UTRECHT UNIVERSITY

BACHELOR THESIS ARTIFICIAL INTELLIGENCE

A performance analysis of applied algorithms in Frequent Itemset Mining

Author:
Nina KRIJGER

Supervisor:
drs. Hans PHILIPPI

A 15 ECTS thesis submitted in fulfillment of the requirements for the degree of Bachelor of Science

in the

Faculty of Humanities

October 19, 2018

Contents

1 Introduction

Within the last few decades data mining has become very popular with academics and large companies. To be able to predict events and solve problems by extracting patterns from large datasets is becoming more realistic with the years. With information becoming more valuable and accessible, we keep improving our algorithms and hardware in order to maximize performance and thereby gather as many results in as little as time possible. And theoretically we are achieving this goal. In practice, however, we often do not know if one algorithm is better than the other. Testing newer algorithms, especially on large datasets, is sometimes neglected, mostly because there is not enough time or resources.

In this work, we will turn to an algorithm of which performance is supposed to exceed its predecessor's. It concerns a data-mining method, called Frequent Itemset Mining (FIM), used for extracting regularities from a dataset (usually containing purchasing records or transactions from a shop). The method is used for determining which items frequently occur together in a transaction, the result from which can be used in making marketing decisions or in choosing product placement.

The algorithm in question, Matrix Based Algorithm with Tags (MBAT), is supposed to work faster than the original FIM method, the Apriori algorithm (see 1.1). Testing this claim, will be the focus of this paper.
FIM is designed to work on datasets that are structured in tables (with a variable amount of columns), as most data-mining methods are. So in order to test the claim, we will be using a database management system (DBMS) where we will keep the data. For communicating with this DBMS we will use `SQL`, since this is the prevailing programming language for databases. We want our algorithms to be able to loop through methods and save results without taking up too much space (since this could affect our performance), therefore we will use `Java` to issue `SQL` queries and record time and temporary results. The actual data will be random, produced by a selfmade `Java` program.

Both algorithms require us to scan the dataset multiple times, this is consuming work and it matters how we access and process that data: one approach might benefit performance, whilst the other might hinder it. If we want to test which algorithm is faster, we need to test them on different platforms. That is the reason we are using two different DBMSs. Both have different ways of accessing and storing the data. Will we be able to use the same `SQL` queries for both DBMSs and if so, will performance of the `SQL` queries stay the same?

The aim is to find the combination of a DBMS and FIM algorithm that delivers the best performance, or better said: which combination finishes within the least amount of time.

Will the newly proposed MBAT be faster than the Apriori algorithm, no matter what DBMS we are using? Or does a DBMS have a significant positive or negative effect on performance, and if so, how?
And is the test conclusive, or are there more methods needed to determine what algorithm is faster?

The structure of this paper will be as follows: Firstly we will look at the two algorithms: How do they work? What is our input, what is our output? The chapter after will expand on this, by showing how we are going to represent our data. After which we discuss our implementation of both algorithms within both DBMSs, this leads to four combinations to consider. Lastly we will run the implementations and state our results, from where we can finally analyse our work and conclude which of these combinations delivers the best performance and if that result is conclusive.

The code will not be added to the report since this will make for a cluttered result. It will of course be available in the form of loose `Java` files.

Algorithms

It should be mentioned that we will only implement part of the FIM method, namely the finding of frequent itemsets. Generating association rules is not part of this research.

These are the algorithms we will use for the mining of frequent itemsets:

Apriori The Apriori algorithm is the classical algorithm in finding frequent itemsets (and generating association rules), the algorithm was introduced by Agrawal and Srikant, 1994. The upside of this algorithm is the lack of using brute-force. The downside is the generation of very large datasets and having to go through the same dataset multiple times, which causes heavy I/O spending, which in turn makes it slower.

MBAT The MBAT is the theoretically improved version of the Apriori algorithm proposed by Singh and Dhir, 2013. This algorithm represents the data in a transactional matrix. Supposedly this should work faster than the Apriori algorithm because it produces smaller datasets. This, however, has not yet been empirically verified by the authors themselves or by any other. As discussed before, this is what we will try to achieve in this paper.

Database Systems

For the DBMSs, we will use `MySQL` and `MonetDB`.

MySQL `MySQL` is a traditional row- and `SQL`-based database. It stores data per row. When issued a query it will brute-force its way through a table and goes from row to row to compare that row's column-values with the query.

MonetDB `MonetDB` is a column- and `SQL`-based database. As can be guessed, it stores data per column. This means whenever a query is parsed, the system will select the column (or columns) upon which is queried and compare the column-values with the values offered. As you can see, this is less intrusive since we do not necessarily have to access every row. Whether it is faster depends on the type of queries.

Combinations

The aforementioned algorithms and DBMSs give us the following combinations which we are going to implement (all make use of the object-oriënted programming language `Java` and query language `SQL`):

- `MySQL` and Apriori
- `MySQL` and MBAT
- `MonetDB` and Apriori
- `MonetDB` and MBAT

1.1 Context Artificial Intelligence

There is no question that Artificial Intelligence (AI) and data mining go hand in hand: AI techniques have proven to be quite fast for classifying data or using data as a learning method. But we are basically just talking about Machine Learning (ML), FIM and specifically *just* mining the frequent itemsets, the subject of this paper, is not a ML method and also not necessarily part of the AI family (Wu, 2004; Feyyad, 1996). What, however, does coincide, is the time and space (or lack thereof) issues that can be solved with intelligent solutions. In AI we use certain tricks to shorten the runtime of our algorithm, the same is done in data mining. For example, pruning. In the context of FIM, pruning is already a large contributor to shortening runtime and continues to be researched (Verma and Kumar, 2013). But what kind of pruning are we talking about here?
Take this next example: say we have a loop that goes on for 300 iterations but we already have the information we need after the 40^{th} iteration. Normally the loop would just finish the remainder of iterations and present you with that data, which is not very efficient. But if we put a check in every iteration, to see if we have reached the maximum amount of possible "answers", we could break out of the loop then and there. This would just give us 40 more steps, instead of 260*n (where n is the amount of steps in an iteration) more steps. Keep in mind that this is only efficient if we can find every possible "answer" before it reaches the 300^{th} iteration, else we have just added 300 extra steps without any reward.
Knowing how your data works and using pruning, belongs to data mining and AI methods. Handling big datasets and going through them in a fast manner will be heavily discussed in this paper, many of which can be implemented in AI methods as well.

2 Algorithms

We have discussed FIM earlier[1], here we will expand on what parts of the method we will use and what parts will not be included in the implementation and analysis. As has been mentioned, FIM consists of two elements. The first element is the algorithm we will use in our code. The second element (represented in *italic*) will not be used in this paper but is an application of the FIM method:

1. Itemset mining
   Where we extract all items from the transaction dataset and combine them with each other to generate every possible itemset. We will count the occurrence (support count σ) of all these itemsets and eliminate those where σ is not as high as the minimal support count (min_sup) i.e. the ones that are not frequent. This will leave us with only the *frequent* itemsets.

2. *Association rule mining*
   This step uses the frequent itemsets to generate rules of the form $X \rightarrow Y$, where X and Y are itemsets. The rule states that the itemset X implies itemset Y. Which means that it is likely that if X occurs, Y will also occur.
   There is more to these rules than explained here, but since we will not be implementing this part of the method, we can freely omit this from the report. More information is offered by the authors in their original paper (Agrawal and Srikant, 1994).

2.1 Apriori

This is one of the first algorithms developed for FIM and is based on the *Apriori principle*. Instead of using a brute-force method: counting all candidate itemsets by scanning the dataset, we divide the itemsets in generations. We reduce the amount of candidates we need to count by using this rule:
If an itemset is frequent, then all of its subsets must also be frequent.

Implementing this gives us the following process:

1 Scan the dataset and count the occurrence (σ) of every item (singular), extract the ones where $\sigma \geq min\_sup$. This is the first generation of frequent itemsets.

2 Combine every frequent itemset with every other frequent itemset. These are our candidate itemsets for the next generation.

3 Scan the dataset and count the occurrence of every candidate itemset, extract the ones where $\sigma \geq min\_sup$. This is the second generation of frequent itemsets.

4 Combine every frequent itemset with every other frequent itemset from the last generation, adding only 1 item to each existing itemset. Keep in mind, there are no duplicate elements in one set and order does not matter.

5 For every candidate itemset check if its subsets are frequent itemsets. If not, the candidate can not be frequent and has to be discarded. If so, the candidate may remain a candidate.

6 Count every candidate itemset in the dataset, extract those where $\sigma \geq min\_sup$. This is a new generation of frequent itemsets.

7 Repeat step 4 through 6 until no new candidates can be formed.

It should be noticed that step 1 to 3 are needed solely for the generation of frequent itemsets with size 1 and 2. The difference between these generations and generations thereafter is determining whether subsets are frequent as well. Generation 1 does not have subsets, and the subsets of generation 2 is generation 1. We are also skipping a step for generation 1: the collecting of candidate itemsets. This deviation is simply because gathering every frequent itemset of size 1 is not very complicated to do within one step. I believe it might even benefit performance.

An example (2.1) will follow on the next page.

Example 1

Table 2.1 is an example of a Transaction dataset. Where we have maximum itemsets of size 4 (as can be seen in Tid = 3). We will loop through the steps, with *min_sup* = 2.

Tid	Item 1	Item 2	Item 3	Item 4
1	1	3	4	
2	2	3	5	
3	1	2	3	5
4	2	5		

TABLE 2.1: Transaction dataset

After completing **step 1** of the algorithm, we will end up with table 2.2.

Itemset	σ
{1}	2
{2}	3
{3}	3
{5}	3

TABLE 2.2: Frequent itemsets: first generation

After **step 2** we have the candidates for generation 2 (2.3) and after **step 3** we have extracted the frequent itemsets (2.4).

Itemset
{1,2}
{1,3}
{1,5}
{2,3}
{2,5}
{3,5}

TABLE 2.3: Candidate itemsets: second generation

Itemset	σ
{1,3}	2
{2,3}	2
{2,5}	3
{3,5}	2

TABLE 2.4: Frequent itemsets: second generation

After **step 4** and **5** we can see there is only one candidate itemset left for generation 3 (2.5), which also happens to be the last frequent itemset.

Itemset
{2,3,5}

TABLE 2.5: Candidate itemsets: third generation

Itemset	σ
{2,3,5}	2

TABLE 2.6: Frequent itemsets: third generation

Normally we would continue with **step 6**, but according to **step 4** no new candidates can be formed, since we only have one frequent itemset in our last generation.

2.2 MBAT

This algorithm, developed by Singh and Dhir, 2013, is not as old as our Apriori algorithm. The biggest difference between Apriori and MBAT is the representation of the transactions, which we will show by giving an example of how this algorithm works. First we will explain the process, where I have quoted many steps from Singh and Dhir, 2013.
As the name, *Matrix Based Algorithm with Tags*, suggests, we will order our data in a matrix instead of a normal table. The algorithm goes as follows:

1 Scan the dataset to find the different items occurring in the dataset and make a transactional matrix by writing all the transactions along the row side and all the items occurring in the dataset along the column side. Complete the matrix by inserting the correct values per row: if a transaction contains an item, insert 1 in the corresponding column, otherwise insert 0. Lastly add 2 columns called Tag 1 and Tag 2. write the lowest valued item under Tag 1 and the highest valued item under Tag 2.

2 Generate the first candidate itemsets directly from the transactional matrix: collect every column-head. Calculate σ by counting every 1 in the corresponding column.

3 For the gathering of the first generation of frequent itemsets, collect every candidate itemset where $\sigma \geq min\_sup$.

4 After mining, trim the matrix by applying the following reduction rules: 1) Delete the transactions not containing a frequent itemset, for they cannot possibly contain a frequent itemset of the next generation. 2) Delete the transactions which do not contain enough items to generate itemsets for the next generation.

5 Generate the next generation candidate itemsets by scanning each row of the matrix and combining every item (having value 1) with each other. Eliminate those itemsets where the subsets of which are not frequent. Then calculate σ per itemset, again by counting along the columns. Whilst calculating, check each row beforehand to see if [Tag 1 $\leq$ the lowest valued item] and if [Tag 2 $\geq$ the highest valued item] per itemset. If it is not the case, the row can be skipped since the itemset is not present in that row.
We can now gather all itemsets where $\sigma \geq min\_sup$, this is our second generation of frequent itemsets

6 Repeat **steps 4** and **5** until the transaction matrix is empty and all frequent itemsets have been extracted.

In their paper the authors present us with quite some advantages of this algorithm over the Apriori algorithm. We will cover these assumptions at the end of this chapter and evaluate them in chapter 6. We will now look at an example, resembling the original example from the authors' paper, only smaller.

Example (2.2) will follow on the next page.

Example 2
In this example we will not show the tables with frequent/candidate itemsets, but rather the transformations the matrix will undergo.

After completing **step 1** of the algorithm, we will end up with the transaction matrix 2.7.

T id	I_1	I_2	I_3	I_4	I_5	I_6	Tag 1	Tag 2
1	1	1	1	0	1	0	1	5
2	0	1	0	1	0	0	2	4
3	0	1	1	0	0	0	2	3
4	1	0	0	1	0	1	1	6
5	1	0	1	0	0	0	1	3
6	0	0	1	0	0	1	3	6
7	1	0	1	0	0	0	1	3
8	1	1	1	0	1	0	1	5
9	1	1	1	0	0	0	1	3
10	0	0	0	1	1	0	4	5

TABLE 2.7: Transaction matrix

With completing **steps 2**, **3** and **4**, we updated the matrix, and it now looks like this 2.8.

T id	I_1	I_2	I_3	I_4	I_5	I_6	Tag 1	Tag 2
1	1	1	1	0	1	0	1	5
8	1	1	1	0	1	0	1	5
9	1	1	1	0	0	0	1	3

TABLE 2.8: Transaction matrix

We now completed **step 5** and **6** and circled back to **step 4**, where we trimmed some more 2.9.

T id	I_1	I_2	I_3	I_4	I_5	I_6	Tag 1	Tag 2
1	1	1	1	0	1	0	1	5
8	1	1	1	0	1	0	1	5

TABLE 2.9: Transaction matrix

And again: only we have nothing left after **step 4**. So according to **step 6**, we are done 2.10.

T id	I_1	I_2	I_3	I_4	I_5	I_6	Tag 1	Tag 2

TABLE 2.10: Transaction matrix

We can already see how different this is going to work from Apriori, since that algorithm does not update its dataset. We will continue examining this in chapter 4.

2.2.1 Assumptions MBAT

According to the authors (Singh and Dhir, 2013) this algorithm exceeds the Apriori algorithm in a number of ways, we will discuss why some of these implementations should surpass those of Apriori:

Matrix This is a rather obvious difference between the algorithms. Instead of using a normal table where each row/transaction consists only of the items it contains, it now represents every possible item and its membership. The authors do not directly address the reason for using this format, but it becomes clear when looking at the rest of the steps from the algorithm: if we want to be able to count the items we first have to search for them. Normally this is a very high-cost step, where we have to go through the entire dataset for just one item. But by using the format of this matrix, we already know where all different items are, since they have their own column. Applying this in `SQL`, we would just have to check every column and we are done. This might very well be an improvement on normal Apriori-like algorithms implemented in `SQL`, where candidate searches are huge obstacles (Shang, 2005)

Tags The algorithm uses two tags: Tag 1 and Tag 2. They keep track of the lowest and highest item respectively in a transaction. The advantages hereof are also not directly explained by the authors. They only state the following:

> Before scanning the columns of the every row for counting, first, Tag 1 is checked to see if the smallest item serial number of the itemset is less than the value of Tag 1 in the corresponding row. If the smallest item serial number is less than the value of the Tag 1 then there is no need to scan columns of that row for counting. If the smallest item serial number is not less than the value of the Tag 1 then Tag 2 is checked to see if the largest item serial number is greater than Tag 2. If the largest item serial number is greater than Tag 2 then we move to the next row for counting. **Hence during the counting of support for candidate itemsets the tags help to reduce counting effort.** Singh and Dhir, 2013, p. 357

It is an example of slight pruning, but it does not look like it necessarily reduces counting effort if we still have to check every row.

Reduction The last adjustment is not one in representation but in (extra) steps, namely the reduction properties. These properties allow us to delete transactions which could not possibly contain frequent itemsets and are therefore useless for analysing any further. This already sounds like a huge advantage, since we reduce our search space increasingly whilst we move through generations of itemsets. And reducing search space is nearly always desired in data mining, it shortens the search, therefore making it faster and improving performance.

3 Dataset

We have already mentioned our data will be random, this is to ensure we can tweak our sets when we experience trouble going through a dataset. Finding an existing dataset that is ideally composed for all our different implementations is going to be difficult: the dataset could be too small, so testing will not deliver interesting results between algorithms. It can also be too big, taking too much time to finish, which is problematic for research which needs to be completed within a certain time-frame. In this chapter we will give a small description of how we will represent our data and what we will use as limits.

3.1 Structure

Considering the algorithms we explained in chapter 2, we have to make our dataset suitable for usage by both.
We have come across transactions, items, (sometimes) certain variables and indices, this is all easy to translate into code, especially with `SQL`.

Item

An item can be represented in many ways: a word, sequence of numbers, or even as its own object. Since we will be handling many items and have to be able to recognize and compare items, the easiest representation is a number. With numbers we will never have to translate them into something else to be able to compare them, such as a word. It is easy to check if one item is smaller/bigger than the other and they can be ordered.

Example 3 : Item 1 = 1, item 2 = 2, item 2363 = 2363. It seems rather trivial but a lot of datasets still have other representations than this one.

Our data is going to be random. Therefore we will make sure every item is as likely to show up in a transaction as any other. Which leads us to the last attribute, the amount of different items we are going to use, a tweaking option we will discuss later on.

Transaction

A transaction, also called a record, is composed of items (which are numbers). The transaction also needs to have something unique so we can find it anywhere in our dataset, so we will give every transaction a unique ID. A transaction can never be empty: if it is empty, it is not a transaction worth looking at and will give a false impression of our dataset. So the size is 1 or bigger, but how much bigger? That leads us to the question, how many items can a transaction have? And it all boils down to: how big can our biggest (frequent) itemset be, which is something we always have to decide beforehand. Again, this is one of our tweaking options which we will discuss later in this chapter.

A few other customs:

- A transaction can never contain duplicate items
- A transaction's size is determined by the amount of items in that transaction. That amount will be random per transaction (but within the limits)

3.2 Limits

If we want our datasets to be suitable, we will need to experiment with some parameters, and so we did. After coding the first prototype of the Apriori algorithm in `MySQL` (and `Java`), we did some test runs. Our initial idea was to use a dataset of 1000000 transactions, 4000 different items and a maximum transaction length of 100. This turned out to be insanely expensive and also quite impossible with the current hardware. When the program could not use any more of memory, it started writing to disk, this turned out to be lethal for the computer. After numerous tests and software re-installations we accepted that we have to dial way back to a more appropriate number for at-home data mining. Eventually the limits where set on the following:

- **Amount of transactions:** 50000
- **Amount of different items:** 20
- **Maximum transaction length:** 8
- *min_sup*: 3

These limits were used throughout testing.

It needs to be said, that this was a very practical test and not based on anything scientific. But for the sake of progress in our research, we had to make the decision in losing some of our integrity or losing a lot of time with the chance of having to fall back on option 1. We chose accordingly.

3.3 Stochastics

We enforced the randomness of our data by using the `Random` class of `Java`. This is shown in our `createdata` class. What is random in our data generation:

- The amount of items in a transaction. It is not possible for that amount to be 0 because then the transaction would not exist.
- The items themselves. The item 0 does not exist.

For further comments and code, see createdata.java

4 Implementation

Our first idea was to work exactly like the steps described in chapter 2, this turned out to be far from ideal in practice. So we made some alterations for most algorithms, which we will illustrate in the same manner as we did with the original algorithms, only we now have an extra step: creating the data.
We will, however, not count this step as part of the overall progress which we are measuring for performance.
Before we continue on with the implementations of the combinations, we digress to justify the order and structure of our tables.

All major changes are displayed in *italic*

Many programming decisions we made, were the result of empirical evidence that some statements did work and some just did not. Although we will not discuss all these here, we made some remarks about other possible implementations in chapter 7.

Table structure

For both `MySQL` and `MonetDB` we noticed that randomly structuring our tables was a deficit for the rest of our program. Hence we added an index on the table(s) which greatly reduced runtime. This makes sense: with both DBMSs and algorithms we have to go through the dataset multiple times, searching for items. But with indexed tables it makes the search go faster, since the DBMSs now have a look-up system. To further explain this:
A clean database in a DBMS does not have an order in its tables, it is random. But with adding an index on a column (or an entire table) a DBMS creates a new data structure where it saves the location of each field (id or item in our case). It is basically just a normal book index. The only downside is that it needs space on our hard disk. The space it needs, however, is negligible in comparison with the size of our original dataset and hard disks.
The order of our tables does not have anything to do with the indexes we put on our tables. We thought that it would look more understandable if our tables were of ascending order.

4.1 Apriori and `MySQL`

Creating data

Creating the dataset in the form of a regular table was our first idea 2.1. This did not work out that well. `MySQL` handles a lot of rows very well, but a lot of columns less so. Just generating the data and saving it in `MySQL` could take an hour. When thinking about all the joins we are supposed to do, there was a realisation that it would not be feasible with a relatively normal computer, it would only go out of bounds memory- and space-wise. So we decided to turn the dataset into a table with just two columns, see table 4.1

Tid	Items
1	1
1	3
1	4
2	2
2	3
2	5
3	1
3	2
3	3
3	5
4	2
4	5

TABLE 4.1: Transaction dataset

A transaction is now spread out over multiple rows: every item within a transaction gets its own row. The only downside is not being able to add an unique index on the IDs. We solved this by adding an index over the combination of the two rows, which is unique.

Program

1 Scan the dataset, and retrieve every itemset of size 1 with $\sigma \geq min\_sup$.

2 Combined every frequent itemset with every other, by the means of a join.

3 Gathered σ for every candidate itemset by pulling a join over a join.

4 By performing an inner join, all frequent itemsets of last generation (n) are selected, where the first n items stay the same and the last item has to be smaller than the other side of the join. This will give us all possible candidate itemsets with the given frequent itemsets.

5 *The internal workings of this part of the algorithm stays the same, only the structure changes. It is very hard to achieve this step in `MySQL`, so we pulled all candidate itemsets to `Java` and started generating all of its subsets of size -1 and simultaneously checked if they are frequent itemsets. We also added one check to prevent unnecessary work: if last generation contains all possible frequent itemsets of that size, we do not have to check subsets, since that automatically means all subsets are frequent. After which we send the updated candidate table back to `MySQL`*

6 Counted the candidate itemsets, retrieved every itemset where $\sigma \geq min\_sup$.

7 Repeat steps 4 to 6 in a for-loop. The for-loop stops when no new candidates can be formed or if the itemsets of maximum size have just been counted.

For further comments and code, see: fimSQL.java

4.2 Apriori and `MonetDB`

The implementation of the Apriori algorithm we defined for `MySQL`, worked surprisingly well with `MonetDB`. The initial expectation was having to reform a lot of our `SQL` queries in order to get it working. This was not the case at all.
We were allowed to use every query using the same syntax, the only thing that changed was exporting and importing tables. But that was just a matter of using different statements, which is negligible in the context of performance.

With that explained, we will refer to the last section, **Apriori and** `MySQL`, for the steps of this program.

For further comments and code, see: fimMonet.java

4.3 MBAT and `MySQL`

Creating data

We discussed a different representation of our data in the first implementation, the same will apply for the datamatrix in MBAT. A transaction matrix consists of many columns, which makes it nigh impossible implementing it in `MySQL` (as has been explained). We came up with a different design, which resembles a normal dataset but still functions as a matrix. We have also added two extra values per transaction, **transaction size** and **amount FI**: **transaction size** is the amount of items in the transaction, **amount FI** notes if the transaction contained a frequent itemset from the last generation. We will keep these values in the matrix. The tags, however, will be exported to a new table, which makes them better accessible and reduces the size of the matrix.

The matrix will look like this 4.2, with the tags table linked to it 4.3:

T id	itemname	itemvalue
1	1	1
1	2	1
1	3	1
1	4	0
1	5	1
1	6	0
1	transaction size	4
1	amount FI	0
2	1	0
2	2	1
2	3	0
2	4	1
2	5	0
2	6	0
2	transaction size	2
2	amount FI	0

TABLE 4.2: Transaction matrix

T id	Tag 1	Tag 2
1	1	5
2	2	4

TABLE 4.3: Tags table

Program

1 *The first step was transforming the actual dataset to a transaction matrix. We chose to not do the transformation but to start off with a matrix. Since it is an annoyingly difficult type of query to compose and execute within a short amount of time, we felt it would not benefit our research. Especially since we do not consider the creating of data to be a part of the overall process.*

2-3 *Calculated frequent itemsets of size 1 by summing up the values in column **itemvalue** and comparing those to min_sup. It was not needed to separately gather candidate itemsets and then start counting*

4 After each generation, we first delete all transactions that did not contain a frequent itemset. For this we can check the **amfim** value per transaction. We then delete all transactions that do not have enough items to contain frequent itemsets for the next generation. For this we can check the **amit** value per transaction.

5 *This is rather big part which we do in little steps: first we extract every item with value 1 and generate subsets per transaction. We then save these and start listing every Tid per itemset. This will give us its σ and we can later use that list for **step 4**, again the downside: we have to do this per itemset.*

6 We repeat the steps until the matrix is empty or if we reached the biggest possible frequent itemsets.

For further comments and code, see: mbatSQL.java

4.4 MBAT and `MonetDB`

With this combination we had to make a little bit of an effort. The `SQL` statements in the last-named program were partly not usable in `MonetDB`, we managed to reform some of those statements without adding too much work. But there were some parts that had to be done with transporting tables from file to `MonetDB` and vice versa. Namely because deleting rows from a column-based DBMS is a very slow query. So the program steps are still the same as **MBAT and** `MySQL` only differently imported and exported.

For further comments and code, see: mbatMonet.java

5 Results

The structure of our results will be as follows:
For every combination we will list the time it took to complete steps of the algorithm. We note the amount of frequent itemsets per generation once, since these will be the same for every combination. The candidate itemsets will be noted separately as they can differ per combination.

5.1 Frequent itemsets

Generation	Amount frequent itemsets
1	20
2	190
3	1140
4	4845
5	15504
6	36086
7	2845
8	3

TABLE 5.1: Frequent itemsets per generation

5.2 Apriori and `MySQL`

Step	Time(s)
Generation 1 frequent itemsets	10
Generation 2 candidate itemsets	0,1
Generation 2 frequent itemsets	4
Generation 3 candidate itemsets	0,2
Generation 3 frequent itemsets	107
Generation 4 candidate itemsets	0,4
Generation 4 frequent itemsets	442
Generation 5 candidate itemsets	2
Generation 5 frequent itemsets	1400
Generation 6 candidate itemsets	24
Generation 6 frequent itemsets	3666
Generation 7 candidate itemsets	160
Generation 7 frequent itemsets	4850
Generation 8 candidate itemsets	0,2
Generation 8 frequent itemsets	0,1
Total	10666

TABLE 5.2: Performance per step

Generation	Amount candidate itemsets
1	20
2	190
3	1140
4	4845
5	15504
6	38760
7	49580
8	3

TABLE 5.3: Candidate itemsets per generation

5.3 Apriori and `MonetDB`

Step	Time(s)
Generation 1 frequent itemsets	1
Generation 2 candidate itemsets	0,1
Generation 2 frequent itemsets	0,1
Generation 3 candidate itemsets	0,1
Generation 3 frequent itemsets	2
Generation 4 candidate itemsets	0,1
Generation 4 frequent itemsets	5
Generation 5 candidate itemsets	1
Generation 5 frequent itemsets	16
Generation 6 candidate itemsets	1
Generation 6 frequent itemsets	46
Generation 7 candidate itemsets	28
Generation 7 frequent itemsets	57
Generation 8 candidate itemsets	1
Generation 8 frequent itemsets	0,1
Total	158,5

TABLE 5.4: Performance per step

Generation	Amount candidate itemsets
1	20
2	190
3	1140
4	4845
5	15504
6	38760
7	49580
8	3

TABLE 5.5: Candidate itemsets per generation

5.4 MBAT and `MySQL`

Step	Time(s)
Generation 1 frequent itemsets	2
Generation 2 candidate itemsets	12
Generation 2 frequent itemsets	44
Generation 3 candidate itemsets	76
Generation 3 frequent itemsets	255
Generation 4 candidate itemsets	92
Generation 4 frequent itemsets	404
Generation 5 candidate itemsets	315
Generation 5 frequent itemsets	2856
Generation 6 candidate itemsets	1347
Generation 6 frequent itemsets	5973
Generation 7 candidate itemsets	710
Generation 7 frequent itemsets	9092
Generation 8 candidate itemsets	85
Generation 8 frequent itemsets	871
Total	22134

TABLE 5.6: Performance per step

Generation	Amount candidate itemsets
1	20
2	190
3	1140
4	4845
5	15504
6	38633
7	40334
8	6136

TABLE 5.7: Candidate itemsets per generation

5.5 MBAT and `MonetDB`

Step	Time(s)
Generation 1 frequent itemsets	4
Generation 2 candidate itemsets	8
Generation 2 frequent itemsets	12
Generation 3 candidate itemsets	108
Generation 3 frequent itemsets	170
Generation 4 candidate itemsets	399
Generation 4 frequent itemsets	823
Generation 5 candidate itemsets	578
Generation 5 frequent itemsets	3428
Generation 6 candidate itemsets	1364
Generation 6 frequent itemsets	9054
Generation 7 candidate itemsets	427
Generation 7 frequent itemsets	11355
Generation 8 candidate itemsets	20
Generation 8 frequent itemsets	0,3
Total	27750,3

TABLE 5.8: Performance per step

Generation	Amount candidate itemsets
1	20
2	190
3	1140
4	4845
5	15504
6	38633
7	40334
8	6136

TABLE 5.9: Candidate itemsets per generation

6 Analysis

First and foremost, we will look at our results: why the time/performance differences with two different DBMSs and why with the two different algorithms? After which we will talk about the advantages named by Singh and Dhir, 2013 and analyse if these advantages were correctly assumed.

6.1 Comparison

The first thing to notice is that the MBAT did not perform all that well in the `MonetDB` environment, which goes against our expectations. Since `MonetDB` is a column-based DBMS and MBAT a matrix, it should be able to perform in under an hour, instead it took almost 8 hours. How can that be? After considering the programming choices and analysing the runtime, it started to make sense: `MonetDB` does not deal well with deleting transactions from a table, which, according to the algorithm, is supposed to happen twice per generation. This makes for some serious time-consuming steps. The goal of these steps does not outweigh the means when it comes to `MonetDB`. Now consider the algorithm in `MySQL`, there is not much of improvement there. Which is weird, since `MySQL` *does* handle deletions well. Upon looking at the sequence of the algorithm again and the history of the console, it became clear that the deletions are not the problem here. They are most likely aiding the program, but then why is it still slower than Apriori? What easily gets forgotten, is the amount of times `Java` has to go through the loop and send statements through to `MySQL`. Instead of the Apriori program, where all frequent itemsets are calculated within one statement, this program has to go through an entire dataset **per** candidate itemset. This in turn, is caused by the inadequacy of `MySQL` for handling multiple (and with multiple, we mean more than 20) columns, if that would be able, the loop could easily be transformed into one or maybe two statements.

Another result which stood out, was the Apriori program in `MonetDB`. It did not even take 3 minutes for it to finish. Proof that `MonetDB` has benefits? Certainly. The beautiful thing about this combination is the almost perfect use of columns. We do not touch the original dataset, we make no updates and do not delete anything. This is already a huge advantage in `MonetDB`. But apart from that, the same `SQL` statements used in `MySQL` perform a lot faster in this DBMS. And this is all thanks to the joins we are performing. Joining is focussed on the columns and with indices even faster.

The combination of Apriori and `MonetDB` seems to triumph amongst the others. Thanks to the few, but large `SQL` statements composed of joins and no updates or deletions, it was able to manage executing and finishing within a meagre 3 minutes.

6.2 MBAT discussion

"In this algorithm the database is scanned only once and that is only generate the transactional matrix" Singh and Dhir, 2013, p. 358.
Yes, in theory. But scanning a database (or, as we call it, dataset) means something completely different in practice. According to **step 3** and **5** of their algorithm we need to collect the candidate itemsets and count occurrence *somehow*, it is never specified how we should do that. That how, unfortunately, is by scanning the database. Which makes this advantage void.

"This method greatly reduces the problem of generation of a large number of candidate itemsets because this method considers only those items in the row of the matrix which are having the value of 1." Singh and Dhir, 2013, p. 358
This is sort of true. Yes, we only take those items which have the value of 1. But the amount of candidate itemsets (as we can see in our results) stays pretty much the same in earlier generations. This is because of the size of our dataset. With 50000 transactions, we are probably going to find every candidate itemset of size 2 (since there are only 190 possible sets). You start seeing the difference in generation 6 to 8.

"The tag columns are very helpful in reducing the effort in counting support for itemsets." Singh and Dhir, 2013, p. 358
Completely valid. Especially in a language such as `SQL` it is a very low-cost query to execute and saves a lot of rows to parse. Which in `MonetDB` makes a difference.

"The combination of above properties and the transaction reduction property provides another advantage of less computational time." Singh and Dhir, 2013, p. 358
That can be made true. In `MonetDB` it is actually an obstacle, because of all the deletions. But in `MySQL` it did assist a good performance, since it is row-based and handles deletions better. It alleviates the pain from scanning the dataset multiple times.

"Method is much easier to implement than apriori and other popular algorithms for association rule mining." Singh and Dhir, 2013, p. 358
This depends a lot on how big your input (datasets) is going to be and how fast you want the program to run. If it is small, as in the examples, then yes, it is definitely easier to implement. Because then it would be possible to represent the matrix as an actual matrix i.e. with multiple columns. It would be easier to come up with `SQL` queries since you can work per column instead of a row. But if your input is big and the amount of different items high, it is going to be a struggle getting your transaction matrix to be flexible, especially when deleting rows.

7 Conclusion

This will be a short summary of what we have already discussed in chapter 6 and what can still be done in future research.

We saw that the MBAT did not perform as we expected with either DBMS: with `MonetDB` because of the deletions in the matrix and with `MySQL` because of the iteration through all itemsets. It should be said that these implementations were our choice and that a different way of translating the algorithm could mean that MBAT is the better choice and has the best performance. But here we showed, with the current implementation, it is not *necessarily* the best choice.

Because MBAT did not live up to its expectations, FIM was automatically the fastest algorithm in this research. In combination with `MonetDB` it took little time to finish. This does not make it a better algorithm, it makes it the *better combination in this implementation*. We can not express this enough, but there are plenty of different designs possible to get these algorithms to work. Which also means that they could have different performances.

The differences between `MonetDB` and `MySQL` were very clear (especially combined with the Apriori algorithm). We noticed that `MonetDB` does not handle deletions well, which is unprofitable for the MBAT. On the other hand, `MonetDB` seems to be advantageous in almost every other aspect, because of its column-based structure. Again, this only matters if the algorithm is designed in a way that a column versus row approach makes a difference. Which is the case in this paper.

7.1 Future Research

Even after applying multiple methods and testing them across various datasets, we know many more possible solutions exist to enhance performance for the combinations we considered in this paper. There simply was not any time left to test them all. We will discuss a few options we either considered and have not tried or we had not considered at all but which might still be fruitful.

- Sparse-Matrix techniques could be used in combination with MBAT. We have not tried to implement this and also have not thought of how to do this. Perhaps in combination with the manual composed by Saad, 1990, a method could be extracted.
- What we have not tried in this examination, is building the matrix from MBAT with all of its columns in `MonetDB` and abstain from deleting or changing the matrix (by either making new tables or just keeping the original matrix as it is without deletions). Since it was able to handle the Apriori dataset, it would not be weird to expect it could handle a matrix of roughly the same size as well.

- What would definitely be worth examining, is using a computer that can handle the workload. For example a supercomputer. We had to change our original dataset to one that could fit in memory and would not take weeks to go through. With a computer that has ample sources, we would not need to, which in turn might have a positive effect on performance.
- Use `Python` instead of `Java`: although it has not been verified, sets (the data structure) in `Python` appear to be easier in use and take less space as a data structure than in `Java`. This might make up for a lot of time, especially in the Apriori algorithm.
- Implement everything in `SQL` without using object-oriënted programming. This would be a very complex structure, but has already been done (Thomas and Sarawagi, 1998). The work would be too extensive for this paper, but we think it might be well worth examining in combination with a stronger computer.

Bibliography

Agrawal, Rakesh and Ramakrishnan Srikant (Sept. 1994). "Fast Algorithms for Mining Association Rules". In: *Proceedings of the 20th International Conference on Very Large Data Bases*. Vol. 1215, pp. 487–499. URL: http://www.cse.msu.edu/~cse960/Papers/MiningAssoc-AgrawalAS-VLDB94.pdf.

Feyyad, U. M. (Oct. 1996). "Data mining and knowledge discovery: making sense out of data". In: *IEEE Expert* 11.5, pp. 20–25. ISSN: 0885-9000. DOI: 10.1109/64.539013.

Saad, Youcef (May 1990). *SPARSKIT: a basic tool kit for sparse matrix computations*. Research Institute for Advanced Computer Science. URL: https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19910023551.pdf.

Shang, Xuequn (2005). "SQL based frequent pattern mining". In: URL: https://opendata.uni-halle.de/bitstream/1981185920/10641/1/xueshang.pdf.

Singh, Harpreet and Renu Dhir (Aug. 2013). "A New Efficient Matrix Based Frequent Itemset Mining Algorithm with Tags". In: *International Journal of Future Computer and Communication* 2.4, pp. 355–358. URL: https://pdfs.semanticscholar.org/9571/a8a823f6e49d2316254a43ce0af32808ae8a.pdf.

Thomas, Shiby and Sunita Sarawagi (1998). "Mining Generalized Association Rules and Sequential Patterns Using SQL Queries." In: *KDD*, pp. 344–348. URL: http://www.aaai.org/Papers/KDD/1998/KDD98-062.pdf.

Verma, Prince and Dinesh Kumar (Sept. 2013). "IP- Apriori: Improved Pruning in Apriori for Association Rule Mining". In: *International Journal of Recent Technology and Engineering* 2.4. ISSN: 2277-3878. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.677.3465&rep=rep1&type=pdf.

Wu, Xindong (Sept. 2004). "Data mining: artificial intelligence in data analysis". In: *Proceedings. IEEE/WIC/ACM International Conference on Intelligent Agent Technology, 2004. (IAT 2004)*. Pp. 7–7. DOI: 10.1109/IAT.2004.1342916.