CS411 PT1 Stage3 - Database Implementation and Indexing

Matt Straczek, Peyton Murray, Qi Yu, Hao Qi

Mar 7, 2023

1. Database Implementation

Our team has implemented our databases on GCP, and we connected GCP using MySQLWorkbench on our local machine for development. Figure 1 shows a screenshot of the connection on GCP.

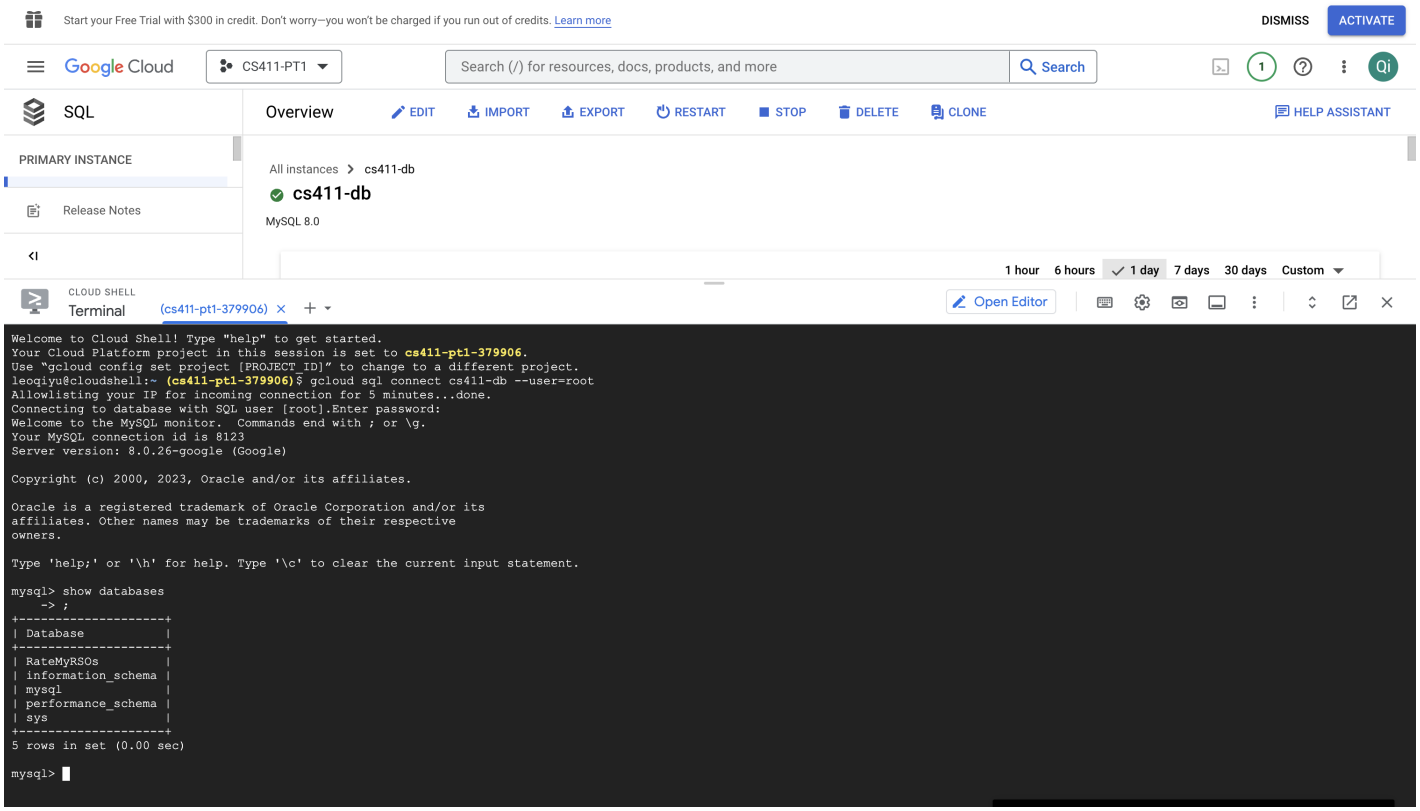


Figure 1: Connection on GCP

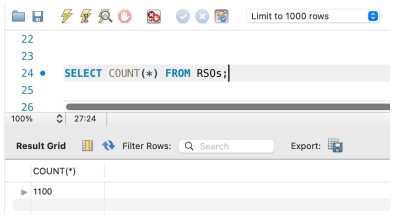
Below is the DDL commands we used for creating our tables

```
CREATE DATABASE IF NOT EXISTS RateMyRSOs;
USE RateMyRSOs;

DROP TABLE IF EXISTS Reports;
DROP TABLE IF EXISTS Memberships;
DROP TABLE IF EXISTS Affiliations;
DROP TABLE IF EXISTS Managements;
DROP TABLE IF EXISTS Reviews;
DROP TABLE IF EXISTS RSOs;
DROP TABLE IF EXISTS Officers;
DROP TABLE IF EXISTS Departments;
DROP TABLE IF EXISTS Users;
```

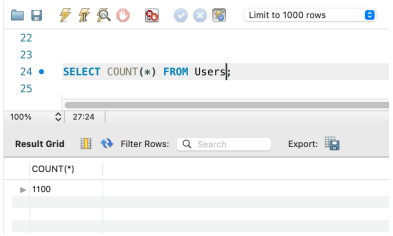
```
CREATE TABLE IF NOT EXISTS RSOs (
    RSOId INT UNIQUE NOT NULL,
    RSOName VARCHAR(255) NOT NULL,
    ContactEmail VARCHAR(255),
    YearEstablished INT NOT NULL,
    Website VARCHAR(255),
    Facebook VARCHAR(255),
    Instagram VARCHAR(255),
    PRIMARY KEY (RSOId)
);

CREATE TABLE Officers (
    OfficerId INT UNIQUE NOT NULL,
    FirstName VARCHAR(255) NOT NULL,
    LastName VARCHAR(255) NOT NULL,
    ContactEmail VARCHAR(255),
    PRIMARY KEY (OfficerId)
);

CREATE TABLE Departments (
    DeptName VARCHAR(255) UNIQUE NOT NULL,
    Address VARCHAR(255),
    ContactPhone VARCHAR(30),
    ContactEmail VARCHAR(255),
    PRIMARY KEY (DeptName)
);

CREATE TABLE Users (
    Username VARCHAR(255) UNIQUE NOT NULL,
    UserPassword VARCHAR(30) NOT NULL,
    UserType VARCHAR(20) NOT NULL,
    FirstName VARCHAR(255) NOT NULL,
    LastName VARCHAR(255) NOT NULL,
    PRIMARY KEY (Username)
);

CREATE TABLE Reviews (
    ReviewId INT UNIQUE NOT NULL,
    Username VARCHAR(255) NOT NULL,
    RSOId INT NOT NULL,
    Content VARCHAR(1024) NOT NULL,
    Rating INT NOT NULL,
    NumLikes INT NOT NULL,
    PostDate DATE,
    PRIMARY KEY (ReviewId),
    CONSTRAINT reviews_users_fk_1 FOREIGN KEY (Username) REFERENCES Users (Username),
    CONSTRAINT reviews_rsos_fk_2 FOREIGN KEY (RSOId) REFERENCES RSOs (RSOId)
);

CREATE TABLE Managements (
    OfficerId INT NOT NULL,
    RSOId INT NOT NULL,
    PRIMARY KEY (OfficerId, RSOId),
    CONSTRAINT manages_officers_fk_1 FOREIGN KEY (OfficerId) REFERENCES Officers (OfficerId),
    CONSTRAINT manages_rsos_fk_2 FOREIGN KEY (RSOId) REFERENCES RSOs (RSOId)
);

CREATE TABLE Affiliations (
    RSOId INT NOT NULL,
    DeptName VARCHAR(255) NOT NULL,
```

```
    PRIMARY KEY (RSOId, DeptName),
    CONSTRAINT affliate_rsos_fk_1 FOREIGN KEY (RSOId) REFERENCES RSOs (RSOId),
    CONSTRAINT affliate_depts_fk_2 FOREIGN KEY (DeptName) REFERENCES Departments (DeptName)
);

CREATE TABLE Memberships (
    Username VARCHAR(255) NOT NULL,
    RSOId INT NOT NULL,
    PRIMARY KEY (Username, RSOId),
    CONSTRAINT members_users_fk_1 FOREIGN KEY (Username) REFERENCES Users (Username),
    CONSTRAINT members_rsos_fk_2 FOREIGN KEY (RSOId) REFERENCES RSOs (RSOId)
);

CREATE TABLE Reports (
    Username VARCHAR(255) NOT NULL,
    ReviewId INT NOT NULL,
    Content VARCHAR(1024),
    PRIMARY KEY (Username, ReviewId),
    CONSTRAINT reports_users_fk_1 FOREIGN KEY (Username) REFERENCES Users (Username),
    CONSTRAINT reports_reviews_fk_2 FOREIGN KEY (ReviewId) REFERENCES Reviews (ReviewId)
);
```

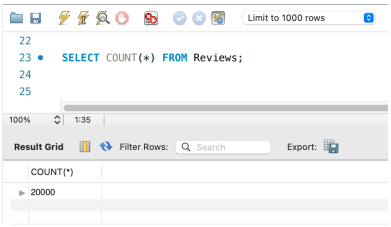
We have inserted more than 1000 rows for table RSOs, Users, Reviews, Managements, Affiliations, Memberships and Officers. Figure 2 shows the screenshots the result of count queries against these tables. The data we inserted are mostly randomly generated data using Python package Faker, except for the departments table which we web crawled from UIUC's official website.



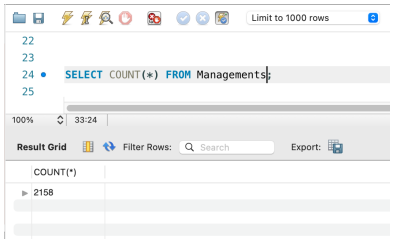
(a) RSOs count



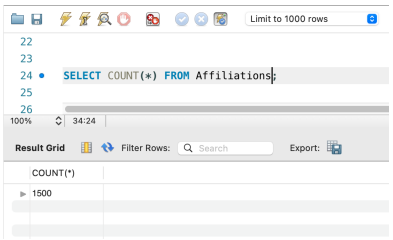
(b) Users count



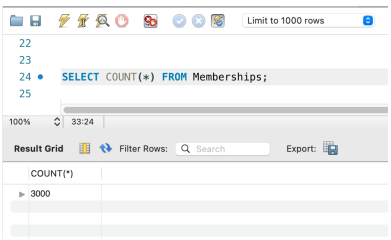
(c) Reviews count



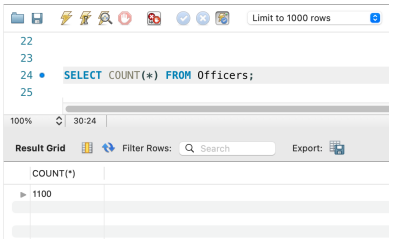
(d) Managements count



(e) Affiliation count



(f) Membership count



(g) Officers count

Figure 2: Number of rows inserted

We have developed two advanced queries in this stage. For the first one, we tried to find out the number of positive reviews that each RSO of good reputations receives. Here, we define positive reviews as the reviews that has an attached rating greater than or equal to 3, and we define RSOs of good reputations as the RSOs

that received at least one reviews starting with the word "Good", "Nice", "Great" or "Best". Finally, we order the result by the number of positive reviews in descending order. Below is the SQL query we used, and Figure 3 shows the result of this query.

```
SELECT RSOId, RSOName, COUNT(*) AS GoodReviewCnt
FROM RSOs NATURAL JOIN Reviews
WHERE Rating >= 3 AND RSOId IN (SELECT DISTINCT RSOId
                                FROM Reviews
                                WHERE Content LIKE "Good%" OR Content LIKE "Great%"
                                OR Content LIKE "Nice%" OR Content LIKE "Best%")
GROUP BY RSOId
ORDER BY GoodReviewCnt DESC
LIMIT 15;
```

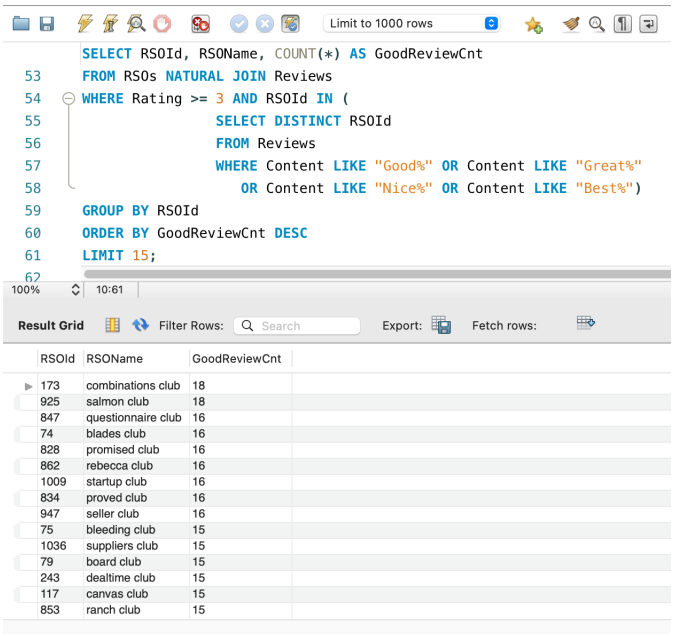
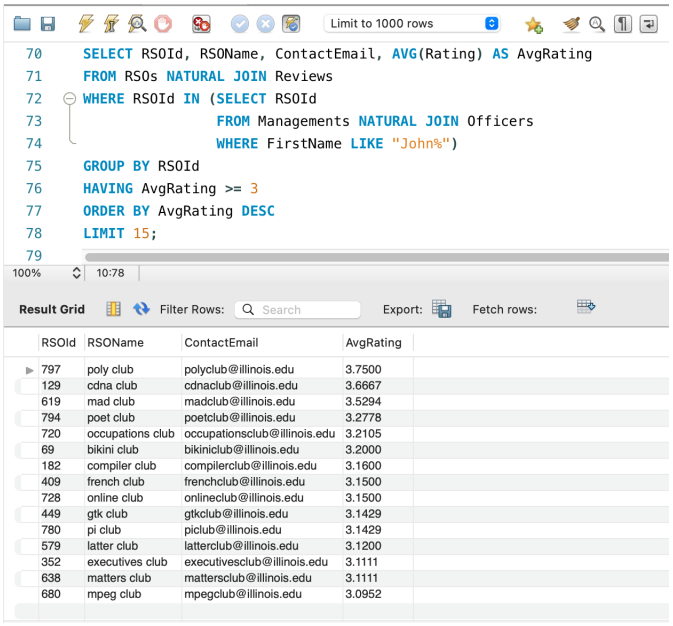


Figure 3: Result of the first advanced SQL query

For the second advanced queries, we tried to find out the RSOs that runs by an officer whose first name starts with John, and receives an average rating above 3. Specifically, we want to find out the RSOId, RSOName, the contact email and the average rating of the RSO, and we want the result to be sorted by the average rating in descending order. Below is the SQL query we used, and Figure 4 shows the result of this query.

```
SELECT RSOId, RSOName, ContactEmail, AVG(Rating) AS AvgRating
FROM RSOs NATURAL JOIN Reviews
WHERE RSOId IN (SELECT RSOId
                FROM Managements NATURAL JOIN Officers
                WHERE FirstName LIKE "John%")
GROUP BY RSOId
HAVING AvgRating >= 3
ORDER BY AvgRating DESC
LIMIT 15;
```



```
Limit to 1000 rows

70  SELECT RSOId, RSOName, ContactEmail, AVG(Rating) AS AvgRating
71  FROM RSOs NATURAL JOIN Reviews
72  WHERE RSOId IN (SELECT RSOId
73                  FROM Managements NATURAL JOIN Officers
74                  WHERE FirstName LIKE "John%")
75  GROUP BY RSOId
76  HAVING AvgRating >= 3
77  ORDER BY AvgRating DESC
78  LIMIT 15;
79
```

100% 10:78

Result Grid | Filter Rows: Search | Export: | Fetch rows:

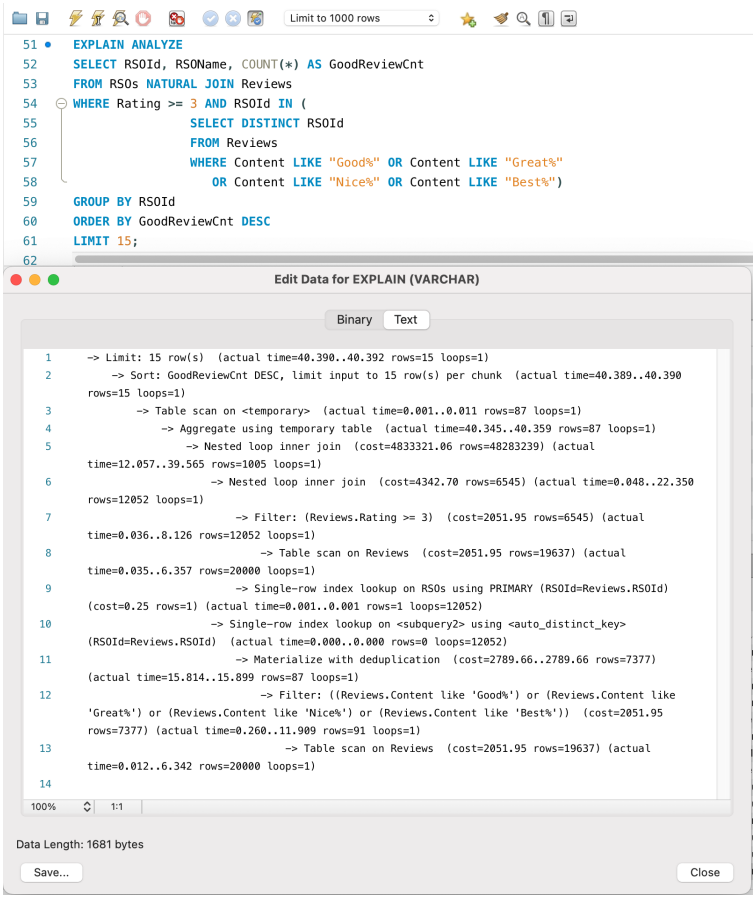
RSOId	RSOName	ContactEmail	AvgRating
797	poly club	polyclub@illinois.edu	3.7500
129	cdna club	cdnaclub@illinois.edu	3.6667
619	mad club	madclub@illinois.edu	3.5294
794	poet club	poetclub@illinois.edu	3.2778
720	occupations club	occupationsclub@illinois.edu	3.2105
69	bikini club	bikiniclub@illinois.edu	3.2000
182	compiler club	compilerclub@illinois.edu	3.1600
409	french club	frenchclub@illinois.edu	3.1500
728	online club	onlineclub@illinois.edu	3.1500
449	gtk club	gtkclub@illinois.edu	3.1429
780	pi club	piclub@illinois.edu	3.1429
579	latter club	latterclub@illinois.edu	3.1200
352	executives club	executivesclub@illinois.edu	3.1111
638	matters club	mattersclub@illinois.edu	3.1111
680	mpeg club	mpegclub@illinois.edu	3.0952

Figure 4: Result of the second advanced SQL query

2. Indexing

(a)

For the first advanced query, Figure 5 shows the result of the EXPLAIN ANALYZE commands before adding any explicit index into any tables.



```
Limit to 1000 rows

51  EXPLAIN ANALYZE
52  SELECT RSOId, RSOName, COUNT(*) AS GoodReviewCnt
53  FROM RSOs NATURAL JOIN Reviews
54  WHERE Rating >= 3 AND RSOId IN (
55                  SELECT DISTINCT RSOId
56                  FROM Reviews
57                  WHERE Content LIKE "Good%" OR Content LIKE "Great%"
58                     OR Content LIKE "Nice%" OR Content LIKE "Best%")
59  GROUP BY RSOId
60  ORDER BY GoodReviewCnt DESC
61  LIMIT 15;
62
```

Edit Data for EXPLAIN (VARCHAR)

Binary | Text

```
1  -> Limit: 15 row(s)  (actual time=40.390..40.392 rows=15 loops=1)
2      -> Sort: GoodReviewCnt DESC, limit input to 15 row(s) per chunk  (actual time=40.389..40.390
   rows=15 loops=1)
3          -> Table scan on <temporary>  (actual time=0.001..0.011 rows=87 loops=1)
4              -> Aggregate using temporary table  (actual time=40.345..40.359 rows=87 loops=1)
5                  -> Nested loop inner join  (cost=4833321.06 rows=48283239) (actual
   time=12.057..39.565 rows=1005 loops=1)
6                      -> Nested loop inner join  (cost=4342.70 rows=6545) (actual time=0.048..22.350
   rows=12052 loops=1)
7                          -> Filter: (Reviews.Rating >= 3)  (cost=2051.95 rows=6545) (actual
   time=0.036..8.126 rows=12052 loops=1)
8                              -> Table scan on Reviews  (cost=2051.95 rows=19637) (actual
   time=0.035..6.357 rows=20000 loops=1)
9                          -> Single-row index lookup on RSOs using PRIMARY (RSOId=Reviews.RSOId)
   (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=12052)
10                     -> Single-row index lookup on <subquery2> using <auto_distinct_key>
   (RSOId=Reviews.RSOId)  (actual time=0.000..0.000 rows=0 loops=12052)
11                         -> Materialize with deduplication  (cost=2789.66..2789.66 rows=7377)
   (actual time=15.814..15.899 rows=87 loops=1)
12                             -> Filter: ((Reviews.Content like 'Good%') or (Reviews.Content like
   'Great%') or (Reviews.Content like 'Nice%') or (Reviews.Content like 'Best%'))  (cost=2051.95
   rows=7377) (actual time=0.260..11.909 rows=91 loops=1)
13                                 -> Table scan on Reviews  (cost=2051.95 rows=19637) (actual
   time=0.012..6.342 rows=20000 loops=1)
14
```

100% 1:1

Data Length: 1681 bytes

Save... Close

Figure 5: First query: EXPLAIN ANALYZE before explicit indexing

We can see that the cost of the filter on table *Reviews* is 2051.95, when the filter trying to find reviews whose content starts with "Good", "Nice", "Great", or "Best". The actual time of this query is in [40.390, 40.392]. Now we add an index on the attributed *Content* in table *Reviews*

```
CREATE INDEX reviews_content ON Reviews(Content(10));
```

Now we use EXPLAIN ANALYZE to analyze the query's performance after indexing. Figure 6 shows the result of the EXPLAIN ANALYZE commands after adding index *reviews_content*

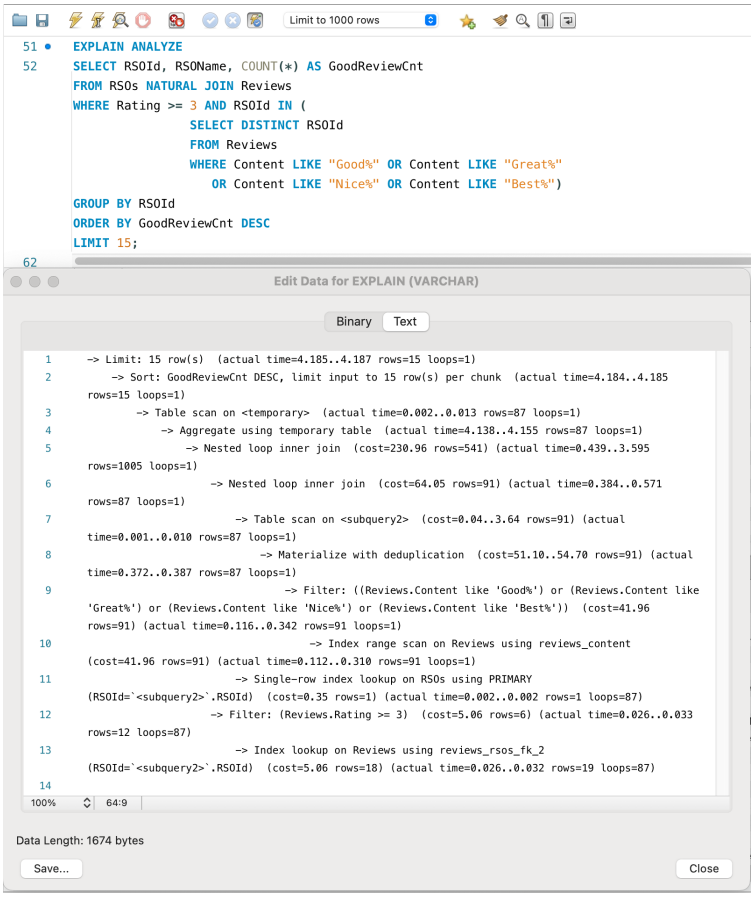


Figure 6: First query: EXPLAIN ANALYZE after index *reviews_content*

Now, we can see that the actual time of this query is in [4.185, 4.187], achieving almost 10x speedup. We can also find that the cost of the filter on table *Reviews* drop significantly to 41.96, which proves that the index *reviews_content* is actually working in speeding up the query.

Now we drop this index and try another one, which is on the attributed *Rating* in table *Reviews*

```
CREATE INDEX reviews_rating ON Reviews(Rating);
```

and use EXPLAIN ANALYZE to analyze the query's performance after indexing. Figure 7 shows the result of the EXPLAIN ANALYZE commands after adding index *reviews_rating*. We can see from the result that this indexing did not give any obvious performance boost. We can even find that for Filter: (Reviews.Rating ¿=3), the cost stays the same. This is probably due to the fact that the database has already has some fast searching algorithms for integer type values, and adding an explicit indexing on this attribute will not yield any speedup.

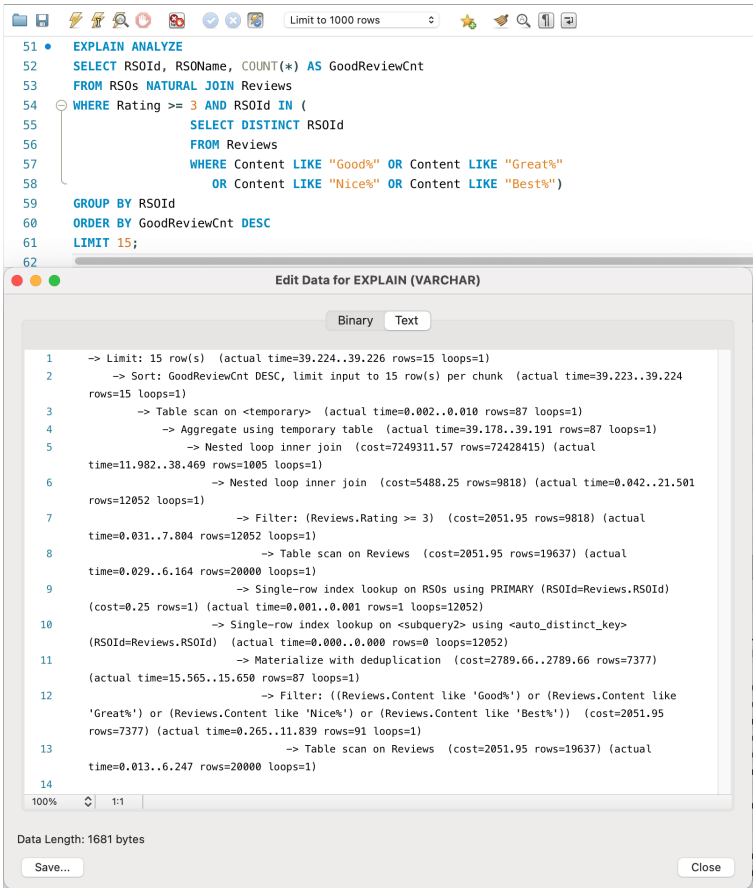


Figure 7: First query: EXPLAIN ANALYZE after index *reviews_rating*

Now we drop this index and try another one, which is on the attributed *PostDate* in table *Reviews*

```
CREATE INDEX reviews_postdate ON Reviews(PostDate);
```

and use EXPLAIN ANALYZE to analyze the query's performance after indexing. Figure 8 shows the result of the EXPLAIN ANALYZE commands after adding index *reviews_postdate*. Obviously, this index will not lead to any speedup for this query because attribute *PostDate* is not used in this query.

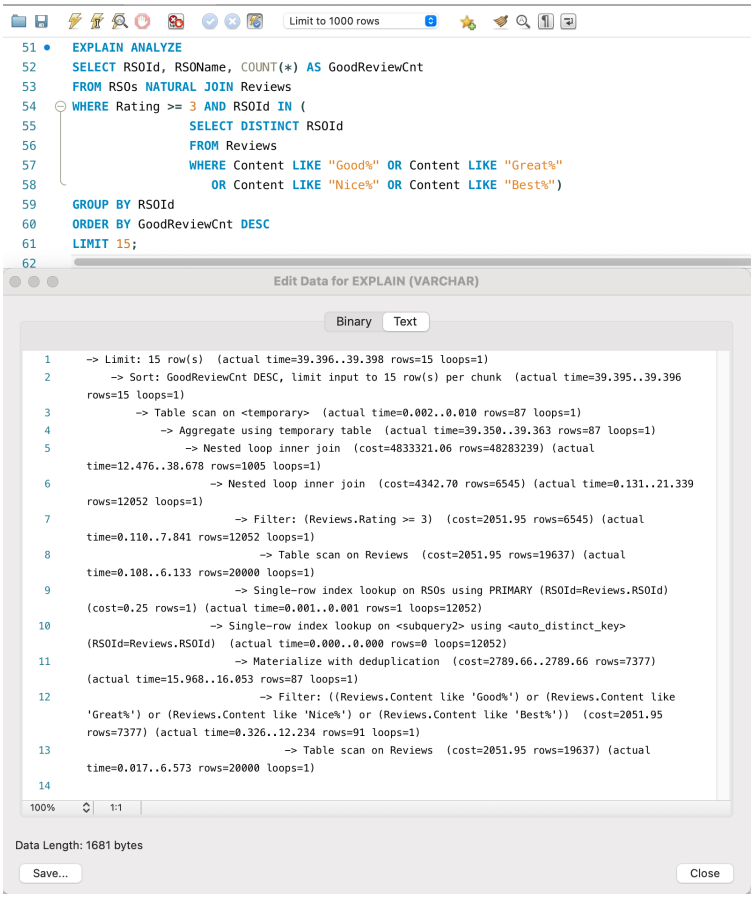


Figure 8: First query: EXPLAIN ANALYZE after index *reviews_postdate*

(b)

For the second advanced query, Figure 9 shows the result of the EXPLAIN ANALYZE commands before adding any explicit index into any tables.

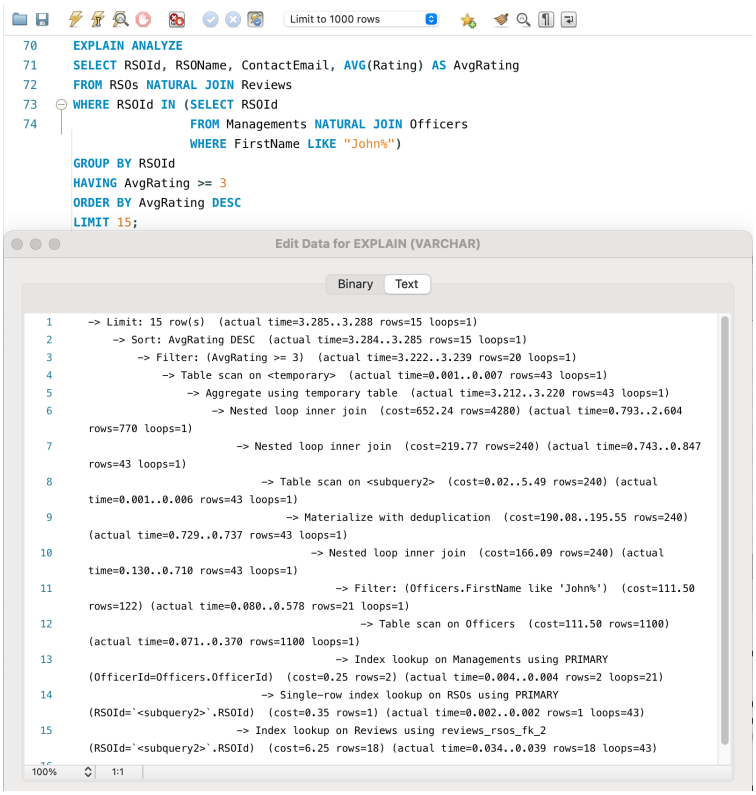


Figure 9: Second query: EXPLAIN ANALYZE before explicit indexing

We can see that the cost of the filter on table *Officers* is 111.50, when the filter trying to find officers whose first name starts with "John". The actual time of this query is in [3.285, 3.288]. Now we add an index on the attributed *FirstName* in table *Officers*

```
CREATE INDEX officers_firstname ON Officers(FirstName(5));
```

Now we use EXPLAIN ANALYZE to analyze the query's performance after indexing. Figure 10 shows the result of the EXPLAIN ANALYZE commands after adding index *officers_firstname*

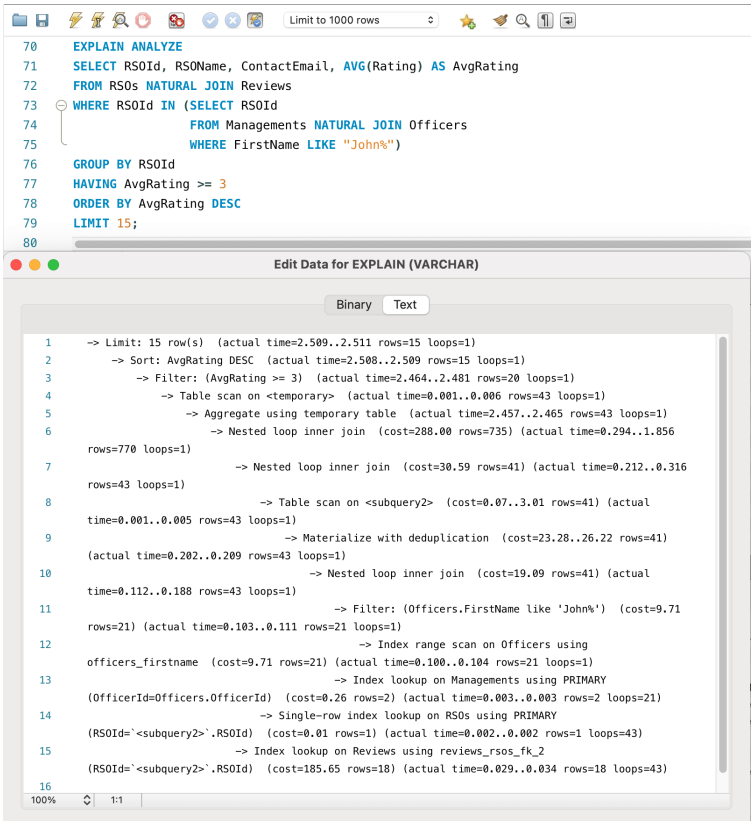


Figure 10: First query: EXPLAIN ANALYZE after index *officers_firstname*

Now, we can see that the actual time of this query is in [2.509, 2.511], achieving approximately a 1.5x speedup. We can also find that the cost of the filter on table *Reviews* drop significantly to 9.71, which proves that the index *reviews_content* is actually working in speeding up the query. However, this index does not give a very high performance boost regarding the actual run time, probably due to the fact the join operations now become the performance bottlenecks, and the speedup is saturated.

Now we drop this index and try another one, which is on the attributed *Rating* in table *Reviews*

```
CREATE INDEX reviews_rating ON Reviews(Rating);
```

and use EXPLAIN ANALYZE to analyze the query's performance after indexing. Figure 7 shows the result of the EXPLAIN ANALYZE commands after adding index *reviews_rating*. We can see from the result that this indexing did not give any performance boost. We suspect that since we did not actually query rows based on the value of *Rating* but merely added up its values, adding an index on *Rating* will not give us a performance boost for this query.

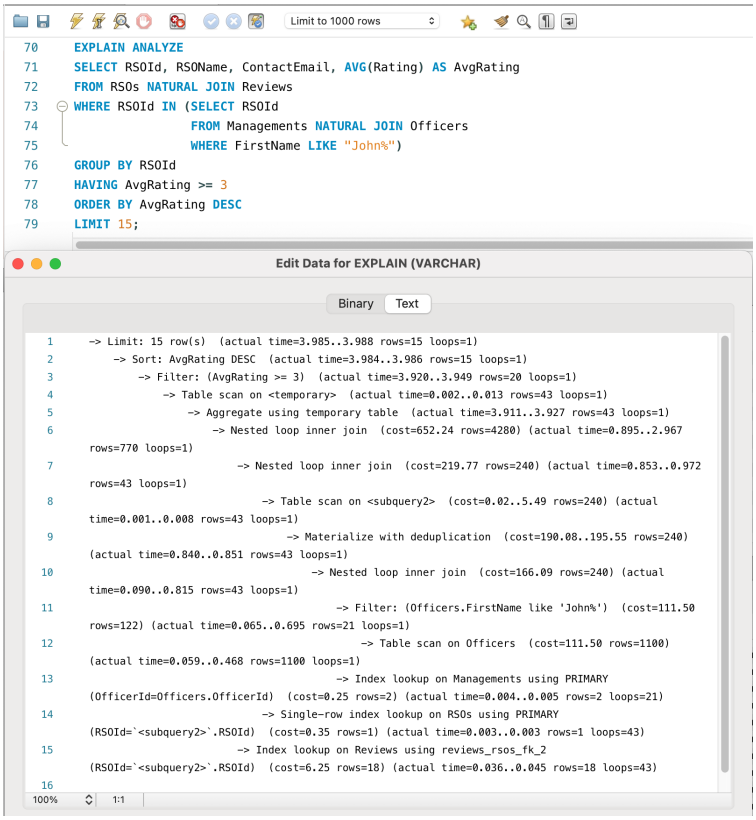


Figure 11: First query: EXPLAIN ANALYZE after index *reviews_rating*

Now we drop this index and try another one, which is on the attributed *LastName* in table *Officers*

```
CREATE INDEX officers_lastname ON Officers(LastName);
```

and use EXPLAIN ANALYZE to analyze the query's performance after indexing. Figure 12 shows the result of the EXPLAIN ANALYZE commands after adding index *officers_lastname*. Similarly, this index will not lead to any speedup for this query because attribute *LastName* is not used in this query.

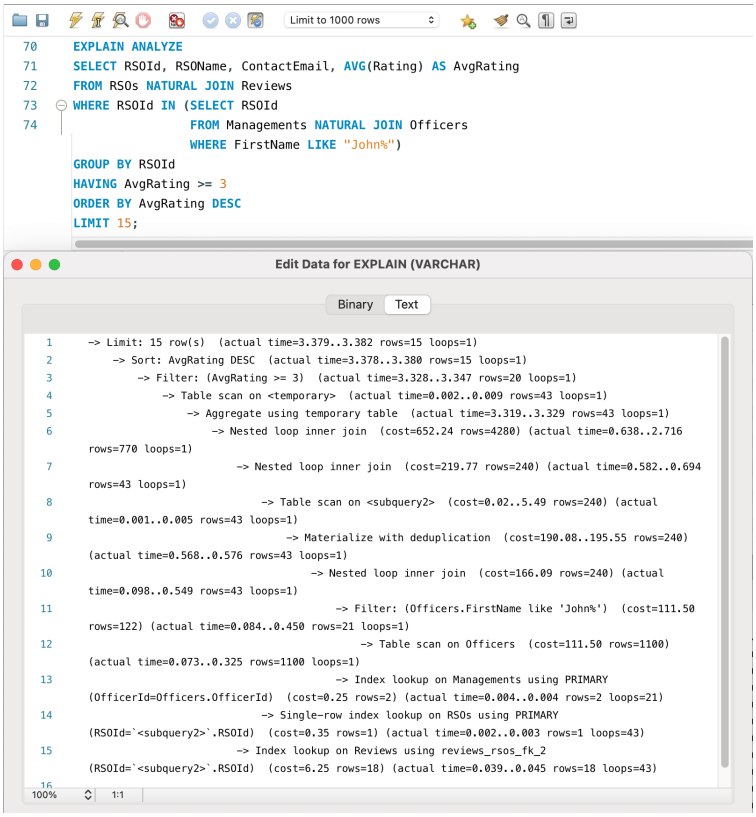


Figure 12: First query: EXPLAIN ANALYZE after index *officers_lastname*