CSCI-Y790
Frequent itemset mining using vertical table and array approach

Suhas Jagadish
under the guidance of Prof. Dirk Van Gucht

December 11, 2016

1. INTRODUCTION

The integration of data mining with database systems is an emergent trend in database research and development area. Database systems provide powerful mechanisms for accessing, filtering, indexing data and SQL parallelization. Most data mining solutions include database-independent applications for the data mining tasks but we believe that coupling data mining with relational databases could remarkably improve the efficiency in knowledge discovery process and simplify the construction of decision support systems.

The computation of frequent itemsets is a computationally expensive preprocessing step for association rule discovery [1], which finds rules in large transactional data sets. Frequent itemsets are combinations of items that appear frequently together in a given set of transactions. Association rules characterize, for example, the purchasing pattern of retail customers or the click pattern of web site visitors. Such information can be used to improve marketing campaigns, retailer store layouts, or the design of a web sites contents.

Although some approaches based on sophisticated in-memory data structures have enriched query languages with mining functionality [2], the query processing power offered by modern database systems, such as PostgreSQL for mining purposes has been widely neglected.

In this project, we analyze several approaches to compute frequent itemsets using SQL. We also propose a new SQL-based approach using arrays or json objects and compare it to the other approaches.

2. FREQUENT ITEMSET DISCOVERY PROBLEM

A fundamental component in data mining tasks is finding frequent patterns in a given dataset. An item is an object of interest, like a product of a shop or a document URL of a web site. An itemset is a set of items and a k-itemset contains k items. A transaction is an itemset representing a fact, like a purchase of products or a collection of URLs requested by a user during a web site visit.

Given a set of transactions, the frequent itemset discovery problem is to find itemsets within the transactions that appear at least as frequently as a given threshold, called minimum support. For example, a user can define that an itemset is frequent if it appears in at least 2% of all transactions.

The itemset discovery algorithm consist of a sequence of steps that proceed in a bottom-up manner: the result of the k-th step is the set of frequent k-itemsets, denoted as F_k. The first step computes the set of frequent items (1-itemsets). Each following step consists of two phases:

(a) The candidate generation phase computes a set of potential frequent k-itemsets from F_{k-1}. The new set is called C_k, the set of candidate k-itemsets. It is a superset of F_k.

(b) The support counting phase filters out those itemsets from C_k that appear more frequently in the given set of transactions than the minimum support and stores them in F_k.

3. TABLE LAYOUT

Before data can be mined with SQL, it has to be made available as relational tables. Typically, the data is stored in tables within the database system or, if the data has to be kept outside of the database system, it has to be accessible to the database system by using wrappers that provide a relational view on the data.

Two types of data objects are relevant for frequent itemset discovery: transactions and itemsets. For each type, there are basically two main layouts for representing these objects in a table. The items of an object can be stored either in a single row, which is called a *vertical layout*, or as an array representation of items, as illustrated in the below table.

<table>
<tr><th>Layout</th><th>Transactions</th><th>Itemsets</th></tr>
<tr><td>Vertical
(multi-row/
single-column)</td><td>(tid, item)
<table><tr><th>tid
integer</th><th>item
character(10)</th></tr><tr><td>100</td><td>A</td></tr><tr><td>100</td><td>C</td></tr><tr><td>100</td><td>D</td></tr><tr><td>200</td><td>B</td></tr><tr><td>200</td><td>C</td></tr><tr><td>200</td><td>E</td></tr></table></td><td>(itemset, position, item)
<table><tr><th>itemsetid
uuid</th><th>pos
integer</th><th>item
charac</th></tr><tr><td>32138f1e-3743</td><td>1</td><td>A</td></tr><tr><td>b835df12-a1c2</td><td>1</td><td>B</td></tr><tr><td>b9f11453-2705</td><td>1</td><td>C</td></tr><tr><td>3fc6f3d4-137c</td><td>1</td><td>D</td></tr><tr><td>a2dd34b6-cff5</td><td>1</td><td>E</td></tr></table></td></tr>
<tr><td>Array representation</td><td>(tid, item)
<table><tr><th>tid
integer</th><th>items
text[]</th></tr><tr><td>100</td><td>{A,C,D}</td></tr><tr><td>200</td><td>{B,C,E}</td></tr><tr><td>300</td><td>{A,B,C,E}</td></tr></table></td><td>(itemset, item)
<table><tr><th>itemsetid
uuid</th><th>items
text[]</th></tr><tr><td>d6cbbe5b-bb4f</td><td>{A}</td></tr><tr><td>6076797e-9304</td><td>{B}</td></tr><tr><td>cf0e7610-aaf8</td><td>{C}</td></tr><tr><td>9baa85d4-a7b0</td><td>{D}</td></tr><tr><td>fbf77ec3-520e</td><td>{E}</td></tr></table></td></tr>
</table>

4. CANDIDATE GENERATION PHASE

The algorithm for both Candidate generation phase and Support counting phase is referenced from [3].

The *Candidate* expression relies on two subexpressions, *Unique* and *PrefixPair*. The *Unique* expression can be regarded as a function that maps a pair of two existing itemset IDs on a new itemset ID that is distinct from all existing IDs and that is guaranteed to be different from any ID that is returned for a different input value pair. *gen_random_uuid()* function of PostgreSQL is used for unique ID generation.

PrefixPair finds ID pairs (a1, a2) of frequent (k1)-itemsets that have a common prefix of size $k2$. Such an itemset pair has the same item value at each position from 1 to $k2$, and the item value of the first itemset at position $k1$ is lexicographically ordered before that of the second itemset. For example, we will create a new itemset ABCD for C if we find the itemsets ABC and ABD in F, which have the common prefix AB.

This approach tries to reduce the number of candidates by ignoring all itemsets that do not fulfill the Apriori property. In general, *Apriori(k)* tests the existence in F_{k1} of those ($k1$)-itemsets that we get when skipping a single item at the positions 1 to $k2$ from a potential candidate k itemset.

The code for generating candidates at each k is attached as 'candidate_phase_k.sql'. For example, 'candidate_phase_1.sql' generates candidates for k=1.

5. SUPPORT COUNTING PHASE

After the candidates are generated in the previous step, the next step is to check if they appear frequently enough in the transactions to qualify for F_k. Once we check if the candidate items are contained in the Transaction, then we can find the support of each candidate by counting the number of distinct transaction values that appear for each candidate.

The code for support counting phase is attached as 'support_phase.sql' for vertical table approach and 'support_phase_array.sql' for array representation.

6. IMPLEMENTATION

(a) Vertical table approach

I assume that we have the transactions data and the initial candidate table ready before we run our code. I have created a stored procedure named '*frequent_itemset_mining()*' which runs the support counting phase and candidate generation phase in sequence to produce three most frequent items bought together. I have set the value of k=3 in this code, it can be changed based on preference and the minimum support is set to 2.

After we execute the stored procedure '*frequent_itemset_mining()*', below is the sequence of results -

tid integer	item character(10)
100	A
100	C
100	D
200	B
200	C
200	E
300	A
300	B
300	C
300	E
400	B
400	E
500	A
500	C
500	E

C1

itemsetid uuid	pos integer	item character(10)
32138f1e-3743-42e6-a9ec	1	A
b835df12-a1c2-4d11-badc	1	B
b9f11453-2705-42c4-8bfe	1	C
3fc6f3d4-137c-45bd-a827	1	D
a2dd34b6-cff5-46d9-974b	1	E

F1

itemsetid uuid	pos integer	item character(10)
32138f1e-3743-42e6-a9ec	1	A
b835df12-a1c2-4d11-badc	1	B
b9f11453-2705-42c4-8bfe	1	C
a2dd34b6-cff5-46d9-974b	1	E

C2

itemsetid uuid	pos integer	item character(10)
0f7a74a4-b706-4414-a614	1	B
0f7a74a4-b706-4414-a614	2	C
51b3084f-fd76-4fce-8516	1	A
51b3084f-fd76-4fce-8516	2	C
7528e2a7-5d8a-41ef-9c52	1	C
7528e2a7-5d8a-41ef-9c52	2	E
c5decf65-00f7-458e-8d0b	1	A
c5decf65-00f7-458e-8d0b	2	B
ca5ca8e3-bf7a-4d55-af29	1	B
ca5ca8e3-bf7a-4d55-af29	2	E
cf62eaf1-7c40-4d52-aafc	1	A
cf62eaf1-7c40-4d52-aafc	2	E

F2

itemsetid uuid	pos integer	item character(10)
0f7a74a4-b706-4414-a614	1	B
0f7a74a4-b706-4414-a614	2	C
51b3084f-fd76-4fce-8516	1	A
51b3084f-fd76-4fce-8516	2	C
7528e2a7-5d8a-41ef-9c52	1	C
7528e2a7-5d8a-41ef-9c52	2	E
ca5ca8e3-bf7a-4d55-af29	1	B
ca5ca8e3-bf7a-4d55-af29	2	E
cf62eaf1-7c40-4d52-aafc	1	A
cf62eaf1-7c40-4d52-aafc	2	E

C3

itemsetid uuid	pos integer	item character(10)
626f6d1b-de41-4462-8bba	1	A
626f6d1b-de41-4462-8bba	2	C
626f6d1b-de41-4462-8bba	3	E
8a2a8beb-53b5-479a-9a20	1	B
8a2a8beb-53b5-479a-9a20	2	C
8a2a8beb-53b5-479a-9a20	3	E

F3

itemsetid uuid	pos integer	item character(10)
626f6d1b-de41-4462-8bba	1	A
626f6d1b-de41-4462-8bba	2	C
626f6d1b-de41-4462-8bba	3	E
8a2a8beb-53b5-479a-9a20	1	B
8a2a8beb-53b5-479a-9a20	2	C
8a2a8beb-53b5-479a-9a20	3	E

(b) Array representation

The greatest advantage of using arrays or json objects in SQL is that it reduces the overhead of data preprocessing and makes it easier to store the data. In the vertical table approach, I had to decompose the transactions data into each row, which contains each item. However by using arrays or json objects, we could import all the items for a particular transaction in a single row, yet perform the same operations.

PostgreSQL supports the usage of arrays and json objects and provides convenient ways to unnest the data when required. So all I had to do was to import the data in the format of arrays, unnest it to perform the support counting and candidate generation operations and nest it back to the array structure. I have created a stored procedure named '*frequent_itemset_mining_array()*' which performs the required operations to generate three most frequent items bought together. Below is the sequence of results -

tid integer	items text[]
100	{A,C,D}
200	{B,C,E}
300	{A,B,C,E}
400	{B,E}
500	{A,C,E}

C1

itemsetid uuid	items text[]
d6cbbe5b-bb4f-439d-99bd	{A}
6076797e-9304-4b24-89d8	{B}
cf0e7610-aaf8-4f56-b5e2	{C}
9baa85d4-a7b0-4b98-a094	{D}
fbf77ec3-520e-4499-bb58	{E}

F1

itemsetid uuid	pos integer	item character(10)
32138f1e-3743-42e6-a9ec	1	A
b835df12-a1c2-4d11-badc	1	B
b9f11453-2705-42c4-8bfe	1	C
a2dd34b6-cff5-46d9-974b	1	E

C2

itemsetid uuid	pos integer	item character(10)
0f7a74a4-b706-4414-a614	1	B
0f7a74a4-b706-4414-a614	2	C
51b3084f-fd76-4fce-8516	1	A
51b3084f-fd76-4fce-8516	2	C
7528e2a7-5d8a-41ef-9c52	1	C
7528e2a7-5d8a-41ef-9c52	2	E
c5decf65-00f7-458e-8d0b	1	A
c5decf65-00f7-458e-8d0b	2	B
ca5ca8e3-bf7a-4d55-af29	1	B
ca5ca8e3-bf7a-4d55-af29	2	E
cf62eaf1-7c40-4d52-aafc	1	A
cf62eaf1-7c40-4d52-aafc	2	E

F2

itemsetid uuid	pos integer	item character(10)
0f7a74a4-b706-4414-a614	1	B
0f7a74a4-b706-4414-a614	2	C
51b3084f-fd76-4fce-8516	1	A
51b3084f-fd76-4fce-8516	2	C
7528e2a7-5d8a-41ef-9c52	1	C
7528e2a7-5d8a-41ef-9c52	2	E
ca5ca8e3-bf7a-4d55-af29	1	B
ca5ca8e3-bf7a-4d55-af29	2	E
cf62eaf1-7c40-4d52-aafc	1	A
cf62eaf1-7c40-4d52-aafc	2	E

C3

itemsetid uuid	pos integer	item character(10)
626f6d1b-de41-4462-8bba	1	A
626f6d1b-de41-4462-8bba	2	C
626f6d1b-de41-4462-8bba	3	E
8a2a8beb-53b5-479a-9a20	1	B
8a2a8beb-53b5-479a-9a20	2	C
8a2a8beb-53b5-479a-9a20	3	E

F_Final

itemsetid uuid	items text[]
17c656db-363b-438b	{A,C,E}
a5b7d6b7-4f3d-408b	{B,C,E}

The same results can be achieved using json objects instead of arrays, without much change in the code.

7. INDEXES and RUNTIME

The unnest operation with arrays unwraps the data into individual rows and can be used further for normal computation. The example shown in the report is for a small dataset, however the unnest operation can be computationally expensive on large datasets. Hence I decided to create an index on the array column(*items* in our case) and tried to compare the runtimes with and without an index. An index can be created as -

create index array_index on T_array using GIN(items);

The above command create an index on the *items* column of our transaction table, which is of an array datatype. Below is the comparison of runtimes before and after creation of index -

Before index creation

QUERY PLAN text
Result (cost=0.00..0.26 rows=1 width=0) (actual time=591.462..591.463 rows=1 loops=1)
Planning time: 0.028 ms
Execution time: 591.485 ms

After index creation

QUERY PLAN text
Result (cost=0.00..0.26 rows=1 width=0) (actual time=343.992..343.992 rows=1 loops=1)
Planning time: 0.030 ms
Execution time: 344.015 ms

As we can see, the runtime reduced by almost 40% after creating indexes. This is however on a small dataset but results show that creation of indexes can reduce the query runtime by a large amount while working on huge data.

8. SYSTEM USED

All the experiments were performed with PostgreSQL 9.5.4 running on EnterpriseDB server, installed on a PC with Intel Core i5 processor, 8GB main memory and Windows 7 operating system.

9. CONCLUSION and FUTURE WORK

In this project, I got an opportunity to investigate the problem of frequent itemset discovery and different solutions using the vertical table approach and array representation.

There are two main approaches to represent transactions and itemsets - vertical table layout, where an object spans as many rows as its number of items, and an array representation of items for each transaction. The advantage of using array representation is to make the data processing easy and work with variety of datatypes. Creation of indexes on specific columns can also speed up processing.

I worked on a small dataset to implement different approaches but more performance experiments are needed to get a more detailed picture of the potential of array or json representation and its competitiveness with other SQL-based approaches.

This project can be extended to work with huge datasets with varying type of data items. With arrays and json objects, PostgreSQL becomes a powerful relational database engine and in my opinion can outperform even noSQL databases. I believe that similar approaches based on a natural representation of the mining problem in SQL will narrow the gap between data mining algorithms and database systems.

References

[1] R. Agrawal, T. Imielinski, A. Swami: Mining Association Rules between Sets of Items in Large Databases. Proceedings SIGMOD, Washington DC, USA.

[2] A. Netz, S. Chaudhuri, U. Fayyad, J. Bernhardt: Integrating Data Mining with SQL Databases: OLE DB for Data Mining. ICDE, Heidelberg, Germany.

[3] Ralf Rantzau: Frequent Itemset Discovery with SQL using Universal Quantification. Breitwiesenstr, Stuttgart, Germany