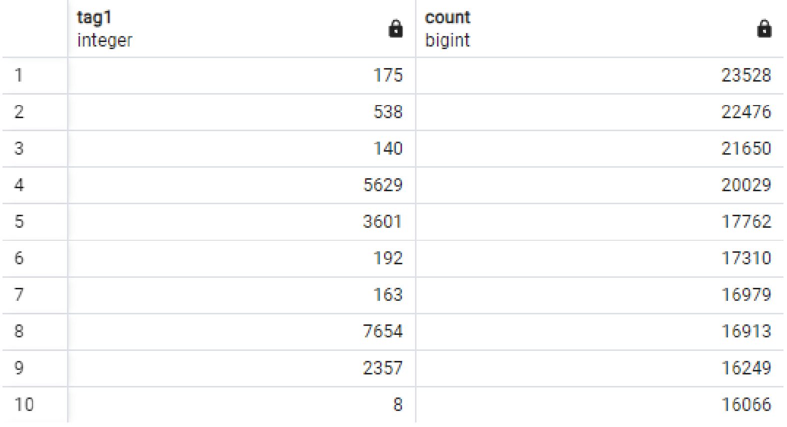
This report describes implementing the "Apriori" algorithm using SQL with Stack Exchange data. For N-item-set mining, it shows the method for creating a new table in each set of lattices, taking tags as items and a set of tags for each post as transactions. The minimum support for all questions is 100.

Note: The program code is provided for questions 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 "sql" subfolder to make the code more manageable and readable. The global settings are provided in "globals.py" inside the root folder.

Question 1: Provide SQL to create a table L1 which contains frequent item sets of size one.

The program in "q_123.py" implements creating a table L1 with frequent item sets of size one. The program creates an L1 table by executing SQL queries for question 1 (named "q1_level1" in "queries.py" in the "sql" subfolder). The SQL command selects a tag (Tag ID) and counts posts (Post ID) this tag was used from the post tags table. As the minimum support is 100, I filter the selection to include tags appearing 100 times or more and grouping those on the tag (Tag ID). Based on the dataset populated from my assignment 2, tags that have support of 100 are 1070. The program executes questions 1, 2, and 3 together by running this code.

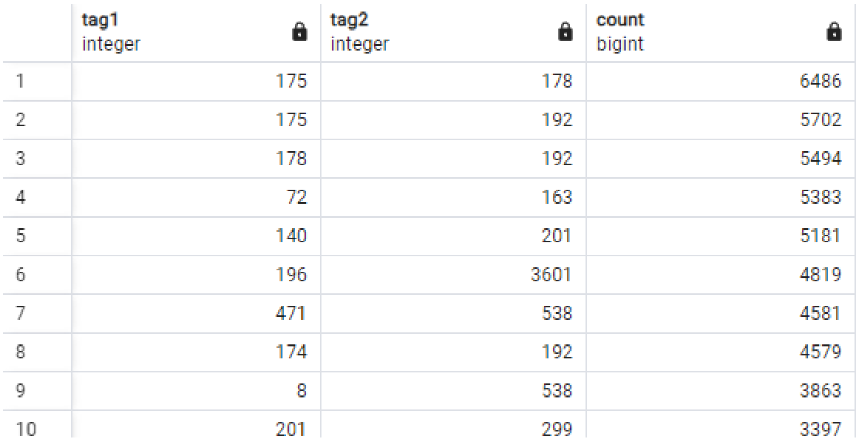


	tag1 integer	count bigint
1	175	23528
2	538	22476
3	140	21650
4	5629	20029
5	3601	17762
6	192	17310
7	163	16979
8	7654	16913
9	2357	16249
10	8	16066

Sample of created L1 table.

Question 2: Provide SQL to create a table L2 containing frequent item sets of size two (tags used on the same post). This should be based only on table L1 and Post Tags written in a single query.

The program in "q_123.py" implements creating a table L2 with frequent item sets of size two. The program creates an L2 table by executing SQL queries for question 2 (named "q2_level2" in "queries.py" in the "sql" subfolder). The SQL command selects a tag (Tag ID) and counts posts (Post ID) this tag was used from the post tags based on Table L1. Based on the dataset populated from my assignment 2, tags that appear together (size 2) and have support of 100 are 1800.

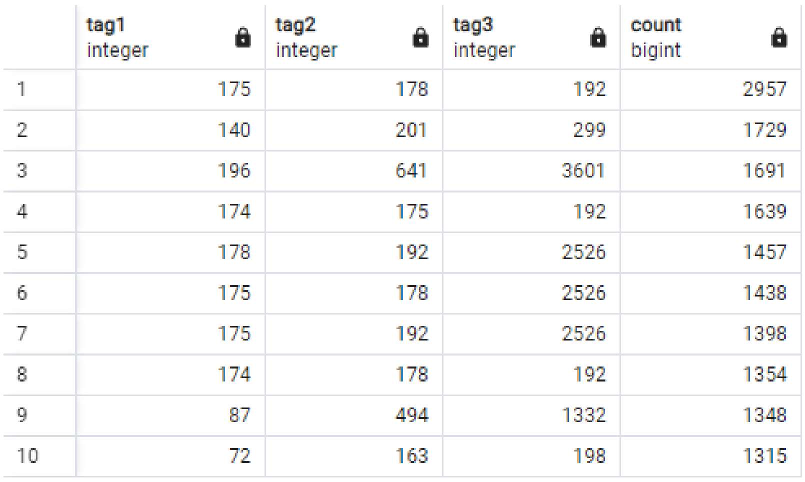


	tag1 integer	tag2 integer	count bigint
1	175	178	6486
2	175	192	5702
3	178	192	5494
4	72	163	5383
5	140	201	5181
6	196	3601	4819
7	471	538	4581
8	174	192	4579
9	8	538	3863
10	201	299	3397

Sample of created L2 table.

Question 3: Provide SQL to create a table L3 containing frequent item sets of size three (tags used on the same post). This should be based only on table L2 and Post Tags written in a single query.

The program in "q_123.py" implements creating a table L3 with frequent item sets of size three. The program creates an L3 table by executing SQL queries for question 3 (named "q3_level3" in "queries.py" in the "sql" subfolder). The SQL command selects a tag (Tag ID) and counts posts (Post ID) this tag was used from the post tags based on the table L2. Based on the dataset populated from my assignment 2, tags that appear together (size 3) and have support of 100 are 385.



	tag1 integer	tag2 integer	tag3 integer	count bigint
1	175	178	192	2957
2	140	201	299	1729
3	196	641	3601	1691
4	174	175	192	1639
5	178	192	2526	1457
6	175	178	2526	1438
7	175	192	2526	1398
8	174	178	192	1354
9	87	494	1332	1348
10	72	163	198	1315

Sample of created L3 table.

Question 4: Provide a program that generalized the approach from previous questions to generate tables for all levels of the lattice (L1, L2… Ln where n is the final level of the lattice). Stop when an empty table is created as the final level of the lattice. Report the count of frequent item sets in each level, and report the names of the tags for the final level of the lattice.

The program in "q4.py" implements generalizing the approach to generate tables from all levels of the lattice. It requires an initial query to first get the set of items meeting the minimum transaction. Hence, I re-use the query from question 1 (named "q1_level1" in "queries.py" in the "sql" subfolder). From level 2 onwards, I automate the query formatting by adding another level of lattice. From level 3 onwards, it is needed to consider matching items specifying the "where" condition. I check the count of results generated in level and report its count (on console and reported as under). When the result count of a level returns zero, the execution stops and reports the tags inserted in the last non-empty level created in the lattice by mapping Tag IDs with the tag names from the Tags table (this is also done programmatically).

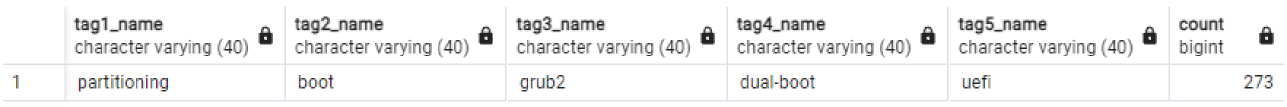
The query generation method is provided in the "q4_lattice_query" method in the "queries.py" in the "sql" subfolder. The final query is printed on the console as well as provided below. To view queries on the console, uncomment the print statement in "q4.py" from the "while" loop (line 27). Results in each level of the lattice are reported as follows;

- L1: 1070 (size 1; item list)
- L2: 1800
- L3: 385
- L4: 37
- L5: 1 (final level)
- L6: 0 (loop breaks here)

The final query is generated programmatically based on the final non-empty level. The query generation method is provided in the "q4_final_query" method in the "queries.py" in the "sql" subfolder. The final query and the result of the count in the final level along with tag names are shown below.

Final Query

SELECT Tags1.TagName AS tag1_name, Tags2.TagName AS tag2_name, Tags3.TagName AS tag3_name, Tags4.TagName AS tag4_name, Tags5.TagName AS tag5_name, L5.count
FROM L5
INNER JOIN Tags AS Tags1 ON Tags1.Id = L5.tag1 INNER JOIN Tags AS Tags2 ON Tags2.Id = L5.tag2
INNER JOIN Tags AS Tags3 ON Tags3.Id = L5.tag3 INNER JOIN Tags AS Tags4 ON Tags4.Id = L5.tag4
INNER JOIN Tags AS Tags5 ON Tags5.Id = L5.tag5
ORDER BY L5.count DESC;



	tag1_name character varying (40)	tag2_name character varying (40)	tag3_name character varying (40)	tag4_name character varying (40)	tag5_name character varying (40)	count bigint
1	partitioning	boot	grub2	dual-boot	uefi	273

Final Query Output (only 1 record)

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