

Report – Assignment 2 on SQL

This report explains the process of creating a relational database and referential constraints and subsequently querying it. The report also includes exploring and visualizing query plans. A section also examines the relational algebra for given queries and creating indexes for improving query performance.

Note: The program code is provided individually for each question in the root folder. The program can be executed by following the README.md instructions. The questions from the root directory contain method calls from the other files in the “code” subfolder to make the code more manageable and readable. The global settings are provided in “globals.py” inside the “code” subfolder.

Question 1: Provide a program to load the Stack Exchange data.

The program provided in “q1.py” creates the relational model using the updated description and provided dataset. The SQL script to create all tables in the dataset, including primary and foreign keys, is provided in the “create_schema.sql” (in the root directory). The script is an independent SQL file and can be executed from “q1.py” in the root directory, along with the insertion. The insertion queries are provided in “queries.py” in the “code” subfolder.

I added the foreign key constraints before the insertion and ensured that those were not violated by checking valid keys on each chunk of the insert. This ensured that no violations occurred during the insertion. The self-referenced foreign keys on the posts table were added after the record insertion to reduce the time and memory consumption for deletion. For this, I first ensured all the parent IDs, accepted answer IDs, and user IDs were valid for adding posts. To ensure that, I created a dummy table with all post IDs to ensure I scanned all the parents and answers over the whole post's data. Table 1 explains the remarks for handling each constraint. The insertion takes around 510 seconds (around 8.5 minutes) across multiple runs on my machine.

Table 1. Remarks for handling each given foreign key constraint.

Sr.	Foreign Key	Remark
1.	OwnerUserId FK Users(Id)	Ensured that posts were inserted where the user was present.
2.	ParentId FK Posts(Id)	Ensured that parent IDs are valid posts before they are inserted. Ignored the null as a post can exist without a parent. This has been achieved first by inserting all post IDs.
3.	AcceptedAnswerId FK Posts(Id)	Ensured that answer IDs were valid posts before they were inserted. Ignored the null as a post can exist without an answer. This has been achieved first by inserting all post IDs.
4.	PostId FK Posts(Id)	Ensured that a post existed before it was inserted into post tags.
5.	TagId FK Tags(Id)	Ensured that the tag exists before it is inserted into post tags.
6.	UserId FK Users(Id)	Ensured that badges were inserted where the user was present.
7.	PostId FK Posts(Id)	Ensured that posts existed before the comments were inserted for a post.
8.	UserId FK Users(Id)	Ensured that comments were inserted where the user was present.

The inserted statistics are as under;

Users Inserted: 1450497, Badges Inserted: 1977517, Tags Inserted: 3155, Posts Inserted: 909005, Post Tags Inserted: 1130897, Comments Inserted: 1453405

Question 2: Provide a program to retrieve the data from the dataset. Report the evidence (e.g., rows returned report the time queries took to run.

The queries are provided in the “queries.py” in the “code” subfolder. Queries are named with query number (i.e., the answer to the first query in this question is named “q2_query1”, and so on). The program can be run from the root folder by running the “q2.py” file. The following provides details on each query's result and execution time.

Report – Assignment 2 on SQL

Query 1. Names of the top 10 most popular badges earned by users within a year of creating their accounts.

Time Taken: 0.93 seconds, Records: 10

```
Names of the top 10 most popular badges earned by users
('Autobiographer', 452480)
('Student', 123270)
('Supporter', 116961)
('Editor', 106203)
('Informed', 95175)
('Teacher', 64725)
('Scholar', 51761)
('Popular Question', 49721)
('Tumbleweed', 41843)
('Notable Question', 20032)
+++++
Query took: 0.9314703941345215 seconds
+++++
```

Query 2. Display names of users who have never posted but have a reputation greater than 1,000.

Time Taken: 0.50 seconds, Records: 12

```
Display names of users who have never posted
('Shiv',)
('Min San',)
('davidaf',)
('Arun. K. P',)
('rosterloh',)
('cmd-not-found',)
('Biber',)
('ninjalj',)
('Anon',)
('pagal pila',)
('MirkoZa',)
+++++
Query took: 0.5041491985321045 seconds
+++++
```

Query 3. Display the name and reputation of users who have answered more than one question with the tag “postgresql”.

Time Taken: 0.14 seconds, Records: 0

```
Display name and reputation of users who have answered
+++++
Query took: 0.1418623924255371 seconds
+++++
```

Empty Results

Query 4. Display the names of users who posted comments with a score greater than 10 within the first week of creating their accounts.

Time Taken: 0.25 seconds, Records: 240

```
Display name of users who posted comments with a score
('David Z',)
('levesque',)
('Jono',)
('David Siegel',)
('jbowtie',)
('crenshaw-dev',)
('Jeremy Kerr',)
('Owais Lone',)
('Mircea Chirea',)
('McDowell',)
('Tommy Brunn',)
+++++
Query took: 0.2504274845123291 seconds
+++++
```

Query 5. The tag names of the tags most commonly used on posts along with the tag “postgresql” and the count of each tag.

Time Taken: 0.26 seconds, Records: 254

```
The tag names of the tags most commonly used on posts
('postgresql', 653)
('apt', 110)
('server', 68)
('package-management', 53)
('14.04', 52)
('software-installation', 33)
('database', 31)
('16.04', 28)
('12.04', 27)
('pgadmin', 27)
('permissions', 24)
+++++
Query took: 0.26323437690734863 seconds
+++++
```

Question 3: Visualize and provide a brief explanation of the execution plan for each query in question 2.

The explained queries are provided in the “queries.py” in the “code” subfolder. Queries are named with query number and embedded with explain (i.e., the answer to the first query in this question is named “q3_query1_explain”, and so on). The program to get the same visualize plan can be run from the root folder by running the “q3.py” file. The plan is saved for each query in a text file in the root folder at successful “q3.py” execution.

Query 1. Names of the top 10 most popular badges earned by users within a year of creating their accounts.

Looking from bottom to up; the query plan shows that there will be a parallel sequence search on the user's table with 604374 rows expected. Before scanning the users the plan shows hashing on the users' table. Moving a step up a sequence scan is performed on badges and a filter is applied on the creation date to filter data for the given range (1 year). This is also accompanied by hashing on join conditions, to map users and badges table. A step above that shows aggregation by badge name. Then there is sorting on the badge name as well and the query defines sorted results in descending order. There are 2 worker groups planned for the query. Then, the gather merge states that the results will be merged from two workers. Then, the grouping in the aggregation is done by the name field of the badges table. Towards the top; it sorts the results by count of badges and there will be 10 rows returned at the end of the query.

Report – Assignment 2 on SQL

Query 2. Display names of users who have never posted but have a reputation greater than 1,000.

The plan shows a parallel sequential scan on the users' table as it reads all rows and applies the filter of reputation having a value greater than 1000. The plan shows creating a hash table for users to perform the join operation. The expected results till this point are 1842. The query plan shows scanning the posts table in parallel without applying any filter. The next step joins posts with the user's hash table by matching the join key. This will return rows without matching users' IDs and having the reputation of 1000 from the join. This query also uses two workers. The final result expected is 4105 rows.

Query 3. Display the name and reputation of users who have answered more than one question with the tag “postgresql”.

This plan shows a sequential search on the tags table to look for the “postgresql” tag to filter the data (1 row expected). A hash table is created for filtered data to create a join hash. The next step scans the post tags table in parallel to collect tag-related posts. It again uses has join with tags table to join with post tags for the filtered tag. A nested loop joins results for tags and post tags. Then, it uses index search using the user's primary key and post-primary key. Subsequently, results are sorted and grouped based on user ID to count posts tagged with the given tag. The results from multiple workers are merged and a final group aggregate is applied to only include users who have more than one question answered with the given tag.

Query 4. Display the names of users who posted comments with a score greater than 10 within the first week of creating their accounts.

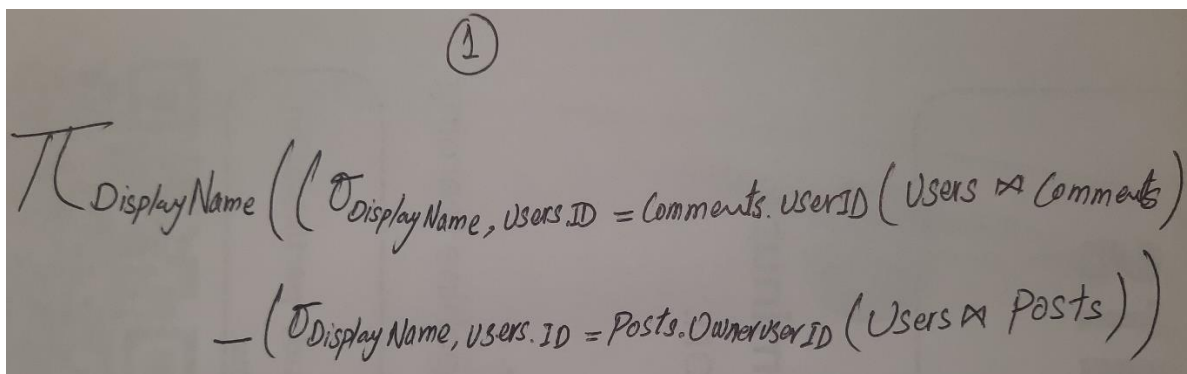
From the bottom up; the first step scans use the index scan user's primary key that matches the comments table. The filter checks whether the creation date of the comment is within 7 days of the user's creation date. Then, it performs a parallel sequential scan on the comments table. The scan also filters the comments table with a score greater than 10. Then, a nested loop combines results from comments and users. Finally, the step gathers the results from workers (2) and merges them.

Query 5. The tag names of the tags most commonly used on posts along with the tag “postgresql” and the count of each tag.

The query plan shows an index scan using the tag primary key with the tag “postgresql”. Then, the query joins the index-only scan using post tags and posts and gets results on each associated post with the joined key. Then, a nested loop combines results from scans and creates a hash aggregate based on post IDs. Then the gather operation collects results from multiple workers (2) and merges the results of parallel scans. Again, nested loops join the results from post tags and tags with a given tag name. Then, the results are aggregated by a count of posts by tag name and sorted in descending order on the tag count. The aggregate ensures distinct tags are counted for final output returning the most commonly used tags alongside "postgresql" (expected 495 rows).

Question 4: Provide relational algebra expressions for the queries below.

Query 1: Display names of users who have commented but never posted.



①

$$\pi_{\text{Display Name}} \left(\left(\sigma_{\text{Display Name}, \text{Users.ID} = \text{Comments.UserID}} (\text{Users} \bowtie \text{Comments}) \right) - \left(\sigma_{\text{Display Name}, \text{Users.ID} = \text{Posts.OwnerUserID}} (\text{Users} \bowtie \text{Posts}) \right) \right)$$

Query 2: Display names of users who have not made any post with the tag “postgres”

②

$$\pi_{\text{Display Name}} \left(\sigma_{\text{ID}, \text{Display Name}} \left(\text{Users} \bowtie_{\text{Users.ID} = \text{Posts.ownerUserID}} \left(\text{Posts} \right. \right. \right. \\ \left. \left. \left. \bowtie_{\text{posts.ID} = \text{PostTags.POSTID}} \left(\text{PostTags} \bowtie_{\text{PostTags.TagID} = \text{Tags.ID} \wedge \text{TagName} \neq \text{'postgres'}} \right. \right. \right. \right. \\ \left. \left. \left. \left(\text{Tags} \right) \right) \right) \right)$$

Query 3: Display names of users who have commented on any post in 2017 and whose name contains “John”.

③

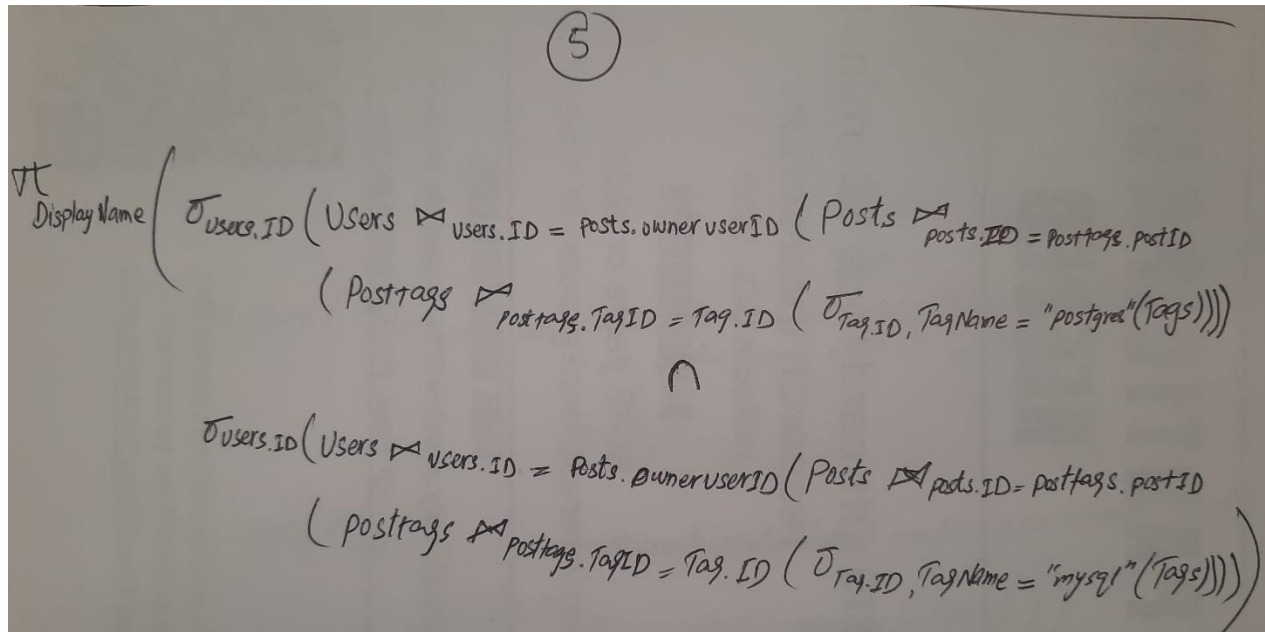
$$\pi_{\text{Display Name}} \left(\sigma_{\text{users.ID}, \text{DisplayName LIKE 'John\%'}} \left(\sigma_{\text{users.ID} = \text{Comments.userID}} \right. \right. \\ \left. \left. \sigma_{\text{2017-01-01} \leq \text{CreationDate} \leq \text{2017-12-31}} \left(\text{Users} \bowtie_{\text{Comments}} \left(\text{Comments} \right) \right) \right)$$

Query 4: The title of posts with a score greater than 1,000 posted by users within the first year of creating their account.

④

$$\pi_{\text{Title, Score}} \left(\sigma_{\text{Score} > 1000 \wedge \text{posts.creationDate} \leq (\text{Users.creationDate} + \text{Interval '1 Year'})} \right. \\ \left. \left(\text{Posts} \bowtie_{\text{posts.ownerUserID} = \text{Users.ID}} \left(\text{Users} \right) \right) \right)$$

Query 5: Users who have made posts with both the tag “postgres” and the tag “mysql”.



Question 5: Create indexes and full-text indexes where they are required. Document your decisions and provide the scripts to generate them. Re-run all the previous queries report performance improvement and briefly explain why such an improvement including references to execution plans.

The code for creating indexes can be run from “q5.py” from the root directory. The queries are provided in “queries.py” in the code subfolder.

Decisions: I created indexes on reputation, user IDs, post IDs, tag IDs, and tag names for referenced fields across the tables (utilized in the queries for question 2). I also created indexes for relationships used in question 2 to optimize the joins. The list of indexes created includes Reputation (users), OwnerUserId (posts), ParentId (posts), PostId (comments), UserId (comments), PostId (posttags), TagId (posttags). I also created text indexes for the only TagName (tags) to handle tags.

After restarting the database server, I executed the queries and query plans again. There is a slight improvement in processing time and query plans also show a decrease and adaptation to using different methods (for some queries).

Query Performance

Table 2. Comparison of each query’s execution time before and after applying indexes.

Sr	Before (Time)	After (Time)
Query 1	0.93 seconds	0.90 seconds
Query 2	0.50 seconds	0.12 seconds
Query 3	0.14 seconds	0.069 seconds
Query 4	0.25 seconds	0.27 seconds
Query 5	0.26 seconds	0.10 seconds

Explanations:

Query 1. Query 1 shows a similar execution plan even after applying indexes. I created an index of user id for the relationship between users and badges. However, I assume the execution plan favored the sequential search rather than the sequential search.

Report – Assignment 2 on SQL

Query 2. Query 2 shows good improvement and the query plan shows switching from Parallel Hash Join to a Nested Loop Join. Cost also reduced from 31126.76 to 1058.18. The new plan uses a Bitmap Index Scan instead of a full sequential scan to filter users having reputations over 1000. The other change is seen with an index-only scan for post owners for the post table to access the joined rows, showing fewer rows to process.

Query 3. Optimization for this query shows a reduction in several steps. The cost is also reduced from 8551.37 to 1418.70. The initial plan includes grouping and gathering merge functions/algorithms which are eliminated. The updated plan uses group aggregation on the sorted input directly. This query plan also uses a Bitmap Heap Scan on the posttags table and Bitmap Index Scan on idx_posttags_tag which could have improved the efficiency as compared to parallel sequence joins in the initial query. The plan still uses a sequential scan on the tags table to filter specific times.

Query 4. Query 4 shows a similar execution plan even after applying indexes. I created an index on user id for the relationship between users and comments. However, I assume the execution plan favored the sequential search rather than the sequential search.

Query 5. Although this query plan shows almost similar steps, however, the query performance is improved substantially by using indexes. The cost is reduced from 8968.33 to 1641.60. The initial query uses group aggregates with multiple sorts which is improved by creating a hash aggregate directly after the join. The idx_posttags_tag also uses the Hash Join with a Bitmap Index Scan to filter tags. Both query plans still use the Seq Scan on tags to filter by the specific tag name. The use of indexes shows good improvement for this query.

Note: Instructions to run the code are provided in the README.md file in the root folder.