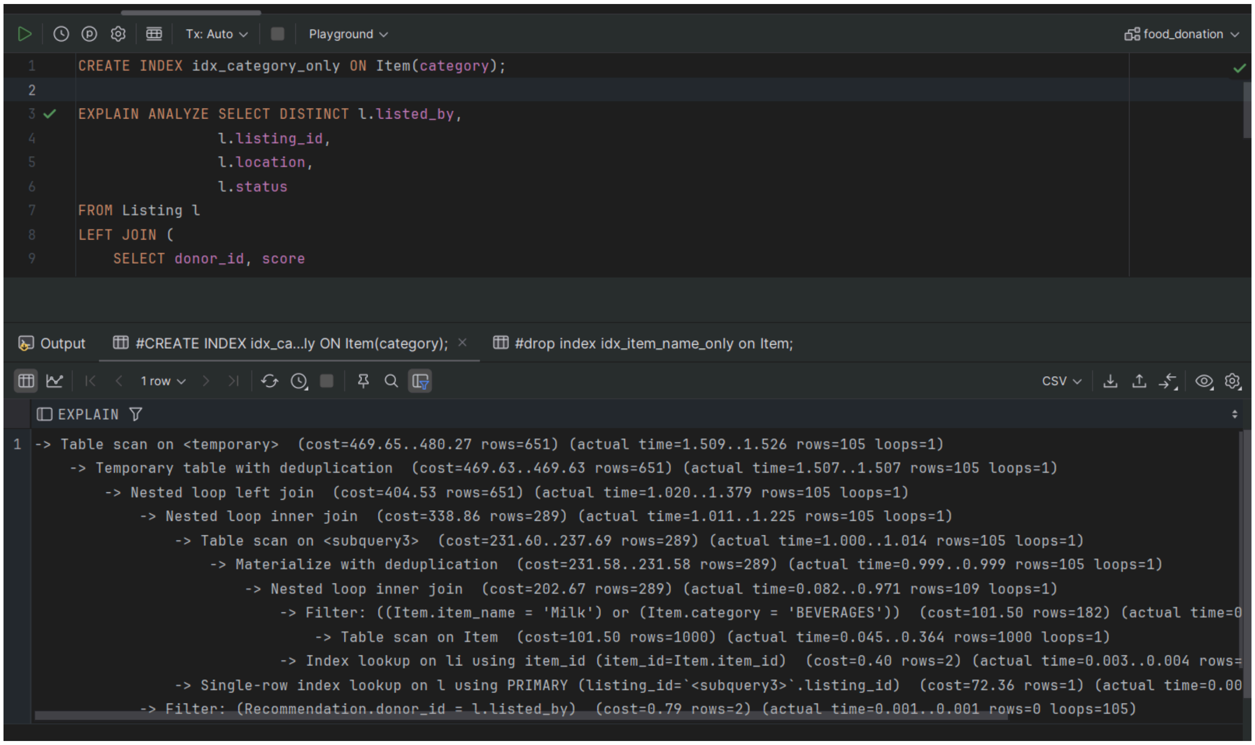


```sql
CREATE INDEX idx_item_name_only ON Item(item_name);

EXPLAIN ANALYZE SELECT DISTINCT l.listed_by,
                l.listing_id,
                l.location,
                l.status
FROM Listing l
LEFT JOIN (
    SELECT donor_id, score
```

```
-> Table scan on <temporary>  (cost=309.85..316.90 rows=366) (actual time=1.562..1.576 rows=105 loops=1)
    -> Temporary table with deduplication  (cost=309.83..309.83 rows=366) (actual time=1.560..1.560 rows=105 loops=1)
        -> Nested loop left join  (cost=273.22 rows=366) (actual time=1.063..1.429 rows=105 loops=1)
            -> Nested loop inner join  (cost=236.05 rows=163) (actual time=1.054..1.301 rows=105 loops=1)
                -> Table scan on <subquery3>  (cost=174.67..179.16 rows=163) (actual time=1.041..1.078 rows=105 loops=1)
                    -> Materialize with deduplication  (cost=174.64..174.64 rows=163) (actual time=1.040..1.040 rows=105 loops=1)
                        -> Nested loop inner join  (cost=158.39 rows=163) (actual time=0.101..1.009 rows=109 loops=1)
                            -> Filter: ((Item.item_name = 'Milk') or (Item.category = 'BEVERAGES'))  (cost=101.50 rows=102) (actual time=0
                                -> Table scan on Item  (cost=101.50 rows=1000) (actual time=0.042..0.377 rows=1000 loops=1)
                            -> Index lookup on li using item_id (item_id=Item.item_id)  (cost=0.40 rows=2) (actual time=0.003..0.004 rows=
                -> Single-row index lookup on l using PRIMARY (listing_id=`<subquery3>`.listing_id)  (cost=40.73 rows=1) (actual time=0.00
            -> Filter: (Recommendation.donor_id = l.listed_by)  (cost=0.79 rows=2) (actual time=0.001..0.001 rows=0 loops=105)
```

The overall cost of the query dropped from 486.07 (no index) to 309.85 (with index)

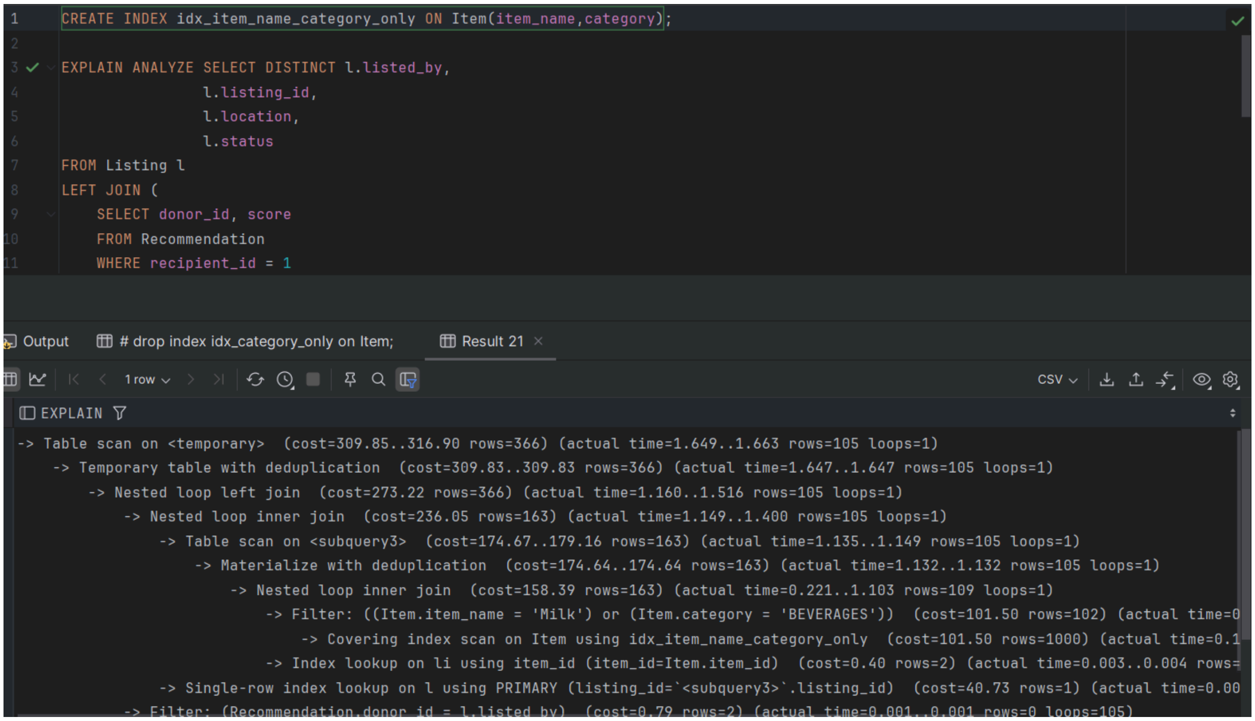
2) <u>Indexing on Item.category only</u>: We see a cost drop from 486.07 (no index) to 469.65


Tx: Auto
Playground
food_donation
Output
#CREATE INDEX idx_ca...ly ON Item(category);
#drop index idx_item_name_only on Item;
1 row
CSV
EXPLAIN

```sql
CREATE INDEX idx_category_only ON Item(category);

EXPLAIN ANALYZE SELECT DISTINCT l.listed_by,
                l.listing_id,
                l.location,
                l.status
FROM Listing l
LEFT JOIN (
    SELECT donor_id, score
```

```
-> Table scan on <temporary>  (cost=469.65..480.27 rows=651) (actual time=1.509..1.526 rows=105 loops=1)
    -> Temporary table with deduplication  (cost=469.63..469.63 rows=651) (actual time=1.507..1.507 rows=105 loops=1)
        -> Nested loop left join  (cost=404.53 rows=651) (actual time=1.020..1.379 rows=105 loops=1)
            -> Nested loop inner join  (cost=338.86 rows=289) (actual time=1.011..1.225 rows=105 loops=1)
                -> Table scan on <subquery3>  (cost=231.60..237.69 rows=289) (actual time=1.000..1.014 rows=105 loops=1)
                    -> Materialize with deduplication  (cost=231.58..231.58 rows=289) (actual time=0.999..0.999 rows=105 loops=1)
                        -> Nested loop inner join  (cost=202.67 rows=289) (actual time=0.082..0.971 rows=109 loops=1)
                            -> Filter: ((Item.item_name = 'Milk') or (Item.category = 'BEVERAGES'))  (cost=101.50 rows=182) (actual time=0
                                -> Table scan on Item  (cost=101.50 rows=1000) (actual time=0.045..0.364 rows=1000 loops=1)
                            -> Index lookup on li using item_id (item_id=Item.item_id)  (cost=0.40 rows=2) (actual time=0.003..0.004 rows=
                -> Single-row index lookup on l using PRIMARY (listing_id=`<subquery3>`.listing_id)  (cost=72.36 rows=1) (actual time=0.00
            -> Filter: (Recommendation.donor_id = l.listed_by)  (cost=0.79 rows=2) (actual time=0.001..0.001 rows=0 loops=105)
```

3) Composite index (item_name, category) on item:



```
CREATE INDEX idx_item_name_category_only ON Item(item_name,category);

EXPLAIN ANALYZE SELECT DISTINCT l.listed_by,
                l.listing_id,
                l.location,
                l.status
FROM Listing l
LEFT JOIN (
    SELECT donor_id, score
    FROM Recommendation
    WHERE recipient_id = 1
```

```
-> Table scan on <temporary>  (cost=309.85..316.90 rows=366) (actual time=1.649..1.663 rows=105 loops=1)
    -> Temporary table with deduplication  (cost=309.83..309.83 rows=366) (actual time=1.647..1.647 rows=105 loops=1)
        -> Nested loop left join  (cost=273.22 rows=366) (actual time=1.160..1.516 rows=105 loops=1)
            -> Nested loop inner join  (cost=236.05 rows=163) (actual time=1.149..1.400 rows=105 loops=1)
                -> Table scan on <subquery3>  (cost=174.67..179.16 rows=163) (actual time=1.135..1.149 rows=105 loops=1)
                    -> Materialize with deduplication  (cost=174.64..174.64 rows=163) (actual time=1.132..1.132 rows=105 loops=1)
                        -> Nested loop inner join  (cost=158.39 rows=163) (actual time=0.221..1.103 rows=109 loops=1)
                            -> Filter: ((Item.item_name = 'Milk') or (Item.category = 'BEVERAGES'))  (cost=101.50 rows=102) (actual time=0
                                -> Covering index scan on Item using idx_item_name_category_only  (cost=101.50 rows=1000) (actual time=0.1
                            -> Index lookup on li using item_id (item_id=Item.item_id)  (cost=0.40 rows=2) (actual time=0.003..0.004 rows=
                -> Single-row index lookup on l using PRIMARY (listing_id=`<subquery3>`.listing_id)  (cost=40.73 rows=1) (actual time=0.00
            -> Filter: (Recommendation.donor_id = l.listed_by)  (cost=0.79 rows=2) (actual time=0.001..0.001 rows=0 loops=105)
```

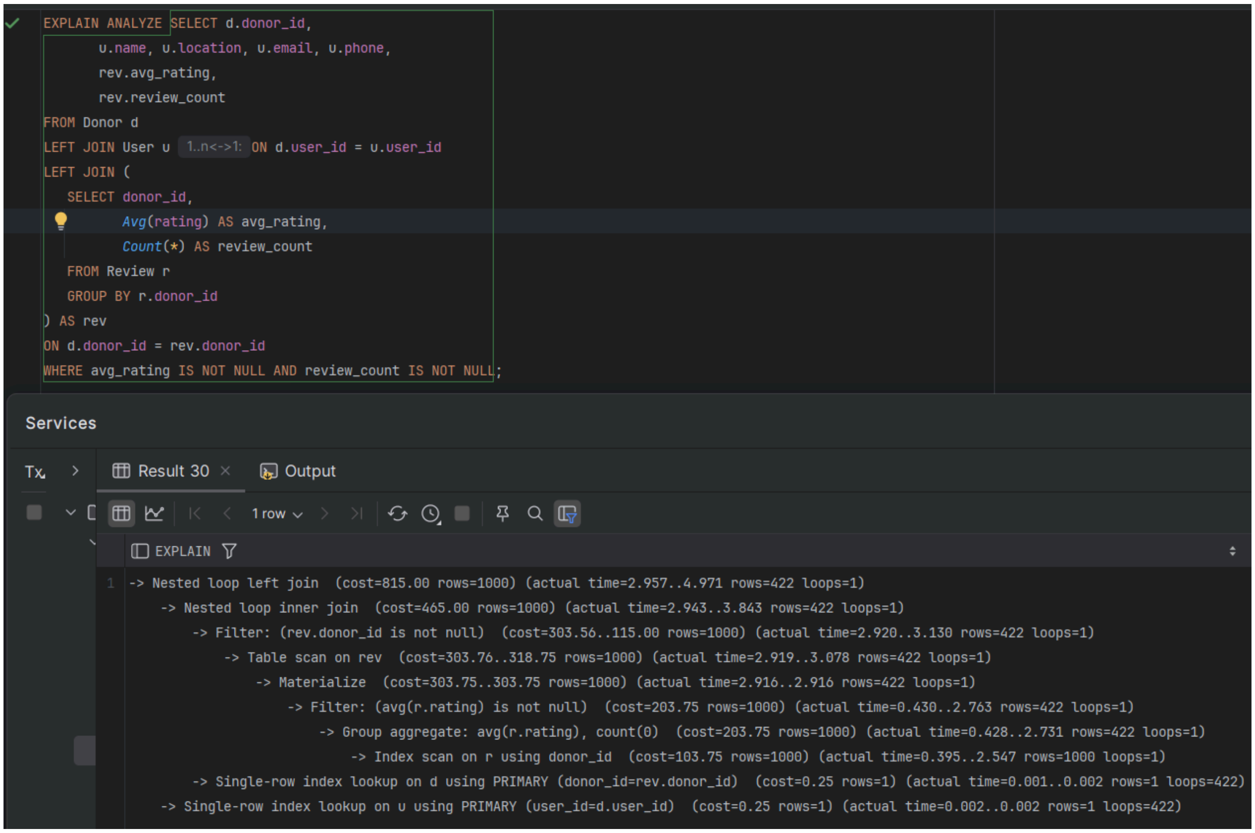
With the Composite Index, we observed some drop in the query cost from 486.07 to 309.85

Analysis :

In short, we can infer that having an index only on item_name provides the same results in cost compared to having a composite index of item_name with category. This can be because the cardinality of item_name is much more than category. Hence, more rows are filtered out while using an index on item_name.

AQ3: Indexing Design and Analysis

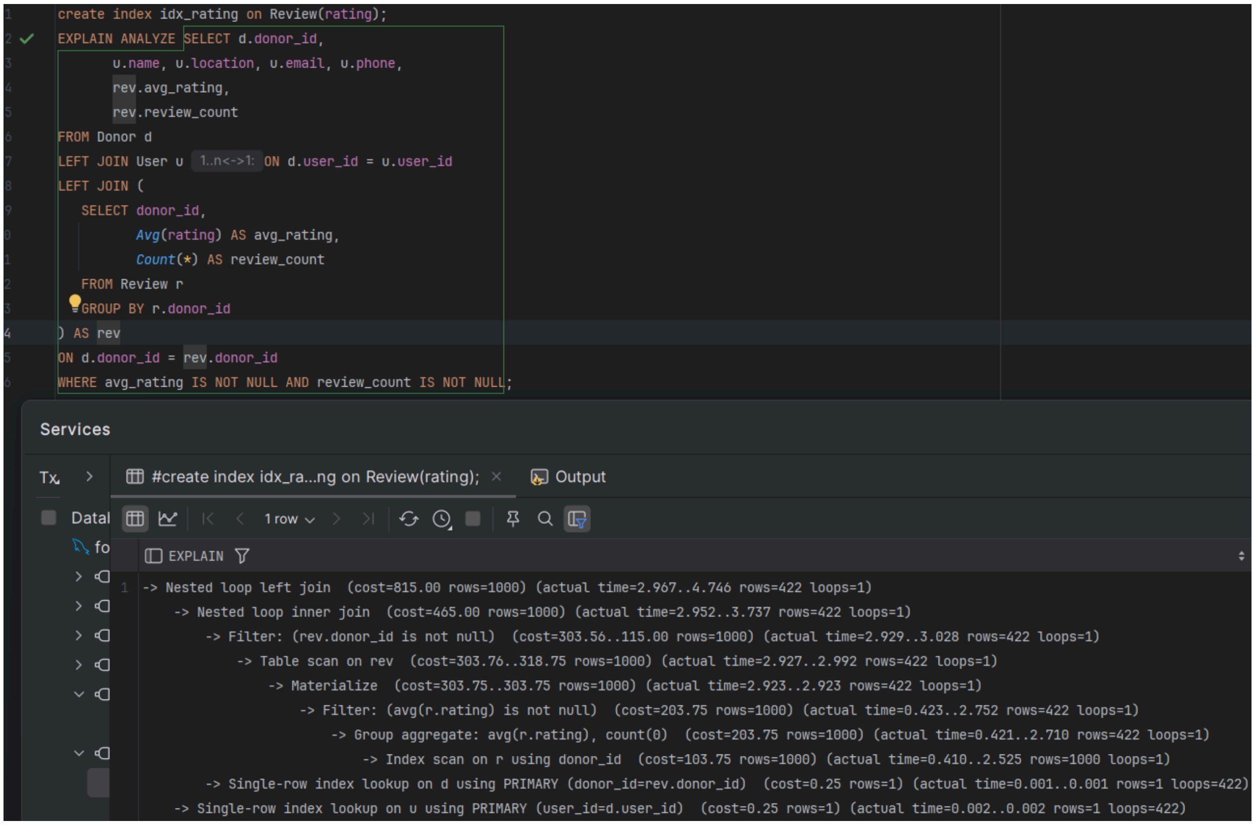
Cost of AQ3 with EXPLAIN ANALYZE before indexing:



```
EXPLAIN ANALYZE SELECT d.donor_id,
       u.name, u.location, u.email, u.phone,
       rev.avg_rating,
       rev.review_count
FROM Donor d
LEFT JOIN User u  1..n<->1:  ON d.user_id = u.user_id
LEFT JOIN (
   SELECT donor_id,
          Avg(rating) AS avg_rating,
          Count(*) AS review_count
   FROM Review r
   GROUP BY r.donor_id
) AS rev
ON d.donor_id = rev.donor_id
WHERE avg_rating IS NOT NULL AND review_count IS NOT NULL;
```

```
-> Nested loop left join  (cost=815.00 rows=1000) (actual time=2.957..4.971 rows=422 loops=1)
    -> Nested loop inner join  (cost=465.00 rows=1000) (actual time=2.943..3.843 rows=422 loops=1)
        -> Filter: (rev.donor_id is not null)  (cost=303.56..115.00 rows=1000) (actual time=2.920..3.130 rows=422 loops=1)
            -> Table scan on rev  (cost=303.76..318.75 rows=1000) (actual time=2.919..3.078 rows=422 loops=1)
                -> Materialize  (cost=303.75..303.75 rows=1000) (actual time=2.916..2.916 rows=422 loops=1)
                    -> Filter: (avg(r.rating) is not null)  (cost=203.75 rows=1000) (actual time=0.430..2.763 rows=422 loops=1)
                        -> Group aggregate: avg(r.rating), count(0)  (cost=203.75 rows=1000) (actual time=0.428..2.731 rows=422 loops=1)
                            -> Index scan on r using donor_id  (cost=103.75 rows=1000) (actual time=0.395..2.547 rows=1000 loops=1)
        -> Single-row index lookup on d using PRIMARY (donor_id=rev.donor_id)  (cost=0.25 rows=1) (actual time=0.001..0.002 rows=1 loops=422)
    -> Single-row index lookup on u using PRIMARY (user_id=d.user_id)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=422)
```

1. <u>Creating an index on rating:</u>



```sql
create index idx_rating on Review(rating);
EXPLAIN ANALYZE SELECT d.donor_id,
       u.name, u.location, u.email, u.phone,
       rev.avg_rating,
       rev.review_count
FROM Donor d
LEFT JOIN User u ON d.user_id = u.user_id
LEFT JOIN (
   SELECT donor_id,
          Avg(rating) AS avg_rating,
          Count(*) AS review_count
   FROM Review r
   GROUP BY r.donor_id
) AS rev
ON d.donor_id = rev.donor_id
WHERE avg_rating IS NOT NULL AND review_count IS NOT NULL;
```

```
-> Nested loop left join  (cost=815.00 rows=1000) (actual time=2.967..4.746 rows=422 loops=1)
    -> Nested loop inner join  (cost=465.00 rows=1000) (actual time=2.952..3.737 rows=422 loops=1)
        -> Filter: (rev.donor_id is not null)  (cost=303.56..115.00 rows=1000) (actual time=2.929..3.028 rows=422 loops=1)
            -> Table scan on rev  (cost=303.76..318.75 rows=1000) (actual time=2.927..2.992 rows=422 loops=1)
                -> Materialize  (cost=303.75..303.75 rows=1000) (actual time=2.923..2.923 rows=422 loops=1)
                    -> Filter: (avg(r.rating) is not null)  (cost=203.75 rows=1000) (actual time=0.423..2.752 rows=422 loops=1)
                        -> Group aggregate: avg(r.rating), count(0)  (cost=203.75 rows=1000) (actual time=0.421..2.710 rows=422 loops=1)
                            -> Index scan on r using donor_id  (cost=103.75 rows=1000) (actual time=0.410..2.525 rows=1000 loops=1)
        -> Single-row index lookup on d using PRIMARY (donor_id=rev.donor_id)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=422)
    -> Single-row index lookup on u using PRIMARY (user_id=d.user_id)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=422)
```

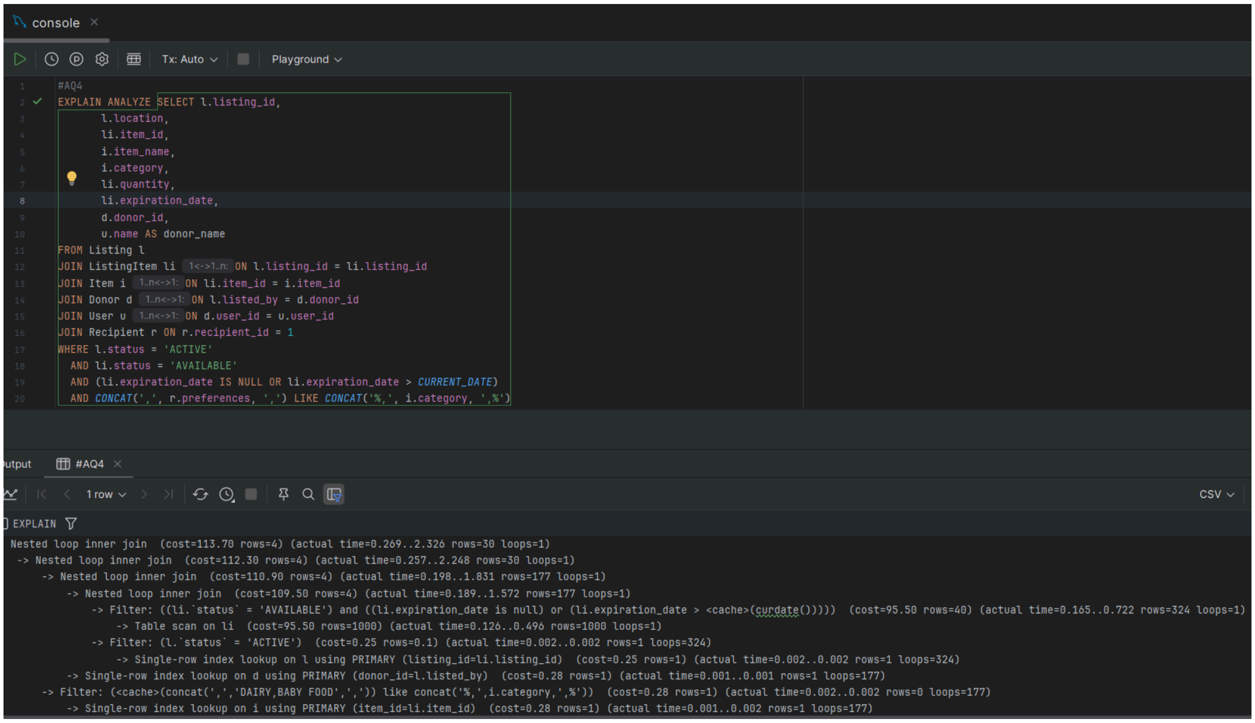
After creating an index on Rating, the overall cost of the query remains the same.

Analysis/Reason for no change in cost:
This result is likely due to the fact that the main filtering and join conditions in this query do not directly leverage the **review.rating** index. Specifically, the query retrieves aggregated data on **avg_rating** and **review_count** at the donor level, which involves grouping on **donor_id** rather than filtering by individual rating values. Consequently, the database engine processes the aggregation independently of the **review.rating** index, rendering it less effective for optimizing this query.

Note: The above query **only** has a **rating** field on which the indexing can be applied on. Other columns(**donor_id, user_id** etc) are already foreign or primary keys, hence we can't index on those. Indexing on other select columns like name, location also may not have any impact since they are not part of WHERE, GROUP BY clause.

AQ4: Indexing Design and Analysis

Cost of AQ4 with EXPLAIN ANALYZE before indexing:

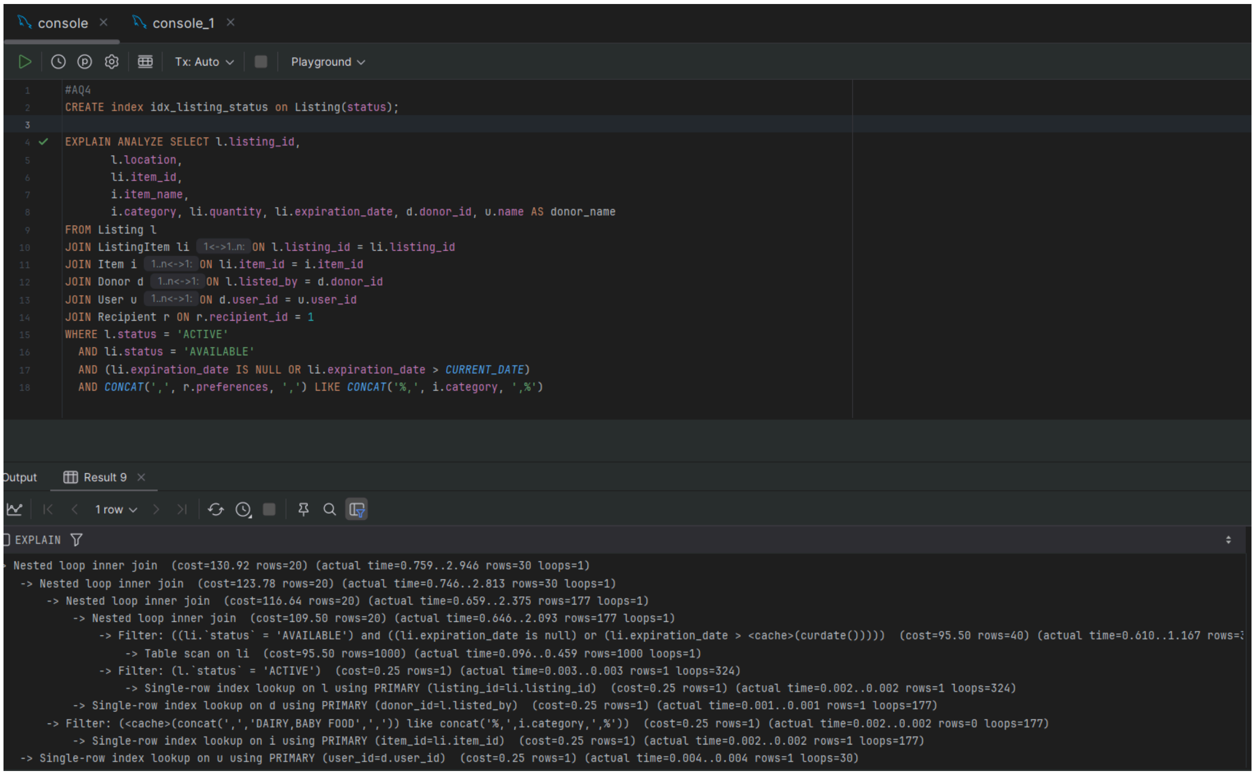


```
#AQ4
EXPLAIN ANALYZE SELECT l.listing_id,
       l.location,
       li.item_id,
       i.item_name,
       i.category,
       li.quantity,
       li.expiration_date,
       d.donor_id,
       u.name AS donor_name
FROM Listing l
JOIN ListingItem li ON l.listing_id = li.listing_id
JOIN Item i ON li.item_id = i.item_id
JOIN Donor d ON l.listed_by = d.donor_id
JOIN User u ON d.user_id = u.user_id
JOIN Recipient r ON r.recipient_id = 1
WHERE l.status = 'ACTIVE'
  AND li.status = 'AVAILABLE'
  AND (li.expiration_date IS NULL OR li.expiration_date > CURRENT_DATE)
  AND CONCAT(',', r.preferences, ',') LIKE CONCAT('%,', i.category, ',%')
```

```
EXPLAIN
Nested loop inner join  (cost=113.70 rows=4) (actual time=0.269..2.326 rows=30 loops=1)
 -> Nested loop inner join  (cost=112.30 rows=4) (actual time=0.257..2.248 rows=30 loops=1)
     -> Nested loop inner join  (cost=110.90 rows=4) (actual time=0.198..1.831 rows=177 loops=1)
         -> Nested loop inner join  (cost=109.50 rows=4) (actual time=0.189..1.572 rows=177 loops=1)
             -> Filter: ((li.`status` = 'AVAILABLE') and ((li.expiration_date is null) or (li.expiration_date > <cache>(curdate()))))  (cost=95.50 rows=40) (actual time=0.165..0.722 rows=324 loops=1)
                 -> Table scan on li  (cost=95.50 rows=1000) (actual time=0.126..0.496 rows=1000 loops=1)
             -> Filter: (l.`status` = 'ACTIVE')  (cost=0.25 rows=0.1) (actual time=0.002..0.002 rows=1 loops=324)
                 -> Single-row index lookup on l using PRIMARY (listing_id=li.listing_id)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=324)
         -> Single-row index lookup on d using PRIMARY (donor_id=l.listed_by)  (cost=0.28 rows=1) (actual time=0.001..0.001 rows=1 loops=177)
     -> Filter: (<cache>(concat(',','DAIRY,BABY FOOD',',')) like concat('%,',i.category,',%'))  (cost=0.28 rows=1) (actual time=0.002..0.002 rows=0 loops=177)
         -> Single-row index lookup on i using PRIMARY (item_id=li.item_id)  (cost=0.28 rows=1) (actual time=0.001..0.002 rows=1 loops=177)
```

Let's Explore the index designs for AQ4:

1) <u>Indexing on Listing.status Only</u>:

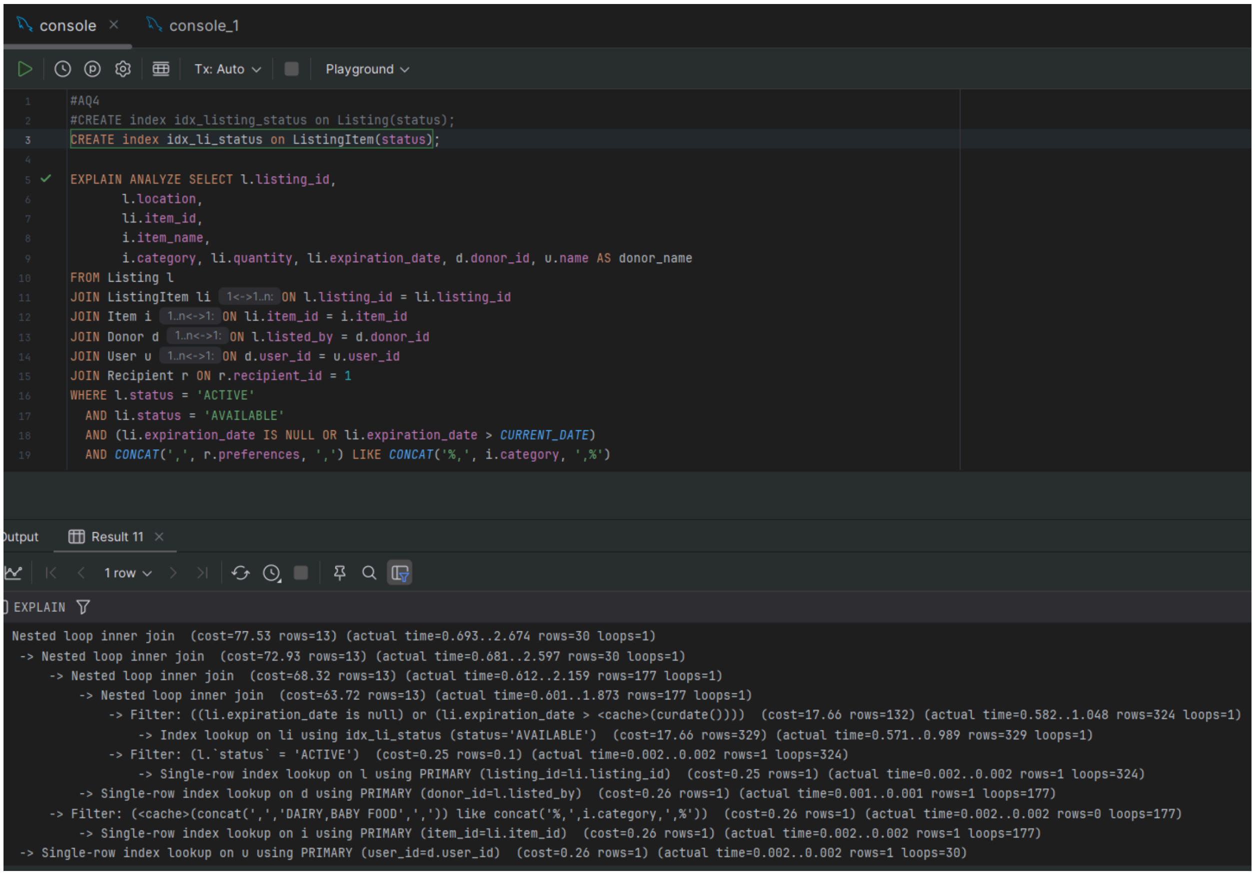


```
#AQ4
CREATE index idx_listing_status on Listing(status);

EXPLAIN ANALYZE SELECT l.listing_id,
       l.location,
       li.item_id,
       i.item_name,
       i.category, li.quantity, li.expiration_date, d.donor_id, u.name AS donor_name
FROM Listing l
JOIN ListingItem li ON l.listing_id = li.listing_id
JOIN Item i ON li.item_id = i.item_id
JOIN Donor d ON l.listed_by = d.donor_id
JOIN User u ON d.user_id = u.user_id
JOIN Recipient r ON r.recipient_id = 1
WHERE l.status = 'ACTIVE'
  AND li.status = 'AVAILABLE'
  AND (li.expiration_date IS NULL OR li.expiration_date > CURRENT_DATE)
  AND CONCAT(',', r.preferences, ',') LIKE CONCAT('%,', i.category, ',%')
```

```
EXPLAIN
Nested loop inner join  (cost=130.92 rows=20) (actual time=0.759..2.946 rows=30 loops=1)
 -> Nested loop inner join  (cost=123.78 rows=20) (actual time=0.746..2.813 rows=30 loops=1)
     -> Nested loop inner join  (cost=116.64 rows=20) (actual time=0.659..2.375 rows=177 loops=1)
         -> Nested loop inner join  (cost=109.50 rows=20) (actual time=0.646..2.093 rows=177 loops=1)
             -> Filter: ((li.`status` = 'AVAILABLE') and ((li.expiration_date is null) or (li.expiration_date > <cache>(curdate()))))  (cost=95.50 rows=40) (actual time=0.610..1.167 rows=
                 -> Table scan on li  (cost=95.50 rows=1000) (actual time=0.096..0.459 rows=1000 loops=1)
             -> Filter: (l.`status` = 'ACTIVE')  (cost=0.25 rows=1) (actual time=0.003..0.003 rows=1 loops=324)
                 -> Single-row index lookup on l using PRIMARY (listing_id=li.listing_id)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=324)
         -> Single-row index lookup on d using PRIMARY (donor_id=l.listed_by)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=177)
     -> Filter: (<cache>(concat(',','DAIRY,BABY FOOD',',')) like concat('%,',i.category,',%'))  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=0 loops=177)
         -> Single-row index lookup on i using PRIMARY (item_id=li.item_id)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=177)
 -> Single-row index lookup on u using PRIMARY (user_id=d.user_id)  (cost=0.25 rows=1) (actual time=0.004..0.004 rows=1 loops=30)
```

Creating an index on listing.status **increased** the query cost from 113.70 (no index) to 130.92. Adding an index on **Listing.status** increased the query cost. This is likely because the filtering condition WHERE **l.status = 'ACTIVE'** in the query has low selectivity, meaning that many records in Listing meet this condition. Since the status is not highly selective, the database engine may still need to scan a large portion of Listing records, making the index less effective.

2) <u>Indexing on ListingItem.status only:</u>

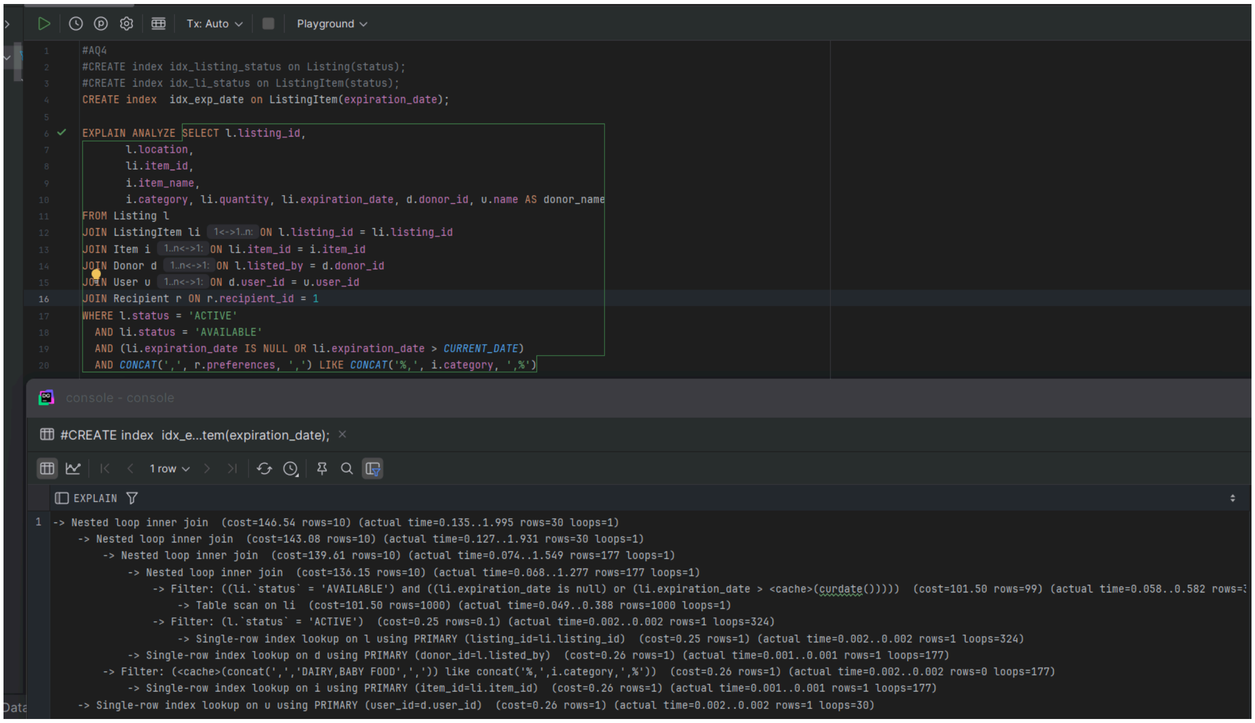


```sql
#AQ4
#CREATE index idx_listing_status on Listing(status);
CREATE index idx_li_status on ListingItem(status);

EXPLAIN ANALYZE SELECT l.listing_id,
       l.location,
       li.item_id,
       i.item_name,
       i.category, li.quantity, li.expiration_date, d.donor_id, u.name AS donor_name
FROM Listing l
JOIN ListingItem li 1<->1..n: ON l.listing_id = li.listing_id
JOIN Item i 1..n<->1: ON li.item_id = i.item_id
JOIN Donor d 1..n<->1: ON l.listed_by = d.donor_id
JOIN User u 1..n<->1: ON d.user_id = u.user_id
JOIN Recipient r ON r.recipient_id = 1
WHERE l.status = 'ACTIVE'
  AND li.status = 'AVAILABLE'
  AND (li.expiration_date IS NULL OR li.expiration_date > CURRENT_DATE)
  AND CONCAT(',', r.preferences, ',') LIKE CONCAT('%,', i.category, ',%')
```

```
Nested loop inner join  (cost=77.53 rows=13) (actual time=0.693..2.674 rows=30 loops=1)
 -> Nested loop inner join  (cost=72.93 rows=13) (actual time=0.681..2.597 rows=30 loops=1)
     -> Nested loop inner join  (cost=68.32 rows=13) (actual time=0.612..2.159 rows=177 loops=1)
         -> Nested loop inner join  (cost=63.72 rows=13) (actual time=0.601..1.873 rows=177 loops=1)
             -> Filter: ((li.expiration_date is null) or (li.expiration_date > <cache>(curdate())))  (cost=17.66 rows=132) (actual time=0.582..1.048 rows=324 loops=1)
                 -> Index lookup on li using idx_li_status (status='AVAILABLE')  (cost=17.66 rows=329) (actual time=0.571..0.989 rows=329 loops=1)
             -> Filter: (l.`status` = 'ACTIVE')  (cost=0.25 rows=0.1) (actual time=0.002..0.002 rows=1 loops=324)
                 -> Single-row index lookup on l using PRIMARY (listing_id=li.listing_id)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=324)
         -> Single-row index lookup on d using PRIMARY (donor_id=l.listed_by)  (cost=0.26 rows=1) (actual time=0.001..0.001 rows=1 loops=177)
     -> Filter: (<cache>(concat(',','DAIRY,BABY FOOD',',')) like concat('%,',i.category,',%'))  (cost=0.26 rows=1) (actual time=0.002..0.002 rows=0 loops=177)
         -> Single-row index lookup on i using PRIMARY (item_id=li.item_id)  (cost=0.26 rows=1) (actual time=0.002..0.002 rows=1 loops=177)
 -> Single-row index lookup on u using PRIMARY (user_id=d.user_id)  (cost=0.26 rows=1) (actual time=0.002..0.002 rows=1 loops=30)
```

Creating an index on ListingItem.status **substantially dropped** the overall cost from 113.70 (no index) to 77.53. This means creating an index on **ListingItem.status** provided a significant performance benefit, as WHERE **li.status = 'AVAILABLE'** is likely a selective filter in our query. This index allows the database to quickly locate only the AVAILABLE items within ListingItem, reducing the number of rows that need to be joined with other tables.

3) <u>Indexing on ListingItem.expiration_date:</u>



```sql
#AQ4
#CREATE index idx_listing_status on Listing(status);
#CREATE index idx_li_status on ListingItem(status);
CREATE index  idx_exp_date on ListingItem(expiration_date);

EXPLAIN ANALYZE SELECT l.listing_id,
       l.location,
       li.item_id,
       i.item_name,
       i.category, li.quantity, li.expiration_date, d.donor_id, u.name AS donor_name
FROM Listing l
JOIN ListingItem li  1<->1..n: ON l.listing_id = li.listing_id
JOIN Item i  1..n<->1: ON li.item_id = i.item_id
JOIN Donor d  1..n<->1: ON l.listed_by = d.donor_id
JOIN User u  1..n<->1: ON d.user_id = u.user_id
JOIN Recipient r ON r.recipient_id = 1
WHERE l.status = 'ACTIVE'
  AND li.status = 'AVAILABLE'
  AND (li.expiration_date IS NULL OR li.expiration_date > CURRENT_DATE)
  AND CONCAT(',', r.preferences, ',') LIKE CONCAT('%,', i.category, ',%')
```

```
1 -> Nested loop inner join  (cost=146.54 rows=10) (actual time=0.135..1.995 rows=30 loops=1)
    -> Nested loop inner join  (cost=143.08 rows=10) (actual time=0.127..1.931 rows=30 loops=1)
        -> Nested loop inner join  (cost=139.61 rows=10) (actual time=0.074..1.549 rows=177 loops=1)
            -> Nested loop inner join  (cost=136.15 rows=10) (actual time=0.068..1.277 rows=177 loops=1)
                -> Filter: ((li.`status` = 'AVAILABLE') and ((li.expiration_date is null) or (li.expiration_date > <cache>(curdate()))))  (cost=101.50 rows=99) (actual time=0.058..0.582 rows=3
                    -> Table scan on li  (cost=101.50 rows=1000) (actual time=0.049..0.388 rows=1000 loops=1)
                -> Filter: (l.`status` = 'ACTIVE')  (cost=0.25 rows=0.1) (actual time=0.002..0.002 rows=1 loops=324)
                    -> Single-row index lookup on l using PRIMARY (listing_id=li.listing_id)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=324)
            -> Single-row index lookup on d using PRIMARY (donor_id=l.listed_by)  (cost=0.26 rows=1) (actual time=0.001..0.001 rows=1 loops=177)
        -> Filter: (<cache>(concat(',','DAIRY,BABY FOOD',',')) like concat('%,',i.category,',%'))  (cost=0.26 rows=1) (actual time=0.002..0.002 rows=0 loops=177)
            -> Single-row index lookup on i using PRIMARY (item_id=li.item_id)  (cost=0.26 rows=1) (actual time=0.001..0.001 rows=1 loops=177)
    -> Single-row index lookup on u using PRIMARY (user_id=d.user_id)  (cost=0.26 rows=1) (actual time=0.002..0.002 rows=1 loops=30)
```

Indexing on ListingItem.expiration_date **increased** the overall cost from 113.70 to 146.54. This is likely because the condition (**li.expiration_date IS NULL OR li.expiration_date > CURRENT_DATE**) requires a complex filter, where the index isn't efficiently utilized due to the need to check for both **NULL** values and dates greater than the current date. The **OR** condition complicates index usage since it doesn't support range scanning efficiently.

Analysis:
In short, we can conclude that the index design having index on `**ListingItem.status**` works best compared to other index design approaches experimented.

Optimization Summary

Best index designs are selected based on the analysis given in the Indexing section.

Query	Best designed Index selected	EXPLAIN ANALYZE Overall Cost		Cost Optimization
		Before Indexing	After Indexing	
AQ1	ListingItem.status	5.44	5.44	No difference
AQ2	Item.item_name	486.07	309.85	Improved
AQ3	Review.rating	815.00	815.00	No difference
AQ4	ListingItem.status	113.70	77.53	Improved