Implementation of the Apriori algorithm for effective item set mining in VigiBase™

Project report in teknisk fysik 15 hp

Niklas Olofsson
# Table of Contents

1. BACKGROUND 3

2. ASSIGNMENT AND PURPOSE 3

3. QUESTIONS REGARDING THE ASSIGNMENT 4

4. THEORY 4
4.1 The principle of the aPriori Algorithm 4

5. METHOD 8
5.1. IMPLEMENTATION IN C# 9
5.2 USER INSTRUCTIONS FOR THE C# IMPLEMENTATION 15
5.3 IMPLEMENTATION IN SQL 16
5.4 USER INSTRUCTIONS FOR THE SQL IMPLEMENTATION 19

5. RESULTS 20

6. DISCUSSION 26
6.1 IMPROVEMENTS OF THE C# IMPLEMENTATION 26
6.2 IMPROVEMENTS OF THE SQL IMPLEMENTATION 27

7. REFERENCES 27

8. APPENDIX 28
APPENDIX 1. 28
APPENDIX 2. 28
Abstract

Implementation of the Apriori algorithm for effective item set mining in VigiBase™

Niklas Olofsson

The assignment was to implement the Apriori algorithm for effective item set mining in VigiBase™ in two different ways. First via an application program written in C# and secondly directly in the database management system. The purpose is to compare the methods efficiency of finding association rules in large amounts of data with respect to execution times and memory consumption and also to list the large item sets. The results are unanimity and shows that the SQL implementation excels the C# implementation in every area as of today.
1. Background

The Apriori algorithm is a classical data mining method for association rule discovery typically applied to market basket data, such as the study of what products tend to be purchased together in an on-line market place (e.g. Amazon etc). VigiBase™ contains 5 million reports of suspected adverse drug reaction (ADR) incidents from across the world. Each report may contain a multitude of different information, among other things lists of drugs taken by the patient and adverse events experienced. Drugs and adverse reactions are encoded using standard terminologies, and can in this context be summarised in terms of binary variables indicating their absence or presence on a given report – much like a transaction in market basket analysis. Thus, the drugs and the ADR terms listed on each report could be seen, separately or together, as item sets where association rule learning would be of interest. Associations of interest could be between drugs (drugs that tend to be used together), between ADRs (related symptoms that tend to occur together or even form a syndrome) but also between groups of drugs and ADRs. Among other things, this can be used to identify data quality problems related to large numbers of very similar reports entered into the database. As the data resides on a MS SQL server and the results should ideally be stored within the same environment, a database-near implementation is strongly preferred.

Vigibase™ is a unique collection of international drug safety data. The data is available in a wide range of services, from advanced analysis to basic case report retrieval.

Vigibase™ is a product developed by Uppsala Monitoring Centre (UMC) which is the WHO collaborating centre for international drug monitoring and was formed 1968. UMC offers medical terminologies and analytical services for the international community of pharma, biotech and CRO companies, as well as academia and software developers.

2. Assignment and Purpose

The assignment was to implement the Apriori algorithm for effective item set mining in VigiBase™ in two different ways. First via an application program written in C# and secondly directly in the database management system. The purpose is to compare the methods efficiency of finding association rules in large amounts of data.
3. Questions regarding the assignment

1. Which of the two implementation methods is most efficient with respect to execution times?
2. Are the memory requirements for the two methods of the same magnitude?
3. List the largest item sets appearing for a few different support thresholds.

4. Theory

The apriori algorithm is a classic algorithm for learning association rules. Association rule learning is an popular method for discovering relations between variables in large databases. An example of association rule learning are the rule {Tshirt,Jeans} => {Shoes} found in the sales data in a clothing store. The rule indicates that customers that are buying both tshirt and jeans also are likely to buy shoes. Association rules are used to show the relationships between data items. The relationships are not inherent in the data as with functional dependencies and they do not represent any sort of causality or correlation. Instead association rules detect common usage of items[1].

4.1 The principle of the aPriori algorithm

**Definition 1**: The support of an item (or set of items) is the number of transactions in which that item (or items) occur[1].

Given a set of transactions in a database where each letter corresponds to a certain product such as Jeans or T-shirt and each transaction corresponds to a customer buying the products A, B, C or D the first step in the apriori algorithm is to count the support (number of occurrences) of each item separately.

<table>
<thead>
<tr>
<th>Transactions</th>
<th>Items</th>
<th>Item</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>A, B, C, D</td>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>T2</td>
<td>B, C, D</td>
<td>B</td>
<td>6</td>
</tr>
<tr>
<td>T3</td>
<td>B, C</td>
<td>C</td>
<td>4</td>
</tr>
<tr>
<td>T4</td>
<td>A, B, D</td>
<td>D</td>
<td>5</td>
</tr>
<tr>
<td>T5</td>
<td>A, B, C, D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T6</td>
<td>B, D</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Table 1*  

The items in the transactions represented in Table 1 has their support represented in Table 2.

**Definition 2**: The support threshold is defined by the user and is a number for which the support for each item (or items) has to be equal or above for the support threshold to be fulfilled[1].
In this example we will use support threshold = 3. This means that in our example all items in table 2 meets this condition since none of them have a support below 2 as seen in Table 2.

**Definition 3:** Given a set of items \( I = \{I_1, I_2, \ldots, I_n\} \) an **item set** is a subset of \( I \).

**Definition 4:** A **large item set** is an item set whose number of occurrences in the transactions are above the support threshold. We use the notation \( L \) to indicate the complete set of large item sets.

In our example the complete set of large itemset \( L \) in this first iteration is \( L = \{A, B, C, D\} \) since all of these terms meets the support threshold. If any of these items had been below the support threshold they had not been included in the subsequent steps. In the next steps we will form all pairs, triples and so on of the items in Table 2. If \( A \) would have a support threshold below three all pairs, triples etc containing \( A \) would also be below the support threshold. This is the fundamental basis of the apriori algorithm since it allows us to prune all transactions having only items under the support threshold, hence reducing the amount of data in each step.

The next step is to form all 2-pair item sets. We do this by making all possible combinations of the large item sets without regarding the order. Note that the number of possible combinations in this first joining step in given by the binomial coefficient \( \binom{n}{k} \).

<table>
<thead>
<tr>
<th>Item</th>
<th>Support</th>
<th>Large itemset</th>
</tr>
</thead>
<tbody>
<tr>
<td>A, B</td>
<td>3</td>
<td>A, B</td>
</tr>
<tr>
<td>A, C</td>
<td>2</td>
<td>A, B</td>
</tr>
<tr>
<td>A, D</td>
<td>3</td>
<td>A, D</td>
</tr>
<tr>
<td>B, C</td>
<td>4</td>
<td>B, C</td>
</tr>
<tr>
<td>B, D</td>
<td>5</td>
<td>B, D</td>
</tr>
<tr>
<td>C, D</td>
<td>3</td>
<td>C, D</td>
</tr>
</tbody>
</table>

*Table 3.*

*Table 4.*

In table 3 the new item sets are illustrated together with respective support. The item set \( A, C \) only have support 2 and since our support threshold is 3 the item set is not a large item set.

Next we generate the 3-sets by joining the full set of large item sets in table 4 over a common item.
The only 3-set that fulfills the support threshold is \{A, B, D\} and \{B, C, D\} as illustrated in table 6. If we continue this process by joining the item sets in the complete large item set over a common pair we get the last possible combination.

The last item set is \{A, B, C, D\} and occurs only two times, hence it is not fulfilling our support threshold. No large itemset is generated and this ends the algorithm.

The process of joining terms in the aPriori algorithm is illustrated in figure 1. Note that the position of a item in the item set doesn't matter. I.e the item set \{A, B, D\} is regarded in the same way as \{D, A, B\} and to keep track of this so we don’t get any redundancies later in the implementation all items in each item set is ordered by its value.
The apriori algorithm cuts some of the branches in the tree in figure 1, for example the item set \{A,C\} did only occur 2 times which was below the support threshold at 3. The apriori algorithm makes use of this by not generating any branches from this node and thus reduces the computational cost. This is as said the foundation of the apriori algorithm.

We can summarize all the steps done in pseudo-code.

\[
\text{Input:} \\
I \ // \text{Itemsets} \\
D \ // \text{Transactions} \\
S \ // \text{support threshold} \\
\text{Output:} \\
L \ // \text{large itemsets} \\
\]

**aPriori algorithm**

\[
k = 0 \ // k \text{ is used as the scan number} \\
L = \emptyset \\
C_1 = I \ // \text{Initial candidates are set to be the items} \\
\text{repeat} \\
k = k + 1 \\
L_k = \emptyset \\
\text{for each } I_i \in C_k \text{ do} \\
\quad c_i = 0 \ // \text{Initial counts for each itemset are 0} \\
\text{for each } t_j \in D \text{ do} \\
\quad \text{if } I_i \subseteq t_j \text{ then} \\
\quad \quad c_i = c_i + 1 \\
\quad \text{for each } I_i \in C_k \text{ do} \\
\quad \quad \text{if } c_i \geq S \text{ do} \\
\quad \quad \quad L_k = L_k \cup I_i \\
\quad \quad \quad L = L \cup L_k \\
\quad \quad \quad C_{k+1} = \text{aPriori-Gen}(L_k) \\
\quad \text{until } C_{k+1} = \emptyset \\
\]

Algorithm 1. aPriori algorithm[1]

\[
\text{Input:} \\
L_{i-1} \ // \text{Large itemsets of size i-1} \\
\text{Output:} \\
C_i \ // \text{Candidates of size i} \\
\]

**aPriori-Gen algorithm**

\[
C_i = \emptyset \\
\text{for each } I \in L_{i-1} \text{ do} \\
\quad \text{for each } J \neq I \in L_{i-1} \text{ do} \\
\quad \quad \text{if i-2 of the elements in } I \text{ and } J \text{ are equal} \\
\quad \quad \quad \text{then} \\
\quad \quad \quad \quad C_k = C_k \cup \{I \cup J\} \\
\]

Algorithm 2. aPriori-Gen algorithm[1]

Notice that the algorithm is divided into two, first the aPriori algorithm itself and then the aPriori-Gen algorithm which is used to generate the new candidate item sets from the large item set. The complete set of candidate item sets have notation \( C \).
In situations with prolific frequent patterns, long patterns or a relatively low support threshold, the algorithm suffers (depending on the transaction tables properties) from the following non-trivial costs[2].

- It is costly to handle a huge amount of candidate item sets.
- It is tedious to repeatedly scan the transaction table and check a large set of candidates by matching.

5. Method

VigiBase™ resides on an MS SQL server and the Apriori algorithm will be applied in two different ways. First via an application-program written in C# and then an implementation directly in the database management system.

Since Vigibase™ consists of around 5 million reports and around 20 million tuples it is important to implement the algorithm as memory efficient as possible. A key question is how to load and represent the database data in the application program since this will have big effects regarding the efficiency. A good implementation will minimize the amount of data that needs to be represented and the number of operations performed on the data.

The data that we are interested in resides on three tables within the database and are illustrated in figure 2.

<table>
<thead>
<tr>
<th>VmDrug</th>
<th>VmReports</th>
<th>VmAdr</th>
</tr>
</thead>
<tbody>
<tr>
<td>VmDrug_Id</td>
<td>int</td>
<td>Report_Id</td>
</tr>
<tr>
<td>Report_Id</td>
<td>int</td>
<td>VmAdrTerm_Id</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Report_Id</td>
</tr>
</tbody>
</table>

*Figure 2. The principal database schema*

The VmDrug_Id represents a certain drug, the Report_Id represents a unique report and VmAdrTerm_Id represents adverse drug reactions. By implementing the apriori algorithm we hope to find association rules like drugs that tend to be used together, symptoms that tend to occur together and also relations between groups of drugs and adr terms.
5.1. Implementation in C#

In figure 3 an Uml schema over the final implementation is illustrated. The two classes Combinations and ElementSet are downloaded from the development resource site CodeProject\(^1\) and are used for making the unique combination of elements as illustrated in figure 3, because of this these two classes will only be reviewed superficially.

![Uml schema of the c# implementation](image)

**Figure 3. Uml schema of the c# implementation**

The RetrieveData class loads the data from the database and performs the first step of the algorithm before sending the data to the aPriori class which contains the algorithm. The Interface class has a visual interface so the user can change different parameters like the support threshold or the SQL query.

The overall thought of the implementation is to throw away as much unnecessary data as soon and as much as possible, this is the "pruning" mentioned in the theory section. The less data that needs to be represented, the less memory is needed and the faster the algorithm will work. Another principle is to minimize the number of loops in the data.

The implemented algorithm will be explained with a simple example together with the functionality of key methods as well as key choices of implementation design.

---

The first thing we need to do is to load the data from the database and represent it in some manner in our application program. This is done by the class RetrieveData and more specifically by the method getData() as seen in figure 3. Via the interface the user can write their own SQL questions but let us assume that he or she wrote the SQL expression below.

**SELECT Report_Id as Id, VmAdrTerm_Id as Term_Id from test.dbo.testTable order by Report_Id;**

The result of this SQL query is illustrated in figure 4.

![Figure 4. Transaction table](image)

As seen in figure 4 each transaction Id can contain several Term Id:s. Each row is however unique. To load the data we iterate through the whole transaction table and group the Term_Ids by corresponding Id. The Id:s are stored separately in a List name report_Id at a certain position and the Term_Id: s are stored as an array at the same position (the same index) as the corresponding report_Id in a List named D. D is our transaction list and is a representation of the transaction table. In our example after we have iterated through all rows in figure 4 we have the following state.

<table>
<thead>
<tr>
<th>report_Id</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{16, 305}</td>
</tr>
<tr>
<td>1</td>
<td>{42, 348, 2916}</td>
</tr>
<tr>
<td>2</td>
<td>{348, 2916}</td>
</tr>
</tbody>
</table>

**State 1.**

Now we need to count each individual term in D since only terms that meets the support threshold are needed for consecutive iterations of the algorithm. This is done by the method countItems() which are storing the number of occurrences in a Dictionary named C where the item set (Term_Id) is used as the key and the value for for each key is the number of unique (with respect to Id) occurrences. Note that this is done simultaneously in the implementation as we are looping through the database to store the values in D to minimize the number of loops. C is having the following state after we have iterated through the rows in figure 4.
State 2.

Now the database data is represented in the application program and we can start using our algorithm. In fact we have already started since we have counted the support for each item. Before we transfer the data to the aPriori class we can prune the transaction table to minimize the amount of data processed in the algorithm. The method `deleteRowsInDOnSupport()` deletes transactions not containing any large item set and the method `deleteRowsInDOnEntries()` deletes transaction with only one item. If a transaction only contains one item it can’t possibly have a large 2-set item set. In this example however we will use a support threshold at 2 which means that the row with index 0 in D (see state 1) will be removed because it only contains items that doesn’t meet the support threshold. The corresponding Id in report_Id (see state 1) will also be removed.

After the pruning we have all information needed for the aPriori algorithm. The methods `getD()`, `getReport_Id()` and `getL()` transfers the data to the aPriori class. The `getL()` method iterates through C in state 2 and store the items that meets the support threshold in a list named L before transferring it.

State 3.

In state 3 L is our first large item set. This first step of the algorithm done by the RetrieveData class differs somewhat from consecutive steps mainly because our candidate item sets only consists of one term. To recap what we done so far, we have first loaded the data from the database and represented it in an suitable manner. At the same time we counted each Term_Id’s support. Transactions in D not containing a supported item or transactions only containing one item are deleted together with corresponding report_Id. The large item sets are generated and are together with the pruned transaction list D and the report_Id transferred to the aPriori class.
The first method in the aPriori class is generateC() which as the name suggests generates the candidate item sets. The method is implemented accordingly to the apriori-Gen algorithm described in the theory section. In this second step of the algorithm it takes the large item sets \( L \) in state 3 and generates all possible 2-sets that can be combined. In later steps it will take all large 2-set item sets and generate all possible 3-sets, all large 3-set item sets to generate all possible 4-sets and so on. If we apply generateC() to our variable \( L \) we get

\[
C = \begin{pmatrix}
{348, 2916} & 0
\end{pmatrix}
\]

where as before our generated pairs (and in subsequent steps 3-sets, 4-sets etc) are stored as keys in a Dictionary named \( C \). Since \( L \) only consisted of two terms the only possible combinations without regard to order is \{1, 2\} and thus \( C \) only consists of one row. Each candidate set is stored as an array and this causes problems when we want to use this array as a key in \( C \). To solve this the array is converted to a comma-separated string by the method makeKeyStringOfCandidate() and this generated string is used as the key in \( C \) instead of the array.

Next step in algorithm is to count the support. This is done by the method calculateSupport() which uses the classes Combinations and ElementSet. The method calculateSupport() iterates through all rows in \( D \), for every row in \( D \) all the items are transferred to the class Combinations which returns all possible combination of a given size without regard to order. The combinations are returned as an array which are converted to a string. If this string exists as a key in \( C \) then we increase the first value by one and also adds the row number from \( D \) which had this combination.

<table>
<thead>
<tr>
<th>Combinations</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 {42, 348}</td>
<td>{348, 2916}</td>
</tr>
<tr>
<td>1 {42, 2916}</td>
<td>{2, 0, 1}</td>
</tr>
<tr>
<td>2 {348, 2916}</td>
<td></td>
</tr>
<tr>
<td>3 {348, 2916}</td>
<td></td>
</tr>
</tbody>
</table>

As seen in state 5 the only combinations that exists in \( C \) is \{348, 2916\} and those combinations comes from row 3 and 4 in \( D \), i.e. the rows with indexes 2 and 3. The first value in \( C \) counts the support for each item and the following values are the rows in \( D \) that contains the item.
We can now generate our new large item set but before we do that we prune our variables. To do this we loop through each row in C and store each row number from D that meets the support in a new variable called markTransacts as a key with value true. This is done by the method markTransactList().

<table>
<thead>
<tr>
<th>markTransacts</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

*State 6.*

In markTransacts we have stored the row number from D which contains item that meets the support threshold. The rows not included in this variable are not interesting any more and we can therefore prune our variables D and report_Id. The method deleteRowsInDOnSupport() deletes all rows in D and in report_Id not included in the markTransacts. In this example both rows in D has support for {348, 2916}, however in the next iteration of the algorithm we have to generate all 3-sets. This means that only rows in D containing at least three items can come into question for having a supported set. Thus we can delete all rows in D containing two items or less and the corresponding report_Id. This is done by the method deleteRowsInDOnEntries() and gives in our example the following state.

<table>
<thead>
<tr>
<th>report_Id</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{10}</td>
</tr>
<tr>
<td>1</td>
<td>{42, 348, 2916}</td>
</tr>
</tbody>
</table>

*State 7.*

We now have all information needed for the third iteration of the algorithm except the complete set of 2-items set. The method getL() loops through C and stores the keys with support value that meets the support threshold.

<table>
<thead>
<tr>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

*State 8.*

We can now perform the third iteration but because L only consists of one row there is no possible way to generate 3-set candidates, hence the iteration stops and all complete sets of large item sets are illustrated below.
State 9.

Figure 5. Principal flow chart of the implemented \textit{aPriori} algorithm

In figure 5 a principal flow chart over the implementation is illustrated to clarify the process performed on the data by the implementation.
In figure 6 the graphical user interface (GUI) is illustrated. The user can change the SQL query and the parameters Support and MaxItem. The size of each complete item set is returned as well as the execution time.

5.2 User instructions for the C# implementation

As said, the user can change different parameters like the support threshold and maxItem in the user interface. Of most importance is however the query parameter. For the program to work the query needs to generate a two attribute relation with report_id to the left and the items to the right and the specified table used needs to exist in the database or else an exception will be thrown. If the user would like to make use of the application on more than one table, i.e. both the drug and the adr table, the user needs to make an union of these tables in the query field. See the example below.

```
SELECT Report_Id, ITEM1 as Term_Id from TABLE_1
UNION
(SELECT Report_Id, ITEM2 + shiftvalue as Term_Id from TABLE_2)
ORDER BY Report_Id
```

Note that the last SELECT query adds a shift-value to ITEM 2. If the values in ITEM1 and ITEM2 overlaps we need to shift one of the values so we get a distinction between the items. For example if we have an ADR term with value 4 and also a DRUG term with the same value
we need to separate them somehow because the number 4 has a different meaning with respect to the context. It is also important that the tuples are ordered by Report_Id.

The easiest way is however to make the query in the database management system and save the result to a table which the query then can refer to.

5.3 Implementation in SQL

A first simple implementation of the aPriori algorithm is quite straightforward and illustrated below. For each iteration k we generate new candidates $C_k$ by joining the previous large item set $L_{k-1}$ with itself. Then we count the occurrences of each candidate in the transactionlist and store only the candidates that meets the support threshold.

```
INSERT INTO C_k
SELECT L1.l1, …, L1.l_{k-1}, L2.l_{k-1}
FROM L_{k-1} AS L1, L_{k-1} AS L2
WHERE L1.l1 = L2.l1 AND …
    L1.l_{k-2} = L2.l_{k-2} AND
    L1.l_{k-1} < L2.l_{k-1}
```

```
INSERT INTO L_k
SELECT l1, …, l_k, count(*)
FROM C_k, T AS t1, …, T AS t_k
WHERE t1.item = C_k.item1 AND …
    t_k.item = C_k.item_k AND
    t1.Tid = t2.Tid AND …
    t_{k-1}.Tid = t_k.Tid
GROUP BY item1, …, item_k
HAVING count(*) >= support
```

Algorithm 3. Suggested SQL implementation of aPriori algorithm

As it turns out this implementation is very slow even with extensive use of indexes. When counting the support k transaction tables are created. The transaction tables contains lot of tuples of no use in iterations later than the first. Perhaps we can modify this algorithm and thus minimizing the information stored in main memory hence reducing the number of operations needed in each iteration. The idea is to minimize the use of the complete transaction table and instead work only with the tuples containing items that meets the support threshold. Consider the transaction table in figure 7. What if we only generate the candidates that are possible over each Id? That is, the only possible 2-set candidates for Id = 1 is \{1,2\}, \{1,3\}, \{2,3\} and for Id = 2 it is \{1,2\}, \{1,4\}, \{2,3\}. This means that the candidate \{3,4\} that had been generated in algorithm 3 never gets generated in this proposal of algorithm.
The SQL query in table 9 generates the corresponding relation as illustrated in state 9. Note that R2’ contains all possible combinations of candidates joined over each Id.

```
INSERT INTO R2'
SELECT p.Id, p.Item, q.Item
FROM Transactions AS p, Transactions AS q
WHERE p.Id = q.Id AND q.Item > p.Item
```

<table>
<thead>
<tr>
<th>Transactions</th>
<th>R2’</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
<td>Item</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 9

State 9.

The next step is to count the support. If we in this example let the support threshold be 2 then the only 2-set candidates in R2’ that meets the support threshold is {1,2}. By counting the number of Id:s that each candidate have we get the support. This is done by the query in table 10 and the corresponding large item set are illustrated in state 10.

```
INSERT INTO C2
SELECT p.Item1, p.Item2, COUNT(*)
FROM R2’ AS p
GROUP BY p.Item1, p.Item2
HAVING COUNT(*) >= 2
```

<table>
<thead>
<tr>
<th>C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

Table 10

State 10.

We can now use the fact that not all of the tuples in R2’ have supported item sets. The query in table 11 stores the supported tuples illustrated in state 11.

```
INSERT INTO R2
SELECT p.Id, p.Item1, p.Item2
FROM R2’ AS p, C2 AS Q
WHERE p.Item1 = q.c1 AND p.Item2 = q.c2
```

<table>
<thead>
<tr>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

Table 11

State 11

The steps described can be summarized in pseudo-code as follows
\begin{align*}
  &k = 1 \\
  &C_1 = \text{generate counts from } R_1 \\
  \text{repeat} \\
  &\quad k = k + 1 \\
  &\quad \text{INSERT INTO } R_k' \\
  &\quad \quad \text{SELECT } p.\text{Id}, p.\text{Item}_1, \ldots, p.\text{Item}_{k-1}, q.\text{Item} \\
  &\quad \quad \text{FROM } R_{k-1} \text{ AS } p, \text{TransactionTable as } q \\
  &\quad \quad \text{WHERE } q.\text{Id} = p.\text{Id} \text{ AND} \\
  &\quad \quad \quad q.\text{Item} > p.\text{Item}_{k-1} \\
  &\quad \text{INSERT INTO } C_k \\
  &\quad \quad \text{SELECT } p.\text{Item}_1, \ldots, p.\text{Item}_k, \text{COUNT}(*) \\
  &\quad \quad \text{FROM } R_k' \text{ AS } p \\
  &\quad \quad \text{GROUP BY } p.\text{Item}_1, \ldots, p.\text{Item}_k \\
  &\quad \quad \text{HAVING COUNT}(*) >= \text{support} \\
  &\quad \text{INSERT INTO } R_k \\
  &\quad \quad \text{SELECT } p.\text{Id}, p.\text{Item}_1, \ldots, p.\text{Item}_k \\
  &\quad \quad \text{FROM } R_k' \text{ AS } p, C_k \text{ AS } q \\
  &\quad \quad \text{WHERE } p.\text{Item}_1 = q.\text{Item}_1 \text{ AND} \\
  &\quad \quad \quad \cdot \\
  &\quad \quad \quad \text{p.}\text{item}_{k} = q.\text{item}_{k} \\
  \text{until } R_k = \{\} \\
\end{align*}

**Algorithm 4. Final SQL implementation of the aPriori algorithm**

where transaction-table is the original database containing all transactions. This is the algorithm used in the results section for comparison with the C# implementation.
To clarify the SQL Sort-Merge join pseudo-code consider figure 8 where the three code snippets from table 9, table 10 and table 11 are being performed as long as the complete large item set is non-empty. T is the transaction-table. Compared to the flow chart of the C# schema this schema have lesser steps and the pruning process is more integrated.

5.4 User instructions for the SQL implementation

Because we have implemented a maxItem parameter in the C# version we have also applied this to the SQL implementation. We extract the attributes we are interested in and counts the number of items for each report. This is done by the SQL code below.

```sql
declare @support int;
declare @maxItem int

set @maxItem = 10;
set @support = 10;
```
CREATE TABLE #R1(Report_Id int not null, Item int not null);
INSERT INTO #R1
SELECT Distinct Report_Id, Item from Table
WHERE Report_Id IN (
SELECT Report_Id from Table
GROUP BY Report_Id
HAVING COUNT(Item) <= @maxItem
)

Only reports that have under or equal @maxItem are allowed to populate the transaction table #R1 used by the algorithm and more precisely the code in table 9. We are also declaring the support threshold as a variable so that, if we want to change the support value only needs to do this at one place and not in every iteration. For the algorithm itself it is just to apply the code in algorithm 4 for each step in the iteration and replace support with the variable @support. Tentatively maxItem number of times.

5. Results

We have examined the two methods of implementation with respect to the stated issues. The system for which this has been done is a HP ProLiant DL385, 2 x AMD Opteron 280 2.4 GHz Dualcore, 9 Gb RAM, disk: 36Gb, 3 x 146Gb, 2 x 300Gb. There are two tables of interest, the table containing the Adr terms and the table containing the Drug terms (see figure 2). The Adr table contains 10182286 tuples and 4759960 reports which gives a mean of 2,14 Adr Terms per report. The Drug table contains 5681941 tuples and the same amount of reports which gives a mean of 1,19 drugs per Report. Because of the difference in the mean we can expect that the efficiency of the algorithm will differ for the same amount of tuples between the two tables.

When running tests on the data we have discovered that some of the reports contain lots of Adr and Drug terms. Some reports have around 60 Adr terms and this causes problems for the two implementation methods, primarily for the C# version. In both methods we have to generate all possible combinations of items for each report. This is done by the class combinations in the C# version and by the SQL code in table 9. For example say that we have 50 Adr terms and we want to generate all combinations containing 7 items without regard to order. The number of combinations is then given by \( \binom{50}{7} = 99884400 \). This is a considerable amount of combinations and slows the algorithms significantly. The SQL code
however uses combinations generated in previous steps when generating new ones and should thus be more tolerant. Reports containing these amount of terms are however of minor interests because it is hard to analyze a large item set containing 50 items. The value of a association rule in this specific case decreases when the size of it is large because of increased problems to determine causality. Because of this we have made a user controlled limitation on how many terms a report can contain to be included in the data processed by the algorithm, see figure 6. This limitation will be referred to as max items.

First we examine the execution times for the two methods on the Adr and the Drug tables. To avoid dependencies in fluctuations of the data composition the tuples are randomly selected.

This is done by the SQL code below which extracts the number of reports we are interested in randomly in a temporary table named #reports. After that, we retrieve all items we are interested in corresponding to the reports and saves them together with the reports in a table named tempTable. As said this is illustrated in SQL code below where we in this example are interested of both drug and adr terms, hence the UNION ALL statement. Note that the Drug terms are already shifted by 100000 in this step for the terms to be non overlapping.

```
CREATE TABLE #reports(Report_Id int)
INSERT INTO #reports
SELECT DISTINCT top NrOfReports Report_Id
FROM Reports
ORDER BY NewId();

CREATE TABLE (Report_Id int, VmAdrTerm_Id int)
INSERT INTO tempTable
SELECT DISTINCT B.Report_Id, B.VmAdrTerm_Id
FROM #reports AS A, A
   adrTerm AS B, AdrTerm_lx AS C
WHERE A.Report_Id = B.Report_Id AND
    B.VmAdrTerm_Id = C.VmAdrTerm_Id AND
    C.VmAdrType_Id = 1
UNION ALL
SELECT Distinct A.Report_Id, A.VmDrug_Id+10000 from Drug AS A, #reports AS B
WHERE B.Report_Id = A.Report_Id
ORDER BY Report_Id;
```
Note that we in the SQL code also checks each AdrType to that we only gets terms from the Adr table with the older denotation. When running the C# version the interface query has been directed to the tempTable.

The support threshold are chosen to 10 and max items are also chosen to 10 for examination of the execution time.

![Graph](image1)

*Figure 9. Execution time as a function of the number of million randomly selected tuples from the Adr table with Support = 10, max Item = 10. Measured data is tabulated in Appendix 1. Note that the C# values are an average of multiple runs.*

![Graph](image2)

*Figure 10. Execution time as a function of the number of million randomly selected tuples from the Drug table with Support = 10, max Item = 10. Measured data is tabulated in Appendix 1.*
As we can see from figures 9 and 10 the SQL implementation is much faster than the C# version. This can partly be explained by the fact that the C# implementation is not parallelized, but the difference in execution time in figure 9 for large amount of tuples is too large to only be explained by this. Probably this is due to the difference in candidate item sets generation and the calculation of support where the SQL version uses the previous candidates with support to generate new candidates as discussed earlier and doesn’t need to generate all possible combinations from each row in D as with case of the C# version.

Another thing to examine is how the two methods react to different supports and max items. The execution time is expected to decrease when increasing the support or decreasing the max item value. The question is how much, and will the sensibility of this differ for the two implementation methods?

Figure 11. Execution time as a function of the support threshold. The test has been done on 2 million randomly selected tuples from the Adr table. Measured data is tabulated in Appendix 1.
Figure 12. Execution time as a function of the max Item threshold. The tests has been done on 2 million randomly selected tuples from the Adr table. Mesured data is tabulated in Appendix 1.

In figure 11 we have plotted the execution time as a function of the support threshold. When the support threshold is low the C# version runs significantly slower whilst when the support threshold is greater the C# version has a tendency to approach the SQL implementations execution times. Notable is that the SQL implementation almost is unaffected by the change in support.

In figure 12 the execution time is plotted as a function of the maximum amount of terms in a Report. Again the SQL implementation reacts very little to changes whilst the C# implementation almost have a exponential growth. This is probably due to the calculation of the support where all possible combinations of terms from a report is generated and then compared with the generated candidate item sets. For example all possible combinations of 9 terms of size 5 is 126 whilst it is 6188 for 17 terms of the same size. This means that the behavior of the C# version due to this is expected.
Another question is how much main memory the two methods consume.

![Graph showing memory usage as a function of million randomly selected tuples from the Adr table with Support = 10, max Item = 10. Mesured data is tabulated in Appendix 1.]

Figure 13. Main memory usage of the C# version as a function of million randomly selected tuples from the Adr table with Support = 10, max Item = 10. Mesured data is tabulated in Appendix 1.

In figure 13 the memory usage is illustrated as a function on million tuples. The system for which this test has been done is a Apple Macbook running Windows Xp with the use of Parallells with 1 GB main memory and a CPU at 2 GHZ dual core. As seen in the figure the amount of main memory is as expected increasing with incresing tuple. It is hard checking the actual memory usage of the SQL version since MS SQL allocates all available main memory regardless of the operations being done. This is an intended behavior of the SQL buffer pool. When running the C# implementation on 6 million tuples we get an out of memory exception, that is, the main memory has run out whilst the SQL implementation manages this amount of tuples. This is an indication of that the C# version is more memory demanding.

The third issue was to list the large item sets for a few different support values. The result of this is listed for some values on maxItem and support in appendix 2. One interesting fact that we have noticed is that some of the drugs in the generated association rules are already banned due to proven severe adverse drug reactions. This indicates that the aPriori algorithm have the capability of finding interesting association rules and therefore can be an important tool for UMC to use.

---

6. Discussion

In the result section figure 9, figure 10 and figure 13 shows that the implemented SQL version excels the C# implementation in both execution time and probably also in memory consumption. Another important observation is that the SQL version seems to be unaffected by changes in support and max allowed items as seen in figure 11 and figure 12. The C# implementation reacts strongly on changes in max Items due to the time and memory consuming design of the support calculation and this seems also to be the bottleneck of the implementation. It should be noted as discussed earlier that MS SQL uses parallelization and the C# implementation does not. This probably affects the results in large degree but it is nevertheless surprising that that the difference is so big. One could expect that a purpose build application would be better than the multi-objective MS SQL application but this is as said not the case. The C# version can however be improved in many ways.

6.1 Improvements of the C# implementation

First off we have the issue of parallelization. It would be interesting to see how much the implementation would be improved by applying this. There is however no time for constructing such an implementation.

In this version of the implementation the method makeKeyStringOfCandidates() takes an array of items and constructs a string item by item when looping through the array.

```csharp
for (int i = 0; i < array.Length; i++) {
    string += Convert.ToString(array[i]);
}
```

This method is used for all generated candidates as well as for every generated combination, i.e. it runs very many times. By construction the string in this way the string gets re-allocated under every iteration and thus very expensive to use. A better implementation perhaps is to use a StringBuilder with fixed size, however preliminary tests suggest that this is not the case. Probably this is due to the small number of iterations done in the for loop.

Since the SQL implementation has proved very effective perhaps an adaption to C# would be better than the existing implementation which follows algorithm 1 to large extent.
6.2 Improvements of the SQL implementation

The tests in the result section has been done without the use of any indexes. This could improve the algorithms efficiency. Indexes has been tested but without any significant effect. Perhaps with a indexing tool, more advanced indexes with effect could be placed.

In the SQL implementation the original transaction table is used in every iteration to construct the candidates for each report. Perhaps this table could be pruned as in the C# version in every iteration and thus increase the efficiency.

Instead or together with improvements of the implementations there is a number of extensions to the apriori algorithm for more effective mining. One of these are the sampling method which after each pass over the database uses simple statistics to limit the amount of data represented[3]. This could be an alternative implementation if execution time and memory consumption is of big importance.

To summarize the SQL implementation is as discussed more effective with respect to execution time and probably also to memory usage. There is however much that can be improved in the C# version. One could say that the SQL implementation is preferred to be used as of today but the C# could with more development time be the version that in the end becomes the most effective.

7. References


[2] X Shang, K-U Sattler, I Geist: Efficient frequent pattern mining in relational databases. 2004 Department of Computer Science, University of Magdeburg. LWA 2004: 84-91 P.O.BOX 4120, 39106 Magdeburg, Germany, Department of Computer Science and Automation, Technical University of Ilmenau {shang|geist}@iti.cs.uni-magdeburg.de, *kus@tu-ilmenau.de

8. Appendix

Appendix 1.

### Measured values from figure 9.

<table>
<thead>
<tr>
<th>#tuples</th>
<th>SQL [s]</th>
<th>C# [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>99489</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>499824</td>
<td>21</td>
<td>14</td>
</tr>
<tr>
<td>998469</td>
<td>31</td>
<td>38</td>
</tr>
<tr>
<td>1998873</td>
<td>47</td>
<td>103</td>
</tr>
<tr>
<td>2999647</td>
<td>56</td>
<td>195</td>
</tr>
<tr>
<td>3997555</td>
<td>70</td>
<td>316</td>
</tr>
<tr>
<td>4999932</td>
<td>84</td>
<td>533</td>
</tr>
<tr>
<td>5999339</td>
<td>97</td>
<td>726</td>
</tr>
<tr>
<td>6994494</td>
<td>110</td>
<td>1172</td>
</tr>
<tr>
<td>7996540</td>
<td>123</td>
<td>1792</td>
</tr>
<tr>
<td>8996965</td>
<td>134</td>
<td>2045</td>
</tr>
<tr>
<td>10182286</td>
<td>153</td>
<td>3276</td>
</tr>
</tbody>
</table>

### Measured values from figure 10.

<table>
<thead>
<tr>
<th>#tuples</th>
<th>SQL [s]</th>
<th>C# [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>100525</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>501488</td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>1002628</td>
<td>16</td>
<td>24</td>
</tr>
<tr>
<td>2005511</td>
<td>30</td>
<td>54</td>
</tr>
<tr>
<td>3009142</td>
<td>40</td>
<td>104</td>
</tr>
<tr>
<td>4013027</td>
<td>50</td>
<td>160</td>
</tr>
<tr>
<td>5681941</td>
<td>62</td>
<td>292</td>
</tr>
</tbody>
</table>

### Measured values from figure 11.

<table>
<thead>
<tr>
<th>Support</th>
<th>SQL [s]</th>
<th>C# [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>46</td>
<td>279</td>
</tr>
<tr>
<td>7</td>
<td>41</td>
<td>162</td>
</tr>
<tr>
<td>9</td>
<td>41</td>
<td>120</td>
</tr>
<tr>
<td>11</td>
<td>40</td>
<td>98</td>
</tr>
<tr>
<td>13</td>
<td>40</td>
<td>85</td>
</tr>
<tr>
<td>15</td>
<td>40</td>
<td>76</td>
</tr>
<tr>
<td>17</td>
<td>39</td>
<td>70</td>
</tr>
</tbody>
</table>

### Measured values from figure 12.

<table>
<thead>
<tr>
<th>max Item</th>
<th>SQL [s]</th>
<th>C# [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>26</td>
<td>48</td>
</tr>
<tr>
<td>7</td>
<td>34</td>
<td>79</td>
</tr>
<tr>
<td>9</td>
<td>39</td>
<td>97</td>
</tr>
<tr>
<td>11</td>
<td>42</td>
<td>119</td>
</tr>
<tr>
<td>13</td>
<td>44</td>
<td>146</td>
</tr>
<tr>
<td>15</td>
<td>47</td>
<td>182</td>
</tr>
<tr>
<td>17</td>
<td>52</td>
<td>236</td>
</tr>
</tbody>
</table>

### Measured value from figure 13.

<table>
<thead>
<tr>
<th>#tuples</th>
<th>C# [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>98215</td>
<td>50</td>
</tr>
<tr>
<td>488776</td>
<td>176</td>
</tr>
<tr>
<td>977403</td>
<td>218</td>
</tr>
<tr>
<td>1952176</td>
<td>460</td>
</tr>
<tr>
<td>2927611</td>
<td>600</td>
</tr>
<tr>
<td>3911242</td>
<td>704</td>
</tr>
<tr>
<td>4890390</td>
<td>895</td>
</tr>
</tbody>
</table>

Appendix 2.

Classified material.