Opleiding Informatica

Querying Frequent Itemsets

in the Browser

Michiel Vrins

Supervisors:
Matthijs van Leeuwen
Arno Knobbe

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

08/12/2020

Abstract

The aim of this thesis is to investigate how non data experts can effectively query a set of frequent itemsets to find interesting patterns in their data set.
Frequent itemset mining is a technique to find repeating patterns in data that consists of rows of transactions such as purchases in a supermarket. These patterns consist of items often occurring together in these transactions. This information can be used by domain experts to, for instance, optimize the positioning of items in a store to increase their sales.
Currently it is difficult for domain experts to find these frequent itemsets and even if they have them, it is difficult to filter through them. It is difficult to find these itemsets because the tools that can do this are not very accessible for people without a background in data mining. Furthermore, if a domain expert finds these frequent itemsets, the list of them will usually be too large to look through without any help. This thesis tries to make it easier for domain experts to find the frequent itemsets and also to exploit their knowledge by allowing the user to filter through the frequent itemsets using an easy to use query language. To do this, we propose a browser-based tool to obtain the frequent itemsets out of a given data set and allow the user to filter through them with an easy to use query language. This program is run completely in the browser, without offloading anything to a server to ensure safety of their data. We also performed a preliminary user study to evaluate the usability and effectiveness of the query language. From these results we can say that the query language is effective for people with a background in computer science.

Contents

1 Introduction **1**
- 1.1 Thesis overview. 1

2 Related work **2**

3 Frequent itemset mining **2**
- 3.1 The Apriori algorithm 3

4 Data exploration **4**
- 4.1 Requirements 5
- 4.2 Interactivity and design 5
- 4.3 Query language 6

5 Implementation **7**
- 5.1 Query language implementation 7
  - 5.1.1 Ohm js 7
  - 5.1.2 Formal definition of the query language 8
- 5.2 Optimizations 9
  - 5.2.1 Bitmasks 9
  - 5.2.2 Pruning 9
  - 5.2.3 Item mapping 10

6 Evaluation **10**
- 6.1 Experiment setup 10

7 Results **12**

8 Discussion **14**

9 Conclusion **16**

10 References **16**

A Appendix **18**
- A.1 Application overview 18
- A.2 Questionnaire 19

1 Introduction

With the increase of collected data in the world, being able to easily analyze all the data is becoming increasingly more important. Analyzing data can however prove very difficult to people with little experience or education in it. Different techniques have been developed over the years to find certain patterns in data which are useful to people even without experience or training in data analysis. One such technique is frequent itemset mining [2] and it can be used to find recurring patterns in a data set. The technique is relatively straightforward and the first fast algorithm, which is still used today, was proposed in 1994 [3]. In short, frequent itemset mining finds all subsets in a binary data set that occur more than a predetermined number of times. As an example, lets say that in a small data set of 20 transactions bread and cheese are bought together 10 times and the minimum threshold is set to 3. The set of bread and cheese occurs more than 3 times in the data set and is therefore frequent. It can however be hard to find frequent itemsets for people without a background in data science. To add to that, even if all the frequent itemsets are found, finding the relevant information can be hard due to the large number of results. This is commonly referred to as the 'pattern explosion'.

This thesis focuses on the question: How can we create a query language that can be used to effectively filter through frequent itemsets by non data experts, while still allowing for advanced queries containing negations, logical OR and logical AND?

This thesis provides an easy to use browser-based program to find the frequent item sets from a data set, and an easy-to-use query language to filter the results. By making the program browser-based it lowers the threshold for non data scientists to use the program. The program accepts data in the CSV format where the data contains rows with items. These rows can for instance contain individual transactions from a supermarket, consisting of only the items bought. To evaluate the effectiveness of this query language a preliminary study on a group of computer science students has been performed.

The target audience are domain experts with little knowledge about data mining with the intent that they can find useful patterns, in the form of frequent item sets, from their data without requiring expensive data scientists. These patterns can then be used to improve their services. For instance a restaurant could adjust the position of items on the menu, or a supermarket the position of items in the store. This is already a widely adopted technique in supermarkets today [9].

1.1 Thesis overview.

This thesis will start with discussing some previous research and programs which are related to pattern mining. This is followed by chapter 3 with an introduction to the concept of frequent itemset mining. In chapter 4 the data exploration features of the application will be thoroughly explained; in chapter 5 the implementation, with a formal description of the query language and in depth explanations of how certain optimizations are implemented. In chapter 6 a user study and its results are presented. Finally in chapter 7 and 8, we finish by discussing our findings and conclusions.

2 Related work

Many different data mining tools have been developed over the years. Some of these tools can be widely used and have a large number of options that can be tuned like Rapid Miner [11] and Cortana [10]. But also smaller more specific tools have been developed for specific data exploration tasks like Viper [13]. Many other recent techniques focus on integrating the users input into the pattern mining and making it more interactive to try to reduce the so-called pattern explosion. This can be done by adjusting the sampling distribution in realtime [4]. Many of these tools also move away from frequent itemset mining and instead focus on other techniques such as subgroup discovery. This technique has the advantage that it uses quality measures which show statistical significance of a found subgroup [7]. The biggest problem with the bigger tools, like Rapid Miner and Cortana, is that they require data mining knowledge in order to effectively use them as can be seen from the preliminary results from [13], where even computer science students needed significantly more time to find answers to general questions about a given data set than using their proposed tool, VIPER. Some tools have been developed which focus on making the exploration of patterns move visual, like MIME [6] and Viper, but these lack the ability to make complicated filters for the resulting itemsets.
The application proposed in this thesis does not focus on trying to use the user input to reduce the 'pattern explosion'. Using extra input to reduce the number of results is a form of constraint-based pattern mining and the most common form of it is the use of minimum support in frequent itemset mining [12].
Instead, this application allows the user to easily filter all the resulting patterns using a query language and allow the user to easily view all original data associated with a chosen pattern.

3 Frequent itemset mining

Frequent itemset mining is a concept where a list of transactions can be processed to find subsets of those transactions which occur more than a specific number of times in a given data set consisting of transactions. A transaction here is a list of items. A transaction can for instance be the different items bought at a supermarket or restaurant in a single sale, but it can be any list of items. An itemset here can be any combination of items from the transactions. The number of times such a subset occurs in a data set is called the support count. This can be formally described by the following.
Let **T** be a database of transactions, **t** an arbitrary transaction and **X** an arbitrary itemset.
The support count of **X** with respect to **T** is defined by the number of transactions **t** in **T** containing **X**.
$\mathbf{supportCount(X)} = |\, \mathbf{t} \epsilon \mathbf{T} \;;\; \mathbf{X} \epsilon \mathbf{t} \,|$

Table 1 and 2 illustrate how such a database of transactions could look like and the corresponding support count of itemsets from that database. Some of the itemsets in Table 2 have been omitted.

For a subset to be frequent it needs to occur a certain minimum number of times in different transactions. This minimum number of times is called the minimum support. This means that the support count of a set needs to be larger than the minimum support for it to be frequent.

Transaction
whole milk, margarine, tropical fruit
whole milk
coffee, bottled water, sausage, rum
canned beer
whole milk, tropical fruit, margarine shopping bags
whole milk, tropical fruit

Table 1: Example of transaction database

Itemset	Support count
whole milk	4
tropical fruit	3
whole milk, tropical fruit	3
margarine	2
whole milk, margarine	2
coffee	1
bottled water	1
whole milk, margarine, tropical fruit	1
...	...

Table 2: Itemsets and corresponding support derived from table 1

There is not a real best value for the minimum support, as it is highly dependent on the data set. Furthermore the computational time required for lower supports is significantly higher. This is because with a higher support we can prune more of the search tree. How this pruning occurs is explained in more detail in 5.2.2. There are however techniques that can estimate the number of frequent itemsets given a specific minimum support. There are also techniques which can estimate a minimum support given a requested number of frequent items [14]. This will be discussed more in section 8.
Having a higher support gives fewer results, this can make it easier to look through the results. However some of the results that are filtered out with a higher support can also be interesting, this makes it hard to find the best value for the support. In order to find all frequent itemsets, the Apriori algorithm can be used. This algorithm dynamically creates itemsets from the database and tests them on all the transactions, checking if they have a support count greater than the min support. The way how these itemsets are dynamically created is explained further in 5.2.2.

3.1 The Apriori algorithm

The following pseudocode is a representation of the Apriori Algorithm [2], which is an algorithm that finds frequent itemsets in a relational database. It has all core functionality needed for calculating the frequent itemsets but it does not contain the full algorithm that is used in the application in order to keep the pseudocode readable. The full implementation can be found in the source code of the application. The pruning optimization, which is explained in more detailed in 6.3.2, is added in the pseudocode. The bitmasks and item mapping optimizations, explained in 6.3.1 and 6.3.3, have

been omitted.
The Apriori algorithm works by starting with a set of all itemsets from the data containing only a single item, in other words all unique items. The number of times these itemsets are subsets of transactions in the database, their support, is counted. If this amount is bigger than the minimum support, the itemset is marked as frequent. New candidate itemsets are created from these frequent itemsets while accounting for duplicates (the way this is done is further explained in section 5.2.2). These new candidates are then tested again for frequency and the process continues until no candidates are left. The algorithm then returns all itemsets marked as frequent.

The algorithm has two arguments: a data set consisting of transactions, and an integer for the minimum support. The algorithm returns a set containing all itemsets which are frequent.

Algorithm 1 Apriori algorithm [2]

Argument1 A transactional database
Argument2 Minimum support

1: $c \leftarrow candidates$ ▷ Initialize candidates with set of all itemsets of size 1 which occur more than minSupport times in the data
2: $j \leftarrow c$ ▷ Copy c into j
3: $Fi \leftarrow empty$
4: **while** $c \neq \emptyset$ **do** ▷ Continue if candidates not empty
5: $Fc \leftarrow empty$
6: **for all** $x \in c$ **do** ▷ Check if candidates are frequent
7: **if** $x.occurrences >= minSupport$ **then**
8: $Fi.push(x)$
9: $Fc.push(x)$
10: **end if**
11: **end for**
12: $c \leftarrow \emptyset$
13: **for all** $y \in Fc$ **do** ▷ Create new candidates for next iteration
14: **for all** $l \in j$ **do** ▷ For all frequent candidates
15: **if** $y.lastItem < l$ **then** ▷ Compare the integer representations
16: $T \leftarrow y.append(l)$
17: $c.push(T)$
18: **end if**
19: **end for**
20: **end for**
21: **end while**
22: **return** Fi ▷ The frequent itemsets in the data

4 Data exploration

In this chapter the requirements of the application are described. Furthermore we describe how a user can interact with the application. Finally we describe the query language it's functionalities

and provide some examples of possible queries.

4.1 Requirements

The application has to fulfill two main tasks. First of all, it needs to be able to process a users input data, generate frequent itemsets from it and finally display them. Secondly, it needs to provide the user the ability to filter itemsets based on a query he supplies. The application should also be intuitive or self explaining in a way that a user does not require external help in order to perform the actions he wishes to perform with the application.

4.2 Interactivity and design

When opening the browser-based application, the user is greeted with the option to load his data into the application. He is also already introduced to the search bar and the table where the frequent itemsets will appear after loading the data as can be seen in Appendix 1 Figure 2. The table contains the following text: "Start by loading your data from a file!" so the user knows he needs to load a file first.

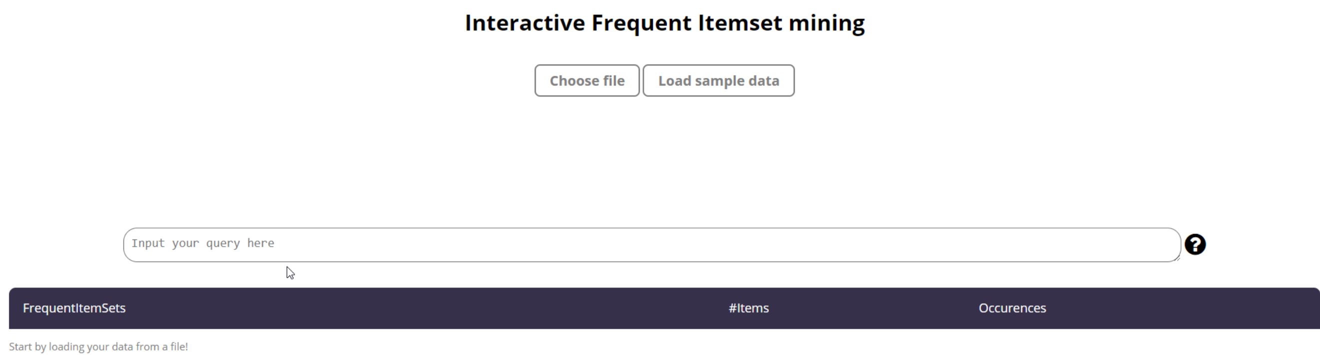

Figure 1: Opening page of the application

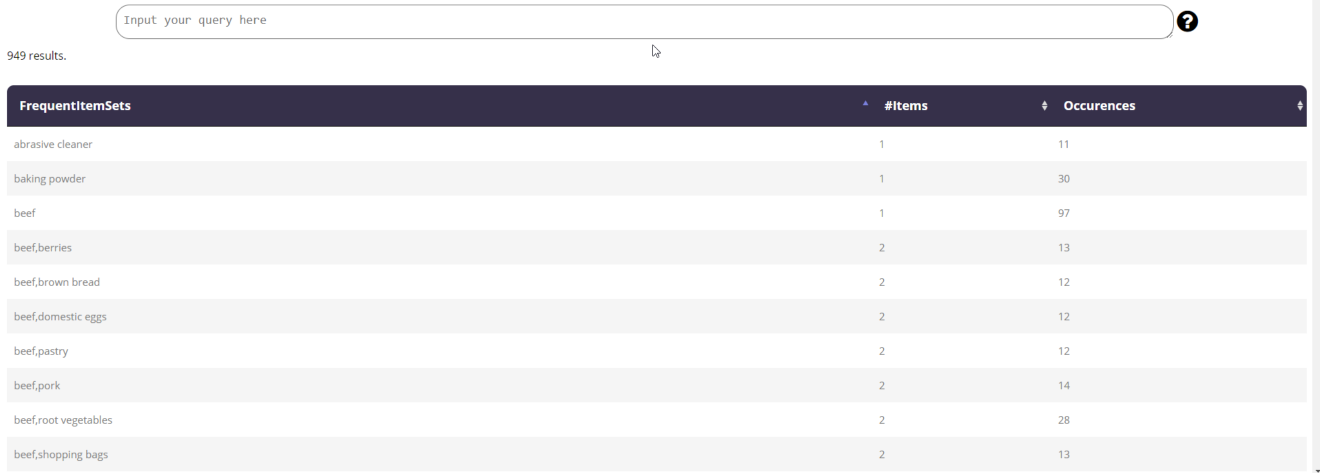

Figure 2: Application after loading a data set

After loading a data set, all frequent itemsets can be viewed in a list in the web application. In order to make it easier for the user to find useful frequent itemsets, the list can be sorted on the number of unique items in the frequent itemset and on the number of total occurrences in the input data. An example of how this would look like with a sample data set can be seen in Appendix 2. Furthermore, the user can click a frequent itemset and instantly see a list of all the transactions containing this itemset. This list also shows the number of unique items and the number of occurrences in the case of duplicate transactions. In the items of the transactions, the frequent itemset contained in the transaction will be displayed in red so it can be easily identified. In Appendix A.1 a screenshot can be found of how this would look like.

4.3 Query language

In order to filter through the resulting frequent itemsets, a query language has been created. The language is designed to be easy even for people without experience in query languages, but still also provide additional functionality for people with experience in query languages such as the ability to filter on the negations and the ability to use logical AND and OR. In order to be easy and intuitive to use, a user can simply type a word and it will filter the results to only show results containing that string. It also allows for filtering on multiple words by separating them with a comma. This functionality is important for domain experts who already have certain ideas of what could be useful or maybe only the data containing a specific word is relevant to them. A user can also search more strictly by surrounding a string with quotation marks, the string must then fully match.
A user can also filter on itemsets that do not contain a word by simply placing it in not() . These results can then further be filtered by appending words after it separated by commas.
For the more advanced part of the query language, a user can choose to perform multiple queries and show them together by surrounding the individual queries in parentheses and separate them

with 'OR' or a space. In these individual queries, a user can for instance filter for frequent itemsets containing an item A and not containing a certain other item B. This result can then be appended to all frequent itemsets containing item B. In order to maintain readability, commas are interchangeable with 'AND'. The 'AND' can only be used with the imprecise filter if it is surrounded with brackets. To make the user aware that the syntax of a query is incorrect, it will color the background in red. It also colors it green when the syntax is correct and show a counter for the number of results, so that small changes can be more easily detected.
Table 3 describes the main functionality supported by the query language, with corresponding examples.

Query functionality	**Example**
Filter on itemsets containing certain string	Margarine
Filter on itemsets containing a precisely matching item	"tropical fruit"
Filter on multiple items(precise and imprecise)	coffee, bottled "coffee", "bottled water"
Combine queries with logical AND	(carrot) AND (potato) carrot, potato
Filter on not containing items	potato, not(carrot) (potato) AND not(carrot) not("whole milk")
Combine queries with logical OR	(carrot) (potato) (carrot) OR (potato) (carrot) OR ("whole milk" AND "potato")
Apply filter on a combination of subqueries	((carrot) OR ("whole milk" AND "potato")) AND not("beef")

Table 3: Query language functionality with examples

5 Implementation

The application is browser-based and is implemented in HTML5, CSS3 and JavaScript. Sending all the data from the user to a server for processing is not a feasible option for security and bandwidth reasons. Therefore all the data processing is done on the user's computer in JavaScript. Since JavaScript is relatively slow, several optimizations had to be made to keep the application responsive.

5.1 Query language implementation

5.1.1 Ohm js

The query language is implemented using an external library called Ohmjs. [15] This open source JavaScript library allows for describing the syntax of the language using a Parsing expression grammar(PEG), which can be used to describe a formal grammar. Furthermore it supports adding semantics to the grammar. This functionality is all available through a single JavaScript file which

can be run in the browser. This library allows for creating functions in JavaScript which handle the processing of individual rules in the formal grammar, which makes it easy to adjust or extend the language created with it.

5.1.2 Formal definition of the query language

The query language is defined using a parsing expression grammar. This grammar was chosen because it is the only supported type of grammar in Ohm js. Parsing expression grammars are different from context free grammars in that they can not be ambiguous and therefore the parsing tree is always the same. This is due to the fact that parsing expression grammars always pick the top most choice possible in the grammar [5]. The following figure describes the grammar used for the query language.

⟨*Exp*⟩ ::= ⟨*Exp*⟩ ‘`,`’ ⟨*Exp*⟩
  | ⟨*Exp*⟩ `and` ⟨*Exp*⟩
  | ⟨*Exp*⟩ `or` ⟨*Exp*⟩
  | ⟨*Exp*⟩ ⟨*Exp*⟩
  | `(` ⟨*Exp*⟩ `)`
  | ⟨*Filter*⟩

⟨*Filter*⟩ ::= ⟨*PreciseWord*⟩ ‘`,`’ ⟨*Filter*⟩
  | ⟨*Word*⟩ ‘`,`’ ⟨*Filter*⟩
  | ⟨*Not*⟩
  | ⟨*PreciseWord*⟩
  | ⟨*Word*⟩

⟨*Not*⟩ ::= ‘`not(`’ ⟨*Filter*⟩ ‘`)`’

⟨*PreciseWord*⟩ ::= ‘`"`’ ⟨*Word*⟩ ‘`"`’

⟨*Word*⟩ ::= ⟨*Word*⟩ ‘`/`’ ⟨*Word*⟩
  | ⟨*Word*⟩ ‘`&`’ ⟨*Word*⟩
  | ⟨*letter+*⟩
  | ⟨*digit+*⟩
  | ‘`""`’

Figure 3: Parsing expression grammar used for the query language

Each of the rules described in figure 3 has its own corresponding semantics. This is defined in Ohm js by adding captions after every rule which define which JavaScript function is called, automatically filling in the correct function arguments. This allows the grammar to be easily changed and extended.
When multiple expressions are combined in the language, all of them are computed individually and contain all items that they would if they were requested separately. They are then combined depending on what operation separates them. This allows the query language to be easily extended

with more selectors, such as minimum number of items in a frequent itemset, by adding it in the PEG and adding a filter for it.

5.2 Optimizations

Frequent itemset mining is relatively straight forward. A set of all possible sets of items can be created from the data and then individually tested with the data to see if their support is above a certain threshold. This however is a very slow and inefficient process, especially in JavaScript. Therefore several optimizations have to be made in order to make the data processing near real time for relatively small data sets.

5.2.1 Bitmasks

An important optimization is the use of bitmasks to save precious computing time. Before the application starts looking for all the frequent item sets, it first finds all the unique items in the data. These unique items are labeled with a number starting from 1 to the number of unique items. Transaction in the data can then be represented as a series of bits with the length of the number of unique items. The positions of zeros and ones then encodes for the corresponding items in a certain transaction. This is implemented in the form of an array of integers, each integer consists of 32 bits in JavaScript and therefore encodes for 32 items. We will call this array of integers a bit-string.
Aside from the obvious required memory reduction, a significant computational performance gain can be obtained using these bit-strings. Since it is relevant to know if a certain set 'A' contains another set 'B' in frequent itemset mining, we can use the logical AND operation on two bit-strings and then compare with the original set 'B'. If the resulting set is the same as the set 'B', then it contains it and otherwise it does not. This logical AND operation is performed here by performing a logical AND on all the individual integers in the array.

5.2.2 Pruning

Due to the nature of frequent itemsets, if an itemset is frequent (has a support bigger than the set threshold), then all subsets of that itemset are also frequent. The opposite is also always true, if an itemset is infrequent, then all supersets of that set are also infrequent. This allows us, in most cases, to prune a lot of candidate frequent itemsets and thus severely decrease our search space. The candidate sets can be created dynamically, or they can all be created before testing if they are frequent. For big itemsets it is necessary to create them dynamically because the number of candidates grows with the number of unique items in the data with a factor 2^d with d the number of unique items. [2] Creating the candidates dynamically can be done by labeling all the items with unique integers. The algorithm then starts with only the individual items as candidates. All the candidates are then checked for frequency and removed if they are infrequent. After finishing the previous step new candidates will be created in the following way.
For all candidates that remain: create new candidates by adding a single item to the candidate which integer label is bigger than the last item of that candidate. This is done for all items which are bigger than the last item.
These new candidates then go through the same process of testing them and creating new candidates. By only adding items which are bigger than the last item of the candidate it ensures that no duplicate candidates are created and tested.

5.2.3 Item mapping

Looking through every transaction in the data to check if a candidate is a subset of that transaction is rather slow, especially for larger data sets. In order to decrease the number of checks it has to perform, references to the transactions are stored in maps based on if a certain item is contained in that transaction. When testing a candidate, the maps corresponding to the items are checked for their size and the smallest is selected. Because this map contains all the items that correspond to 1 of the items of the candidate, this map contains all the transactions that are possible supersets of this candidate. We can then test if the candidate is frequent based on the transactions in this map, this drastically decreases the number of checks we have to perform in most cases. And thus gives a reduction in overall computation time needed.

6 Evaluation

This chapter contains the results and discussion of a preliminary user study that was performed to evaluate the usability and effectiveness of the query language. First, the testing methodology will be discussed, then the setup of the experiments will be described in 6.1 and the results can be found in Chapter 7.

Testing the usability and effectiveness of the application would preferably be done with an A/B-test [16] where the application would be put head to head with another, widely used, application such as RapidMiner [11]. Ideally, two different evenly sized groups of people would then be instructed to perform the same tasks with the two different applications. These results could then be compared and the applications performance in such tasks could be evaluated. The people that would test the application should also be a representative group for the people that the application is intended for, i.e., domain experts with little to no experience in data mining. This testing methodology however also requires a large sample size compared to a qualitative test. For this reason, a compromise was made and a qualitative test was chosen and due to it being significantly easier to find computer science students to test the application they were selected. Computer science students however are not a representative group of people for domain experts without experience in data mining. Computer science students however do have experience with other query languages. This means that if a computer science student has difficulties with an application with its own query language it is logical to assume that people without any experience with query languages would have difficulties as well. Furthermore, there are some parts of the application that can be tested without the need of a representative group such as the general interaction with the application by the user.

6.1 Experiment setup

For the user testing, 8 computer science students were individually tested. In this section the individual that is being tested will be referred to as the user. During the test, the user is asked to fill in several questions, in the form of a questionnaire in Google Forms, while exploring and performing actions in the application. While this test is being performed, we observe all the actions the user makes through the use of the screen share option in Discord. To make sure that the user is able to continue the test if he gets stuck on something, he is allowed to ask

citrus fruit,semi-finished bread,margarine,ready soups
tropical fruit,yogurt,coffee
whole milk
pip fruit,yogurt,cream cheese,meat spreads
other vegetables,whole milk,condensed milk,long life bakery product
whole milk,butter,yogurt,rice,abrasive cleaner
rolls/buns
other vegetables,UHT-milk,rolls/buns,bottled beer,liquor (appetizer)
potted plants
whole milk,cereals
tropical fruit,other vegetables,white bread,bottled water,chocolate
citrus fruit,tropical fruit,whole milk,butter,curd,yogurt,flour,bottled water,dishes
beef
frankfurter,rolls/buns,soda
chicken,tropical fruit
butter,sugar,fruit/vegetable juice,newspapers
fruit/vegetable juice

Table 4: A snapshot of the data set

questions to the observer but is asked to keep it to a minimum. If the user makes the application behave in a way that is unintended, the observer will also be able to intervene and explain what it should be doing and how to bring the application in a state from which the user can continue.

The application has a pre-loaded sample data set that is used for testing. To load the data the user has to click a button next to the button where he would normally upload a data set. This data set [1] consists of transactions in a supermarket in the form of a list of unique items that were bought in a single transaction. The data set contains 1576 transactions with an average of 4.5 items per transaction from a total of 166 unique items. A sample of this data set can be found in 4. This data set was chosen because it is easy to understand associations between items in a supermarket especially for computer science students. In other words, everyone is to some degree a domain expert when talking about transactions in a supermarket. This allows the user to be able to more easily find patterns in the data compared to other data sets where he would not be a domain expert.

The experiment starts off with a question to the user to rate their experience in data analysis. The observer then shortly verbally outlines a scenario to the user. In this scenario, the user is described that he works for a supermarket and that he is a domain expert. He is tasked with finding out interesting information that could be useful for that specific supermarket which could be used to develop a strategy to possibly increase their revenue. He has then uploaded a CSV containing 1600 transactions from this supermarket, which is the data loaded by clicking load sample data in the application. He is also briefly explained what frequent itemsets are in the following form: "Frequent itemsets that are seen in the application are subsets of items from transactions in the database that occur in at least ten different transactions."

With this information in mind, the user is asked in the questionnaire to find three frequent itemsets that he thinks are interesting. He is also asked to briefly describe in once sentence why he thinks each frequent itemset is interesting. This question is designed to introduce the user to the application and to see if he is able to explore the application without problems and if he can find interesting patterns. The user is then asked to find five frequent itemsets that contain 'whole milk'. The purpose of this question is to see if a user will use the query field to search for whole milk.

The next three questions that are asked are there to test the query language and to find barriers that the user might encounter trying to work with the query language. The user is asked to create three queries matching the following.

1. A query that shows all Frequent Itemsets that contain 'tropical fruit' and 'whole milk' but not 'root vegetables' in the same Frequent Itemset.
2. A query that shows all Frequent Itemsets that contain any item containing the word: vegetables
3. A query that shows all Frequent Itemsets that contain 'whole milk' or 'berries' but without showing any Frequent Itemsets containing both 'berries' and 'beef' together.

Three examples of correct answers are the following.

1. ("tropical fruit", "whole milk"), not("root vegetables")
2. vegetables
3. (("whole milk") OR ("berries")) AND not("berries", "beef")

The user is then asked to rate certain parts of the application on a scale from 1 to 10. He is asked to rate how useful he finds the data exploration part of the application, i.e., being able to sort the table and being able to see all transactions containing a certain frequent itemset. He is asked to rate how useful he found the query language. Finally he is asked to rate how well the data is displayed in the table.
The user is then asked to describe any problems or barriers that he encountered trying to find the frequent itemsets. After that, the user is given room to give constructive feedback on the application.
The complete questionnaire can be found in appendix A.2

7 Results

In this chapter the results from the preliminary user study are shown.
The participants were asked to give a list of the 3 frequent itemsets which they found the most interesting with an explanation, why they chose them. From these results only 4 of them were given by more than one participant. The reason why participants chose these frequent itemsets was all the same: for all frequent itemsets with a certain number of unique items, they chose the itemset that occurred the most. The most commonly answered frequent itemset was the itemset with the most number of occurrences from all the sets with the most number of unique items. This was followed by

the set with the most occurrences with only one unique item, in other words, the most sold item. Followed by two frequent itemsets which had the most occurrences of all sets with only two unique items.

In the first question, participants were asked to create queries based on a sentence describing the results. With only one exception; every question was answered correctly by all participants. The only query that was answered incorrectly was vegetable, while the question was to give all frequent itemsets containing the word vegetables. The following table shows the answered queries given by the individual participants, where P1 through P8 refers to the different participants.

	Create a query that shows all Frequent Itemsets that contain 'tropical fruit' and 'whole milk' but not 'root vegetables' in the same itemset.	Create a query that shows all Frequent Itemsets that contain any item containing the word: vegetables
P1	((berries) (whole milk)), not(berries, beef)	vegetables
P2	(tropical fruit, whole milk), not(root vegetables)	vegetable
P3	(tropical fruit, whole milk) AND not(root vegetables) (tropical fruit, whole milk), not(root vegetables)	vegetables
P4	(Tropical Fruit,whole milk), not(root vegetables)	vegetables
P5	tropical fruit, whole milk, not(root vegetables)	vegetables
P6	(tropical fruit, whole milk), not(root vegetables)	vegetables
P7	("tropical fruit") AND ("whole milk") AND not("root vegetables")	vegetables
P8	whole milk, tropical fruit, not(root vegetables)	vegetables

	Create a query that shows all Frequent Itemsets that contain 'whole milk' or 'berries' but without showing any transactions containing both 'berries' and 'beef' together.
P1	((berries) (whole milk)), not(berries, beef)
P2	((whole milk) (berries)), not(berries, beef)
P3	((whole milk) OR (berries)), not(berries, beef)
P4	((whole milk) or (berries)), not(berries, beef)
P5	((whole milk) OR (berries)) AND not(berries, beef)
P6	((whole milk) OR (berries)), not(berries, beef)
P7	("whole milk" OR "berries") AND not("berries" , "beef")
P8	(whole milk, not(berries, beef)) or (berries, not(beef))

Table 5: User submitted queries

The results from the user scoring can be found in table 6. It contains the mean, standard deviation(Std Dev), minimum (Min), and maximum (Max) of the scores.

Questions	Mean	Std Dev	Min	Max
Rate how useful you find the data exploration part of the application. (being able to sort the table, click on frequent itemsets to see all occurences)	8	1,414213562	6	10
Rate how useful you find the query language.	8	1,309307341	6	10
Rate how well the data is displayed in the table.	7.125	1,125991626	5	8

Table 6: User scoring from 1 to 10

The participants were asked to describe any problems or barriers they encountered while trying to find and evaluate frequent itemsets. The most common problem the participants seemed to have was finding that there was an explanation to the query language. This was also clearly visible while observing the participants. Some participants tried random queries to see if it would do anything and two of the participants had to be told that there was an explanation balloon after they got stuck. After that the most common issue was that some users created queries that seemed to be correct but did not return the correct results. Lastly, some users seemed to have trouble seeing which column the table was sorted on due to the location of the sorting arrows.

In the final question, the participants were asked to give any additional feedback. The following list shows the improvements suggested by the participants that they would like to see. These suggestions are slightly paraphrased.

- Add the ability to filter on number of items and number of occurrences in the query language.
- Show for instance only "beef, water" and compare that to only "beef" and only "water" in one single query.
- Add the ability to sort all items in the frequent itemset.
- When querying for items, entries would be easier to compare if the items that the user is looking for are always in the same position, preferably at the start of the entry.
- Add support for regex in the query language.
- Filter with Natural Language Processing.

8 Discussion

This chapter discusses the results and limitations of the research. Furthermore, it will present some suggestions for future research.

In this thesis, there were two main problems with the user testing that could have been done better. First of all, the user testing was done through the use of of a qualitative questionnaire. This would have been done better through the use of an A/B test so the results would have been more objective. The next problem is that the participants were not domain experts and also had experience with data analysis which is in contrast with the research question this thesis tried to answer. The data acquired from these tests however does contain valuable information. This is because if people with

experience with query languages have difficulties with a new query language, people without any experience with them would likely have more problems.
From the results we can see that computer science students answered all questions correctly where they had to create a query based on a sentence describing the results correctly, with only one exception. From this we can say that the explanation for the query language is likely enough for computer science students since the questions were already fairly complicated. This however does not necessarily mean that domain experts without any experience with query languages will be able to easily understand it.
The participants also seemed to have trouble finding the explanation of the query language, with some even requiring to be told where the info button was located. This could be improved in later work by either making the help button stand out more, or by giving a popup menu when clicking the query field for the first time.
From the answers to the questions where the participants were asked to rate parts of the application we can see that they gave high scores with almost every answer being above a 6 with the only answer lower than a 6 being for how the data is displayed in the table. Furthermore the averages for the usefulness of the query language and the data exploration part of the application were both an 8 with a relatively low standard deviation. From this we can fairly confidently say that computer science students find the application useful, but there is room to improve on how the data is displayed.
Another issue that the participants brought up was the ability to filter on number of occurrences and number of items. To find all frequent itemsets with only two items, the user would have to scroll down with the list sorted on number of items. This could be improved by allowing the user to filter on number of items or occurrences. This can be done either through the use of the query language, through the use of an extra filter or both.

To get more accurate results as to how well the application will work for domain experts without a background in data analysis another user study will have to be performed. We suggest that this user study consists of an A/B test with the other application being a widely used data mining tool. The participants in this test should then also be domain experts without a background in data analysis.

The application itself was implemented without the use of an large front end framework which led to some sub optimal design choices later on. This could have been done better by starting with a framework and building the application on that. The processing of the data and generation of frequent itemsets is fast for small data sets, but it can become rather long for large data sets even with several optimizations. From our testing we found that 1576 transactions with 166 unique items with an average of 4.5 items in every transaction took slightly less than a second to process on a ryzen 3600x.
For future work in this field we would like to see natural language processing [8] being combined with frequent itemset mining. In such an application a user would be able to describe what he wants to see in plain English and the application will filter the data accordingly. A user would for instance load a data set in CSV format and the application will generate the frequent itemsets just like our application, but instead of making a query which is prone to error, he could ask the application a question without translating it to a query himself. This would make it easier for domain experts without any background in query languages as they would not have to learn it to

find useful information from their data.
Another expansion to the application could be the use of a Pattern Frequency Spectrum as proposed in [14]. This would allow the user to specify how many frequent itemsets he would like to see and the application will then set the minimum support accordingly, thus saving computational time especially for larger itemsets.

9 Conclusion

This thesis tried to answer the question: How can we create a query language that can be used to effectively filter through frequent itemsets by non data-experts, while still allowing for advanced queries? In this thesis we have created an application to create a list of frequent itemsets from a database of transactions in the browser. We also created a query language which is able to successfully filter the results based on written queries in real time.

The query language that was created however seems to have its limitations, as follows from the responses from computer science students who have tested the application. We can not fully answer the research question, due to the participants not being a representative group for the users that the application was originally designed for. To do this, another user test would have to be conducted with a representative group of users and preferably an A/B test would be performed instead of a qualitative questionnaire. Aside from its limitations, the application allows computer science students to effectively filter frequent itemsets and explore them, as follows from Table 6 and the result that all participants were able to provide interesting frequent itemsets with an explanation for why.

References

[1] Groceries dataset. https://www.kaggle.com/irfanasrullah/groceries. Accessed: 2020-04.

[2] C. C. Aggarwal and J. Han. *Frequent Pattern Mining*. Springer International Publishing, 1 edition, 2014.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of 20th Intl. Conf. on VLDB*, pages 487–499, 1994.

[4] M. Bhuiyan, S. Mukhopadhyay, and M.A. Hasan. Interactive pattern mining on hidden data: a sampling-based solution., 2012.

[5] B. Ford. Parsing expression grammars: a recognition-based syntactic foundation. In Neil D. Jones and Xavier Leroy, editors, *POPL*, pages 111–122. ACM, 2004.

[6] B. Goethals, S. Moens, and J. Vreeken. Mime: a framework for interactive visual pattern mining. In *In: Proceedings of KDD'11.*, pages 757–760, 2011.

[7] F. Herrera, C. J. Carmona, P. González, and M. J. Del Jesus. An overview on subgroup discovery: Foundations and applications. *Knowledge and Information Systems*, 29:495–525, 12 2011.

[8] D. Jurafsky and J. H. Martin. *Speech and language processing : an introduction to natural language processing, computational linguistics, and speech recognition*. Pearson Prentice Hall, 2009.

[9] A. Loraine Charlet and D. Ashok Kumar. Market basket analysis for a supermarket based on frequent itemset mining. *IJCSI International Journal of Computer Science Issues, www.IJCSI.org*, 9:1694–0814, 09 2012.

[10] M. Meeng and A. J. Knobbe. Flexible enrichment with cortana—software demo. In *In: Proceedings of Benelearn*, pages 117–119, 2011.

[11] I. Mierswa and R. Klinkenberg. Rapidminer studio (9.1) [data science, machine learning, predictive analytics]. retrieved from https://rapidminer.com/, 2018.

[12] S. Nijssen and A. Zimmermann. Constraint-based pattern mining. In C. C. Aggarwal and Han J., editors, *Frequent Pattern Mining*, pages 147–163. Springer, 2014.

[13] M. van Leeuwen and L. Cardinaels. VIPER - visual pattern explorer. In *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2015, Porto, Portugal, September 7-11, 2015, Proceedings, Part III*, volume 9286 of *Lecture Notes in Computer Science*, pages 333–336. Springer, 2015.

[14] M. van Leeuwen and A. Ukkonen. Fast estimation of the pattern frequency spectrum. In Toon Calders, Floriana Esposito, Eyke Hüllermeier, and Rosa Meo, editors, *Machine Learning and Knowledge Discovery in Databases*, pages 114–129, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[15] A. Warth. Ohm. https://github.com/harc/ohm, 2020.

[16] S. Young. Improving library user experience with a/b testing: Principles and process. *Weave: Journal of Library User Experience*, 1, 08 2014.

A Appendix

A.1 Application overview

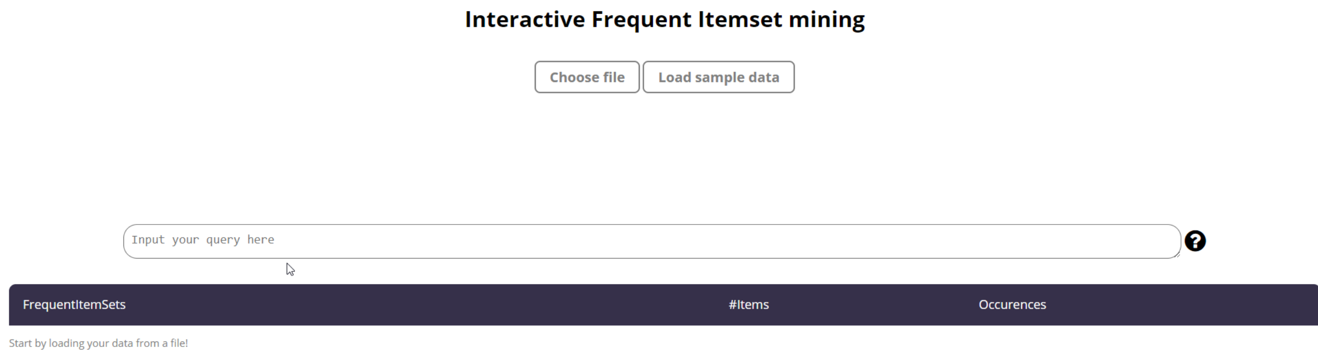

Figure 4: Opening page of the application

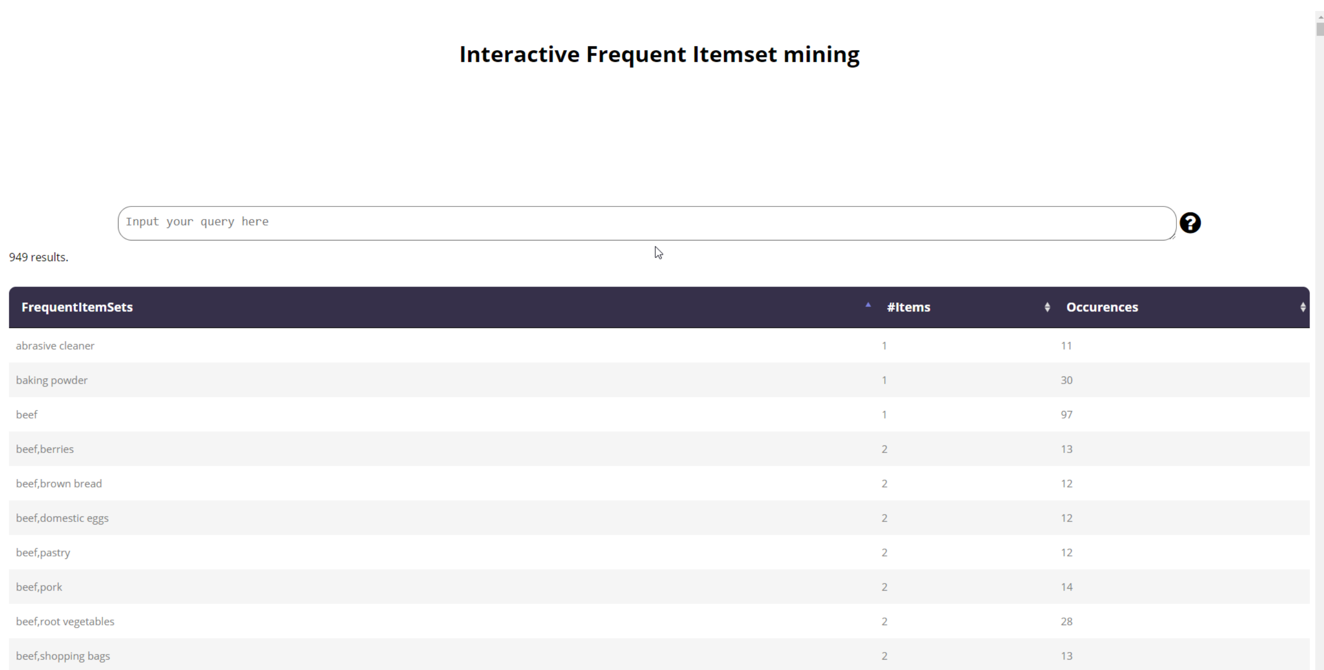

Figure 5: Application after loading a data set

Transactions	#Items	Occurences
Beef, Berries, **Root Vegetables**, Whole Milk, Dessert, Spread Cheese, Brown Bread	7	1
Beef, Berries, **Root Vegetables**, Whole Milk, Whipped/Sour Cream, Abrasive Cleaner	6	1
Beef, Citrus Fruit, Berries, **Root Vegetables**, Brown Bread, Detergent	6	1
Beef, Citrus Fruit, **Root Vegetables**, Onions, Curd, Hard Cheese, Rolls/Buns, Fruit/Vegetable Juice	8	1
Beef, Grapes, Nuts/Prunes, **Root Vegetables**, Onions, Other Vegetables, Cream, Soft Cheese, Sliced Cheese, Hard Cheese, Domestic Eggs, Rolls/Buns, Pastry, Mustard, Specialty Chocolate, Specialty Bar, Dental Care, Shopping Bags	18	1
Beef, Pip Fruit, **Root Vegetables**, Rolls/Buns, Margarine	5	1
Beef, **Root Vegetables**	2	1
Beef, **Root Vegetables**, Bottled Beer, Chocolate, Shopping Bags	5	1
Beef, **Root Vegetables**, Other Vegetables, Butter, Yogurt, Rolls/Buns, Flour	7	1
Beef, **Root Vegetables**, Other Vegetables, Rolls/Buns, Sugar	5	1
Beef, **Root Vegetables**, Other Vegetables, Soda, Bottled Beer, Hygiene Articles	6	1
Beef, **Root Vegetables**, Other Vegetables, Whole Milk, Butter, Soft Cheese, Hard Cheese, Domestic Eggs, Pasta, Coffee, Candles	11	1
Beef, **Root Vegetables**, Whole Milk, Butter Milk, Yogurt, Beverages, Cream Cheese, Sweet Spreads, Soda, Chocolate	10	1
Beef, **Root Vegetables**, Whole Milk, Cream Cheese	4	1
Beef, Tropical Fruit, Pip Fruit, Grapes, **Root Vegetables**, Processed Cheese, Rolls/Buns, White Bread, Pastry, Pasta, Specialty Vegetables, Cat Food, Soda, Misc. Beverages	14	1
Beef, Tropical Fruit, **Root Vegetables**, Herbs, Other Vegetables, Whole Milk, Yogurt, Hard Cheese, Rolls/Buns, Brown Bread, Bottled Water, Bottled Beer, Salty Snack, Shopping Bags	14	1
Beef, Tropical Fruit, **Root Vegetables**, Onions, Other Vegetables, Instant Coffee, Soda	7	1

949 results.

FrequentItemSets

abrasive cleaner
baking powder
beef
beef,berries
beef,brown bread
beef,domestic eggs
beef,pastry
beef,pork
beef,root vegetables
beef,shopping bags 2 13

Figure 6: Application showing all transactions containing a certain frequent itemset

A.2 Questionnaire

The questionnaire was created and presented in google forms and is attached below.

Frequent Itemset exploration

*Vereist

1. Name *

2. Rate your experience with data analysis. *

Markeer slechts één ovaal.

	1	2	3	4	5	
unexperienced	◯	◯	◯	◯	◯	experienced

3. Compose a list of 3 Frequent Itemsets that you think are the most interesting. And for each give a 1 sentence explanation as to why you think it is interesting. *

4. Compose a list of 5 Frequent Itemsets that contain 'whole mik'. *

5. Create a query that shows all Frequent Itemsets that contain 'tropical fruit' and 'whole milk' but not 'root vegetables' in the same itemset. *

6. Create a query that shows all Frequent Itemsets that contain any item containing the word: vegetables *

7. Create a query that shows all Frequent Itemsets that contain 'whole milk' or 'berries' but without showing any transactions containing both 'berries' and 'beef' together. *

8. Rate how useful you find the data exploration part of the application. (being able to sort the table, click on frequent itemsets to see all occurences) *

Markeer slechts één ovaal.

1	2	3	4	5	6	7	8	9	10
⬭	⬭	⬭	⬭	⬭	⬭	⬭	⬭	⬭	⬭

9. Rate how useful you find the query language. *

Markeer slechts één ovaal.

1	2	3	4	5	6	7	8	9	10
⬭	⬭	⬭	⬭	⬭	⬭	⬭	⬭	⬭	⬭

10. Rate how well the data is displayed in the table. *

Markeer slechts één ovaal.

1	2	3	4	5	6	7	8	9	10
⬭	⬭	⬭	⬭	⬭	⬭	⬭	⬭	⬭	⬭

11. Which problems(barriers) did you encounter trying to find the Frequent Itemsets and evaluating them? *

12. Additional feedback. *

Deze content is niet gemaakt of goedgekeurd door Google.

Google Formulieren