

# 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.

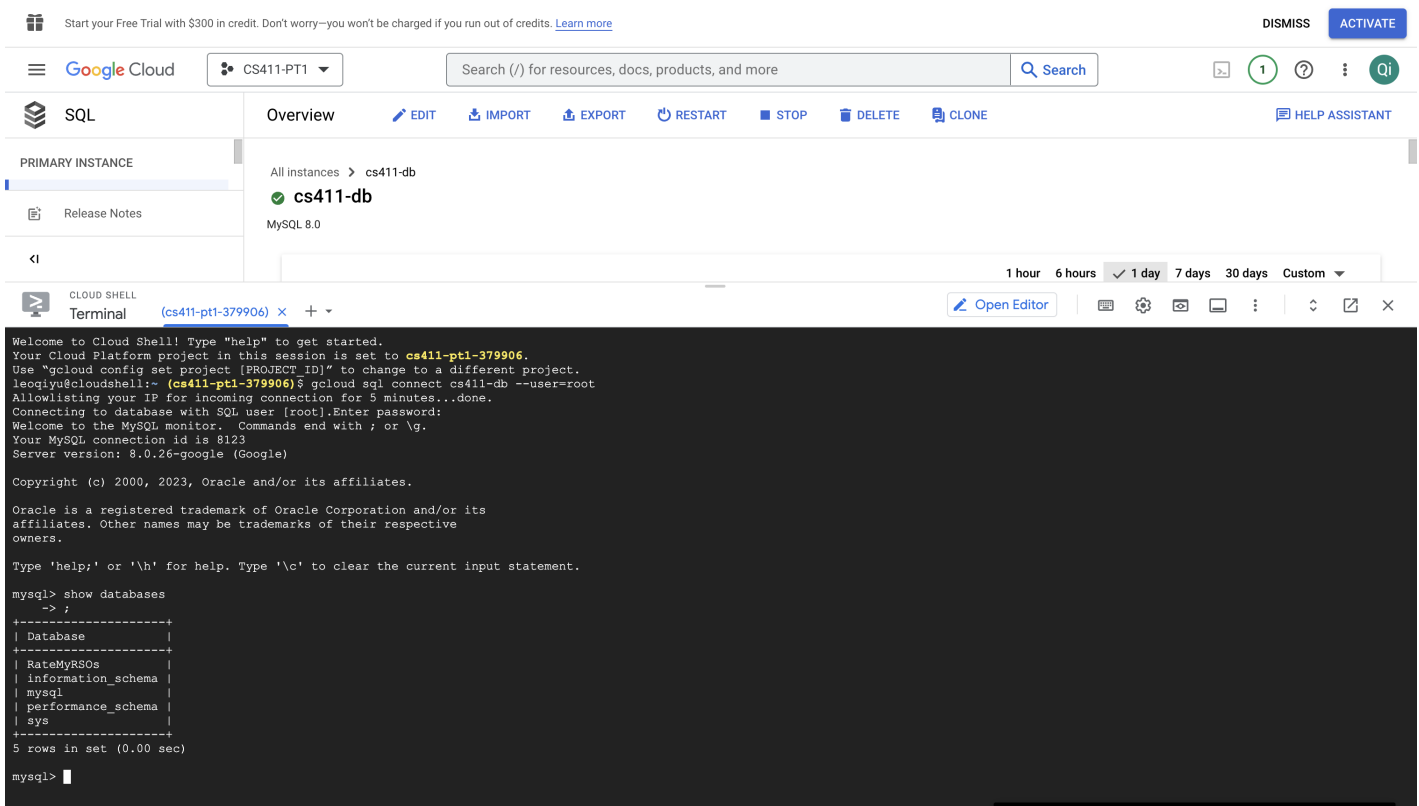


Figure 1: Connection on GCP

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 affiliate_rsos_fk_1 FOREIGN KEY (RSOId) REFERENCES RSOs (RSOId),
CONSTRAINT affiliate_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.

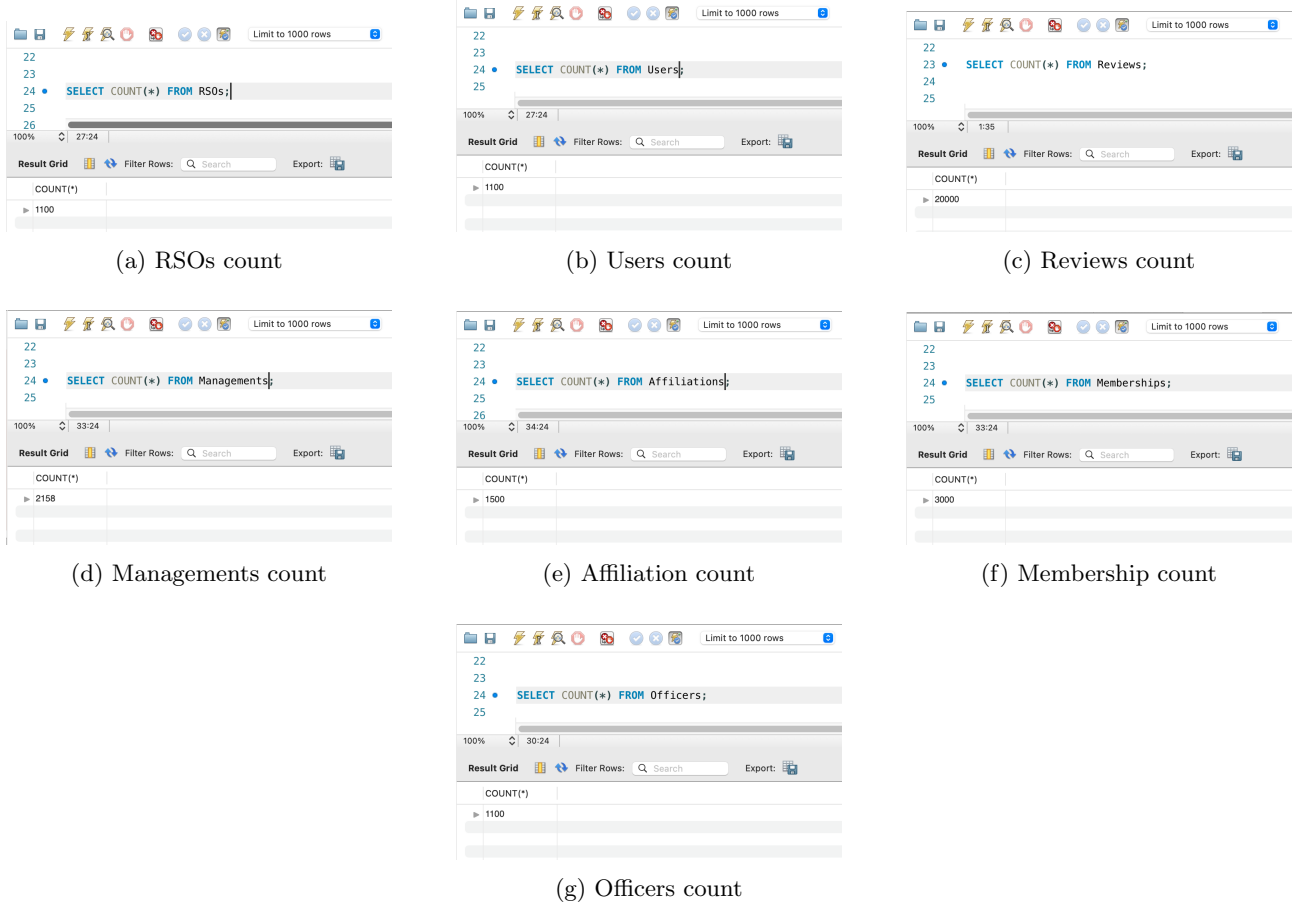


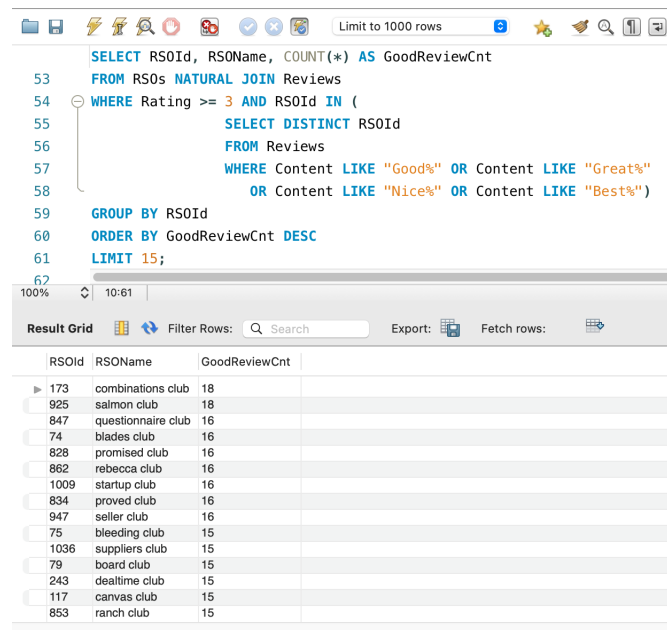
Figure 2: Number of rows inserted

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;
```



The screenshot shows a SQL query editor with a toolbar at the top. The query is entered in the main text area. Below the query, there is a 'Result Grid' section showing the results of the query. The results are displayed in a table with three columns: RSOId, RSOName, and GoodReviewCnt. The table contains 15 rows of data, sorted by GoodReviewCnt in descending order.

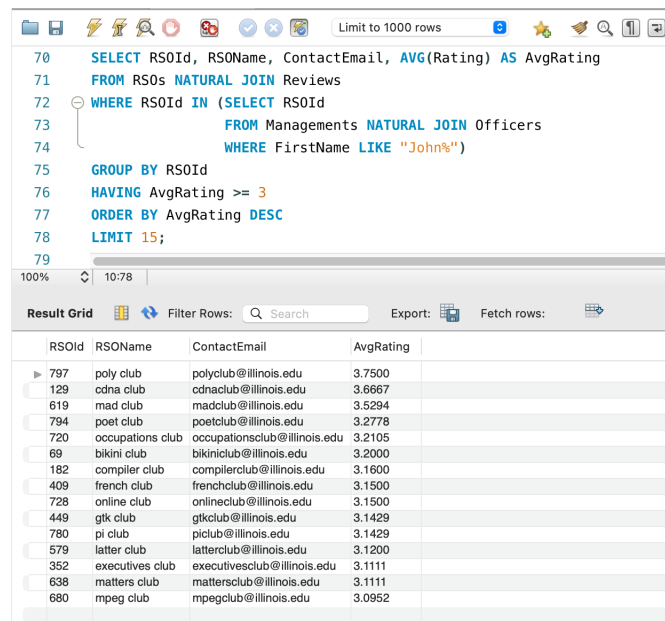
RSOId	RSOName	GoodReviewCnt
173	combinations club	18
925	salmon club	18
847	questionnaire club	16
74	blades club	16
828	promised club	16
862	rebecca club	16
1009	startup club	16
834	proved club	16
947	seller club	16
75	bleeding club	15
1036	suppliers club	15
79	board club	15
243	dealttime club	15
117	canvas club	15
853	ranch club	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

```

Result Grid

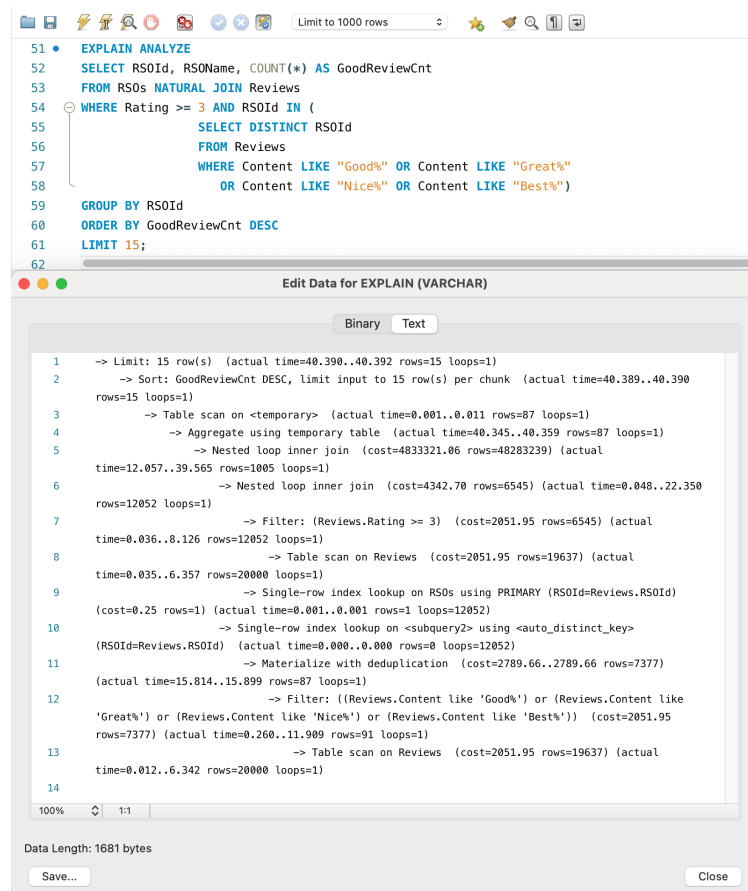
RSOId	RSOName	ContactEmail	AvgRating
797	poly club	polyclub@illinois.edu	3.7500
129	cdna club	cdnaclub@illinois.edu	3.6667
619	mad club	madclub@illinois.edu	3.5294
794	poet club	poetclub@illinois.edu	3.2778
720	occupations club	occupationsclub@illinois.edu	3.2105
69	bikini club	bikiniclub@illinois.edu	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.



```

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

```

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*

```

51 • EXPLAIN ANALYZE
52 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;
62

```

Edit Data for EXPLAIN (VARCHAR)

Binary Text

```

1  --> Limit: 15 row(s) (actual time=4.185..4.187 rows=15 loops=1)
2  --> Sort: GoodReviewCnt DESC, limit input to 15 row(s) per chunk (actual time=4.184..4.185
   rows=15 loops=1)
3  --> Table scan on <temporary> (actual time=0.002..0.013 rows=87 loops=1)
4  --> Aggregate using temporary table (actual time=4.138..4.155 rows=87 loops=1)
5  --> Nested loop inner join (cost=230.96 rows=541) (actual time=0.439..3.595
   rows=1005 loops=1)
6  --> Nested loop inner join (cost=64.05 rows=91) (actual time=0.384..0.571
   rows=87 loops=1)
7  --> Table scan on <subquery2> (cost=0.04..3.64 rows=91) (actual
   time=0.001..0.010 rows=87 loops=1)
8  --> Materialize with deduplication (cost=51.10..54.70 rows=91) (actual
   time=0.372..0.387 rows=87 loops=1)
9  --> Filter: ((Reviews.Content like 'Good%') or (Reviews.Content like
   'Great%') or (Reviews.Content like 'Nice%') or (Reviews.Content like 'Best%')) (cost=41.96
   rows=91) (actual time=0.116..0.342 rows=91 loops=1)
10 --> Index range scan on Reviews using reviews_content
   (cost=41.96 rows=91) (actual time=0.112..0.310 rows=91 loops=1)
11 --> Single-row index lookup on RSOS using PRIMARY
   (RSOId='<subquery2>'.RSOId) (cost=0.35 rows=1) (actual time=0.002..0.002 rows=1 loops=87)
12 --> Filter: (Reviews.Rating >= 3) (cost=5.06 rows=6) (actual time=0.026..0.033
   rows=12 loops=87)
13 --> Index lookup on Reviews using reviews_rsos_fk_2
   (RSOId='<subquery2>'.RSOId) (cost=5.06 rows=18) (actual time=0.026..0.032 rows=19 loops=87)
14

```

Data Length: 1674 bytes

Save... Close

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  $\geq$  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.

```

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=39.224..39.226 rows=15 loops=1)
2  -> Sort: GoodReviewCnt DESC, limit input to 15 row(s) per chunk (actual time=39.223..39.224
rows=15 loops=1)
3  -> Table scan on <temporary> (actual time=0.002..0.010 rows=87 loops=1)
4  -> Aggregate using temporary table (actual time=39.178..39.191 rows=87 loops=1)
5  -> Nested loop inner join (cost=7249311.57 rows=72428415) (actual
time=11.982..38.469 rows=1005 loops=1)
6  -> Nested loop inner join (cost=5488.25 rows=9818) (actual time=0.042..21.501
rows=12052 loops=1)
7  -> Filter: (Reviews.Rating >= 3) (cost=2051.95 rows=9818) (actual
time=0.031..7.804 rows=12052 loops=1)
8  -> Table scan on Reviews (cost=2051.95 rows=19637) (actual
time=0.029..6.164 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=15.565..15.650 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=0.265..11.839 rows=91 loops=1)
13 -> Table scan on Reviews (cost=2051.95 rows=19637) (actual
time=0.013..6.247 rows=20000 loops=1)
14
100% 1:1
Data Length: 1681 bytes
Save... Close

```

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.

The screenshot shows a database IDE with a query editor and an execution plan window. The query is as follows:

```

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

```

The execution plan window, titled "Edit Data for EXPLAIN (VARCHAR)", shows the following details:

```

1  -> Limit: 15 row(s) (actual time=39.396..39.398 rows=15 loops=1)
2      -> Sort: GoodReviewCnt DESC, limit input to 15 row(s) per chunk (actual time=39.395..39.396
      rows=15 loops=1)
3          -> Table scan on <temporary> (actual time=0.002..0.010 rows=87 loops=1)
4              -> Aggregate using temporary table (actual time=39.350..39.363 rows=87 loops=1)
5                  -> Nested loop inner join (cost=4833321.06 rows=48283239) (actual
      time=12.476..38.678 rows=1005 loops=1)
6                      -> Nested loop inner join (cost=4342.70 rows=6545) (actual time=0.131..21.339
      rows=12052 loops=1)
7                          -> Filter: (Reviews.Rating >= 3) (cost=2051.95 rows=6545) (actual
      time=0.110..7.841 rows=12052 loops=1)
8                              -> Table scan on Reviews (cost=2051.95 rows=19637) (actual
      time=0.108..6.133 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=15.968..16.053 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=0.326..12.234 rows=91 loops=1)
13                                                  -> Table scan on Reviews (cost=2051.95 rows=19637) (actual
      time=0.017..6.573 rows=20000 loops=1)
14

```

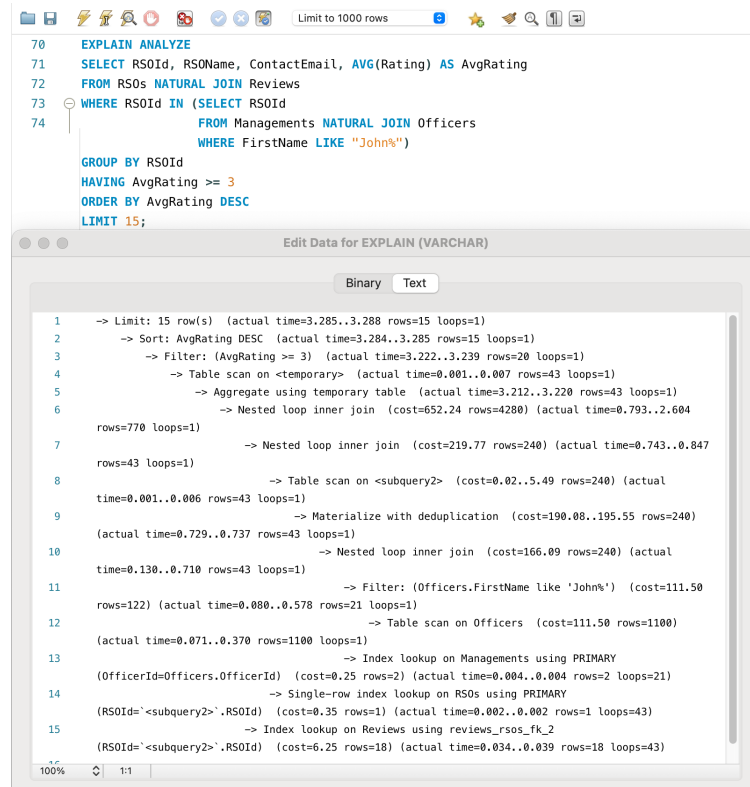
At the bottom of the execution plan window, it states "Data Length: 1681 bytes" and has "Save..." and "Close" buttons.

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.



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

Edit Data for EXPLAIN (VARCHAR)

Binary Text

```
1  --> Limit: 15 row(s) (actual time=3.285..3.288 rows=15 loops=1)
2    --> Sort: AvgRating DESC (actual time=3.284..3.285 rows=15 loops=1)
3    --> Filter: (AvgRating >= 3) (actual time=3.222..3.239 rows=20 loops=1)
4    --> Table scan on <temporary> (actual time=0.001..0.007 rows=43 loops=1)
5    --> Aggregate using temporary table (actual time=3.212..3.220 rows=43 loops=1)
6    --> Nested loop inner join (cost=652.24 rows=4280) (actual time=0.793..2.604
7      rows=770 loops=1)
8    --> Nested loop inner join (cost=219.77 rows=240) (actual time=0.743..0.847
9      rows=43 loops=1)
10   --> Table scan on <subquery2> (cost=0.02..5.49 rows=240) (actual
11     time=0.001..0.006 rows=43 loops=1)
12   --> Materialize with deduplication (cost=190.08..195.55 rows=240)
13     (actual time=0.729..0.737 rows=43 loops=1)
14   --> Nested loop inner join (cost=166.09 rows=240) (actual
15     time=0.130..0.710 rows=43 loops=1)
16   --> Filter: (Officers.FirstName like 'John%') (cost=111.50
17     rows=122) (actual time=0.080..0.578 rows=21 loops=1)
18   --> Table scan on Officers (cost=111.50 rows=1100)
19     (actual time=0.071..0.370 rows=1100 loops=1)
20   --> Index lookup on Managements using PRIMARY
21     (OfficerId=Officers.OfficerId) (cost=0.25 rows=2) (actual time=0.004..0.004 rows=2 loops=21)
22   --> Single-row index lookup on RSOs using PRIMARY
23     (RSOId='<subquery2>'.RSOId) (cost=0.35 rows=1) (actual time=0.002..0.002 rows=1 loops=43)
24   --> Index lookup on Reviews using reviews_rsos_fk_2
25     (RSOId='<subquery2>'.RSOId) (cost=6.25 rows=18) (actual time=0.034..0.039 rows=18 loops=43)
```

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*

```

70 EXPLAIN ANALYZE
71 SELECT RSOId, RSOName, ContactEmail, AVG(Rating) AS AvgRating
72 FROM RSOs NATURAL JOIN Reviews
73 WHERE RSOId IN (SELECT RSOId
74                 FROM Managements NATURAL JOIN Officers
75                 WHERE FirstName LIKE 'John%')
76 GROUP BY RSOId
77 HAVING AvgRating >= 3
78 ORDER BY AvgRating DESC
79 LIMIT 15;
80
Edit Data for EXPLAIN (VARCHAR)
Binary Text
1  -> Limit: 15 row(s) (actual time=2.509..2.511 rows=15 loops=1)
2    -> Sort: AvgRating DESC (actual time=2.508..2.509 rows=15 loops=1)
3      -> Filter: (AvgRating >= 3) (actual time=2.464..2.481 rows=20 loops=1)
4        -> Table scan on <temporary> (actual time=0.001..0.006 rows=43 loops=1)
5          -> Aggregate using temporary table (actual time=2.457..2.465 rows=43 loops=1)
6            -> Nested loop inner join (cost=288.00 rows=735) (actual time=0.294..1.856
rows=770 loops=1)
7              -> Nested loop inner join (cost=30.59 rows=41) (actual time=0.212..0.316
rows=43 loops=1)
8                -> Table scan on <subquery2> (cost=0.07..3.01 rows=41) (actual
time=0.001..0.005 rows=43 loops=1)
9                  -> Materialize with deduplication (cost=23.28..26.22 rows=41)
(actual time=0.202..0.209 rows=43 loops=1)
10                    -> Nested loop inner join (cost=19.09 rows=41) (actual
time=0.112..0.188 rows=43 loops=1)
11                      -> Filter: (Officers.FirstName like 'John%') (cost=9.71
rows=21) (actual time=0.103..0.111 rows=21 loops=1)
12                        -> Index range scan on Officers using
officers_firstname (cost=9.71 rows=21) (actual time=0.100..0.104 rows=21 loops=1)
13                          -> Index lookup on Managements using PRIMARY
(OfficerId=Officers.OfficerId) (cost=0.26 rows=2) (actual time=0.003..0.003 rows=2 loops=21)
14                            -> Single-row index lookup on RSOs using PRIMARY
(RSOId='<subquery2>'.RSOId) (cost=0.01 rows=1) (actual time=0.002..0.002 rows=1 loops=43)
15                              -> Index lookup on Reviews using reviews_rsos_fk_2
(RSOId='<subquery2>'.RSOId) (cost=185.65 rows=18) (actual time=0.029..0.034 rows=18 loops=43)
16
100% 1:1

```

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.

```

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

```

Edit Data for EXPLAIN (VARCHAR)

Binary Text

```

1  --> Limit: 15 row(s) (actual time=3.985..3.988 rows=15 loops=1)
2  --> Sort: AvgRating DESC (actual time=3.984..3.986 rows=15 loops=1)
3  --> Filter: (AvgRating >= 3) (actual time=3.920..3.949 rows=20 loops=1)
4  --> Table scan on <temporary> (actual time=0.002..0.013 rows=43 loops=1)
5  --> Aggregate using temporary table (actual time=3.911..3.927 rows=43 loops=1)
6  --> Nested loop inner join (cost=652.24 rows=4280) (actual time=0.895..2.967
   rows=770 loops=1)
7  --> Nested loop inner join (cost=219.77 rows=240) (actual time=0.853..0.972
   rows=43 loops=1)
8  --> Table scan on <subquery2> (cost=0.02..5.49 rows=240) (actual
   time=0.001..0.008 rows=43 loops=1)
9  --> Materialize with deduplication (cost=190.08..195.55 rows=240)
   (actual time=0.840..0.851 rows=43 loops=1)
10 --> Nested loop inner join (cost=166.09 rows=240) (actual
   time=0.090..0.815 rows=43 loops=1)
11 --> Filter: (Officers.FirstName like 'John%') (cost=111.50
   rows=122) (actual time=0.065..0.695 rows=21 loops=1)
12 --> Table scan on Officers (cost=111.50 rows=1100)
   (actual time=0.059..0.468 rows=1100 loops=1)
13 --> Index lookup on Managements using PRIMARY
   (OfficerId=Officers.OfficerId) (cost=0.25 rows=2) (actual time=0.004..0.005 rows=2 loops=21)
14 --> Single-row index lookup on RSOS using PRIMARY
   (RSOId=<subquery2>`.RSOId) (cost=0.35 rows=1) (actual time=0.003..0.003 rows=1 loops=43)
15 --> Index lookup on Reviews using reviews_rsos_fk_2
   (RSOId=<subquery2>`.RSOId) (cost=6.25 rows=18) (actual time=0.036..0.045 rows=18 loops=43)
16
100% 1:1

```

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.

The screenshot shows a database IDE with two windows. The top window contains an SQL query, and the bottom window shows the execution plan for that query.

```

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

```

The bottom window, titled "Edit Data for EXPLAIN (VARCHAR)", shows the execution plan for the query above. It includes details such as the number of rows, loops, and the cost of each operation.

```

1  -> Limit: 15 row(s) (actual time=3.379..3.382 rows=15 loops=1)
2    -> Sort: AvgRating DESC (actual time=3.378..3.380 rows=15 loops=1)
3      -> Filter: (AvgRating >= 3) (actual time=3.328..3.347 rows=20 loops=1)
4        -> Table scan on <temporary> (actual time=0.002..0.009 rows=43 loops=1)
5          -> Aggregate using temporary table (actual time=3.319..3.329 rows=43 loops=1)
6            -> Nested loop inner join (cost=652.24 rows=4280) (actual time=0.638..2.716
7              rows=770 loops=1)
8              -> Nested loop inner join (cost=219.77 rows=240) (actual time=0.582..0.694
9                rows=43 loops=1)
9                -> Table scan on <subquery2> (cost=0.02..5.49 rows=240) (actual
10                  time=0.001..0.005 rows=43 loops=1)
11                  -> Materialize with deduplication (cost=190.08..195.55 rows=240)
12                    (actual time=0.568..0.576 rows=43 loops=1)
13                    -> Nested loop inner join (cost=166.09 rows=240) (actual
14                      time=0.098..0.549 rows=43 loops=1)
15                      -> Filter: (Officers.FirstName like 'John%') (cost=111.50
16                        rows=122) (actual time=0.084..0.450 rows=21 loops=1)
17                        -> Table scan on Officers (cost=111.50 rows=1100)
18                          (actual time=0.073..0.325 rows=1100 loops=1)
19                          -> Index lookup on Managements using PRIMARY
20                            (OfficerId=Officers.OfficerId) (cost=0.25 rows=2) (actual time=0.004..0.004 rows=2 loops=21)
21                            -> Single-row index lookup on RSOs using PRIMARY
22                              (RSOId='<subquery2>'.RSOId) (cost=0.35 rows=1) (actual time=0.002..0.003 rows=1 loops=43)
23                              -> Index lookup on Reviews using reviews_rsos_fk_2
24                                (RSOId='<subquery2>'.RSOId) (cost=6.25 rows=18) (actual time=0.039..0.045 rows=18 loops=43)

```

Figure 12: First query: EXPLAIN ANALYZE after index *officers\_lastname*