Apriori-based Frequent Pattern Mining
Apriori-based Frequent Pattern Mining

by Ferenc Bodon

Under the supervision of
Dr. Lajos Rónyi, Tadeusz P. Dobrowiecki

Department of Computer Science and Information Theory
Budapest University of Technology and Economics

Budapest
2006
Alulfrott Bodon Ferenc kijelentem, hogy ezt a doktori értekezést magam készítettem és abban csak a megadott forrásokat használtam fel. Minden olyan részt, amelyet szó szerint, vagy azonos tartalomban, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.


Bodon Ferenc

Az értekezés bírálatai és a védésről készült jegyzőkönyv megtekinthető a Budapesti Műszaki és Gazdaságtudományi Egyetem Villamosmérnöki és Informatikai Karának Dékáni Hivatalában.
Acknowledgement

First and foremost, I wish to thank my supervisor Lajos Rónyai. He patiently and thoroughly guided me for an almost 6-year period starting from a read-and-research course of my graduate years to reading the chapters of this thesis. He introduced me to the subject of data mining; kindly accepted me at his Informatics Laboratory at the SZTAKI in 2000; and since then he has been providing me with constant support for my research. I would like to thank my second supervisor Tadeusz P. Dobrowiecki for helping me starting my academic career.

I am grateful to the Department of Computer Science and Information Theory at the Budapest University of Technology and Economics and the Web Search and Data Mining Group at MTA SZTAKI, the two pleasant and stimulating environments in which my PhD research was carried out. I wish to thank the head of the department, András Recski for his support, encouragement and advice about career decisions.

Among my colleagues in the Informatics Laboratory, I am especially grateful to Balázs Rácz for a successful research period that we spent together. The thesis benefited a lot from his comments and from our discussions. I am much in debt to Lars Schmidt-Thieme for his ideas and for sharing his experiences with me. Above all, I would like to thank him for the invitation to the University of Freiburg. It made me very proud to be an equal partner during my research with him and Balázs. The joint work in Freiburg gave the final and most important impulse to this thesis.

I am also grateful to Zoltán Hornák for inviting me to the Remote Supervision Project and for helpful discussions. The results of the cooperation are presented in Chapter 6.

I am much in debt to Tamás Sarlós, Katalin Friedl, András Lukács, Dániel Marx and Nimród Major who read some of my manuscripts and provided useful comments and suggestions.

I acknowledge the financial support of the OTKA grants T 42481, T 42706, TS-04733 of the Hungarian National Science Fund, the NKFP-2/0017/2002 project Data Riddle, and a Madame Curie Fellowship (IHP Contract nr. HFMN-CT-2001-00251).

I would like to thank the trust and love of my family following me through the university years. Without the constant pushing and asking for the thesis I might be still working on other papers and postpone repeatedly this summarizing work.

Last but not least, I am grateful to my friends for pulling me out of my research
cave and provided funny and diverse programs throughout many years. In no particular order: Tamás Bereczki, for being the most productive partner in extending our transience-graph; László Gillemot, for the adventures in the Leon; Norbert Oláh, for opening a new world with sharing his windsurf with me; János Tapolcai, whose brassy smile after his PhD defense gave me a great influence to pull myself together and start writing my thesis; Keppány Kelemen for being my partner in the Bubu Fan Club; Tamás Csoknyai for the unforgettable parties at Velence; Csaba Kiss for revealing the undiscovered possibilities of the English language and of course to Dávid Horváth, Réka Tabák, Gergely Wesztergem-Szabó, Gergő Kovács, Gábor Zsoldos, Barna Kiss, Dániel Tófalvi, etc.
Notation

General
\( \mathcal{J} \) the sequence of transactions, i.e., the transaction database
\( \text{cover}_\mathcal{J}(p) \) the cover of pattern \( p \) in transaction database \( \mathcal{J} \)
\( \text{sup}_\mathcal{J}(p) \) the support of pattern \( p \) in transaction database \( \mathcal{J} \)
\( \text{uw-sup}_\mathcal{J}(p) \) the unweighted support of pattern \( p \) in transaction database \( \mathcal{J} \)
\( \text{minsup} \) minimal support threshold
\( \text{minfreq} \) minimal frequency (relative support) threshold
\( F_\ell \) set of frequent patterns of size \( \ell \)
\( C_\ell \) set of candidate patterns of size \( \ell \)
\( t, t_i, t_k \) transactions
\( \prec \) a partial order
\( \prec \) a transitive binary relation
\( NB(F) \) negative border of set \( F \)
\( NB^\prec(F) \) order-based negative border of set \( F \)

Frequent Itemset Mining
\( \mathcal{I} \) the set of all items
\( S \) a set of itemsets with downward closure property
\( i, i_j, i_k \) items
\( I, I', \ldots \) subsets of \( \mathcal{I} \)
\( \mathcal{T}^\prec(F) \) The trie that stores set of itemsets \( F \) and uses ordering \( \prec \)
\( \mathcal{T}_{OPT}(F) \) A minimum-size trie that stores set of itemsets \( F \)
\( \prec_{DESC} \) the descending order of items according to the supports
\( d \) depth of a node in the trie
\( u, v \) nodes in the trie

Unified Framework of Frequent Pattern Mining
\( (P, \leq) \) a poset
\( | \cdot | \) a size function
\( PC \) a pattern context
\( MPUB(F) \) the minimal proper upper bound of set of patterns \( F \)
# Contents

1 Introduction 1
  1.1 The importance of Apriori-based approach 1
  1.2 Our contributions 2
    1.2.1 Improving the support count procedure 3
    1.2.2 Improving candidate generation 4
    1.2.3 Further results 5
  1.3 Impact of the results 6
  1.4 A Frequent Pattern Mining Template Library 7
  1.5 Organization 8

2 Basic Concepts 11
  2.1 The Frequent Itemset Mining Problem 11
  2.2 The arena of FIM algorithms; a short history 13
  2.3 Techniques 14
  2.4 Algorithmic aspects of the modern processors’ features 15
    2.4.1 Memory hierarchies, data locality 15
    2.4.2 Pipeline processing, branch prediction 16
  2.5 Graphical presentation of the experiments 16
  2.6 Bottom-up FIM algorithms 17
  2.7 Breadth-first, iterative vs. depth-first, recursive algorithms 18
  2.8 Apriori algorithm 19

3 Methods to Improve Support Count 21
  3.1 The trie and its variants 22
    3.1.1 The representation of the list of edges 23
    3.1.2 Index vs. pointer-based trie 24
    3.1.3 Patricia trie 26
  3.2 The trie of Apriori 26
3.2.1 Support Counting ........................................ 26
3.2.2 Removing Infrequent Candidates .................. 27
3.2.3 Candidate Generation ................................. 27
3.3 Compactness of the trie .............................. 27
3.4 Improvements used in Apriori ....................... 28
3.5 Inhomogeneous trie .................................... 31
  3.5.1 Routing strategies at the nodes ................. 33
3.6 Removing Dead-end Branches ......................... 39
3.7 Caching the transactions ............................. 41
3.8 Equisupport pruning .................................. 45
  3.8.1 Prefix equi-support pruning .................... 45
  3.8.2 Level 2 equisupport pruning ................. 48
  3.8.3 Level 2 equisupport pruning and further dead-end pruning .......................... 50
3.9 Further techniques .................................. 54
  3.9.1 Support count of Christian Borgelt .......... 54
  3.9.2 Filtering unimportant items from the transactions ..... 55

4 Candidate Generation .................................. 57
  4.1 Intersection-based pruning ......................... 57
  4.2 Omitting complete pruning ........................ 60
  4.3 Summary of the techniques ......................... 62
  4.4 The influence of item ordering .................... 63
    4.4.1 The order-preserving assumption ............ 63
    4.4.2 The number of candidates .................... 68
    4.4.3 Size of the trie ................................ 68
    4.4.4 Support count ................................ 71
    4.4.5 Pruning efficiency ............................. 72
  4.5 The battle of Apriori implementations .......... 74

5 A general model of FPM ................................ 77
  5.1 Frequent Pattern Mining ........................... 77
  5.2 Describing some algorithms at the general level ........................................... 81
    5.2.1 Toivonen technique .......................... 81
    5.2.2 Common description of Eclat and FP-growth ........................................... 83
  5.3 Conclusion ........................................ 85

6 Filtering False Alarms ................................ 87
  6.1 Introduction .................................. 87
  6.2 Related Work .................................. 88
  6.3 False Alarms ................................. 89
  6.4 A Formal Definition of Episode Mining .......... 90
6.4.1 Support and Alarm Support .................................. 93
6.4.2 The Aim of Data Mining ..................................... 94
6.5 The Algorithm ABAMSEP ....................................... 95
   6.5.1 Candidate Generation ..................................... 97
   6.5.2 Determining the Support .................................. 98
   6.5.3 An Optional Step ........................................... 100
   6.5.4 Completeness and nonredundant generation ............. 100
   6.5.5 Theoretical Remarks on the Time and Memory need ...... 102
6.6 Experimental Results ........................................... 103
6.7 Consequences and Future Research ............................. 104

7 References ....................................................... 109
Chapter 1

Introduction

Frequent pattern mining (FPM) is a relatively young subfield of data mining. It aims to find frequently occurring patterns in an input data set. We focus on the largest family of FPM, where the input data set is given as a sequence of transactions and the frequency of a pattern is given by the number of transactions that contain the pattern. The frequency threshold is set by the user. The types of the patterns and transactions may be of many kind, such as sets, sequences, trees, graphs.

Agrawal et al. [7] presented the problem that appeared as a subproblem in analyzing supermarket transaction data to better understand customers' behavior. The customers' habits were modeled by association rules. This new approach turned out to be very useful tools for marketing purposes. An association rule shows a connection between sets of products. For example the rule “digital camera \(\Rightarrow\) memory card (85%, 1300, 3)” states that 85% of the people who bought digital camera also bought a memory card. The two products were contained in 1300 baskets. This parameter is called the support of the rule. The third parameter tries to capture the dependence of the antecedent and consequence. It states that the relative frequency of the memory card is three times higher in the baskets that contain digital camera than the relative frequency in all baskets. The most profitable pricing strategies, however, are based on association rules whose support parameter is high, i.e the products of the rule are bought by many people; they are frequent product sets.

To determine the interesting rules, the frequently occurring sets of products (itemsets) have to be mined first. Many algorithms were proposed to solve the frequent itemset mining (FIM) problem efficiently. The methods and algorithms developed here turned out to be useful in other data mining fields like classification [54], clustering [49, 60], and functional dependency discovery [55, 56]. The association rules and the frequent itemsets were used successfully not just in marketing but in other application domains such as improving the efficiency of electronic commerce [94], bioinformatics [110, 116], DNA and protein analysis, medical diagnosis [41, 76, 77], inductive databases [57, 69],
query expansion [89], Web log [73, 109] and bank account analysis.

The basic FIM problem was extended in many applications, which has led to the introduction of hierarchical [39, 50, 100], negative [96], incremental [10, 29, 30, 104], parallel [4, 48] and quantitative FIM [40, 92, 101], FIM with non-universal support threshold [66, 108], temporal [63], cyclic [81] and fuzzy [9, 28] association rules. It became clear, that a disadvantage of association rules is their large number. To select the interesting rules, constraint based solutions were proposed [45, 75, 86]. To avoid the inefficiency of post processing, constraint-based FIM methods were presented, and also received a great deal of attention. Two important kinds of the frequent sets are the maximal [2, 26, 27, 46] and the closed [84, 85, 112, 114] frequent sets; they are also in the focus of intensive research.

In some application domains the itemset model was not general enough [98]. There are efficient algorithms to mine frequent sequences [6, 67, 102], trees [32, 31, 33, 111], graphs [36, 58, 62], and Boolean formulas. These generalizations also enjoy a broad range of applications, including webpage personalization, stock sequence and chemical compound analysis, fraud detection, operations research, etc.

In this work we concentrate on the basic FPM problem.

1.1 The importance of Apriori-based approach

Algorithm Apriori proposed by Agrawal and Srikant [5] in 1994 was the first FIM algorithm that could cope with large datasets. Its modifications and improvements ruled the world of FIM till the prefix based, depth-first, recursive algorithms such as Eclat [115] in 1997 and FP-growth [51] in 2000 entered the scene. Two years later a highly optimized Apriori algorithm proposed by Orlando et al. [79] called DCI that adapted a hybrid support count function overtook the gold medal in the arena of FIM implementations in 2002.

The open source competitions of FIM implementations [43] brought many interesting observations. First of all, it has become clear that there exists no best algorithm that beats all the other algorithms on every database. Second, the basic Apriori algorithm outperformed many new and sophisticated algorithms and in fact, Apriori turned out to be the best at extremely large databases where FP-growth and Eclat based algorithms aborted due to the large memory need. Third, the improved version of DCI was one of the four outstanding implementations. These facts prove that Apriori is an important and appreciated member of the family of FIM algorithms.

Moreover, the merits of Apriori do not end with border or itemset discovery. Its adaptations in frequent sequence, tree and graph mining are also among the most important and most efficient algorithms. Techniques for decreasing the run-time and memory need are often applicable in searching for frequent patterns of other types. Our experiments proved, for example, that a naive adaptation of the basic Apriori to sequence mining
often outperforms an adaptation of a highly optimized Eclat version. It is not clear that prefix-based, depth first recursive algorithms outperform Apriori-based solutions, if we step out of the world of itemsets.

1.2 Our contributions

This thesis deals with the Apriori based solution of the basic frequent pattern mining problem. Apriori is one of the oldest, and it is without doubt the most widely-known solution. Despite the immense research efforts focused on the method and its applications, some major data structure/implementations issues and possible speed-up methods have left without much consideration. Our main goal was to fill this void.

Apriori scans the transaction datasets several times. After the first scan the frequent 1-itemsets are found, and in general after the \(l\)th scan the frequent \(l\)-itemsets are extracted. The method does not determine the support of every possible itemset. In an attempt to narrow the domain to be searched, before every pass it generates candidate itemsets and only the support of the candidates are determined. In each iteration three major steps are repeated: candidate generation, support count of the candidates, removing infrequent candidates.

1.2.1 Improving the support count procedure

The main bottleneck of Apriori is the support count method. To solve this computation-intensive task, the authors of Apriori proposed hash-tree to store the candidates. Here, we propose an Apriori-optimized version of trie. Our inhomogeneous trie uses a special block allocator and hybrid edge representations (that allows a hybrid support count technique) to store candidate itemsets. This highly optimized solution achieves a much better data locality and fewer number of operations than the traditional implementation of a trie, and hence improves running time.

Beside examining data structure issues, we also investigate algorithmic solutions in order to speed up support count. Three new techniques are proposed. First we point out that unimportant branches of the candidate trie necessarily slows down support count. We show that by using dead-end pruning, we can get rid of such subtrees. We merge this technique with the candidate generation and the infrequent removal procedures. Therefore, we also speed up candidate generation and avoid the two extra traversals of the trie to do the dead-end pruning, which would be the naive solution of dead-end pruning. By a preorder depth-first traversal in the candidate generation, we do candidate generation and dead-end pruning simultaneously.

The second method for improving the support counting is based on the observation that we get many equal transactions if we delete infrequent items from all transactions. The same transactions result in the same traversals in the trie. If a transaction occurs
n times, then it is useful to call the expensive support count procedure just once (with counter increment \(n\)) instead of \(n\) times (with counter increment 1). Thus the number of calls to the most expensive method may be considerably reduced and the I/O and string to integer parsing costs are reduced. This technique results in a faster Apriori only if collecting the same transactions requires less time than the time we spare by reducing the number of support count calls. Here we propose a *Patricia-tree based solution* that combines the advantages of trie and red-black tree based solution. Experiments prove that the run-time of Apriori drops to its fraction at some databases, if transaction collection is used.

These two speed-up techniques reduce run-time greatly (sometimes the gain is of several orders of magnitude), but still they do not make it possible to process dense datasets, in which many nonclosed itemsets occur. Our new *equisupport pruning with a special dead-end pruning* raises Apriori to the elite group of those FIM algorithms that are able to cope with dense databases. The technique is based on a lemma that states that the support of certain itemsets can be calculated directly from the support of some subsets. An important property of our new methods is that they do not require any extra traversal in the candidate trie. Therefore, in those rare databases where the methods do not allow any savings (for example in databases with no non-closed itemsets the equisupport trick is never used), the run-time deterioration is insignificant.

### 1.2.2 Improving candidate generation; safe and hardware independent techniques

As a result of the above techniques, the run-time of support count reduces so much that the candidate generation method becomes the bottleneck of Apriori at many databases. To achieve further improvements, we propose an *intersection-based candidate generation* technique, which determines all possible candidate extensions of a given itemset in one step. This technique requires a modest fraction of movements in the candidate trie, compared to the original method. Again, we try to avoid traversing multiple times the same part of the trie at different points of the candidate generation.

Many techniques are proposed by the FIM research community in the recent years. Most of them are *unsafe*, i.e., they work satisfactorily for databases with certain characteristics, but ruin efficiency if the input data does not belong to the “preferred” group. The limitations, weaknesses and sensitivities of the techniques are hardly ever discussed by the authors. In contrast, the methods proposed by us are safe. Applying them may lead to speed-ups of several orders of magnitude, while the efficiency is not ruined if the database is not preferred by the technique (e.g. a technique based on equisupport pruning is never applied if the database contains no non-closed itemsets). Note, that this kind of run-time and memory safety is not attributed to most of the well-known techniques, including the ones used in algorithms Apriori-Hybrid, DIC, DHP, Partition
or in support lower bound and iteration merging solutions, etc.

Our techniques are also hardware independent, i.e., we believe that they result in more efficient algorithms on all types of modern processors. Our beliefs are supported by run-time results on different processors, but more importantly, we present hardware friendliness diagrams. These diagrams provide much more information than single run-times and allow the generalization to other processors. Note, that this does not hold for run-time based consequences. To illustrate, this let us assume that we propose a method and the experiments show that a new algorithm is faster on Pentium 4 processors. By using profilers and processor counters we understand the reasons of the speed-up. We find that the technique improves the branch prediction facility of the processor, nevertheless the technique decreases data locality, which results in more cache misses. In this case we can not conclude that the technique will also result in a faster algorithm on other architectures. For example in AMD processors the pipe-line is shorter therefore improving branch-prediction may not reduce run-time as much as the increased number of cache misses decrease it. Our experiments on hardware friendliness show that our techniques are not “hardware specific”, improving the efficiency of a processor feature never ruins the efficiency of another feature.

The hardware friendliness analysis is a new approach, which helps better understanding data mining solutions. Since this approach has been worked out together with Balázs Rácz and Lars Sømidt Thieme, it is not a part of this thesis as a contribution.

1.2.3 Further results

Frequent pattern mining is full of beliefs that turned out to be false. One such misbelief is that the efficiency of Apriori roots from its ability of performing complete pruning, i.e., a candidate is not generated, if it has an infrequent subset. Here, we contradict this view. We show that if a proper ordering is used, then a slight modification of Apriori that does not adapt complete pruning, outperforms traditional Apriori at most databases. This issue leads to the analysis of ordering used to convert itemsets to sequences, which greatly influences both run-time and memory need. We investigate the ordering issue and show which orderings are favored by different techniques and methods of Apriori.

We reject the independence assumption (which declares that the frequency of itemset \( I_1 \cup I_2 \) equals to the product of the frequencies of \( I_1 \) and \( I_2 \)) and present a new, more versatile notion, the order-preserving assumption, which makes possible to formally prove the effectiveness of many heuristics.

One of the main reasons for the chaos in frequent pattern mining is the lack of a unified theoretical framework. Concepts, techniques, ideas introduced in a special subfield of FPM are often useful in other subfields of FPM and are often introduced independently by others. Our research in frequent itemsequence mining has shown that techniques and solutions of frequent itemset mining provide a good basis for an efficient algorithm in frequent sequence mining. Didactically we would say, that the hierarchy of
the type of patterns should be examined through a “sliding window”. Many techniques are applicable in the next generalization level, many of them are not applicable, and many new problems also enter the scene. Here, we present a partial order based unified framework of frequent pattern mining, and show some examples of treating certain techniques and notions within this framework.

In the last chapter, we investigate the applicability of frequent pattern mining in intrusion detection. Based on Boolean formulas, we provide a formal description and give an Apriori based solution of the resulting computational problem.

1.3 Impact of the results

After several years of development and research our Apriori implementation became one of the fastest and most memory-efficient implementation. It achieves this strong performance without sacrificing the object-oriented approach, which does not apply to the competitor code [24]. Our Apriori implementation is ranked to the first place by Google, it is downloaded by researchers, students and users from all around the world; the website of the code and the introductory paper is cited by many data mining papers [11, 12, 23, 43, 64, 97]. The implementation is used in basic research of frequent pattern mining and in applied research of other domains. Here we list a few of them.

In an excellent paper about upper and lower bound for the support of itemsets, our Apriori implementation was chosen and extended to support the theoretical foundation with experimental results [97]. Algorithm Apriori is also used in recommender systems. Our code was employed in a subspace clustering approach [1].

A genome research group of Sanger Institute (University of Cambridge) is using the sequence mining version of our Apriori [16] in analyzing DNA sequences. In the business application domain our implementation is employed in detecting performance antipatterns in component-based enterprise systems. For more information the reader is referred to [83].

The most important application of our code may be in epilepsy research conducted by the Intelligent Control Systems Laboratory (ICSL) at Georgia Institute of Technology. Literature suggests that seizure precursors within the electroencephalogram (EEG) may identify brain tissue that is key to the generation of recurrent seizures in patients with epilepsy. Their project investigates if seizure precursors can be exploited to track the development of the diseased tissue, thereby mapping an “epileptic network” in the brain.

The objective of the research at ICSL is to develop an automated methodology to map the epileptic network based upon the detection and statistical analysis of seizure precursors. Our program was one of the data mining tools they employed to process their extremely large datasets. The volume of data made it impossible for them to work with known commercially available systems (such as SAS Enterprise miner, SPSS, etc.) in order to find interesting association rules.
1.4. A FREQUENT PATTERN MINING TEMPLATE LIBRARY

Another usage of our code is in the Pipelined Data Mining Framework. The framework designed and developed by SZTAKI allows the handling and processing of very large datasets using a data stream approach. It combines an abstraction of data source, run-time modularity and configurability with keeping performance and resource management issues in hand. This enables a versatile platform for data mining and data analysis that allows an ordinary desktop or workstation-range computer to perform DM tasks and/or run queries on such large datasets that are much outside the scope of boxed DBMS or statistical systems. Our Apriori implementation was chosen to be embedded in the Pipeline.

We were pleased to learn that the data structures, implementation issues and the methods for speeding-up the search and reducing the memory need proposed by Christian Borgelt and the author, become the basics issues regarding algorithm Apriori. Due to the outstanding efficiency of our trie-based approach and the scalability of this solution the trie become the de facto standard concerning the main data structure of Apriori: all the major and competitive systems have adopted this approach.

1.4 A Frequent Pattern Mining Template Library

Those who believe that their work is of high value, often say, that the main problem of frequent pattern mining is the lack of reproducibility and the impossibility of verification.

In the beginning of the FPM era a typical paper proposed some new techniques, reasoned with some intuitive, informal thoughts and showed its efficiency on some carefully chosen datasets. This procedure led to indignation, because the efficiency of the implementation of the rival algorithm was often significantly below the efficiency of the implementation done by the original authors. The generality, drawbacks, limits of the proposed algorithm were rarely discussed.

Fortunately, this era quickly closed after some famous implementations were made publicly available, and at the conferences of high standards it was required that the proposed algorithms be compared with the known implementations. The level was raised further by the two FIMI competitions. Now we have ultrafast FIM implementations, nevertheless nobody exactly knows why do they perform so well, what are the limitations of the solutions, what kind of input data they prefer. They are like black-boxes, and only the authors can change the parts of the implementation, which is attributed to the highly optimized, non-object oriented codes, which are almost impossible to read by other researchers.

If we would like to understand the performance effect of all parts of a code, we have to make it modularized. This is not a trivial task in a highly optimized environment. In [91] we presented some techniques, which are based on templated and in-line functions, to make a code object-oriented without sacrificing efficiency. To achieve a perfect FPM world, object oriented codes are not enough yet. The codes have to be in a library,
where any part of an implementation can be replaced by another element of the same functionality and any technique can be switched on and off. This way each part of an algorithm can be tested separately and together with other techniques. We can measure how does a certain solution contribute to the final performance, how do different techniques assist or hold back each other.

These principles were followed in building up our FPM template library, which contains our fully pluggable Apriori, Eclat and FP-growth implementations that are competitive with (and in most of the cases outperform) the black-box implementations. For example in our Apriori algorithm different template classes are responsible for doing the support counting, the candidate generation, coding and decoding the items, caching the transaction. All techniques like, dead-end pruning, equisupport extension, etc. can be turned on and off by a template parameter. The data structure is also a template parameter. If it is a trie, then the representation of the list of edges is given by an other template class, in which even the vector representation is pluggable, therefore we can chose STL vector or our lightweight, self-made vector.

The FPM template library made possible to conduct a comprehensive set of experiments with reasonable effort. In a black-box system this would have required a lot of laborious and error-prone work. The library is made publicly available and started to be used by other researchers.

Similar template library, called DMTL, was proposed by Hasan et al. [52]. The purpose of DMTL differs from the purpose of our library. The DTML is more pattern and application oriented, we concentrate on algorithms and data structures. DMTL is an excellent tool for if we would like to mine a new type of patterns using the existing algorithms, and our library comes at hand if we would like to change a part of an existing algorithm. In DTML the algorithms are the building blocks, while in our library we disassemble the methods as much as it makes sense.

1.5 Organization

In Chapter 2 we explain the FIM problem and describe the most important FIM algorithms.

Chapter 3 presents the special trie and our solutions to improve the efficiency of support counting of Apriori. Due to the techniques, it is possible to process large databases with smaller support threshold values. The speed-up of support counting emphasizes the importance of the fine-tuning of candidate generation. Chapter 4 is dedicated to this task.

The lack of a unified FPM framework (and the numerous publications) led to disarray and inconsistency in the consideration of the more or less related methods and problems. In Chapter 5 we present our model and show how to use a pattern specific algorithm in a more general context.
In the last chapter we take a closer look at an application of frequent pattern mining and introduce a formal framework for filtering false alarms. It is based on the discovery of frequently occurring Boolean formulas.
Chapter 2

Basic Concepts

In this section we introduce the basic problem, give a brief overview of the existing algorithms and present a detailed description of algorithm Apriori.

2.1 The Frequent Itemset Mining Problem

In the first application (i.e., market basket analysis) of the frequent itemset mining, the input database contained baskets; each consists of some products (without quantities). In the formal FIM problem the baskets and products are called transactions and items, respectively.

Let \( \mathcal{I} \) be a set of uninterpreted symbols called items. Any subset \( I \subseteq \mathcal{I} \) is called an itemset. Let \( \mathcal{J} = \langle t_1, \ldots, t_n \rangle \) be a sequence of itemsets called data (also called as transaction database). Its elements \( t \in \mathcal{J} \) will be called data itemsets or transactions\(^1\). For any itemset \( I \subseteq \mathcal{I} \) we define the set

\[
\text{cover}_\mathcal{J}(I) := \{ t \in \mathcal{J} \mid I \subseteq t \}
\]

of data itemsets containing \( I \) as the cover of \( I \). The size of the cover

\[
\text{sup}_\mathcal{J}(I) := |\text{cover}_\mathcal{J}(I)|
\]

is called support. Given a lower support threshold \( \text{minsup} \) called minimum support, the set

\[
F_{\mathcal{J}, \text{minsup}} := \{ I \subseteq \mathcal{J} \mid \text{sup}_\mathcal{J}(I) \geq \text{minsup} \}
\]

\(^1\)A large part of the research community defines the data as a multi-set of itemsets or as a binary relation over a set of items and a set of transactions (bipartite graph-based definition). It is actually a matter of taste since the three definitions result in an equivalent problem statement. We have decided for sequence-based definition because, in practice, the data is actually given as sequence.
is called the set of frequent itemsets.

The frequent itemset mining (FIM) task then is, given data \(J\) and a lower support threshold \(\minsup\), to compute the set \(F\) of all frequent itemsets.

Historically, the support threshold was defined as a relative measure to the number of transactions, i.e. \(\frac{\text{freq}_J(I)}{|J|}\) and a relative support threshold in interval \([0,1]\) was given. The data mining community tended to change the definition, and by today, the absolute support is the default. In the rest of the paper we refer to the relative support as frequency and denote \(\frac{\text{freq}_J(I)}{|I|}\) by \(\text{freq}_J(I)\) and \(\frac{\minsup}{|J|}\) by \(\text{minfreq}\).

We will often illustrate definitions and methods by examples where the items are denoted by capital letters of the English alphabet. For the sake of simplicity, we often omit braces and commas when denoting an itemset. For example, we write \(AEDG\) instead of the precise form \(\{A, E, D, G\}\).

There are some notions that are heavily used throughout the paper. Next, we give the definitions for them.

In a set of itemsets \(S\) the downward closure property holds, if \(I' \subseteq S\) for all \(I' \subseteq I\) and all \(I \in S\). A frequent itemset \(I\) is maximal if there exist no proper superset of \(I\) in \(J\) that is frequent. An itemset \(I\) is closed \([84][114]\) if there exist no proper superset of \(I\) that has the same support as \(I\).

**Corollary 1** All maximal frequent itemsets are closed.

**Definition 2.1.1** The negative border of a set of itemsets \(F\) (denoted by \(\text{NB}(F)\)) contains the itemsets that are not elements of \(F\), but all their proper maximal subsets are in \(F\). Formally

\[
\text{NB}(F) := \{I | I \notin F \text{ and } I' \in F \text{ for all } I' \subset I \text{ such that } |I'| + 1 = |I|\}.
\]

In poset theory the negative border is called the minimal, proper upper bound.

**Example 1**
Let \(J = \{A, B, C, D\}\) and \(F = \{\emptyset, A, B, C, AB, AC,\}\). Then \(\text{NB}(F) = \{BC, D\}\).

**Definition 2.1.2** Let \(\prec\) denote a total order on \(J\). The \(\ell\)-element prefix of itemset \(I\) (\(\ell \leq |I|\)), which is denoted by \(P^I_{\ell}\), is the \(\ell\)-element subset of \(I\) that contains the \(\ell\) smallest elements of \(I\) with respect to the ordering \(\prec\).

By definition \(P^I_0 = \emptyset\) for any \(I\) itemset, i.e., the empty set is the zero-size prefix of all itemsets.

**Example 2**
Let \(J = \{A, B, C, D, E\}\) and \(\prec\) denote the alphabetic order over \(J\). Here, \(P^2_{ABC} = AB\) and \(P^3_{BDE} = B\).
Definition 2.1.3 The order based negative border of a set of itemsets $F$ contains the itemsets $I$ that are not elements of $F$, but their prefix of size $|I| - 1$ and the subsequent subset of size $|I| - 1$ are elements of $F$. Here, subsequent is understood with respect to the ordering defined on the power set of $I$. Formally:

$$NB^-(F) := \{ I \mid I \notin F \text{ and } P_I^{[|I|-1]} \in F, Q \in F, \text{ where } P_I^{[|I|-1]} < Q < Q' \}
\text{ for all } Q' \subset I \text{ such that } |Q'| + 1 = |I|, Q' \neq P_I^{[|I|-1]}, Q' \neq Q \}.$$  (2.1)

By definition item $i$ is in $NB^-(F)$ if $\{i\}$ is not in $F$ and the empty set is in $F$.

Example 3
Let $I = \{ A, B, C, D, E \}$, $F = \{ \emptyset, A, B, C, AB, AC \}$ and for any itemsets of the same size $I$, $J$ let $I \prec J$ if $I$ lexicographically precedes $J$. Then $NB^-(F) = \{ ABC, BC, D \}$.

Corollary 2 For any itemset $J$, $F \subseteq 2^I$ and $<$ we have

$$NB(F) \subseteq NB^-(F).$$

In depth-first like algorithms the notion projected database plays an important role.

Definition 2.1.4 Let $\mathcal{J}$ be a transaction database over $I$. The $I$-projected database of $\mathcal{J}$ (which is denoted by $\mathcal{J}[I]$) consists of the elements of $\mathcal{J}$ that contain $I$.

The sequence of transactions that are not contained in the $I$ projected database is denoted by $\mathcal{J}[\overline{I}]$ and called the complement of the projected database. Obviously, no element of $\mathcal{J}[\overline{I}]$ contains $I$.

For example $\langle ABC, AE, BCE, BCE \rangle | \{ B \} = \langle ABC, BCE, BCE \rangle, \langle ABC, AE, BCE \rangle | \{ AE \} = \langle AE \rangle$ and $\langle ABC, BF, BCE \rangle | \{ AF \} = \langle ABC, BCE \rangle$.

2.2 The arena of FIM algorithms; a short history

The first FIM algorithm AIS (together with the FIM problem itself) was presented by Agrawal et al. [7]. A year later the same authors published Apriori, which is the widest-known algorithm even nowadays. In the next few years many Apriori modifications were proposed, DHP [82], DIC [25], Partition [95] and the sampling algorithm [105] are the most famous ones. These algorithms are regarded obsolete; there exists no public implementation of any of them that is competitive with today's algorithms. In 1996 Zaki et al. [115] published algorithm Eclat and four years later Han et al. [51] presented FP-growth. Since FP-growth was shown not to perform well on sparse datasets, the
authors improved their solution and published H-mine [87]. A very efficient Apriori
mutant DCI, which adapts hybrid support count was presented by Orlando et al. [79].

2003 was a milestone in the history of frequent itemset mining. The first open
FIM competition was organized [43]. Two FP-growth implementations (FP-growth* by
Grahne and Zhu [47] and Patricia by Pietracaprina and Zandolin [88]), a modification
of DIC [80] and a highly optimized Eclat called lcm by [106] were standing out from
the field of competitors. Lcm was further improved to the second FIM competition,
where a brand new FP-growth implementation by Rácz [90] overtook the first place
from FP-growth* and Patricia.

There have been many different algorithms proposed for frequent itemset mining.
Although most of these algorithms are variants of other algorithms, sometimes small or
obvious, sometimes larger or more intricate, for marketing purposes most of them come
with a highly individual names, making it rather hard to see the common features as
well as the specific differences.

All these algorithms can be categorized as variants of one of three different base
algorithms, Apriori, Eclat and FP-growth. Furthermore, Eclat and FP-growth are the
same algorithms except that they use different data structures.

2.3 Techniques

Most published algorithms are the modifications of the base algorithms. A typical FIM
paper presents some technique that decreases the run-time, memory need or I/O demand
of a known method. In fact, there is much more to discuss about techniques and data
structure issues than about the base algorithms.

In the next sections we describe the most important FIM algorithm, i.e., Apriori.
It is first described at semantic level, and then we check what kind of data structure
supports best the functions of the algorithm. Then we present our methods to improve
efficiency and also give a glimpse on techniques proposed by others.

We call a technique memory safe if it never increases the memory need of the
algorithm significantly (let us say more than 25%). A memory-safe technique is called
strictly memory-safe if it required the same or less amount of memory than the
algorithm without the in all test databases with every support threshold. Similarly a
technique is run-time safe if it never results in a significant run-time degradation. We
call a technique dangerous if the performance drops to its fraction at some benchmark
dataset.
2.4. ALGORITHMIC ASPECTS OF THE MODERN PROCESSORS’ FEATURES

2.4 Algorithmic aspects of the modern processors’ features

Many researchers tend to analyze their algorithms by using the external memory model. Due to the huge memory sizes, most databases fit into the main memory, which leads to the usage of the simpler random access model (RAM) (also called von Neumann model [107] named after the Hungarian born John von Neumann who proposed first this architecture). The precise model of the modern processors, however, is more sophisticated than the RAM model, which is the reason that the analysis has often nothing to do with the real run-times. For an excellent overview about the changes in classical algorithms required by the new model, the reader is referred to [72].

The most important features of modern processors, which have to be kept in mind by a data mining programmer, are the memory hierarchy and the pipeline processing.

2.4.1 Memory hierarchies, data locality

The memory is not one big block but rather a hierarchy of memories with different sizes, access latencies and access numbers. The larger the memory the longer it takes to access it. The members of the hierarchy are registers, (few kilobytes of) L1 cache, (few megabytes of) L2 cache, sometimes L3 cache, (few gigabytes of) main memory and hard disk. The data are copied from the main memory to the L2 cache and from L2 to L1 cache in blocks. The size of block (also called cache line size) for copying from L2 to L1 cache is 128 bytes in the case of Pentium 4 processors.

The block processing brings in some important algorithmic aspects. Reaching a single bit from a slower memory takes the same time as reaching a whole block. Processing the data that is in the same block does not require an other slow memory access operation. Therefore data locality, the requirement that data items which are processed close to each other in time, should be located close to each other in memory, is a immensely important issue, which affect significantly the running time. Data near the currently processed data should contain many items, which will be processed in the near future.

When a data has to be processed, it has to be moved into the registers. Sometimes it is already there, because it was used in the previous instructions. Due to the limited number of registers, it is more probably that the data is located in L1, L2 cache or in main memory. It may even be located on the hard disk, if the memory usage of the algorithm is so large, that the operating system has to swap. We say a data access causes cache miss if it is located in L2 cache or main memory. Although the processor may perform another operations while the data is fetched, the performance of the processor get far from its maximum. The processor is capable to do 1000 basic operations (like addition) during the time the data is fetched from the main memory. In summary, when designing the data structure – algorithm pair, we have to endeavor to reach high data
locality so that cache misses are avoided.

2.4.2 Pipeline processing, branch prediction

The instructions a programmer works with are executed as a sequence of many microoperations (u-ops). The operations are not processed individually, one-by-one after each other. Instead, a parallel processing is done by using a pipeline. Unfortunately, the data dependency and the conditions ruin the efficiency of parallelism. Data dependency occurs when an instruction depends on the results of a previous instruction. Branch prediction means predicting the output of a condition and loading the predicted operations into the pipeline. If the prediction turns out to be false, then the pipeline has to be flushed and the correct values have to be reloaded to the registers. These problems can be often overcome by different techniques (like code reordering), which are done automatically by the compiler. We still have to take data independence and branch prediction into consideration when designing a computation intensive algorithm.

The pipeline processing makes it possible to execute more than one instruction during a clocktick. The problems mentioned above are the reasons for being the average performance of the processor much less than the optimal. We say the processor stalls, if it can not execute an operation in the actual clocktick.

Unnecessary conditions may ruin efficiency, but this is not always the case. The branch prediction is "intelligent" in the sense that it learns if the outcome of the condition never changes, and sets the prediction accordingly. Therefore, a 100% true (or) false condition never ruins efficiency at all.

2.5 Graphical presentation of the experiments

This work is based on thorough theoretical analysis and on a very comprehensive set of experiments. To increase readability we avoid using tables of numbers but rather trying to visualize the experiments. In the literature the authors present their experiments by run-time and memory plots. Displaying the plots for all databases takes too much space, therefore only a few (unfortunately the ones that give a favorable view of the proposed technique) are selected. The FIMI contests showed that the published algorithms do not perform so well in general as they do in certain, carefully chosen databases. For fairness, we test each technique on 16 well-known test databases, most of them can be downloaded from http://fimi.cs.helsinki.fi. To avoid space problems, we restrict our attention to test results at low support thresholds.

In many experiments we compare two solutions (s and s_new), one (s_new) is expected to be faster. The advantage of the faster solution is presented on 16 databases mainly at very low support thresholds. We use bar-charts, where the height of a bar is \( \frac{m(s)}{m(s_{new})} \), where \( m \) denotes the measurement (in most of the cases it is run-time and memory-
need). Sometimes the new technique results in an improvement of a several orders of magnitude. To present such cases we use the logarithm of the measurements.

In many cases we are not only interested in the run-times but we would like to visualize the way the technique suits to the features of the modern processor. For this we use a diagram like the following.

![Diagram showing uops over time](image)

The height of the wide bars centered around the ticks show the actual run-time (the total clockticks used by the program). The colors/patterns of these bars show how well the program utilized these clockticks: the top-most part shows the amount of clockticks during which three u-ops were executed, while the bottom-most part shows the time during which the program execution was stalled for some reason (i.e., no operations were executed during that clocktick).

The narrow bars centered around the ticks show the total number of u-ops that were executed. The bar is divided into two, the upper part show the bogus u-ops, those u-ops that were speculatively executed on a mispredicted branch, and thus were rolled back. The ratio of the lower-to-upper part of this bar shows the branch prediction inefficiency.

The narrow bars beside the wide ones show the front-side bus activity, the total number of clockticks during whose at least one read/write operation was pending (i.e., data transfer time including memory latency). The upper part of these bars show the time consumed by prefetch reads (when the processor speculatively transfers data from the memory into the cache for further availability), while the lower part shows actual reads or writes. The main difference is that the delivery of data during actual reads and writes presumably stalls the execution pipeline (these are the cache misses). If the ratio of prefetch (top part) to actual wait (bottom part) is high, then a huge amount of cache misses are avoided by the prefetch mechanism, thus achieving a considerable performance gain.

### 2.6 Bottom-up FIM algorithms

The initial step is common in all algorithms. We scan the database once to determine the support of every item, and then select the frequent ones. Without loss of generality,
we assume that the frequent items are denoted by consecutive integers starting from 0.

In the latter phases of the algorithms each transaction is filtered before being processed, i.e., infrequent items are removed. Most of the techniques make the assumption that the (frequent) items are coded with nonnegative integers. Therefore each transaction is filtered, and recoded. Obviously, before writing out the results the items have to be coded back.

Apriori. Eclat and FP-growth perform a bottom-up traversal of the search space, i.e., starting from the empty set they determine the frequent itemsets in a growing manner. To avoid duplicate checking of the same itemset all FIM algorithm are based on an ordering of the items. The lexicographic extension of this ordering makes it possible to order the itemsets. It would be impossible to determine the support of every possible itemset (their number is exponential in $|I|$) therefore the algorithms restrict their attention to the so called candidates. In general a candidate is an itemset whose support is determined.

Bottom-up search algorithms turned out to be more efficient algorithms than those that perform top-down or a middle-way top-down bottom-up search (such as algorithms Pincer [65] and CBW [103]). This is attributed to the fact that the maximal frequent itemset border is closer to the empty set than to $I$, i.e., in general the size of the largest frequent set is much less than $|I|$.

### 2.7 Breadth-first, iterative vs. depth-first, recursive algorithms

Apriori is an iterative, breadth-first algorithm. In the iteration step $\ell$ it determines the frequent itemsets of size $\ell$. Eclat and FP-growth, on the contrary, are recursive, depth-first-like algorithms. Given a set of frequent itemsets (denoted by $F^+\ell$) with a common maximal proper prefix $P$ and of size $|P| + 1$, it takes the itemsets $I \in F^+\ell$ one-by-one and determines the frequent itemsets whose prefix is $I$. The search is done recursively; initially the empty set is considered as a prefix and the set of frequent 1-itemsets is the given set.

The definition of a candidate in Apriori differs from the definition in Eclat and FP-growth. In Apriori the set of candidates at iteration $\ell$ is equal to the negative border of frequent itemsets found till the iteration step $\ell$. In Eclat and FP-growth the set of candidates in the next recursive step belonging to itemset $I \in F^+\ell$ is the subset of the order-based negative border of $F^+\ell$ whose element's prefix is $I$ (formally \{$I'|I' \in NB^-(F^+\ell) \text{ such that } F^{|I'|-1} = I$\}). The recursive step is terminated if no candidate is generated.

It would be inefficient to check all itemsets of a given size if they meet the definition for candidates. Instead, we generate the candidates. Here we make use of the fact that
in all three algorithms the smallest and the subsequent subset of the candidate must be frequent. The itemsets form a lattice, therefore each candidate is a union of two frequent itemsets, that have same prefix of size $\ell - 1$. This is the reason the maximal proper prefix and the subsequent itemset are called the *generators* of the candidate. The item that is added to get the candidate (i.e., the largest item of the second generator) is called the *extender*.

The set of infrequent candidates is the the negative border of the frequent itemsets in Apriori and is the order-based negative border of the frequent itemsets in the case of Eclat and FP-growth. It follows from Corollary 2 that the number of candidates is never less in Eclat and FP-growth than in Apriori.

### 2.8 Apriori algorithm

Apriori is regarded to be the first FIM algorithm that can cope with large datasets and large search space. It was proposed by Agrawal and Srikant [5] and Mannila et al. [70] independently at the same time. Their cooperative work was presented in [8].

The algorithm scans the transaction datasets several times. After the first scan the frequent 1-itemsets are found, and in general after the $\ell^{th}$ scan the frequent $\ell$-itemsets are extracted. The method does not determine the support of every possible itemset. In an attempt to narrow the domain to be searched, before every pass it generates *candidate* itemsets and only the support of the candidates are determined. An itemset becomes a candidate if all its proper subsets of are frequent. Due to the bottom-up search, all frequent itemsets of size smaller than the candidate are already determined, therefore it is possible to do the subset validations.

After all the candidate $(\ell + 1)$-itemsets have been generated, a new scan of the transactions is effected and the precise support of the candidates are determined. The candidates with low support are discarded. The algorithm ends when no candidates are generated. The pseudo code of Apriori is given below.

The intuition behind candidate generation is based on the following simple fact:

**Property 1** *Every subset of a frequent itemset is frequent.*

This is immediate, because if a transaction $t$ contains an itemset $X$, then $t$ contains every subset $Y \subseteq X$.

Using the fact indirectly, we infer that, if itemset $I$ has a subset that is infrequent, then $I$ cannot be frequent. In the algorithm Apriori only those itemsets are candidates whose all subsets are frequent. It is not necessary to check all subsets; if all maximal proper subsets are frequent, then the anti-monotone property of the support function guarantees that all subsets are frequent as well.

It would be inefficient to go through on all itemsets of size $(\ell + 1)$ and do the subset check, instead, we generate the candidates. All itemsets that meet the subset check
Algorithm 2.8.1: Apriori

Require: \( D \): database over the set of items \( \mathcal{I} \),
          \text{minsup} support threshold

Ensure: \( F \): the set of frequent itemsets

\[
\ell \leftarrow 1 \\
C_\ell \leftarrow \emptyset \\
\textbf{while } |C_\ell| \neq 0 \textbf{ do} \\
\quad \text{support.count}( D, C_\ell ) \\
\quad \textbf{for all } j \in C_\ell \textbf{ do} \\
\quad \quad \text{if } j.\text{support} \geq \text{minsup} \textbf{ then} \\
\quad \quad \quad F_l \leftarrow j \\
\quad \quad \quad C_{\ell+1} \leftarrow \text{candidate.generation}(F_l) \\
\quad \ell \leftarrow \ell + 1; \\
\quad F = \bigcup_{j=1}^{\ell} F_j \\
\textbf{end while}\]

The requirement must be the union of two different \( \ell \)-itemset that are frequent and have \( \ell - 1 \) common items. Different pairs can have the same union (for example the pairs \( (AB, AC) \) and \( (AB, BC) \)). In order the candidate generation to be non-redundant we take the union of those \( \ell \)-itemsets whose intersection is the \((\ell - 1)\)-element prefix. Pairs \( (I_1, I_2) \) and \( (I_2, I_1) \) generate the same candidate therefore we assume \( I_1 \prec I_2 \). The pseudo code of the candidate generation is found in Algorithm 2.8.2.

Algorithm 2.8.2: candidate.generation

Require: \( F_l \): frequent itemsets of size \( \ell \)
Ensure: \( C_{\ell+1} \): the set of candidates of size \( \ell \)

\[
\textbf{for all } \{i_1, \ldots, i_{\ell-1}, i_\ell\}, \{i_1, \ldots, i_{\ell-1}, i_\ell'\} \in F_l \textbf{ such that } i_\ell \prec i_\ell' \textbf{ do} \\
\quad c \leftarrow \{i_1, \ldots, i_{\ell-1}, i_\ell, i_\ell'\} \\
\quad \textbf{if all } \ell \text{-subsets are frequent}(c, F_l) \textbf{ then} \\
\quad \quad C_{\ell+1} \leftarrow c
\]

After the candidate generation the supports of the candidates are calculated. This is done by reading transactions one by one. A counter with 0 initial value is associated with each candidate. For each transaction \( t \) the algorithm decides which candidates are contained in \( t \). The counter of these candidates are incremented.
Chapter 3

Methods to Improve Support Count

In this chapter we show how to carry out efficiently the support counting of Apriori. We start by describing our special trie-based solution, then we introduce our three methods to reduce run-time and memory need. These are (1.) dead-end pruning, (2.) transaction caching, (3.) equisupport pruning.

Let us recall the support counting procedure. For each transaction \( t \) the algorithm decides which candidates are contained in \( t \). A simple solution of this is to check each candidate if it is contained in the transaction. This is an elementary operation (determining if an ordered sequence contains an other ordered sequence) if the transaction and the candidates are stored ordered. The drawback of this solution is that the transaction is checked and partially traversed as many times as the number of candidates, which is quite slow at low support thresholds, where there are many candidates.

To save numerous transaction traversals it is useful to store the candidates in a special data structure. In the original paper [5] a hash-tree was proposed for this purpose. The first trie-based Apriori implementation is reported in [84]. Independent from each other Borgelt, Goethals and the author (and maybe several others) published the first open-source Apriori implementations. In [20] trie and hash-tree were compared, and suggested that the trie is a better data structure in Apriori w.r.t run-time, memory need but most importantly the flexibility. The main disadvantage of hash-tree is that it is non-parametric, i.e., it requires a hash function. The efficiency of the hash-tree is greatly influenced by the hash-function. Different hash-functions are suitable for different databases and even different hash-functions are suitable for the same database with different support threshold. There exists no available and efficient Apriori implementation that uses a hash-tree.

A vector-trie middle-way solution was proposed in [78]. Candidates with the same 2-element prefix are stored in a vector. The addresses of the vectors are directly accessible by a triangular array. Vector of prefix \( i,j \) belongs to the element at index \( i,j - i - 1 \) of the array. To save memory, the common 2-element prefixes are not stored in the
elements of the vectors. The authors declared that this solution is more efficient than trie-based solution, because of the "pointerless" approach, the high data locality and the predictable code branches. Our experiments do no support this claim.

The following plots show that although this is a much better solution than simply storing the candidates in a list, it is still not competitive with trie-based solution at medium or low support thresholds. This observation holds in all databases.

![Comparison of simple vector, prefix-array and trie-based solution for storing the candidates in Apriori](image)

Figure 3.1: Comparison of simple vector, prefix-array and trie-based solution for storing the candidates in Apriori

Due to the outstanding efficiency of the trie-based solution, we restrict our attention to this data structure.

### 3.1 The trie and its variants

Since the trie (prefix-tree) data structure comes into play in Apriori, FP-growth and many other FIM algorithms (like MaxMiner [93] and TreeProjection [3]), we begin with the description this central data structure.

The data structure trie was originally introduced by de la Briandais [35] and Fredkin [38] to store and efficiently retrieve words of a dictionary. Mueller [74] was the first to use trie in a FIM algorithm.

A trie is a rooted, labeled tree. Each label is a character and each node represents a word (sequence of characters) which is the concatenation of the characters that are on the path from the root to the node. The root is defined to be at depth 0, and a node at depth d can point to nodes at depth d + 1. A pointer is also referred to as edge or link. We will use the notations parent, child, sibling, ancestor and descendant as they are defined in the classical oriented tree data structures.

Tries are suitable for storing and retrieving not only words, but any finite sequences over arbitrary alphabet as well. In the FIM setting a link is labeled by a frequent item, and a node represents a sequence of items. To obtain a sequence from a set, we have to
3.1. THE TRIE AND ITS VARIANTS

define a total order on the items. For this, we always use the same order that is used to order the edges. In this case the preorder depth-first search traversal corresponds to the ascending lexicographical ordering of the itemsets.

If the trie stores sequences of different lengths, then a boolean value is also associated to each inner node. A true value denotes that the sequence that is represented by the inner node is also contained in the dictionary not just the sequences represented by the leaves. Figure 3.2 presents a trie that stores the itemsets A, C, F, AC, AF, EF, AEF. The order used to convert sets to sequences corresponds to the alphabetic order. Inner nodes with false and true boolean values are denoted by squares and circles, respectively.

\( \mathcal{T}(A, C, F, AC, AF, EF, AEF) \)

![Figure 3.2: Example: a trie that stores sets \{A\}, \{C\}, \{F\}, \{AC\}, \{AF\}, \{EF\}, \{AEF\}](image)

A trie that stores all subsets of a given set is quite unbalanced. Figure 3.3 shows the trie that stores all subsets of itemset \{ABCDE\}.

Originally the tries are child-linked, i.e., from each node only its children can be reached with one step. In case of a parent-linked trie we can only reach the parents directly. Obviously, the two approaches can be combined.

### 3.1.1 The representation of the list of edges

The list of edges belonging to an inner node can be represented in many ways. The representation used in the algorithms greatly affects run-time and memory-need. Let us assume that we have a node \( u \) with \( n \) children. This means that \( n \) edges start out from \( u \). Denote the smallest and largest label of these edges by \( l_{\text{min}} \) and \( l_{\text{max}} \) respectively. The most frequently used representations are:

**ordered list:** Each edge is represented by a pair, whose first element is the label, and the second is a pointer to the child. The edges are stored in a vector, which is
ordered according to the labels. The memory need of this solution (ignoring the overhead of a vector) is $2n$ cells.

**indexvector:** The child pointers are stored in a vector whose length equals to the number of frequent items. A node at index $i$ is the endpoint of the edge whose label is item $i$. If there is no edge with such label, then the element is NIL.

Obviously the elements at index less than the smallest label and greater than the largest label are NIL. We save memory if these elements are not stored. In **offset-indexvector** representation the smallest element (the offset) and a pointer vector of size $l_{max} - l_{min} + 1$ is stored. The child pointer of label $i$ is given by the element at index $i - l_{min}$.

**hybrid solution:** Notice, that neither of the above representations needs always less memory than the other. If $2n < l_{max} - l_{min} + 1$, then the ordered list needs less memory, otherwise the offset-indexvector. In the hybrid edge representation we dynamically choose the edge representation based on the memory requirements.

### 3.1.2 Index vs. pointer-based trie

The nodes of the trie (together with the lists of edges) can be stored consecutively or scattered in the memory. We distinguish two types of Trie according to the memory layout (such tries are depicted in Figure 3.4).
3.1. THE TRIE AND ITS VARIANTS

contiguous-memory based:

\[ 2, 167, B, 6, D, 8, 0, 123, 0, 102 \]

\[ 123 \]

\[ 102 \]

pointer-based:

\[ 167, \text{[B, D]} \]

\[ 123, \text{[]} \]

\[ 102, \text{[]} \]

Figure 3.4: different representations of the same trie

**pointer-based trie**: The nodes are scattered in the memory. The counter and the list of edges are associated with the node. The nodes are identified by their address in the memory, and a link is represented by a pointer. When adding a new leaf into the tree we search for a free space in the memory and reserve it to the new leaf. Deleting a leaf means simply freeing the memory occupied by the leaf and removing the pointer (together with the label) from the edgelist of its parent.

If we store the edges in an ordered vector, then the memory need of a node is the memory need of a counter and a list. The total memory need of a trie is \( n s_i + n s_{\text{eo}} + (n - 1)s_i + (n - 1)s_p \), where \( n \) is the number of nodes in the trie, \( s_{\text{eo}} \) is the memory need of the overhead of the vector, \( s_i, s_p \) is the size of an integer and a pointer, respectively. If the vector of C++ STL is used then the overhead of a vector equals three times the size of the pointer, therefore the total memory need is approximately \( 2n(s_i + s_p) \) which is 26\( n \) bytes in a Pentium 4 and 40\( n \) bytes in an Opteron.

**contiguous-block trie**: The trie is represented by one big vector. The counter, the number of edges and the list of edges are associated with the node. Each node is identified by the position in the vector. Adding (and erasing) a leaf is quite a laborious work. We expand the vector, then insert a new edge into the edgelist of the parent. This results in an increase of the positions of the nodes coming after the parent, therefore the indices have to be updated. This requires a total scan of the vector.

It may be difficult to find a free big block in the memory, hence a list of medium-size blocks are used in practice. The blocks are of the same size, therefore we can quickly determine the block (and the offset) of a node it has been placed into.

If the edgelists are stored in an ordered vectors, then the memory need of a node equals to the memory need of the counter the memory need of the variable that
stores the number of children, and the edges (without overhead). The total memory need is \( s_m + n(s_i + s_i + 2s_i) \approx 4ns_i \) which is 16\( n \) in a Pentium and Opteron as well. Note that we assume that the size of the vector that stores the trie is not greater than \( 2^{2n} \), otherwise we cannot address an element by an integer value.

In our implementation leaves are added and deleted from the trie, therefore we use the pointer-based approach.

### 3.1.3 Patricia trie

A directed path is called chain if all inner nodes on the path have only one child. A tree that is obtained from a trie by collapsing maximal chains to a single edge is called patricia tree. The new edge points to the last node of the chain and its label is the sequence of the labels on the chain. If chain collapse is restricted to chains that end in leaves then we talk about leaf-patricia tree.

Patricia trees consume less memory if the trie contains many chains. Otherwise, it need more memory, because the labels are represented by vectors, which is an inefficient solution when it contains just one element.

### 3.2 The trie of Apriori

Throughout the algorithm one child-linked trie is maintained. In this trie a counter is associated with each node. This counter stores the support of the itemset the node represents. In candidate generation phases new leaves are added with zero counters, in support count phases the counters are updated, and when we eliminate infrequent subsets (infrequent removal phase), leaves with counter value less than minsup are pruned.

Next, we examine Apriori’s main procedures from the perspective of the trie.

#### 3.2.1 Support Counting

In the support counting phase, we take the transactions one-by-one. With a recursive traversal we traverse some part of the trie. If a node is reached, then the itemset represented by the leaf is contained in the transaction. The counters of such leaves are increased. The traversal of the trie is driven by the elements of transaction \( t \) and starts in the root. No step is performed on edges that have labels which are not contained in \( t \). More precisely, if we are at a node at depth \( d \) by following a link labeled with the \( j \)th (let \( j \) be 0 in the root) item in \( t \), then we move forward on those links that have the labels \( i \in t \) with index greater than \( j \), but less than \(|t| - \ell + d\), if we denote the size of the candidates by \( \ell + 1 \). The upper bound is obtained by the fact that \( \ell - d \) another steps are required to reach a leaf from a child.
3.2.2 Removing Infrequent Candidates

After support counting, the leaves that represent infrequent itemsets have to be deleted from the trie. Leaves are reached in a depth-first traversal.

3.2.3 Candidate Generation

Here we make use of another nice feature of tries; \( \ell \)-itemsets, that share the same \(( \ell - 1 )\)-prefix, are represented by sibling leaves [14]. Consequently, the extender of a node must be in the label set of edges pointing to a sibling. This is just a necessary requirement. For an \(( \ell + 1 )\)-itemset \( I \) to become a final new leaf, it has to meet Apriori’s pruning condition: the \( \ell \)-subsets of \( I \) have to be frequent.

To obtain the itemsets represented by the nodes, we have to maintain a stack and perform a depth first traversal. Whenever we step down along an edge we push its label to the stack, and pop it when a backward step is performed.

3.3 Compactness of the trie and the run-time of Apriori

The growth of available memory sizes follows Moore’s law. Today memory sizes are so large that most of the databases fit in the main memory if the proper filtering and compression is applied (in FIM setting this means removing infrequent items from the transactions and recoding items to integers). The cheap and huge memorie devices encourages the implementors of data mining algorithms to handle memory issues generously.

The reader will, however, observe the opposite in our case; we try to keep memory consumption as small as we can, and we spend serious efforts on keeping the trie as compact as possible. This has two main reasons. First, memory allocations and deallocations require processor operations. Second, by increasing compactness, we increase data locality, which improves the efficiency of the prefetching the caching features of modern processors.

To illustrate this we have done the following experiment. We measured the run-time and memory need of our Apriori. However, we manipulated the candidate trie a little bit; a vector of uninitialized integers was inserted into each node. The size of the vector was a parameter. The larger this parameter is, the more the nodes are scattered from each other, and hence the worse the data locality is. The following plots show the run-time and memory need.

The reason of the run-time increase is prompted by Fig. 3.6, which shows more information about the utilization of the clockticks, the number of u-ops that were executed on properly and improperly predicted branches, the total number of clockticks during
Figure 3.5: The influence of node’s size of the trie on run-time and memory need
whose at least one read/write operation was pending on database BMS-WebView-2 with minsup = 6. The left bar chart belongs to vector size 0 the right one belongs to the vector size 50.

Figure 3.6: Complex hardware-friendliness diagram of two implementations

We see, that the two implementations perform approximately the same number of instruction, and there is no significant difference in branch prediction efficiency. However, in the second implementation the processor stalls much more than in the first case, which results the slowing down of the program. The processor stalls are caused by bad data locality resulting in cache misses.

3.4 Improvements used in Apriori

Before we turn to our methods that speed up algorithm Apriori, we have to find what is worth improving, i.e., what takes significant time of the running. We have already mentioned that in the beginning of the FIM research the efforts were focused on reducing I/O costs and later reducing the number of candidates. Now, we know that these two
3.4. IMPROVEMENTS USED IN APRIORI

Factors are not so important, but rather the data structure and its usage, the memory management, and the level the implementation suits the architecture of the modern processors are the issues that really matter.

The following table shows the distribution of processor time usage between the main functions of Apriori. We measured the three main functions of Apriori (generating candidates, determining the supports and deleting infrequent candidates), the time required for reading in, sorting and recoding (removing infrequent items and assign 0,1,... values to the frequent items) the transactions and determining the support of the two element candidates. Methods that required less than half percent of the run-time are indicated by blank entries. For the sake of readability numbers above 25 are rounded. To see the correlation between the ratio of the methods and the characteristics of the database and search space, we also provide some statistics about the data sets and the frequent itemsets (see Tables 3.2 and 3.3). In these tests we have used a highly optimized Apriori implementation, which is based on an inhomogeneous trie using our special block allocator, dead-end branch removal, a triangular array-based solution to find efficiently frequent pairs, and a sophisticated depth-first, buffered input/output manager performing the input/output routines.

The data show that Apriori is so fast at high support thresholds, that its operation require less time than processing the input. Thus we concentrate on low support thresholds.

The tables support the widely-known observation, that determining the support of the candidates takes most of the time of Apriori. This is, however, not always true. In mining tasks where the number of frequent itemsets is high (databases BMS-WebView-1, BMS-WebView-2, retail) but the size of the dataset is medium with modest average transaction sizes (T1C1S1K5P5K0, T1C1I4D100K) the candidate generation contributes significantly to the run-time. Consequently, we first focus on the support count procedure and then turn to speed up the candidate generation method.

The distribution changes by employing certain heuristics, and then other parts may become the bottleneck of the algorithm. For example if equisupport pruning is applied (see section 3.8) then it becomes possible to process dense databases at much lower support threshold, and subset enumeration and output writing dominates the run-time. Nevertheless, we regard these issues of more advanced nature. We believe that our data gives good indicators about the bottleneck of Apriori and possible targets for improvement.

We see three principal ways to reduce the run-time of support counting.

1. We fine-tune and optimize the elementary operation of support counting, i.e., finding the candidates that are contained in a given transaction.

2. We reduce the number of support count method calls.

3. We make use of the fact that some operations are done repeatedly (for example
## Table 3.1: The distribution of run-time of Apriori’s methods in %

<table>
<thead>
<tr>
<th>Database</th>
<th>minsup</th>
<th>counting support</th>
<th>generating candidate</th>
<th>input sort recode</th>
<th>infrequent removal</th>
<th>frequent pair mining</th>
</tr>
</thead>
<tbody>
<tr>
<td>T4CH0D10CK</td>
<td>3,000</td>
<td>14</td>
<td>53</td>
<td></td>
<td></td>
<td>31.6</td>
</tr>
<tr>
<td>kowarak</td>
<td>7,000</td>
<td>21</td>
<td>65</td>
<td></td>
<td></td>
<td>1.5</td>
</tr>
<tr>
<td>T1CH4D10CK</td>
<td>150</td>
<td>68</td>
<td>24</td>
<td></td>
<td></td>
<td>4.1</td>
</tr>
<tr>
<td>connect</td>
<td>55,000</td>
<td>73</td>
<td>25</td>
<td></td>
<td></td>
<td>1.4</td>
</tr>
<tr>
<td>accidents</td>
<td>210,000</td>
<td>77</td>
<td>21</td>
<td></td>
<td></td>
<td>1.4</td>
</tr>
<tr>
<td>pumsb</td>
<td>41,000</td>
<td>57</td>
<td>2.5</td>
<td></td>
<td></td>
<td>10.6</td>
</tr>
<tr>
<td>retail</td>
<td>65</td>
<td>64</td>
<td>22</td>
<td></td>
<td></td>
<td>3.8</td>
</tr>
<tr>
<td>BMS-POS</td>
<td>5,000</td>
<td>38</td>
<td>56</td>
<td></td>
<td></td>
<td>0.7</td>
</tr>
<tr>
<td>BMS-WebView-1</td>
<td>39</td>
<td>67</td>
<td>0.1</td>
<td></td>
<td></td>
<td>2.7</td>
</tr>
<tr>
<td>BMS-WebView-2</td>
<td>30</td>
<td>55</td>
<td>23.3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>webdocs</td>
<td>700,000</td>
<td>1</td>
<td>53</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mushroom</td>
<td>1,000</td>
<td>55</td>
<td>1.3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1CH5N1KF5KC0</td>
<td>500</td>
<td>8</td>
<td>67</td>
<td></td>
<td></td>
<td>22.6</td>
</tr>
<tr>
<td>T2CH10N1KF5KC0</td>
<td>2,000</td>
<td>76</td>
<td>17.8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T3CH15N1KF5KC0</td>
<td>1,300</td>
<td>25</td>
<td>73.6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pumsb*</td>
<td>23,000</td>
<td>55</td>
<td>41</td>
<td></td>
<td></td>
<td>2.5</td>
</tr>
</tbody>
</table>

### High Support Threshold

<table>
<thead>
<tr>
<th>Database</th>
<th>minsup</th>
<th>counting support</th>
<th>generating candidate</th>
<th>input sort recode</th>
<th>infrequent removal</th>
<th>frequent pair mining</th>
</tr>
</thead>
<tbody>
<tr>
<td>T4CH0D10CK</td>
<td>220</td>
<td>53</td>
<td>6.5</td>
<td>0.5</td>
<td>0.5</td>
<td></td>
</tr>
<tr>
<td>kowarak</td>
<td>860</td>
<td>54</td>
<td>2.6</td>
<td>2.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1CH4D10CK</td>
<td>3</td>
<td>33</td>
<td>63</td>
<td>0.7</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>connect</td>
<td>43,000</td>
<td>55</td>
<td>3.1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>accidents</td>
<td>130,000</td>
<td>58</td>
<td>1.4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pumsb</td>
<td>32,000</td>
<td>65</td>
<td>1.6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>retail</td>
<td>3</td>
<td>20</td>
<td>63</td>
<td>1.3</td>
<td>1.8</td>
<td>0.7</td>
</tr>
<tr>
<td>BMS-POS</td>
<td>67</td>
<td>84</td>
<td>13</td>
<td>0.8</td>
<td>0.5</td>
<td></td>
</tr>
<tr>
<td>BMS-WebView-1</td>
<td>33</td>
<td>44</td>
<td>54</td>
<td>0.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BMS-WebView-2</td>
<td>4</td>
<td>12</td>
<td>83</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>webdocs</td>
<td>200,000</td>
<td>77</td>
<td>21.3</td>
<td></td>
<td></td>
<td>1.3</td>
</tr>
<tr>
<td>mushroom</td>
<td>250</td>
<td>85</td>
<td>12.5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1CH5N1KF5KC0</td>
<td>4</td>
<td>53</td>
<td>39</td>
<td>1.8</td>
<td>0.7</td>
<td>0.8</td>
</tr>
<tr>
<td>T2CH10N1KF5KC0</td>
<td>90</td>
<td>84</td>
<td>13.0</td>
<td>1.7</td>
<td>0.6</td>
<td></td>
</tr>
<tr>
<td>T3CH15N1KF5KC0</td>
<td>300</td>
<td>84</td>
<td>12.2</td>
<td></td>
<td></td>
<td>0.8</td>
</tr>
<tr>
<td>pumsb*</td>
<td>13,000</td>
<td>59</td>
<td>0.5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Low Support Threshold
Table 3.2: Some statistics about the databases

<table>
<thead>
<tr>
<th>database</th>
<th>number of transactions</th>
<th>number of items</th>
<th>average size of the transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>mushroom</td>
<td>8 124</td>
<td>119</td>
<td>23.3</td>
</tr>
<tr>
<td>pumsb</td>
<td>49 046</td>
<td>2 088</td>
<td>70.4</td>
</tr>
<tr>
<td>pumsb</td>
<td>49 046</td>
<td>2 113</td>
<td>74.0</td>
</tr>
<tr>
<td>EMS-WebView-1</td>
<td>59 662</td>
<td>497</td>
<td>2.5</td>
</tr>
<tr>
<td>connect</td>
<td>67 557</td>
<td>120</td>
<td>42.0</td>
</tr>
<tr>
<td>EMS-WebView-2</td>
<td>77 512</td>
<td>3 340</td>
<td>4.5</td>
</tr>
<tr>
<td>retail</td>
<td>88 163</td>
<td>16 470</td>
<td>10.3</td>
</tr>
<tr>
<td>T4CH10109K</td>
<td>100 000</td>
<td>870</td>
<td>10.1</td>
</tr>
<tr>
<td>T4CH10109K</td>
<td>100 000</td>
<td>942</td>
<td>36.5</td>
</tr>
<tr>
<td>T4CH10109K</td>
<td>193 379</td>
<td>3 950</td>
<td>10.3</td>
</tr>
<tr>
<td>T4CH10109K</td>
<td>197 446</td>
<td>4 408</td>
<td>26.2</td>
</tr>
<tr>
<td>T4CH10109K</td>
<td>190 055</td>
<td>4 550</td>
<td>30.9</td>
</tr>
<tr>
<td>accidents</td>
<td>340 183</td>
<td>468</td>
<td>33.8</td>
</tr>
<tr>
<td>EMS-POS</td>
<td>515 567</td>
<td>1 657</td>
<td>6.5</td>
</tr>
<tr>
<td>kosarak</td>
<td>990 002</td>
<td>41 270</td>
<td>8.1</td>
</tr>
<tr>
<td>webdoc</td>
<td>1 592 052</td>
<td>5267 656</td>
<td>177.2</td>
</tr>
</tbody>
</table>

traversing the same part of the tree several times) at different steps of the support count phase, and by merging these support counts we may spare some redundant work.

First we investigate fine-tuning of the support count procedure by introducing a special data structure, optimizing the routing strategies and applying dead-end pruning. Then we turn to a technique that significantly reduces the number of support count calls at many databases. Finally, we consider databases with many closed itemsets and present equisupport pruning.

### 3.5 Inhomogeneous trie and a special block allocator

From programming point of view a trie can be declared in many ways. The simplest one is the following: “Trie is a recursive structure; it has a counter and a list of edges. An edge is pair of a label and a trie pointer”. A trie is called leaf if its list is empty. Another definition is, that “A leaf is a counter. The trie is a leaf (a counter) or a counter and a list of edges.”. The first type of trie is called homogeneous trie, because it is declared by a singe data structure (not taking into consideration the data structure list). The second is inhomogeneous trie because in the definition we use two data structures (leaf and trie). Distinguishing the above definitions seems to have no meaning.

To understand the contrary, we have to dig down to implementation level. The main point of the difference comes from the facts that:

1. the compactness of the trie is crucial, and greatly affects both run-time and memory need.
<table>
<thead>
<tr>
<th>database</th>
<th>minsup</th>
<th>number of frequent items</th>
<th>number of frequent item pairs</th>
<th>size of the maximal frequent itemset</th>
<th>average size of the frequent itemsets</th>
<th>average size of the filtered transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>webdocs</td>
<td>700 000</td>
<td>8</td>
<td>14</td>
<td>34</td>
<td>4</td>
<td>2.0</td>
</tr>
<tr>
<td>kosarak</td>
<td>2 000</td>
<td>472</td>
<td>0</td>
<td>473</td>
<td>1</td>
<td>6.99</td>
</tr>
<tr>
<td>connect</td>
<td>65 000</td>
<td>12</td>
<td>72</td>
<td>916</td>
<td>5</td>
<td>2.6</td>
</tr>
<tr>
<td>pubmed</td>
<td>23 000</td>
<td>34</td>
<td>126</td>
<td>1165</td>
<td>8</td>
<td>4.0</td>
</tr>
<tr>
<td>BMS-POS</td>
<td>5 000</td>
<td>145</td>
<td>408</td>
<td>1171</td>
<td>5</td>
<td>2.5</td>
</tr>
<tr>
<td>T1011N5K5P</td>
<td>500</td>
<td>1450</td>
<td>90</td>
<td>1655</td>
<td>5</td>
<td>1.1</td>
</tr>
<tr>
<td>accidents</td>
<td>210 000</td>
<td>21</td>
<td>125</td>
<td>1685</td>
<td>8</td>
<td>4.17</td>
</tr>
<tr>
<td>T3011N5K5P</td>
<td>1 300</td>
<td>1067</td>
<td>3</td>
<td>1671</td>
<td>2</td>
<td>1.0</td>
</tr>
<tr>
<td>retail</td>
<td>45</td>
<td>2665</td>
<td>4958</td>
<td>11684</td>
<td>5</td>
<td>2.1</td>
</tr>
<tr>
<td>T1011D10K</td>
<td>150</td>
<td>767</td>
<td>5549</td>
<td>19127</td>
<td>10</td>
<td>3.39</td>
</tr>
<tr>
<td>pubmed</td>
<td>41 000</td>
<td>25</td>
<td>249</td>
<td>36811</td>
<td>11</td>
<td>5.8</td>
</tr>
<tr>
<td>mushroom</td>
<td>1 600</td>
<td>43</td>
<td>380</td>
<td>53652</td>
<td>15</td>
<td>7.1</td>
</tr>
<tr>
<td>BMS-WebView-1</td>
<td>39</td>
<td>262</td>
<td>3802</td>
<td>60376</td>
<td>12</td>
<td>4.8</td>
</tr>
<tr>
<td>BMS-WebView-2</td>
<td>39</td>
<td>2122</td>
<td>6052</td>
<td>194262</td>
<td>15</td>
<td>6.5</td>
</tr>
</tbody>
</table>

**high support threshold**

| webdocs     | 200 000 | 155                       | 1 596                          | 58 297                              | 10                                   | 5.0                                      |
| accidents   | 100500  | 32                        | 408                           | 160 874                             | 12                                   | 6.7                                      |
| pubmed      | 13 000  | 63                        | 960                           | 1 253 920                           | 17                                   | 8.8                                      |
| T1011N5K5P  | 4       | 3 524                     | 45 0812                       | 1 600 477                           | 14                                   | 3.7                                      |
| kosarak     | 860     | 1 437                     | 11 469                        | 3 578 574                           | 19                                   | 8.36                                     |
| pubmed      | 32 000  | 36                        | 535                           | 6 613 656                           | 20                                   | 1.6                                      |
| T1011D10K   | 3       | 866                       | 220 988                       | 5 169 854                           | 14                                   | 4.43                                     |
| mushroom    | 250     | 82                        | 1 684                         | 9 544 484                           | 17                                   | 8.9                                      |
| T3011D100K  | 220     | 501                       | 104 161                       | 10 174 500                          | 20                                   | 8.48                                     |
| connect     | 43 100  | 34                        | 483                           | 11 869 442                          | 19                                   | 10.1                                     |
| BMS-POS     | 350     | 3 486                     | 13 037                        | 15 747 841                          | 20                                   | 6.4                                      |
| T2011N5K5P  | 57      | 884                       | 37 377                        | 16 637 252                          | 13                                   | 6.4                                      |
| retail      | 90      | 4 621                     | 86 776                        | 16 964 579                          | 20                                   | 8.3                                      |
| BMS-WebView-2 | 5     | 186                       | 106 070                       | 60 163 674                          | 23                                   | 5.8                                      |
| BMS-WebView-1 | 35   | 272                       | 6 544                         | 60 417 674                          | 25                                   | 11.5                                     |

**low support threshold**

Table 3.3: Some statistics about the frequent itemsets
2. any list has some overhead (at least 8 bytes, but in the case of C++ STL's `vector` it is 12 bytes on a 32 bit processor), i.e., the size of an empty list is not zero.

An inhomogeneous trie spares memory by saving the overhead of the lists at the leaves. Since tries of FIM algorithms are very large, and contain many leaves, the saving may be significant. Note that the size of a leaf of an inhomogeneous trie is merely the size of a counter, i.e., 4 bytes. On the contrary the leaf takes 12+4=16 bytes in a homogeneous trie. The cache line (the block that is the basic unit in transferring data from the memory to the cache) size is 32 bytes in the case of Pentium 4 processor, which means 8 and 2 leaves fit in a cache-line in the case of inhomogeneous and homogeneous trie, respectively. In 64 bit architectures (like Opteron) the difference is even larger (the size of a leaf is the same, however, the size of a pointer is 8 bytes).

Notice, that if a transaction contains an itemset represented by a leaf, then it contains its siblings many times. It is important that the siblings be as "close" to each other in the memory as possible to obtain better data locality.

Leaves being generated in the candidate generation phase, deleted or converted into inner node in the infrequent removal phase require a lot of allocations/deallocations. We can reduce the overhead of this and improve data locality at the same time by applying a special block allocation mechanism. The leaves are stored in a block\(^1\) and there is an extra stack that stores pointers of the freed places. When a leaf is freed, a pointer to its place is popped to the stack. When a new leaf is allocated, we check if the stack is empty. If not, we reallocate the memory that is pointed by the top element of the stack. If the stack is empty, then we simply allocate a new element in the current block. Since a leaf is practically a counter (and integer), reallocation means a value assignment.

This solution can be further improved by merging together the stack and the blocks, i.e., each position of a block is either a leaf or a pointer that points to the next empty position (if there is any, otherwise its value is NULL). In C++ this solution is supported by the `union` data structure and by the fact that a pointer and an integer needs the same amount of memory in 32 bit processors.

Table 3.4 shows some experiments concerning this design detail.

An inhomogeneous trie with our special block allocator reduces both run-time and memory need significantly. In the forthcoming experiments we always use inhomogeneous tries and our block-allocator.

### 3.5.1 Routing strategies at the nodes

*Routing strategy* at an inner node refers to the principle used to select the edges to follow during the recursive traversal of the support count method. Given a node with a list of edges and a part of the transaction \( t \) denoted by \( t' \) we have to find the edges whose labels are included in \( t' \). This is the main step of support count in APRIORI,

\(^1\)Actually we used a list of medium-size blocks instead of one big block in our implementation.
it is called many times, and this is the step that primarily determines the run-time of the algorithm. In this section we analyze some possible solutions. The number of edges having the node we investigate (at depth $d$) is denoted by $n$. For the sake of efficiency the elements of the transaction are ordered.

Different routing strategies can be applied with different edgelist representations (see section 3.1.1). In an indexvector-based solution the edge that has a given label can be found in one step, thus we adapt the simple method that checks for each element $i$ of $t'$ if there exists an edge with label $i$. In our implementation we skip those elements that are smaller than the smallest label (this equals to the offset if the offset trick is applied), and terminate the search if the actual element of $t'$ is larger than the largest label (i.e., offset plus the size of the vector).

With an ordered list representation several solutions are applicable:

**Simultaneous traversal (merge):** Two pointers are maintained: one goes through the elements of $t'$ and the other goes through on the $n$ edges. Both pointers are initialized to the first element of the corresponding list. The pointer that points to the smaller item is increased. If the pointed items are the same, then a match is found (recursive step is called), and both pointers are increased. We terminate the search if any pointer reaches the end of its list. The worst case number of comparisons (and pointer increases) is $n + |t'|$, the best case is $\min\{n, |t'|\}$.

**Find corresponding edge:** For each item in $t'$ we find the corresponding edge (if there is any). We can use a binary search for finding the proper label. Notice that the run-time of the binary search is proportional to $\log_2 n$. Since the labels are ordered,
it is enough to perform binary search from the position that the previous binary search returned.

**find corresponding transaction item:** For each label we find the corresponding transaction item. For this a binary search starting from the previously returned index is applicable.

The logarithmic run-time need of the binary search can be reduced to constant time by applying an offset-bitvector representation of $t'$, whose value at index $i$ is **true** if item $i$+offset is the element of $t'$ otherwise **false**. The offset is the smallest element of $t'$.

The problem with bitvectors is that they do not exploit the fact that at a certain depth only a part of the transaction needs to be examined. For example, if the item of the first edge is the same as the last item of the basket, then the other edges should not be examined. The bitvector-based approach does not take into consideration the positions of items in the basket.

We can easily overcome this problem if the indices of the items are stored in the vector. For example transaction \{2, 4, 7\} is stored as [1, 0, 2, 0, 0, 3] with offset 2. The routing strategy with this vector is the following. First we step through those edges whose labels are less than the offset. Then we take the remaining labels one-by-one. If we reach for item $i$ in $t'$, then we check the element $i$−offset of the vector. There are three possibilities. If it is 0, then the item is not contained; we proceed with the next label. If the element is smaller than $|t| − \ell + d + 1$ then match is found (and the support count procedure is continued with the next label). Otherwise the procedure is terminated.

For each routing strategy we could give an upper bound on the number of comparisons in the worst case. Comparing these theoretical values, however, predict the efficiency of the routing strategies much worse than the degree each method suits to the features of the modern processor and memory structures. Now let us turn to the experiments we have carried out.

### 3.5.1.1 Routing strategies in the case of ordered-list edge representation

First we tested the routing strategies that can be applied when the edges are stored in an ordered list. Two typical plots are depicted in Figure 3.7.

Some hardware friendliness diagrams are given in Figure 3.8.

Observations based on all the tests are the following:

1. There exists no single routing strategy that outperforms all other routing strategies on every database with every support threshold. The run-time differences between routing strategies is sometimes up to ten-fold.
Figure 3.7: Routing strategies in the case of ordered edgelist representation

Figure 3.8: Hardware friendliness diagrams of some routing strategies
2. Except for merge, there exists a dataset for each routing strategy where its performance is quite bad compared to the best one.

3. merge outperforms the binary-search based approaches most of the cases by a significant margin.

4. Binary search-based approaches always get faster if the position returned by the previous binary search is stored and used to decrease the search space.

5. Bitvector based solutions performed poorly most of the times; it was always slower than merge.

Let us explain the observations one-by-one.

1. The efficiency of a routing strategy depends on \( n \), the length of \( t' \) and the number of matches. Different data have different characteristics concerning these values, thus different routing strategies perform well.

2. The merge strategy produces the simplest code (its code contains the fewest lines) and it does not wait for the data because the items are read sequentially and the prefetch feature is very effective.

3. If only the number of comparisons (in the worst/average case) is taken into consideration then binary search is always faster than linear search. If we, however, also consider the way modern processors' features are utilized, we conclude that the linear search outperforms binary search significantly when the lists we are searching in are small. Notice that pipelining, prefetching performs poorly since the element of the list to process depends on the outcome of the previous comparison. This also results in an inefficient branch-prediction.

4. Storing the index that was returned form the previous binary search reduces the average number of theoretical comparisons from \( n \log_2 n \) to \( \log_2 n! \). This simple trick is also greatly supported by the modern processor's cache system. Storing and using the value that was returned by the last binary search is performed quite fast most of the times since it is likely to be stored in the L1 cache.

5. The bitvector-based approach does not take into consideration that only a part of the transaction has to be examined. This results in many superfluous traversals. Let us see an example. Assume that the only 4-itemset candidate is \( \{D, E, F, G\} \) and we have to find the candidates in transaction \( \{A, B, C, D, E, F\} \). Except for the bitvector-based approach all the techniques considered will not visit any node except the root, because there is no edge of the root whose label corresponds to any of the first \( 6 - 4 + 1 = 3 \) items in the transaction. On the contrary, the bitvector-based approach uses the whole transaction and starts with a superfluous
travel that goes down even to depth 3. The indexvector-based solution overcomes this drawback.

3.5.1.2 Routing strategies in the case of different edge representation

Next we compared the “winner” (i.e., merge) to the routing strategies that can be applied when offsetindex-vector and hybrid edge representation is used. In the case of hybrid edge representation (i.e., ordered list or offsetindex-based representation is selected depending on the sizes, in other words, the node representation is not unique but changes dynamically) a hybrid routing strategy is used: lookup-edge if the current node uses offsetindex-vector, merge otherwise. For the sake of memory compactness we used the uppermost bit of the nodes’ counter to store the type of representation of the nodes’ edges.

The hybrid solution almost always outperformed the other two solutions concerning both run-time and memory need. The offsetindex-vector approach performed quite poorly in most of the cases. This is attributed to its large memory need. The correlation between the memory need and run-time is quite apparent, the solution is competitive in run-time only when it is competitive in memory-need.

![Figure 3.9: Ratio of run-time and memory-need of ordered list-based Apriori compared to hybrid edge representation-based Apriori](image)

Some hardware friendliness diagrams are given in Figure 3.10.

The hybrid solution is more efficient than the ordered-list edge representation with the merge routing. The advantage is not very significant, the largest difference was 62% in run-time and 37% in memory-need.

In the rest of the experiments we use hybrid edge representation and hybrid routing strategy.
3.6 Removing Dead-end Branches

Frequent itemsets of size $\ell$ are only needed in (1) writing out the results and (2) generating candidates of size $\ell + 1$. The results can be written out either in candidate generation or at the infrequent candidate removal phase. In candidate generation some leaves are extended (if adding an item to its representation results in an itemset all of whose subsets are frequent) some are not. This means that there are leaves that represent candidates and there are leaves that do not. We call the second kind of leaves dead-end leaves and a subtrie is a dead-end branch if all its leaves are dead-end leaves. Dead-end branches are also generated in infrequent removal phase. If all (or all but one) children of a node are infrequent, then the node becomes a leaf and is never extended again.

The nodes of a dead-end branch are not needed for candidate generation thus its nodes’ itemsets can be written out and such nodes can be purged from the trie. This technique has many advantages. First, the trie gets smaller. Second, the support count is faster. To illustrate this, let us assume that only one candidate (itemset $ABC$) is generated. Figure 3.11 shows two candidate tries. The second is obtained by applying the dead-end branch pruning. The advantage of dead-end branch removal can be easily seen if we consider finding the candidates in transaction $\langle A, B, C, D, E \rangle$. In both cases the whole trie is traversed, which means visiting only half as many nodes in the second case as in the first case.

Dead-end branch pruning does not require any movement in the trie, if the nodes in the candidate generation phase are visited in a preorder depth first manner. This is based on the Property 2.

Property 2 Let $I$ be an itemset and $\prec$ an ordering on the elements of $I$. $I$ strictly precedes all subsets of $I$ according to the lexicographical ordering of $\prec$.

Consequently, an itemset $I$ can be a subset of those candidates whose generators
strictly precede \( I \) in the preorder traversal. Therefore a node can be pruned if no new candidates are generated from any descendants of it.

Dead-end branch pruning does not necessarily speed up Apriori. If there exist no dead-end paths, then the dead-end branch checks mean unnecessary condition check. For example if all maximal candidates have the same size, then dead-end pruning is never used, and this technique neither results in a faster nor a more memory-efficient algorithm. Fortunately, in most cases the negative border of frequent itemsets (i.e., the maximal candidates) is not "straight" and the size of the maximal candidates varies. Also note that conditions that are always true (or always false) are 100% predictable, and they do not deteriorate branch prediction efficiency. Figure 3.12 shows the ratio of run-time and memory need of Apriori that does not use the dead-end pruning and the Apriori that does.

Some hardware friendliness diagrams are given in Figure 3.13.

The experiments show that dead-end pruning is an efficient technique. It always resulted in a faster and more memory-efficient algorithm.
3.7. CACHING THE TRANSACTIONS

The problem of traversing dead end paths was also considered in [23] as an influence of our earlier paper [15]. The author of [23] has chosen another solution. For each node a boolean value was attributed (more precisely the uppermost bit of the counter was dedicated for this purpose) whose value is true if the node is on a path to the deepest level (i.e., to a candidate), otherwise false. Recursion during support counting proceeds only on such children whose boolean value is true.

This solution has two drawbacks. First, dead end branches are not erased and therefore the space is not freed. Second, the boolean value check is just a second test after a matching of items is found during support count (see routing strategy merge on page 34). Thus the items with false boolean values are also considered in finding the edges to follow. This problem could be solved by not just distinguishing the edges but actually storing different edges in two different lists. This requires, however, more than one bit overhead.

It is easy to see the consequence of the two drawbacks if we compare the experiments (for details see [23]). It reached 20-40% speed-up at database BMS-WebView-1, while our solution resulted in a more than twice so fast program.

In the rest experiments with Apriori we use dead-end pruning.

3.7 Caching the transactions

I/O and string to integer parsing costs are reduced if the transactions are stored in the main memory instead of disk. It is useless to store the same transactions multiple times. It is better to store them once and employ counters representing the multiplicities. This way, memory is saved and run-time may be significantly decreased. This technique is used in FP-growth and can be used in APRIORI as well.

The advantage of this idea is the reduced number of support count method calls. If a transaction occurs \( n \) times, then the expensive procedure is called just once (with
counter increment \( n \) instead of \( n \) times (with counter increment 1). Thus the number of calls to the most expensive method may be considerably reduced. Unfortunately, the data structure needs memory, and its build-up (i.e., collecting the same transactions) requires processor time.

We refer to the data structure that stores the transactions together with the multiplicities as transaction cache. The transactions are cached after the first scan, so that infrequent items can be removed from the transactions. Different data structures can be used as transaction caches. We have three requirements:

1. inserting an itemset has to be fast,
2. the data structure has to be memory-efficient,
3. listing the transactions and the multiplicities has to be fast.

A simple solution is an ordered vector, each element stores an itemset and its multiplicity counter. Inserting a transaction becomes slow as the number of transactions becomes large. A better solution is a vector of ordered vectors where the \( j \)-th vector stores transactions of size \( j \). We refer to this solution as order-array based cache.

The most famous Apriori implementation [24] uses trie and in our previous implementation we have used a red-black tree (denoted by RB-tree). In an RB-tree cache each node stores a transaction. Due to the success of Patricia-trees in FP-growth based algorithms [88] we also tested this solution.

<table>
<thead>
<tr>
<th>database</th>
<th>( \minsup )</th>
<th>ordered-array</th>
<th>RB-tree</th>
<th>trie</th>
<th>patricia</th>
</tr>
</thead>
<tbody>
<tr>
<td>k scarcity</td>
<td>48 000</td>
<td>1.74</td>
<td>0.63</td>
<td>0.63</td>
<td>0.84</td>
</tr>
<tr>
<td></td>
<td>840</td>
<td>212.8</td>
<td>2.79</td>
<td>2.56</td>
<td>1.68</td>
</tr>
<tr>
<td>accidents</td>
<td>3.96</td>
<td>3.49</td>
<td>1.13</td>
<td>0.50</td>
<td>0.8</td>
</tr>
<tr>
<td>BMS-PCS</td>
<td>100 500</td>
<td>94.4</td>
<td>1.88</td>
<td>1.48</td>
<td>1.23</td>
</tr>
<tr>
<td></td>
<td>5 000</td>
<td>128.05</td>
<td>1.46</td>
<td>0.68</td>
<td>0.53</td>
</tr>
<tr>
<td></td>
<td>67</td>
<td>153.28</td>
<td>1.51</td>
<td>1.16</td>
<td>0.72</td>
</tr>
<tr>
<td>webdocs</td>
<td>700 000</td>
<td>27.05</td>
<td>25.62</td>
<td>25.68</td>
<td>24.98</td>
</tr>
<tr>
<td></td>
<td>200 000</td>
<td>1030.25</td>
<td>38.45</td>
<td>45.56</td>
<td>31.89</td>
</tr>
</tbody>
</table>

| | database  | \( \minsup \) | ordered-array | RB-tree | trie  | patricia |
| | k scarcity | 48 000       | 0.69          | 0.69    | 0.69  | 1.03     |
| |           | 840          | 28.02         | 32.16   | 22.55 | 10.02    |
| | accidents | 210 000      | 2.91          | 2.72    | 1.45  | 1.51     |
| | BMS-PCS   | 100 500      | 21.68         | 15.15   | 13.60 | 9.2      |
| |           | 5 000        | 15.09         | 17.86   | 24.84 | 10.56    |
| |           | 67           | 23.31         | 22.87   | 38.21 | 13.12    |
| | webdocs   | 700 000      | 48.66         | 48.66   | 534.01| 254.84   |
| |           | 200 000      | 278.16        | 286.98  | 934.01| 254.84   |

Table 3.5: Transaction caching with different data structures

The experiments proved our expectation, that ordered-vector and vector of ordered-vector solutions are not competitive with tree based solutions (the table does not even
include the order vector-based solution, since its run-time exceeded the acceptable run-time threshold most of the cases). Tries slightly outperforms RB-trees concerning run-time, but their memory need is much larger, even larger than the memory need of order-array solutions. Trie is said to be an efficient data structure in compressing data sets because it stores the same prefixes once instead of the number of times it appears (which is the case with ordered-arrays and RB-trees). Experiments, however, do not support the statement about compression efficiency.

The reason for this comes from the fact that a trie has much more nodes – therefore much more edges – than an RB-tree has (except for one bit per node, RB-trees need the same amount of memory as simple binary trees). In a trie each node stores a counter and a list of edges. For each edge we have to store the label and the identifier of the node the edge points to. Thus adding a node to a trie increases memory need by at least 5 · 4 bytes (if items and pointers are stored in 4 bytes). In a binary tree, like an RB-tree, the number of nodes equals to the number of transactions. Each node stores a transaction and its counter.

When inserting the first \( \ell \)-itemset transaction in a trie, \( \ell \) nodes are created. However in an RB-tree we create only one node. Although the same prefixes are stored only once in a trie, this does not reduce the memory difference so much. This is the reason for the empirical fact we observed, that a binary tree consumes \( 3-10 \) times less memory than a trie does.

A Patricia tree overcomes the defect of a trie that stems from the inefficient storage of single paths. It substitutes a single path with one link with a label equal to the set of labels that are on the path. This spares many pointers but more importantly, the memory need caused by the overhead of a list is greatly reduced. Thus Patricia trees keep the advantage of trie-based solution without suffering from large memory need.

In this section we avoid discussing the run-time and memory need effect of the ordering used to convert itemsets to sequences. An in-depth analysis is provided in section 4.4.

After finding the best data structure for a transaction cacher, we investigated if transaction caching really speeds up Apriori. In these experiments (see some results in Figure 3.14) we have used a Patricia-tree as a transaction cacher.

Some hardware friendliness diagrams are given in Figure 3.15.

Experiments show, that transaction caching is a great speed-up technique, it sometimes (in the case of connect, pumsb) decreases run-time by several orders of magnitude, sometimes the run-time "just" drops to its fraction (accidents, BMS-WebView-1, T20I1ON1KP5KCC, 25D200K, pumsb*). Due to the last tree-based solution, this technique is regarded run-time safe, i.e., even at databases where the number of support count method calls do not decrease significantly, building up the cacher does not reduce overall run-time. Building-up the cacher never takes significant time compared to frequent itemset mining (the largest run-time increase was 10% and 5% at databases retail and BMS-WebView-2, respectively) at low support thresholds.
Figure 3.14: Caching the transactions

Figure 3.15: Hardware friendliness diagrams of Aprioris with and without transaction caching
This technique is obviously not memory safe. The cacher may need a lot of memory, even more than the memory needed by the candidates. With most of the databases the memory increase was not too large and we found no databases where the increased memory assumption resulted in swapping. In the remaining experiments we will turn transaction caching on.

3.8 Equisupport pruning

The search space pruning based on equisupport itemsets is perhaps the most widely used speed-up trick in the FIM field. Omitting equisupport extensions means excluding from the support counting the proper supersets of those \( \ell \)-itemsets that have the same support as one of their \((\ell - 1)\)-subsets. This comes from the following simple property.

**Property 3** Let \( X \subset Y \subseteq J \). If \( \text{sup}(X) = \text{sup}(Y) \), then \( \text{sup}(Y \cup Z) = \text{sup}(X \cup Z) \) for any \( Z \subseteq J \).

This property holds for all \( Z \subseteq J \), nevertheless we restrict our attention to itemsets \( Z \subseteq J \setminus Y \).

The connection between the equisupport pruning and closed itemset mining is obvious. Itemset \( X \) is a non-closed set, with closure \( Y \), if there exists no proper superset of \( Y \) with support equal to \( \text{sup}(Y) \). An itemset \( X \) can be an antecedent of an exact association rule (rule with confidence 100\%) if and only if it is a non-closed itemset. Itemset \( X \) is called a **key pattern** [13] if there exist no proper subset of \( X \) with the same support.

Equisupport pruning can be easily used in prefix-based, depth-first algorithms. Here we show how to employ this technique in Apriori. The complete pruning of Apriori allows us to perform a more sophisticated equisupport pruning than the one that is used in prefix-based, depth-first algorithms.

3.8.1 Prefix equisupport pruning

If candidate \( Y \) has the same support as its prefix \( X \), then it is not necessary to generate any superset \( Y \cup Z \) of \( Y \) as a new candidate. Based on the above property its support can be calculated directly from its subset \( X \cup Z \) [44].

The support of the prefix is always available at bottom-up FIM algorithms, thus **prefix equisupport pruning** (i.e., \( X \) is the prefix of \( Y \), such that \( |X| + 1 = |Y| \)) can be applied at any time. The technique works the following way. After determining the supports of the children of itemset \( P \), we check at the infrequent removal phase if their support are equal to \( \text{sup}(P) \). Children with such supports are not considered as generators in later phases and the extending items that belong to them are stored in a set (called **equisupport set**) and associated with itemset \( P \). Notice, that due to the
non-redundant traversal of the itemset lattice, $Y \setminus X \prec z$ for all $z \in Z$ where $\prec$ denotes the order used to define the prefix.

When writing out a frequent itemset $I$, we also output the union of $I$ with itemset $E'$ for all $E' \subseteq E$, where $E$ is the union of all equisupport sets for the prefixes of $I$.

**Example 4**

Let us assume that the following itemsets of size two with prefix $A$ are found to be frequent $AB, AC, AD$ and $\sup(A) = \sup(AB) = \sup(AC) = 4, \sup(AD) = 3$. Only itemset $AD$ is considered as generator for further candidates with prefix $A$. At least two itemsets are needed to generate a candidate in Apriori, Eclat and FP-growth, thus processing prefix $A$ terminates. When writing out itemsets $AD$ and $A$ we also append all subsets of $BC$ to them, thus we write itemsets $AD, ABD, ACD, ABCD$ with support 3, and $A, AB, AC, ABC$ with support 4.

If the database contains only closed sets, then equisupport pruning is never used and the large number of support equivalence checks just slows down the algorithm. Experiments, however, show that in all algorithms the equisupport check can be performed quite fast (for example in the case of Apriori it requires no traversal in the trie) and results no cache misses. Even at databases that contain insignificant number of non-closed sets the run-time increase is absolutely insignificant.

Prefix equisupport pruning can be applied in all bottom-up FIM algorithms, where candidates are generated on the basis of prefixes. From an Apriori’s trie point-of-view, each node has to be extended with a list that stores equisupport items. In the infrequent candidate removal phase we check if a leaf has the same support as its prefix generator. If it has, then the leaf is purged from the trie and the label of the link is added to the parent’s equisupport set. Each item $i$ in an equisupport set can be regarded as a loop edge with label $i$. Loop edges are not considered in support count, but must be considered in the complete pruning step of candidate generation.

**Example 5**

Let itemsets $AB, AC, BC$ be the only frequent pairs, $\sup(AB) \neq \sup(A) \neq \sup(AC)$ and $\sup(B) = \sup(BC) = \sup(BD)$. Figure 3.16 shows the trie obtained after infrequent candidates removal phase. Notice that if loop edges were not considered in the previous step of the candidate generation, then itemset $ABC$ would not be generated as a candidate even though all its subsets are frequent.

This example draws attention to the connection between equisupport pruning and dead-end branch removal. We see that node $B$ does not lead to a leaf at depth 2 therefore dead-end branch removal would erase this node, and itemset $ABC$ would not be generated. The depth of a node for dead-end branch removal must be redefined so that it does not purge leaves that may be needed for a proper complete pruning. We have to see, that an itemset obtained by taking the union of a leaf $X$ and any item that
is in the equisupport set of some prefix of \(X\) has the same support as \(X\). Thus when considering the depth of node \(X\) during dead-end branch removal, we have to add to the actual depth of \(X\) the size of the equisupport sets that are on the path from the root to \(X\). For example the depth of node \(B\) in Figure 3.16 is 3 instead of one.

The astute reader may notice that edge that points to node \(B\) from the root is considered in support count, however it does not lead to any candidate. We have seen the run-time impact on the support count method of ignoring such nodes when we analyzed dead-end pruning (see section 3.6). If they cannot be pruned (so that complete pruning can be applied), then they should be distinguished. Edges that are on a path to a candidate should be type one (let us call them normal edges), while the rest including the equisupport loops should be of type two (denoted by dashed edges). Such “colored trie” is depicted in Figure 3.17. The frequent pairs are \(AB, AC, AD, BC, BD, CD, CE\) and \(\text{sup}(A) = \text{sup}(AD), \text{sup}(B) = \text{sup}(BD), \text{sup}(C) = \text{sup}(CD)\). The upper trie stores the frequent two itemsets. Below, on the left a trie is depicted, which is obtained after candidate generation if equisupport pruning and coloring is used. The trie on the right is generated if no equisupport pruning is used.

Notice that when determining which candidates are contained in transaction \(\langle ABCD EFGH \rangle\), only four nodes are visited in the colored trie, nine in the original equisupport and 13 in the non-equisupport case.

Although distinguishing the edges seems to be a good practice, it also has some drawbacks. Each node stores two lists of edges, that means double overhead. In databases that do not contain non-closed itemsets, the second type of edges are never used. We have seen (in section 3.3) that increasing the size of the trie nodes deteriorates run-time and memory need. With an other solution we may get rid of large part of the overhead. Instead of this technique, here we propose a different solution that we call level 2 equisupport pruning.
3.8.2 Level 2 equisupport pruning

It seems contradictory to restrict equisupport pruning to prefixes in the case of Apriori since all subsets together with the supports are available and the equisupport Property 3 (see page 45) is fulfilled for every subset. To understand why we can not apply a general equisupport pruning we have to understand, that

- complete pruning does not allow simple removing of equi-support leaves. A loop edge can be regarded as a classic edge that leads to a node that is fairly similar to its parent. It is like copying an identical subtree of a child to the node itself. Thus a node with many self loops is a compact representation of a whole imaginary subtrie, which is traversed during the complete pruning.

- for efficient support counting and candidate generation the trie has to store ordered sequences, i.e., the labels on all paths that start from the root and lead to a leaf have to be ordered. In other words when an inclusion of an itemset \( X \) is checked we start from the root and check if there exist a link with label equal to the smallest element of \( X \). If there exists we follow the link, and then check the second smallest element, etc.

Based on a non-prefix subset equivalence, removing a leaf and adding a loop link, however, may invalidate the second assumption. Let us consider the example, where \( F_2 = \{ AB, AC, BC, BD, CD \} \) and \( \text{sup}(BC) = \text{sup}(C) \). Since leaf \( BC \) has same support
as its subset, it can be removed, and a loop edge with label $B$ has to be added to node $C$. This is seen in Figure 3.18.

![Figure 3.18: Example: nonprefix equisupport pruning ruins ordering](image)

The trie obtained by a nonprefix equisupport pruning does not meet the ordering requirement. Node $BC$ cannot be reached from the root, by first checking item $B$ and then $C$. Therefore, itemset $ABC$ is not generated as a candidate because its subset $BC$ can not be verified.

Fortunately, there exists a set of subsets that allows a second type of equisupport technique, because it does not invalidate the ordering.

Here we propose a new equisupport pruning technique, which meets the ordering requirement of the trie, thus it can be applied. It can be used only if the prefix equisupport pruning is used as well.

**Property 4** Let $Y$ be the prefix of itemset $Y \cup z$, where $|z| = 1$. If there exists a subset $X$ of $Y$ such that $|X| + 1 = |Y|$ and $\text{sup}(X \cup z) = \text{sup}(X)$, then $\text{sup}(Y \cup z) = \text{sup}(Y)$.

The above property is a special case of the general equisupport pruning property. We emphasized on the purpose to better illustrate which itemsets play role in this pruning technique. To use the pruning, it requires that we know the equisupport sets of all subsets. This information is only available in Apriori.

This special equisupport pruning can be easily adapted in the candidate generation phase. The second step of the candidate generation is checking all $\ell$-subsets if they are frequent. These are reached by the $(\ell - 1)$-element prefixes of them. We can add an extra check to apply the equisupport pruning. If the largest item of the candidate is in the equisupport set of a subset of the prefix, then the candidate is pruned and this largest item is placed in its generator’s equisupport set.

**Example 6**
Suppose that the set of frequent two itemsets are \{AB, AC, AD, BC, BD\} and the only equisupport is $\text{sup}(BC) = \text{sup}(C)$. We do not generate $ABC$ as a candidate because it has a 2-element subset that contains $C$ in the equisupport set of its prefix. Figure 3.19
depicts the trie before and after the candidate generation. Please keep in mind, that
death-end branch pruning is applied during candidate generation.

Figure 3.19: Example: special prefix equisupport pruning

The example also shows that this technique may also reduce the number of iterations
of Apriori. Consider the above example except that itemset $BD$ is not frequent.
Three iterations are needed in non-equisupport case because $ABC$ would be a can-
didate. Equisupport pruning, however, prevents us from generating $ABC$ as a candidate,
and terminates Apriori after the second iteration.

3.8.3 Level 2 equisupport pruning and further death-end prun-
ing

Further pruning can be applied if level 2 equisupport pruning is used. This is based on
the following lemma.

**Lemma 3.8.1** In the candidate generation phase when checking all subsets of an $(\ell+1)$-
itemset, no equisupport sets of nodes at depth $d$ for all $d < \ell - 1$ need to be considered,
if level 2 equisupport pruning is used.

**PROOF:** We prove this statement by contradiction. Let us assume the prefix of the
candidate is denoted by $P$ and item $i_j$ of subset $Q = \{i_1, i_2, \ldots, i_j, \ldots, i_t\}$, is in the
equisupport set of itemset $P_Q = \{i_1, i_2, \ldots, i_{j-1}\}$. We claim that itemset $Q' = P_Q \cup (P \setminus
Q) \setminus \{i_j\}$ could not have been generated as a candidate at iteration $j$. If $i_j < P \setminus Q$, then the
prefix equisupport check prunes $Q'$ (because it prevents extending $P_Q$), otherwise the
level 2 pruning does this work, because the largest item of $Q'$ is in the equisupport set
of its subset $P_Q$. □

Table 3.6 illustrates the rationale of the proof ($P = \{ABCD\}$). The table contains
the subset of $P$ that is not generated as a candidate, if the items corresponding to the
3.8. EQUISUPPORT PRUNING

indices of the row and column, are \( i_j \) and \( Q \), respectively. For example item \( B \) cannot be in the equisupport set of itemset \( A \) because it contradicts to the fact that \( ABC \) was a candidate. Also, if item \( C \) is in the equisupport list of itemset \( B \), then equisupport pruning in candidate generation prevents generating itemset \( ABC \) as a candidate. In general, the existence of itemsets above the diagonal as a candidate contradicts to prefix equisupport pruning, while under the diagonal the itemset contradicts to equisupport pruning in the candidate generation phase.

<table>
<thead>
<tr>
<th>( P \setminus Q )</th>
<th>Q</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABC</td>
<td>D</td>
<td>AD</td>
<td>ABD</td>
<td>ABCD</td>
<td>−</td>
</tr>
<tr>
<td>ABD</td>
<td>C</td>
<td>AC</td>
<td>ABC</td>
<td>−</td>
<td>ABCD</td>
</tr>
<tr>
<td>ACD</td>
<td>B</td>
<td>AB</td>
<td>−</td>
<td>ABC</td>
<td>ABCD</td>
</tr>
<tr>
<td>BCD</td>
<td>A</td>
<td>−</td>
<td>AB</td>
<td>ABC</td>
<td>ABCD</td>
</tr>
</tbody>
</table>

Table 3.6: Illustration of the proof of Lemma 3.8.1

Lemma 3.8.1 allows us to (1.) simplify the code (equisupport sets need to be considered only at level \( \ell - 1 \)) and (2.) remove some dead-end branches. Nodes at depth \( \ell - 1 \) with no children can be removed after the candidate generation, even if their equisupport sets are not empty. This pruning does not require any extra movement in the trie. The preorder traversal of the trie ensures that any \( \ell \)-itemset can be a subset of an \( (\ell + 1) \)-itemset that is generated by the preceding nodes. This corresponds to the property 2 (see page 39) used in dead-end pruning. We call level 2 pruning together with dead-end pruning presented in this section as level 3 equisupport pruning.

Example 7

The Figure 3.20 illustrates level 3 equisupport pruning. The trie on the left side is obtained after infrequent removal phase at iteration 2. After candidate generation and the new dead-end pruning, we get the trie that is depicted on the right side of the figure. Notice that nodes A and B are present in the next iteration if equisupport pruning in candidate generation is not applied because their virtual depth is 4 and 3. These unnecessary branches slow down support count throughout two iterations.

The example also shows that this dead-end pruning also reduces the number of iteration in Apriori. The virtual depth of node A is 4, therefore this node is removed during the candidate generation in iteration 5. Dead-end branch removal, however, terminates the algorithm before the support count of the 4th iteration begins.
CHAPTER 3. METHODS TO IMPROVE SUPPORT COUNT

Figure 3.20: Example: removing dead-end branches when level 3 equisupport pruning is applied

3.8.3.1 Experiments with equisupport pruning

Equisupport pruning is not necessarily run-time safe. If the database does not contain non-closed itemsets, then the memory allocations of the never used equisupport lists require extra processor operations. Furthermore, this technique is not necessarily memory safe. The equisupport sets need memory even if they are empty and never used. Experiments, however, show that the performance deterioration is not significant. The highest run-time and memory need degradation were 26% and 20%, respectively. We believe that this is attributed to the fact that equisupport check does not require any extra movement in the trie and can be performed quickly. In the experiments, whose results are shown in Figure 3.21, level 3 equisupport pruning was employed.

Figure 3.21: Equisupport pruning

Some hardware friendliness diagrams are given in Figure 3.22.

The results meet our expectation. In dense datasets the run-time and memory need drop to their fraction. The decrease may be of several orders of magnitude. Please notice the logarithmic scale.
Figure 3.22: Hardware friendliness diagrams of Apriori with different equisupport pruning techniques

Next, we tested if the speed-up is attributed to prefix equisupport or the other two prunings also play significant role. The answer is found in Fig. 3.23.

Figure 3.23: prefix equisupport pruning vs. level 3 equisupport pruning

Experiments show that equisupport pruning proposed in candidate generation and this special dead-end pruning do not only possess a nice theoretical foundation but it is an efficient speed-up technique in practice as well. In some cases the run-time dropped to its half.

**historical remark:** Similar pruning technique based on itemsets with equal support was first presented in algorithm PASCAL proposed by Bastide et al. [13]. Their solution differs from our in many aspects. First of all, they apply full equisupport pruning, i.e., they do not calculate the support of any proper superset of itemset $I$ if $\text{sup}(I) = \text{sup}(I')$ for any $I' \subset I$. They use the term key pattern for those itemset that have no proper subsets with the same support. The authors of PASCAL describe full equisupport removal in conceptual terms. This description suggests a naive/straightforward implementation that keeps the whole combinatorial set of equisupport expansions.
The edges may be distinguished so that many of them are not considered during support count, but the nodes have to exist in order to perform full pruning. We declare that the main merit of equi-support pruning is the fact that many nodes can be deleted and even more need not be generated. In dense databases the main bottleneck of Apriori is the heavy memory need of the large candidate trie. This is not reduced by the PASCAL technique. On the contrary, our solution solves this problem. The results of the experiments shown in Figure 3.21 justifies this argumentation.

3.9 Further techniques

In this section we give a short description of the two most important speed-up techniques that are used in other Apriori implementations.

3.9.1 Support count of Christian Borgelt

When the transactions are stored in a trie or in a Patricia tree then another support count technique can be applied. This clever idea was already mentioned in [24] and was sketched in [22]. This technique is used in the recent versions of Borglet's famous Apriori implementation.

The observation behind the idea is that two transactions result in the same program flow till the common element, i.e., till the common prefix. Storing the transactions in a trie gives the necessary information about the common prefixes. It is possible to process the same prefixes only once instead of the number of times it appears. The counter of itemset \( I \) in the transaction trie stores the number of transaction whose prefix is itemset \( I \). In this respect this solution differs from the one used in transaction caching (and rather it resembles to an FP-tree that is deprived of cross-links.) Another difference is that the ordering used in the transaction trie must correspond to the ordering used in the candidate trie. In section 4.4 we will see, that this is a drawback since the two tries prefer different orderings.

Unfortunately, the algorithm is not detailed in [22], but we believe it works as follows. We simultaneously traverse the candidate trie and the transaction trie in a double recursive manner. We maintain two node pointers respectively that are initialized to the roots. We go through on the edges of both node. If the label belong to the transaction trie is smaller or equal than the other label, then the recursion is continued on the child of the given transaction node, and with the same candidate node. If the two labels are equal, then the recursion is continued with the pointed children. A slightly optimized version is found in Algorithm 3.9.1.

The solution above suffers from the disadvantage of many redundant traversal in the transaction trie. It does not take into consideration the fact that only a part of a
Algorithm 3.9.1 BORGELETT_SUPPCOUNT

Require: \( n_c \): a node of the candidate trie,
\( n_t \): a node of the transaction trie,
\( \ell \): number of step from \( n_c \) that needs to be done to reach a leaf,
\( \hat{i} \), the smallest index of the edge of \( n_c \) that is larger than the label of edge that led to \( n_t \).

if \( \ell = 0 \) then
  \( n_c\text{-counter} \leftarrow n_c\text{-counter} + n_t\text{-counter} \)
else
  for \( j = 0 \) to \( n_t\text{-edge}\text{-number} - 1 \) do
    while \( i < n_c\text{-edge}\text{-number} \) AND \( n_c\text{-edge}[i]\text{.label} < n_t\text{-edge}[j]\text{.label} \) do
      \( i \leftarrow i + 1 \)
    if \( i < n_c\text{-edge}\text{-number} \) AND \( n_c\text{-edge}[i]\text{.label} \geq n_t\text{-edge}[j]\text{.label} \) then
      BORGELETT_SUPPCOUNT\((n_c, n_t\text{-edge}[j]\text{.child}, \ell, i)\)
    if \( n_c\text{-edge}[i]\text{.label} = n_t\text{-edge}[j]\text{.label} \) then
      BORGELETT_SUPPCOUNT\((n_c, n_t\text{-edge}[j]\text{.child}, n_t\text{-edge}[j]\text{.child}, \ell - 1, 0)\)
    \( i \leftarrow i + 1 \)
  else
    break

transaction needs to be evaluated. To overcome this problem we can employ a counter
for each node \( n_t \) of the transaction trie that stores the length of the longest path that
starts from node \( n_t \). During the support count we do not proceed the recursion on a
node whose counter is less than \( \ell - 1 \). Several other optimizations can be applied that is
based on removing unvisited or unimportant paths from the transaction trie. For more
details the reader is referred to \[22\].

3.9.2 Filtering unimportant items from the transactions

Filtering unimportant items from the transactions means removing those items from
each transaction that do not play role in determining the support of the candidates.
Obviously as the algorithm proceeds more and more items can be filtered from the
transactions. We have already mentioned a very simple filtering, i.e., after the first scan
we remove infrequent items from the transactions. A similarly simple filtering is when we
delete the transactions of size smaller than \( \ell \) at iteration \( \ell \).

Further filtering can be applied. To illustrate this imagine that the candidates of
size two are \( AB, AC, BC \) and \( DF \) and transaction \( ABCD \) is processed. Item \( D \) is not
contained in candidates of size 2 that are contained in the transaction, therefore it can be
deleted from the transaction. In general an element of the transaction can be removed
at iteration \( \ell \) if it is not contained in any candidate that occurs in the transaction \[22\].
A more sophisticated solution was proposed by Park et al. [82]. It is based on the fact that for a candidate \( I \) of size \( \ell + 1 \) to occur in a transaction each element of \( I \) must be contained in at least \( \ell \) candidates of size \( \ell \) that occur in the transaction. This is a necessary condition, therefore an item in the transaction can be trimmed if it does not appear in at least \( \ell \) of the candidates in the transaction. For example transaction \( ACDE \) is deleted if the candidates are the same as used in our previous example. Notice that the previous simple filtering does not remove any element from the transaction.

This technique often results in a large number of item erase, however, to evaluate its efficiency we have to take into consideration the overhead of removing an item from the transaction, which depends on the way the transactions are handled. There are different solutions in the literature.

Algorithm DCI [79] processes and filters each transaction one-by-one and writes them out to the disk, i.e., the database is reduced progressively. It uses optimized I/O operations for the efficient disk usage. If we employ an ordered vector, ordered array or a binary tree as a transaction cacher, then removing an item from a transaction can be replaced by removing the original transaction and inserting the filtered transaction. These transaction cacher, however, are not competitive with red-black tree, trie or patricia tree based solutions. Unfortunately, removing an item from a stored transaction is not an easy task in the case of trie and patricia tree, and it is a slow operation in the case of red-black tree (deletion may need the expensive rotation operation).

This drawback was also observed in [22] where the following heuristics were proposed. Rebuild the transaction cacher if the filtering result in a significant node decrease, otherwise use the original transaction cacher. The threshold of rebuild was determined experimentally.
Candidate Generation

The techniques proposed in the previous chapter speed-up the support count so much that candidate generation becomes the bottleneck of Apriori at many databases. In this chapter we propose an improved candidate generation method. Then, we examine the necessity of the second step of candidate generation and draw the surprising conclusion, that this step is not the main merit of Apriori, furthermore it slows down the algorithm in most of the cases. The last part of this chapter is dedicated to the influence of ordering used to convert sets to sequences. We present the order-preserving assumption that makes it possible to formally analyze many heuristics.

4.1 Intersection-based pruning

The classical candidate generation consists of two steps. First taking the union of two frequent itemsets that have common $(\ell - 1)$-prefix, and then we check the subsets. This latter step is called the \textit{complete pruning} of Apriori. From a trie point of view, each itemset that fulfills the complete pruning requirement can be obtained by taking the union of the representations of two sibling nodes in the trie. In the so called \textit{simple pruning} we go through all nodes at depth $\ell - 1$, take the pairwise union of the children and do the complete pruning check. Two straightforward modifications can be applied to reduce unnecessary work. On one hand, we do not check those subsets that are obtained by removing the last and the one before the last elements of the union (the resulting sets are the generators). On the other hand, the prune check is terminated as soon as a subset is infrequent, i.e., not contained in the trie.

A problem with the simple pruning method is that it unnecessarily traverses some parts of the trie many times. We illustrate this by an example. Let $ABCD$, $ABCE$, $ABCF$, $ABCG$ be the four frequent 4-itemsets. When we check the subsets of potential candidates $ABCD$, $ABCD$, $ABCD$, then we travel through nodes $ABD$, $ACD$
and $BCD$ three times. This gets even worse if we take into consideration all potential candidates that stem from node $ABC$. We travel to each subset of $ABC$ 6 times.

To save these superfluous traversals, we propose an intersection-based pruning method [21]. Let us assume that we want to add new leaves to node $P \cup x$, where $P$ denotes the prefix. When checking the subsets of itemset $P \cup \{x, y\}$, we check $P \cup x$, $P \cup y$ and $Q \cup \{x, y\}$ where $Q \subset P$ and $|Q| + 1 = |P|$. $P \cup x$, $P \cup y$ are the generators, they have to be frequent. Therefore when checking the subsets of $P \cup \{x, y\}$ it is enough to examine if item $y$ extends nodes $Q \cup x$ for all $Q \subset P$. Similarly, when checking subsets of $P \cup \{x, z\}$ we examine if item $z$ extends nodes $Q \cup x$ for all $Q \subset P$. Consequently node $P \cup x$ is extended by those sibling items that extend all $Q \cup x$ nodes, i.e. the extending set equals to the intersection of labels of edges that start from nodes $Q \cup x$. This is the point where we save the traversals. If nodes that represent $Q$ itemsets are stored, then checking the subsets of $P \cup \{x, z\}$ means determining the child nodes of $Q$ nodes that are reached by label $z$ and doing the intersection. Furthermore, if the edges are stored ordered and we memorize the index of edges used in the actual search (and it at a starting point in the next search), then in determining the items that extend the children of $P$ the edges of all $Q$ nodes are traversed at most once.

In intersection-based candidate generation when extending the children of $P$, we first find nodes $Q$, where $Q \subset P$, $|Q| + 1 = |P|$. Then we take each label $i$ of nodes that start from $P$ and determine if $x$ extends all $Q$ nodes. If not, then $P \cup x$ can not be extended, otherwise we take the intersection of the extender labels of $Q \cup x$ and the label of siblings $P \cup x$. The elements of the result set are the items that extend $P \cup x$, because they meet the complete pruning requirement.

Note the real advantage of this method. The $(\ell-2)$-subset nodes of the $P$ are reused, hence the paths representing the subsets are traversed only once, instead of $\binom{n}{2}$, where $n$ is the number of the children of the prefix.

**Example 8**

Let us assume that the trie obtained after removing infrequent itemsets of size 4 and dead-end paths is depicted in Fig. 4.1.

![Diagram](image)

**Figure 4.1:** Example: intersection-based pruning
4.1. INTERSECTION-BASED PRUNING

To get the children of node ABCD that fulfill the complete pruning requirement (all subsets are frequent), we find the nodes that represent the 2-subsets of the prefix (ABC). These nodes are denoted by $Q_1$, $Q_2$, $Q_3$. Next, we find their children that are reached by edges with label $D$. These children are denoted by $Q'_1$, $Q'_2$ and $Q'_3$ in the trie. The intersection of the label sets associated to the children of the prefix, $Q'_1$, $Q'_2$ and $Q'_3$ is: $\{D, E, F, G\} \cap \{E, F, G\} \cap \{F, G\} \cap \{F\} = \{F\}$, hence only one child is added to node ABCD, and $F$ is the label of this new edge.

When determining the extender items of node ABCE, we find the new $Q'_j$ node, i.e., children of nodes $Q_j$, that are reached by edge with label $E$. The lack of any such node indicates that ABCE cannot be extended, because it has a proper subset that is infrequent.

Intersection-based candidate generation is not necessarily faster than the traditional candidate generation. If the first, non-generator subset of the candidate is infrequent, then the traditional method terminates quickly. On the contrary intersection-based method first determines the nodes for all subsets of the prefix. Therefore the intersection-based method is faster under the negative border, and the traditional method may be the better solution when the elements of the negative border are generated. The distance from the negative border, however, is not know in advance.

We tested intersection-based pruning with and without the equisupport technique (Figure 4.2).

![Figure 4.2: Speed-up ratios of intersection-based candidate generation without and with Level 3 equisupport pruning](image)

Some hardware friendliness diagrams are given in Figure 4.3.

Obviously at databases where support count dominates, the overall run-time decrease is insignificant. Experiments shows that at databases where candidate generation takes a significant time of the overall run-time, the intersection-based candidate generation is an efficient technique.
Figure 4.3: Hardware friendliness diagrams of Aprioris with the simple classic and with the intersection-based candidate generation

Equisupport pruning influences efficiency of intersection-based pruning at databases which contain non-closed itemsets. Equisupport pruning reduces the number of support count and candidate generation calls, because it replaces these operations with subset enumeration. It is not known, however, how does the ratio of support count and candidate generation changes (this depends on the characteristics of the database). If the candidate generation becomes more significant, then the advantage of the intersection-based pruning grows.

4.2 Omitting complete pruning

Complete pruning is declared to be an inherent and important step of algorithm Apriori. It seems to be natural to use pruning, since — in contrast to the DFS algorithms — all subsets of a potential candidate are available. The main merit of Apriori against DFS algorithms is that Apriori generates a smaller number of candidates. In [21] it was shown that the efficiency of Apriori is not necessarily attributed to complete pruning, furthermore, complete pruning slows down Apriori most of the times. In the rest of the paper we refer to Apriori that does not apply complete pruning (i.e., the second step of the candidate generation is omitted) as Apriori-Noprun.

The advantage of the pruning is to reduce the number of candidates. The number of candidates in Apriori equals to the number of frequent itemsets plus the number of infrequent candidates, i.e., the negative border of the frequent itemsets. If pruning is not used, then the number of infrequent candidates becomes the size of the order-based negative border of the frequent itemsets, where the order corresponds to the order used in converting the sets to sequences (An itemset \( I \) is an element of the order-based negative border of \( F \) if it is not in \( F \), but its prefix \( P_{|I|-1} \) and the subsequent subset of \( I \) of the same size are in \( F \)). It follows, that if we want to decrease the redundant work (i.e., determining a support of the infrequent candidates), then we have to use...
the order that results in the smallest order-based negative border. This issue is further investigated in Section 4.4, here let us accept that the ascending order according to supports is expected to result in the minimal negative border.

The disadvantage of the pruning strategy is simple: we have to traverse some part of the trie to decide if all subsets are frequent or not. Obviously this needs some time.

Here we state that pruning is not necessarily an important part of Apriori. This statement is supported by the following observation, that applies in most cases:

\[ |NB^{\leq \lambda}(F) \setminus NB(F)| \ll |F|. \]

The left-hand side of the inequality gives the number of infrequent itemsets that are not candidates in the original Apriori, but are candidates in Apriori-Noprun. So the left-hand side is proportional to the extra work to be done by omitting pruning. On the other hand, \(|F|\) is proportional to the extra work done by pruning. Candidate generation with pruning checks all the maximal proper subsets of each element of \(F\), while Apriori-Noprun does not. The outcomes of the two approaches are the same for frequent itemsets, but the pruning-based solution determines the outcome with much more effort (i.e., traverses the trie many times).

Although the above inequality holds for most cases, this does not imply that pruning is unnecessary, and slows down Apriori. The extra work is just proportional to the quantities in the formulas above. Extra work caused by omitting pruning means determining the support of some candidates. The resource requirement of this is affected by many factors, such as the size of these candidates, the number of transactions, the number of elements in the transactions, and the length of matching prefixes in the transaction. The extra work caused by pruning comes in a form of redundant traversals of the tree during checking the subsets. This also depends on many other parameters.

As soon as the pruning strategy is omitted, Apriori can be further tuned by merging the candidate generation and the infrequent node deletion phases. After removing the infrequent children of a node, we extend each child the same way as we would do in candidate generation. This way we spare an entire traversal of the trie. This solution combines candidate generation and infrequent candidates removal phases.

This trick can also be used in the original Apriori, however – as opposed to the application of Apriori-Noprun – it does not necessarily speed up the algorithm. To understand this, we have to observe that candidate generation is always after the infrequent node deletion phase, in which some leaves and even entire branches of the trie may be removed. Since the trie is traversed many times during the complete pruning checks of the candidate generation, this trie purge may result in a significant run-time decrease. If the second step, and thus the numerous trie traversals are omitted, then we can merge infrequent candidate removal and candidate generation phase without the threat of causing performance degradation.

Figure 4.4 shows the performance gain of Apriori-Noprun compared to Apriori with classical pruning. We also check the results when equisupport pruning was turned on.
This means full equi-support pruning in the case of classic Apriori and prefix equisupport pruning in Apriori-Noprune.

Some hardware friendliness diagrams are given in Figure 4.5. Experiments show that complete pruning is not necessarily an important step of Apriori, furthermore it increases run-time most of the times. The highest difference was at database BMS-WebView-1, where the run-time dropped to its quarter as soon as complete pruning was omitted. Similar to intersection-based candidate generation, the equisupport pruning also changes the importance of complete pruning.

4.3 Summary of the techniques

We have presented many techniques that aim to reduce run-time or memory need. The following table summarizes our results. The tick in the second (third) column denotes that the technique is run-time (memory) safe. The sign S stands for the strict
4.4. THE INFLUENCE OF ITEM ORDERING

Figure 4.5: Hardware friendliness diagrams of Apriori and Apriori-Noprune

safety, i.e., for all databases the technique did not result in a slower (less memory-efficient) implementation. If no sign is found, then this technique has no influence on that measurement. For example routing strategies, when the edges are stored in an ordered vector do not have effect on memory need.

The fourth column stores the largest run-time drop. For example if the run-time of the base algorithm was 20 sec, and with the technique it dropped to 10 sec, then this value is 2. Therefore higher numbers here mean more efficient algorithms. If the technique resulted in a slower algorithm – for example the run-time increased to 30 sec – then the fifth column stores the largest performance degradation (2/3 in this case). The last two columns store the same indicators but for memory consumption.

4.4 The influence of item ordering

At the theoretical level we work with sets. In the implementations there exist no sets but vectors, lists, arrays, trees. Sets are converted to sequences using a total order on the items. The lexicographic order according to this order defines a total order on the itemsets. The order greatly affects the algorithms and the speed-up techniques. Till this point we carefully avoided this issue, but this subsection is dedicated to this topic.

4.4.1 The order-preserving assumption

In many FIM papers certain algorithms and speed-up techniques are explained with the independence assumption. Independence assumption states that if the frequencies of disjoint itemsets $I_1$ and $I_2$ are respectively $\text{freq}(I_1)$ and $\text{freq}(I_2)$, then the frequency of itemset $I_1 \cup I_2$ is (or at least close to) $\text{freq}(I_1) \cdot \text{freq}(I_2)$. This tries to encapsulate the independence of two binary random variables, but the probabilities are substituted by frequencies (relative supports). The assumption seems to contradict our original goal which is discovering unusual, unexpected, correlated patterns in the form of association
Table 4.1: Summary of the techniques

<table>
<thead>
<tr>
<th>technique</th>
<th>runtime need</th>
<th>largest runtime ratio</th>
<th>smallest runtime ratio</th>
<th>largest memory need</th>
<th>smallest memory need</th>
</tr>
</thead>
<tbody>
<tr>
<td>inhomogeneous trie with special block allocator</td>
<td>S</td>
<td>2.18</td>
<td>2.86</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dead-end pruning</td>
<td>S</td>
<td>4.24</td>
<td>2.56</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hybrid edge representation</td>
<td>✓</td>
<td>1.02</td>
<td>0.94</td>
<td>1.37</td>
<td>0.97</td>
</tr>
<tr>
<td>transaction caching</td>
<td>✓</td>
<td>0.94</td>
<td>0.17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>level 3 equisupport pruning</td>
<td>✓</td>
<td>105</td>
<td>0.77</td>
<td>42</td>
<td>0.88</td>
</tr>
<tr>
<td>prefix vs. level 3 equisupport pruning</td>
<td>S</td>
<td>3.35</td>
<td>1.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>intersection-based pruning</td>
<td>✓</td>
<td>6.12</td>
<td>0.98</td>
<td></td>
<td></td>
</tr>
<tr>
<td>omitting complete pruning</td>
<td>–</td>
<td>7.36</td>
<td>0.81</td>
<td>0</td>
<td>0.75</td>
</tr>
</tbody>
</table>

rules. If independence holds then the itemset that consists of the most frequent items would be largest itemset with the highest support. If we assume that item frequencies are \( \text{freq}(i_1) \geq \text{freq}(i_2) \geq \cdots \geq \text{freq}(i_k) \), then the size of the largest itemset would be \( k \) where \( \text{freq}(i_1) \text{freq}(i_2) \cdots \text{freq}(i_k) \geq \text{min}_\text{freq} \) but \( \text{freq}(i_1) \text{freq}(i_2) \cdots \text{freq}(i_{k+1}) < \text{min}_\text{freq} \). In general the number of frequent itemsets of size \( \ell \) would be \( |\{I = \{i_1, i_2, \ldots, i_\ell\} : \text{freq}(i_1) \text{freq}(i_2) \cdots \text{freq}(i_\ell) \geq \text{min}_\text{freq}\}| \).

We compared the distribution of frequent itemsets of real databases to their "independent version". The latter has the same item frequencies ad the original one, and the frequencies for larger sets are derived from the independence assumption (formula). The results of two randomly selected databases are seen in Figure 4.6.

We can see that reality is quite far form the assumption. We get similar consequences when we compare the number of frequent sets, the size of the largest frequent set, the average size of a frequent sets, etc.

When using a model we expect the consequences drawn from the model to be close to reality. It seems that almost all observable consequences that are drawn from the independence assumption have nothing to do with reality.

Does there exist a model that suits the characteristics of the frequent itemsets and at the same time it can be used to make further consequences?

Here we propose the following assumption.

**Definition 4.4.1** The order-preserving assumption requires that \( \text{sup}(X \cup Y) \leq \text{sup}(X \cup \text{sup}(Y)) \).
4.4. THE INFLUENCE OF ITEM ORDERING

Figure 4.6: Distribution of the size of the frequent itemsets and the distribution of frequent itemsets under independence assumption

\[ Z \text{ holds whenever } \sup(Y) \leq \sup(Z) \text{ for any disjoint sets } X, Y, Z. \]

We get an equivalent definition if support is substituted with frequency. The order-preserving assumption follows from the independence assumption, but not conversely. An immediate consequence of the independence assumption is that \( \sup(X \cup Y) = \sup(X \cup Z), \) if \( \sup(Y) = \sup(Z). \) If we want that the relative orders according to frequencies of two itemsets are not changed when adding certain items to both itemsets, then we have to modify slightly the definition.

**Definition 4.4.2** The soft order-preserving assumption requires that \( \sup(X \cup Y) \leq \sup(X \cup Z) \) holds whenever \( \sup(Y) < \sup(Z) \) for any disjoint sets \( X, Y, Z. \)

Some immediate consequences for later use are listed in the following.

**Corollary 3** Let \( I = \{i_1, i_2, \ldots, i_\ell\}. \) If \( \sup(i_1) \leq \sup(i_2) \leq \cdots \leq \sup(i_\ell), \) then \( \sup(\{i_1, i_2, \ldots, i_{\ell-1}\}) \leq \sup(I') \) for all \( I' \subset I, \) with \( |I'| = \ell - 1. \) Also, \( \sup(\{i_2, i_3, \ldots, i_\ell\}) \geq \sup(I'') \) for all \( I'' \subset I, |I''| = \ell - 1. \)

**Proof:** If \( I' = I \setminus \{i_j\}, \) then set \( X = I \setminus \{i_j, i_\ell\}, Y = \{i_j\} \) and \( Z = \{i_\ell\}. \) The order-preserving assumption gives the first claim. The second claim can be obtained similarly.

The soft order-preserving assumption version of the above corollary is the following.

**Corollary 4** Let \( I \) be a set of items of size \( \ell. \) If soft order-preserving assumption holds, then the subset of size \( \ell - 1 \) that consists of the most (least) frequent items, that has the largest (smallest) support among the subset of \( I \) of size \( \ell - 1. \)
The corollary claims, that the subset of $I$ that contains the most (least) frequent items has the largest (smallest) support among all the subsets of $I$ of the same size.

According to the following corollary (which gives an equivalent version of definition 4.4.1), the order-preserving assumption is hereditary to the projected databases, i.e., the ordering based on the supports of the items is equal to the ordering based on the supports of the items in projected databases.

**Corollary 5** Let $\mathcal{I}$ be a set of itemsets in which the order-preserving assumption holds. Then $\text{sup}_{\mathcal{I}\mid X}(Y) \leq \text{sup}_{\mathcal{I}\mid X}(Z)$ if and only if $\text{sup}_{\mathcal{I}}(Y) \leq \text{sup}_{\mathcal{I}}(Z)$ holds for any disjoint sets $X, Y, Z$.

**Proof:** Using the fact that the support of $X \cup Y$ in $\mathcal{I}$ equals to the support of $Y$ in $\mathcal{I}\mid X$ we get the claim, since the definition of order-preserving assumption can be rewritten such as: $\text{sup}_{\mathcal{I}\mid X}(Y) \leq \text{sup}_{\mathcal{I}\mid X}(Z)$ holds whenever $\text{sup}_{\mathcal{I}}(Y) < \text{sup}_{\mathcal{I}}(Z)$ for any disjoint sets $X, Y, Z$.

The property, however, does not hold in the complement of the projected database. This is proven by the following example. Let $\mathcal{I} = \{Y, XZ, XWZ\}$. It is easy to verify that the order-preserving assumption holds. Nevertheless $\text{sup}(Y) < \text{sup}(Z)$ while $\text{sup}_{\mathcal{I}\mid X}(Y) > \text{sup}_{\mathcal{I}\mid X}(Z)$.

The order-preserving assumption is quite rigid, and its validity is sensitive to noise, which is always present in real-world databases. If the probabilities of the occurrences of two itemsets are equal, then it is quite likely that in their support in a dataset will be close to each other but the chance of equality is small and converges to 0 as the number of transactions increases. This applies to all of their extensions with independent itemsets. Consequently, half of the extension does not fulfill the order preserving assumption. Here we propose a relaxation of our assumption.

**Definition 4.4.3** Let $0 \leq \alpha \leq 1$ be a given constant. The $\alpha$ order-preserving assumption requires that $\alpha \cdot \text{sup}(X \cup Y) \leq \text{sup}(X \cup Z)$ holds whenever $\text{sup}(Y) < \text{sup}(Z)$ for any disjoint sets $X, Y, Z$.

Obviously, if $\alpha = 1$, then we get the soft order-preserving assumption.

It is quite easy to verify the validity of the $\alpha$ order-preserving assumption in a set of itemsets $\mathcal{S}$, in which downward closure property holds, in a sequence of itemset $\mathcal{I}$. We check all different itemset pairs $I, I' \in \mathcal{S}$ if their intersection is nonempty. For such itemset pairs we calculate $I_1 = I \setminus I'$, $I_2 = I' \setminus I$. If the order of supports according to $I, I'$ differs from the order of support according to $I_1, I_2$ then the order-preserving assumption fails, otherwise holds. The *order-preserving ratio* is then given by the number of itemset pairs that result a positive check divided by the number of itemsets pairs considered (i.e., $I$ and $I'$ are not disjoint sets). The order-preserving ratio can similarly be calculated for the $\alpha$ order-preserving assumption. Table 4.2 contains the order preserving ratio of the frequent itemsets in our benchmark databases.
<table>
<thead>
<tr>
<th>database</th>
<th>minsup</th>
<th>1</th>
<th>0.95</th>
<th>0.9</th>
</tr>
</thead>
<tbody>
<tr>
<td>T40t10D100</td>
<td>900</td>
<td>0.912</td>
<td>0.998</td>
<td>0.999</td>
</tr>
<tr>
<td>kosarak</td>
<td>1800</td>
<td>0.817</td>
<td>0.980</td>
<td>0.998</td>
</tr>
<tr>
<td>T10t4D100K</td>
<td>8</td>
<td>0.690</td>
<td>0.693</td>
<td>0.726</td>
</tr>
<tr>
<td>connect</td>
<td>56000</td>
<td>0.725</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>pumsb</td>
<td>41000</td>
<td>0.863</td>
<td>0.994</td>
<td>1.000</td>
</tr>
<tr>
<td></td>
<td>38000</td>
<td>0.219</td>
<td>0.974</td>
<td>0.999</td>
</tr>
<tr>
<td>accidents</td>
<td>114000</td>
<td>0.882</td>
<td>0.960</td>
<td>0.988</td>
</tr>
<tr>
<td>retail</td>
<td>1</td>
<td>0.870</td>
<td>0.876</td>
<td>0.909</td>
</tr>
<tr>
<td>BMS-POS</td>
<td>400</td>
<td>0.809</td>
<td>0.860</td>
<td>0.901</td>
</tr>
<tr>
<td></td>
<td>350</td>
<td>0.116</td>
<td>0.354</td>
<td>0.544</td>
</tr>
<tr>
<td>BMS-WebView-1</td>
<td>37</td>
<td>0.857</td>
<td>0.942</td>
<td>0.984</td>
</tr>
<tr>
<td></td>
<td>36</td>
<td>0.351</td>
<td>0.802</td>
<td>0.961</td>
</tr>
<tr>
<td>BMS-WebView-2</td>
<td>30</td>
<td>0.790</td>
<td>0.819</td>
<td>0.853</td>
</tr>
<tr>
<td>webdocs</td>
<td>220000</td>
<td>0.877</td>
<td>0.966</td>
<td>0.990</td>
</tr>
<tr>
<td>mushroom</td>
<td>1600</td>
<td>0.910</td>
<td>0.955</td>
<td>0.990</td>
</tr>
<tr>
<td></td>
<td>900</td>
<td>0.868</td>
<td>0.896</td>
<td>0.913</td>
</tr>
<tr>
<td>T10t510K5K0</td>
<td>100</td>
<td>0.915</td>
<td>0.961</td>
<td>0.967</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0.790</td>
<td>0.809</td>
<td>0.819</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.714</td>
<td>0.729</td>
<td>0.739</td>
</tr>
<tr>
<td>T20t10N1K5K0</td>
<td>400</td>
<td>0.963</td>
<td>0.999</td>
<td>0.999</td>
</tr>
<tr>
<td>T30t15N1K5K0.25D200K</td>
<td>650</td>
<td>0.999</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>pumsb*</td>
<td>17000</td>
<td>0.850</td>
<td>0.963</td>
<td>0.986</td>
</tr>
<tr>
<td></td>
<td>15000</td>
<td>0.434</td>
<td>0.833</td>
<td>0.928</td>
</tr>
</tbody>
</table>

Table 4.2: The order-preserving ratio of the frequent itemsets
The figures show that the order-preserving assumption holds in most of the cases.
Now let us turn to the consequences of the order-preserving assumption that are quite valuable in frequent itemset mining.

4.4.2 The number of candidates

The number of candidates is independent of the ordering in the case of Apriori. In contrast, it depends on the prefixes — and thus on the ordering as well — in the case of Eclat, Fp-growth and Apriori-Noprun. The set of infrequent candidates is equal to the order based negative border of the frequent itemsets. An $\ell$-itemset is an element of the order-based negative border if it is infrequent and its $(\ell - 1)$-element prefix and the subsequent (with respect to the ordering) subset of the same size are frequent. The following lemma indicates which ordering results in the smallest order based negative border.

Lemma 4.4.4 If the order-preserving assumption holds, then the ascending order with respect to the supports results in the smallest order based negative border.

Proof: For each element $I = \{i_1, i_2, \ldots, i_\ell\}$ of the order-based negative border the proper prefixes of $I$ are frequent. Without loss of generality we can assume that $i_1 < i_2 < \cdots < i_\ell$. If $\sup(i_j) \leq \sup(i_{j+1})$ for all $j = 1, 2, \ldots, \ell - 1$ and the order-preserving assumption holds, then the corollary 3 states that $\sup(\{i_1, i_2, \ldots, i_{\ell-1}\}) \leq \sup(P')$ for all $P' \subseteq I$, where $|P'| = \ell - 1$. Itemset $\{i_1, i_2, \ldots, i_{\ell-1}\}$ is the prefix which is frequent and hence all proper subsets of $I$ are frequent. Consequently $NB^-(F) = NB(F)$ if $<$ denotes the ascending order according to frequencies. By Corollary 2 (see page 13) no other ordering results in smaller number of candidates, hence the lemma follows.

Corollary 6 If order-preserving assumption holds, then

$$NB(S) = NB^{<_{\text{ASC}}}(S),$$

where $S$ denotes a set of itemset, in which downward closure property holds, and $<_{\text{ASC}}$ denotes the ascending ordering according to the supports.

4.4.3 Size of the trie

Itemsets inserted into a trie are first converted to sequences based on an ordering. The ordering affects the shape and the number of nodes of the trie. This is illustrated by the tries depicted in Figure 4.7. Both tries stores sets $AB$ and $AC$. The first trie uses ordering $A < B < C$ the second uses the reverse.

For the sake of reducing the memory need which has strong correlation to the traversal times (see page 27), it would be useful to use the ordering that results in a trie with
minimal size. The size of the trie is given by the number of nodes. Comer and Sethi proved in [34] that the minimal trie problem, i.e., to determine the ordering which gives a minimal trie (denoted by $T_{OPT}$), is NP-complete. On the other hand, a simple heuristic (which was employed in FP-growth) performs very well in practice: use the descending order according to the frequencies. This is inspired by the fact that tries store any given prefix only once, and there is a higher chance of itemsets having the same prefixes if the more frequent items are closer to the root.

Different orderings may result isomorphic tries (and different orderings can result in a minimum-size trie). For example, try the store sets $A, B, AB, AC$ and use ordering $A < B < C$ and $A < C < A$ are isomorphic and minimal. Furthermore different orderings may result different, non-isomorphic minimal tries. This is shown in Figure 4.8.

![Diagram](image)

Figure 4.8: Example: minimal non-isomorphic tries

Note that we have to distinguish two frequencies of the items; frequency in the database, and frequency among the itemsets inserted into the trie. We call this latter frequency as *unweighted frequency*. Obviously the ordering based on the two frequency measures are not necessary equal. If the elements of the database are $AB, AC, AD, BC, RC, RC$ then $A$ is the most frequent according to unweighted frequency, but according to database frequency it is only in third place. Next we prove that under the order-preserving assumption the two orderings are equal.
Definition 4.4.5 Let $\mathcal{I}$ be a sequence of itemsets and denote by $\mathcal{I}^*$ the sequence that is obtained from $\mathcal{I}$ by keeping only the different elements (i.e., $\mathcal{I}^*$ contains the same itemsets as $\mathcal{I}$ but with multiplicity exactly one). The unweighted support of itemset $I$ in $\mathcal{I}$ equals to the support of $I$ in $\mathcal{I}^*$, i.e.

$$\text{uw-sup}_{\mathcal{I}}(I) = \text{support}_{\mathcal{I}^*}(I).$$

Lemma 4.4.6 Let $\mathcal{I}$ be a sequence of itemsets over $\mathcal{J}$. If order-preserving assumption holds, then the ordering with respect to the unweighted support equals to the ordering with respect to the support, i.e., if $\text{sup}(\{i_j\}) < \text{sup}(\{i_k\})$, then $\text{uw-sup}(\{i_j\}) \leq \text{uw-sup}(\{i_k\})$.

**Proof:** We prove the statement by contradiction. Let us assume that $\text{sup}(i) > \text{sup}(i')$ but $\text{uw-sup}(i) < \text{uw-sup}(i')$. Let us denote the elements of the cover of $i'$ in $\mathcal{I}^*$ by $t'_1, t'_2, \ldots, t'_n$ ($i' \in t_j, t_j \in \mathcal{I}^*$, $t'_k \neq t'_l$). According to the order-preserving assumption $\text{sup}(\{t \setminus i' \cup i\}) \geq \text{sup}(t)$ for all $t \in \text{cover}_{\mathcal{I}^*}(i')$. This is a contradiction, because the number of items in $\mathcal{T}$ is smaller or equal than $\text{uw-sup}(i')$, they are different and all contain $i$, therefore the size of $\text{uw-sup}(i)$ cannot be less than $\text{uw-sup}(i')$. \qed

The heuristic (use the descending order on the frequencies) does not always result in a minimal trie. This is illustrated by the following example. Let us store itemsets $AB, AC, ZB, ZC, ZBC, Z$ in a trie. A trie that uses descending order according to frequencies ($\text{sup}(Z) = 4, \text{sup}(B) = \text{sup}(C) = 3, \text{sup}(A) = 2$) is depicted on the left side of Figure 4.9. On the right side we can see a trie that uses an other order (items $A$ is moved right behind item $Z$) and has fewer nodes.

![Figure 4.9: Example: descending order does not result in the smallest trie](image)

The failure of the descending order producing the minimum size trie stems from the fact that the order-preserving assumption does not hold. Note that in the example $\text{sup}(Z) > \text{sup}(A)$, but $\text{sup}(ZB) < \text{sup}(AB)$.

Conjecture 4.4.7 Let $\mathcal{I}$ be a set of itemsets and denote $\ll_{\text{DESC}}$ the descending order of items according to the number of occurrences of the items in $\mathcal{I}$. If order-preserving
assumption holds then \( T^{DESC}(\mathcal{J}) \) is the minimum-size trie among the tries that store \( \mathcal{J} \), i.e., there exists no ordering \( \prec \) such that \( T^{\prec}(\mathcal{J}) \) has fewer nodes than \( T^{DESC}(\mathcal{J}) \).

If the conjecture holds, then we know that the heuristic (also employed by \textit{FP-growth}) works fine under ideal circumstances (the order-preserving assumption holds for all sets). Table 4.2 shows that the real-world is "close" to the ideal, but still slightly different. One of the most valuable knowledge of frequent itemset mining would be a formula about the relationship of the \( \alpha \) order-preserving ratio of a set of itemset \( \mathcal{J} \) and the ratio of \( |T^{DESC}(\mathcal{J})| \) and \( |T_{GFT}(\mathcal{J})| \), where \( T_{GFT} \) denotes a minimum-size trie.

### 4.4.4 Support count

One may tend to follow the observation, that a smaller memory need results in better data locality and hence faster algorithms. Therefore, we should use the descending order according to the frequencies when building the candidate trie. This is, however, just one side of the problem.

To understand the other side we have to recall the support count procedure. To decide which candidates are contained in a given transaction, a part of the trie has to be traversed. Each path of the traversals starts from the root. Some paths reach a leaf, others do not. The number and the length of paths that reach a leaf is independent of the ordering. This, however, does not apply to the length of the remaining paths. To reduce the expected number of unnecessarily visited nodes, first we have to check if the transaction contains the least frequent item since this is likely to provide the strongest filtering among the items of the candidate, i.e., this is the item that is contained in the least amount of transactions. Next, the second least frequent is advised to check, then the third, and so on. The edges are checked from the root to the leaves, hence we expect the least amount of redundant checks and thus the best run-time, if the order of items corresponds to the ascending order according to the supports.

![Figure 4.10: Example: Tries with different orders](image-url)

To illustrate this rationale, let us consider the tries at Figure 4.10. The two tries store the same sets, but one in the left uses the descending order \((A \prec B \prec C \prec D)\) and the other the ascending order according to unweighted frequencies. When determining the candidates in transaction \(\{A, B, E, F\}\). Nodes 0, 1 and 2 are visited if descending
order is used, while the search is terminated immediately at the root in the case of the ascending order.

We know that transaction caching using a trie or a patricia tree requires descending order according to the frequencies in order to be storage efficient. In contrast, the minimal number of redundant steps in the candidate trie during the support count prefers ascending order. These two requirements can be satisfied at the same time by a little trick. The items are recoded according to ascending order according to the supports, but the items are stored descending in each transaction when inserting into the cachter. Since the efficient support count (i.e., \texttt{merge}) requires the items of the transaction to be stored ascendingly, we simply reverse each transaction when it is retrieved from the cachter.

In summary, descending order is good for the compactness (and does not require to reverse the transactions before being processed), while ascending order results in fewer redundant steps in the trie. Experiments show also that there is no absolute winner: most of the times the ascending order results in the faster algorithm, sometimes the descending order. For some results pertaining to this dichotomy, see Figure 4.11. Values less than one mean that the Apriori that uses descending order according to the frequencies is the faster.

Some hardware friendliness diagrams are given in Figure 4.12.

In all experiments the transaction caching does not changes which ordering results in the first place. This is attributed to the fact that we used low support threshold. In such cases the memory need of a transaction cachter and the run-time of building it are insignificant comparing to the memory need of the candidate trie and the run-time of support count. Different ordering may be a better choice if we raise the support threshold.

4.4.5 Pruning efficiency

There is a strong connection between the ordering and efficiency of the Apriori that does not use complete pruning. We want to use the ordering that minimizes the number of false candidates. Candidates in Apriori-Noprun are the same as candidates in Eclat or FP-growth (see section 4.4.2) therefore the fastest Apriori-noprun is expected when ascending order according to the frequencies is used. Experiments support this conclusion.

Table 4.3 shows the ratio of the number infrequent candidates and the number of frequent itemsets in the case of complete pruning, Apriori-Noprun with ascending and descending order according to the supports.

We can see that the number of infrequent candidates is much larger when the descending order is used (check the values in the last column).

It follows from the rational that Apriori differs from Apriori-Noprun in terms of sensitivity of the ordering. Both orderings has their advantage in Apriori, but in Apriori-
4.4. THE INFLUENCE OF ITEM ORDERING

![Graph 1](image1.png)

**Figure 4.11:** Ascending vs. Descending order according to the frequencies

![Graph 2](image2.png)

**Figure 4.12:** Hardware friendliness diagrams of Aprioris with ascending and descending order according to the frequencies
4.5 The battle of Apriori implementations

We have enrolled our three selected implementations (Apriori, Apriori—Noprun e and Apriori—MEMSAFE) in a competition with three known Apriori codes. Apriori—MEMSAFE employs on-line candidate 2-itemset generation [44] and does not use transaction caching. Apriori—Noprun e omits the complete pruning phase. Apriori and Apriori—MEMSAFE adapt the intersection-based candidate generation. Apriori—Noprun e and Apriori use transaction caching and apply a diagonal array for determining the supports of candidates of size two. All three implementations use inhomogeneous trie with the special block allocator, dead-end branch pruning, hybrid edge representation and full equisupport pruning.

We compared our implementation to three C/C++ implementations coded by Christian Borgelt\(^1\), Bart Goethals\(^2\) and Tingshao Zhu\(^3\) respectively. This later was finally excluded from the race, because it ran extremely slow, several orders of magnitude slower than the others. We used the latest versions that are available on the authors’ website at 15th December 2005.

We have tested two implementations from Christian Borgelt, the one that was submitted to FIMI’04 (Apriori—Borgelt—FIMI) and other that can be downloaded from the webpage. We ran this implementation with two different parameters, in order to test the

\(^1\)http://fuzzy.cs.uni-magdeburg.de/~borgelt/apriori.html
\(^2\)http://www.adrem.us.ac.be/~goethals/software/
\(^3\)http://www.cs.ualberta.ca/~tszhu/software.html
4.5. THE BATTLE OF APRIORI IMPLEMENTATIONS

speed- and memory-optimized version respectively (denoted by Apriori-Borgelt-Speed and Apriori-Borgelt-Mem respectively). Apriori-Borgelt-Speed always consumed the same amount of memory as Apriori-Borgelt-FIMI but sometimes ran slower.

In the memory optimized version hybrid edge representation is used and transactions are not stored in the memory. The speed-optimized version uses a trie with offset-index edge representation, and a trie storing the transactions. It adapts a novel support counting method, (the basis of which was described in section 3.9.1) together with the simple unimportant item filtering technique (see section 3.9.2).

Due to the space restrictions we show only a small number of test results. We uploaded all results to the page http://www.cs.bme.hu/~boden/fim/test.html. Three typical run-time plots are depicted in Figure 4.13.

![Figure 4.13: Battle of the Apriori implementations, run-times](image)

Goethals’ implementation is not competitive in speed with the other Apriori implementations. Concerning just the lowest support threshold, Apriori-Borgelt-FIMI finished in the first place in 5 cases and our Apriori in 11 cases. The following figure shows the advantage of Apriori over Apriori-Borgelt-FIMI. Values greater than one means that Apriori was faster than Apriori-Borgelt-FIMI.

![Figure 4.14: Borgelt vs. Bodon (run-times)](image)
The highest advantage of Apriori-Borgelt-FIMI is at database T10I5N1KP5KC0-25D200K where it is two times faster than Apriori. On the contrary, our Apriori often outperformed Apriori-Borgelt-FIMI with an order of magnitude, and in several cases Apriori-Borgelt-FIMI could not even cope with the task.

Concerning memory-optimized versions, our implementation outperformed Borgelt’s implementation in run-time in 13 cases.

The advantage of our solution is quite clear if we take a look at the memory need. Our implementations consumed only a fraction of the memory need of Borgelt’s implementation. This applies to all databases at all support thresholds. Three typical memory-need plots are in Figure 4.15.

![Figure 4.15: Battle of the Apriori implementations, memory need](image)

The comparison of the two main rivals, i.e., Apriori and Apriori-Borgelt-FIMI is found in Fig. 4.16.

![Figure 4.16: Borgelt vs. Bodon (memory need)](image)

In summary, our code results in the fastest Apriori implementation in most of the cases, and its memory requirement is outstanding in the field.
Chapter 5

A general model of frequent pattern mining

Up to this point we have been absorbed in frequent itemset mining, more specifically in Apriori-based FIM. Frequent patterns are not only searched in forms of itemsets, but frequently occurring sequences, rooted trees, directed and undirected graphs play also an important role in many application domains.

Algorithms and techniques used in frequent itemset mining often turn out to be useful when searching for other types of patterns. Unfortunately, different patterns are examined by different research groups, and the interaction and cooperation between them is often not satisfactory. This is the reason that the same techniques are often reinvented and labeled by different names.

In this section we propose a general framework for frequent pattern mining and raise some definition and algorithm to this level.

5.1 Frequent Pattern Mining

Before we give the precise definition let us clear some definition of the poset theory.

A partial order (often simply referred to as an order or ordering) is a relation \( \leq \subseteq A \times A \) that satisfies the following three properties:

1. Reflexivity: \( a \leq a \) for all \( a \in A \)
2. Antisymmetry: If \( a \leq b \) and \( b \leq a \) for any \( a, b \in A \), then \( a = b \)
3. Transitivity: If \( a \leq b \) and \( b \leq c \) for any \( a, b, c \in A \), then \( a \leq c \)

In the rest of the paper we say \( a \) is smaller than \( b \) (\( b \) is larger than \( a \)) if \( a \leq b \), and it is strictly smaller if \( a \neq b \). A set and a partial order on that set define a Poset, that is, a poset is a pair \( (P, \leq) \) where \( \leq \) is a partial order relation on set \( P \). We say \( z \in P \) is a lower bound of \( P' \) if \( z \leq x \) for all \( x \in P' \).
The epithet proper is added if we exclude equality from the conditions used in the definitions above. Also, we write \( a < b \) if \( a \leq b \) and \( a \neq b \).

**Definition 5.1.1** A poset \((P, \preceq)\) is locally finite if every interval \([x, y]\) in \(P\) is finite, i.e., there exists a finite number of \(z\) such that \(x \preceq z \preceq y\).

**Definition 5.1.2** A poset \((P, \preceq)\) is graded if there exists a size function \(\|\| : P \to \mathbb{Z}\), such that \(|b| = |a| + 1\), where \(a, b \in P\) and \(a\) is a proper lower bound of \(b\).

Let us define the most important component of frequent pattern mining.

**Definition 5.1.3** The poset \(PC = (P, \preceq)\) is called a pattern context if (1) it is locally finite, (2) graded and (3) there exists exactly one minimal element of \(P\). The elements of \(P\) are called patterns.

The first element of the pattern context is the pattern space. We say \(p'\) is a subpattern of \(p\) and \(p\) is a superpattern of \(p'\) if \(p' \preceq p\). The minimal element of a pattern space is called the empty pattern and denoted by \(\emptyset\). Without loss of generality we assume that \(|\emptyset| = 0\). A pattern of size \(\ell\) is also called an \(\ell\)-element pattern.

Let us summarize the pattern context that are presented in the literature.

**itemset** \((PC_{\text{SET}})\): The pattern context is \(PC_{\text{SET}} = (2^\mathcal{J}, \subseteq)\), where \(\mathcal{J}\) is a given set and \(\subseteq\) denotes the traditional subset relation for sets.

**itemsequence** \((PC_{\text{SEQ}})\): Let \(\mathcal{J}\) be a set of items. The patterns are sequences over \(\mathcal{J}\). The sequence \(S = \langle i_1, i_2, \ldots, i_n \rangle\) is a subsequence of \(S' = \langle i'_1, i'_2, \ldots, i'_m \rangle\) if there exist \(1 \leq j_1 < j_2 < \cdots < j_n \leq m\) such that \(i_1 = i'_{j_1}, i_2 = i'_{j_2}, \ldots, i_n = i'_{j_n}\).

**sequence of itemset** \((PC_{\text{SEQ.SET}})\): Let \(\mathcal{J}\) be a set of items. The patterns are sequences over \(2^\mathcal{J}\). The sequence \(S = \langle I_1, I_2, \ldots, I_n \rangle\) is a subsequence of \(S' = \langle I'_1, I'_2, \ldots, I'_m \rangle\) if there exist \(1 \leq j_1 < j_2 < \cdots < j_n \leq m\) such that \(I_1 \subseteq I'_{j_1}, I_2 \subseteq I'_{j_2}, \ldots, I_n \subseteq I'_{j_n}\).

**directed, labeled ordered/unordered tree** \((PC_{\text{TREE}})\): An itemsequence can be represented by a vertex labeled, directed path. Similarly an itemset is a graph that has no edges. A natural generalization of the path is the tree. If the edges are also labeled then ordered and unordered trees are distinguished. In ordered trees the edges of an inner node must be ordered by labels. The purpose of ordering is to avoid distinguishing similar trees. Please note, that this does not mean the pattern space does not contain isomorphic trees. The trees depicted in Figure 5.1 are isomorphic, but are different elements of the pattern space.
directed, acyclic graphs ($PC_{DAG}$): The natural extension of the directed trees are the directed, acyclic graphs (DAG). In some application domains like telecommunication network or gene mining the patterns are called *episodes*. Pattern graph $G$ is a subpattern of pattern graph $G'$ if $G$ is isomorphic to a subgraph of $G'$.

labeled graph ($PC_{GRAPH}$): In chemical databases the frequent patterns are searched for in a form of labeled graphs [58] [36] [62].

A tree/dag/graph $g'$ is a subpattern of tree/dag/graph $g$ if $g'$ is the subgraph of $g$.

Boolean formulas ($PC_{CNF}$): To analyze telecommunication networks such conditions are searched that often hold. Conditions are given by Boolean formulas of conjunctive normal form over event variables. For the formal definition please see section 6.4.

**Definition 5.1.4** Let $(P, \preceq)$ and $(P', \preceq')$ be two posets. The function $f : P \rightarrow P'$ is anti-monotone, if $f(b) \preceq' f(a)$ for any $a \preceq b$.

The definition of the frequent pattern mining (FFM) is the following.

**Definition 5.1.5** Given $\mathcal{I}$ input data, $(P, \preceq)$ pattern context, $\text{supp}_{\mathcal{I}} : P \rightarrow \mathbb{N}$ anti-monotone function, which is called support function and $\text{minsup} \in \mathbb{N}$ constant, find the following set of patterns

$$F = \{ p : p \in P, \text{supp}_{\mathcal{I}}(p) \geq \text{minsup} \}.$$

The min-up constant is called support threshold and the elements of $F$ are the frequent patterns. For the sake of simplicity the index $\mathcal{I}$ is often omitted from the support function and $\text{supp}(p)$ is called the support of $p$.

The definition of the frequent pattern mining is quite general, and all frequent pattern mining problems that were defined in the literature are specializations of this definition. The most important family of FFM is the frequent pattern mining in a transactional database. Here, the input data is given as a sequence of transactions and the support
function is defined by a match predicate. We say, that pattern $p$ is contained by $t$, or $p$ occurs in $t$ if the match predicate returns true. The support count of a pattern equals to the number of transactions that contain $p$. The anti-monotonicity of support count requires that the match predicate is an anti-monotone predicate.

The following property follows from the anti-monotonicity of the support function.

**Property 5** All subpatterns of a frequent pattern are frequent.

A pattern is always a structure of a basic itemset like sequences, trees, graphs. If we define a total order on the basic set, then generalize this ordering to the patterns themselves in an easy (e.g., lexicographic ordering) or a sophisticated (e.g., canonical form at graphs) way. A total order on the patterns is a requirement of some algorithm (like Eclat or FP-growth) and at some algorithm (like Apriori) it is guarantee of the effectiveness. The notion of prefix is also often used. To give the definition of prefix we also have to be able to define a total order on the patterns.

**Definition 5.1.6** Let $(P, \preceq)$ be a poset. The total order $\prec'$ is called a linear extension of $\preceq$ if $a \preceq' b$ for all $a \preceq b$.

Not all linear extensions are suitable for the FPM algorithms. We restrict our attention to the size preserving linear extension, i.e., if $|a| < |b|$ holds then $a \prec' b$ must also hold. Therefore when we give a linear extension of $\preceq'$ of $(P, \preceq)$ pattern context, then we actually define a total ordering among the elements of the same size. By default all linear extensions are understood as size preserving linear extension in the rest of the work.

**Definition 5.1.7** Let $(P, \preceq)$ pattern context and $\prec'$ a linear extension of $\preceq$. The prefix of size $\ell$ of $p$ (denoted by $P_\ell(p)$) is the smallest according to $\preceq'$ among the $\ell$-element subpatterns of $p$

$$P_\ell(p) = p' \iff p' \subseteq p, |p'| = \ell, \exists p'' \subseteq p \text{ such that } p'' \prec' p' \text{ and } |p''| = \ell.$$

**Definition 5.1.8** A frequent pattern $p$ is maximal if there exist no other frequent pattern $p'$ such that $p < p'$.

**Definition 5.1.9** A pattern $p$ is closed if there exists no other pattern $p'$ with $\sup(p) = \sup(p')$ and $p < p'$.

Closed patterns are interesting because of two reasons. First, the support function is anti-monotonic, therefore we expect that any superpattern of $p$ has smaller support than $p$, and equality of support brings interesting insights in many applications. Second, frequent closed patterns turned out to be a compact representation of the frequent patterns in the case of itemsets. We state that this property does not apply to other types of patterns.
5.2. DESCRIBING SOME ALGORITHMS AT THE GENERAL LEVEL

Definition 5.1.10 Pattern \( p' \) is a closure of pattern \( p \), if \( \text{supp}(p') = \text{supp}(p) \) and there exist no other \( p'' \) such that \( p' \prec p'' \) and \( \text{supp}(p'') = \text{supp}(p) \).

Definition 5.1.11 The pattern context \( (P, \preceq) \) is unique with respect to closure, if the closure of any pattern is unique.

Lemma 5.1.12 \( PC_{\text{SET}} \) is unique with respect to closure.

Proof: We prove the statement by contradiction. Let us assume that there are two closures of itemset \( I \), which are denoted by \( I' \) and \( I'' \). From the definition of the closure it follows that \( I' \not\subseteq I'' \) and \( I'' \not\subseteq I' \). Based on the definition of support function and the closure we get that transactions that contain itemset \( I \) also contain itemset \( I' \) and \( I'' \) and consequently itemset \( I' \cup I'' \). Since \( I' \) and \( I'' \) are not identical, their union is a proper superset of both \( I' \) and \( I'' \) and it has the same support as \( I \). This contradicts to the definition of the closure, i.e., \( I' \) and \( I'' \) cannot be closed. \( \Box \)

Lemma 5.1.13 \( PC_{\text{SEQ}} \) is not unique with respect to the closure.

Proof: We prove this by a contradictory example. In database \( \mathcal{I} = \{\langle A, B \rangle, \langle B, A \rangle\} \) the following holds: \( 2 = \text{supp}(\langle \rangle) = \text{supp}(\langle A \rangle) = \text{supp}(\langle B \rangle) \) and no proper superpattern of \( \langle A \rangle \) and \( \langle B \rangle \) has support 2. Therefore the empty pattern has two closures. \( \Box \)

Since \( PC_{\text{SEQ-SET}}, PC_{\text{TREE}}, PC_{\text{DAG}}, PC_{\text{GRAPH}} \) are the generalizations of \( PC_{\text{SEQ}} \), we obtain:

Corollary 7 \( PC_{\text{SEQ-SET}}, PC_{\text{TREE}}, PC_{\text{DAG}}, PC_{\text{GRAPH}} \) are not unique pattern contexts with respect to closure.

5.2 Describing some algorithms at the general level

In this section we show some examples how we can describe certain algorithms in our environment. This has two major advantages. First, some algorithms and techniques presented at a special type of pattern become applicable at other types as well. Second, we can identify the equivalent (with respect to the functionality) parts of different algorithm. Based on this we can clear some theoretical background, place the technique in a unified framework and resolve naming inconsistencies.

5.2.1 Toivonen technique

The input data in frequent pattern mining is very large in many cases. This leads to a long processing time, which can be reduced by sampling. In the simple sampling
algorithm we take a sample of the input data, run an FPM algorithm to obtain the
frequent patterns in the sample (locally frequent items). Finally, the precise support of
the locally frequent items are determined in the original input.

This algorithm is not complete; some frequent patterns may be locally infrequent.
One may decrease the support count in order to reduce the chance biased sampling, this
however may lead to a computational blow-up. Notice for example, that database BMS-
POS can be processed at support threshold 33, but that does not hold for minsup = 32.
This is not attributed to the large database size, but the extreme growth of the number
of frequent patterns at the lower support threshold.

A better solution was proposed by Toivonen [105]. We present its description in our
FPM context. The algorithm is a small modification of the simple sampling algorithm.
Its advantage is that it reports if the FPM was complete, and also if it is possible that
some patterns are missed.

The difference from simple sampling algorithm is that it not only determines the
support of the locally frequent patterns, but also the minimal proper upper bound of
them. A pattern is the element of the minimal proper upper bound if it is locally
infrequent, but all its proper subpatterns are locally frequent. Formally:

\[ MPUB(F) = \{ p : p \not\in F, \forall p' \prec p, p' \in F \} , \]

where \( F \) denotes a set of patterns that posses the download closure property. The
minimal proper upper bound is also called negative border in many applications.

The following lemma provides a basis for the report.

**Lemma 5.2.1** Let \( D \) be an input database and \( D' \) a sample of it. The frequent patterns
in \( D \) and \( D' \) are denoted by \( F \) and \( F' \) respectively. The set of patterns that are frequent
and are also in \( F' \cup MPUB(F') \) is denoted by \( F^* \), i.e., \( F^* = F \cap (F' \cup MPUB(F')) \). If

\[ F^* \cup MPUB(F^*) \subseteq F' \cup MPUB(F') , \quad (5.1) \]

then the set of frequent patterns equals to \( F^* \).

**Proof:** We prove the statement by contradiction. Let us assume there exists \( f \in F \),
but \( f \not\in F' \) and the equation (5.1) holds. Because of the definition of \( F^* \), \( f \) is not
element of \( (F' \cup MPUB(F')) \). Let us examine the smallest \( f' \preceq f \), such that \( f' \in F \)
but \( f' \not\in F^* \). Such \( f' \) must exist, because \( f' \) may be identical to \( f \). From the minimal
property of \( f' \) it follows that all proper subsets of \( f' \) are in \( F' \cup MPUB(F') \). Then all
proper subsets of \( f' \) are element of \( F^* \) and \( f' \in MPUB(F^*) \). This is a contradiction,
because there is an pattern \( (f') \) that is element of the left side of equation (5.1) but not
element of the right side. \( \Box \)

After the algorithm determines \( F^* \) it checks the condition. If it holds, then it reports
that the algorithm was complete, it found all frequent patterns.
5.2. DESCRIBING SOME ALGORITHMS AT THE GENERAL LEVEL

Toivonen algorithm was presented to mine frequent itemsets. Thanks to this more general description of the algorithm it is possible to apply it to other types of patterns like trees or graphs.

The locally frequent itemsets are proposed to be found by Apriori algorithm. In this case \( MPUB(F') \) equals to the set of infrequent candidates, therefore determining them needs no extra work. It easy to see that if Apriori is replaced by Eclat or FP-growth and \( MPUB(F') \) is the set of infrequent candidates (i.e., the order-based negative border of the frequent patterns), then the lemma still holds. Consequently, we extended the applicability of Toivonen method, and also relaxed its relation with the algorithm Apriori.

5.2.2 Common description of Eclat and FP-growth

Eclat, which was proposed by Zaki et al. [115], was the first algorithm that did not follow the breadth-first iterative approach, used in Apriori and its relatives. It employed a recursive, depth-first like, database projection-based search. Three years later FP-growth was proposed by Han et al. [51]. The authors declared the algorithms as a novel solution. This was accepted by many researchers and authors of different data mining textbooks. Here we show that Eclat and FP-growth are equivalent with respect to the functionality of the main methods. The only difference is the data structure used to achieve efficiency.

5.2.2.1 Depth-first-like search

The pattern space can be represented by a tree, each node representing a pattern. The parent of node \( p \) represents the prefix of \( p \). The pattern space traversal method reflects the method algorithms traverse this tree in order to find the frequent patterns. Apriori performs a breadth-first traversal, i.e., first it determines the frequent patterns of size 1, then the frequent patterns of size 2, and so on. A pure depth-first method would go down till it finds an infrequent pattern then it would backtrack and move further down till it bumps into an other infrequent pattern again. On the contrary, a depth-first-like algorithm, like Eclat and FP-growth, determines all frequent children of a node, then takes the children one-by-one and calls a recursive search.

Example 9

Let us assume that in \( PCSET \) the only maximal itemset is \{ABC\}. The following tables show the order different approaches output the results.

In breadth-first traversal the algorithms can do full pruning, i.e., it can check all subpatterns of a candidate if they are frequent. Depth-first algorithm can only check if the prefix is frequent, while depth-first-like algorithms the prefix and the subsequent subpattern are frequent.
5.2.2.2 Database projection

When determining the frequent patterns of a subtree with root (prefix) \( p \), then it is enough to examine the transactions that contain \( p \). A transaction that does not contain \( p \) does not contain any superpattern of \( p \) either.

5.2.2.3 Depth-first-like, recursive algorithms

Here we give the scheme of depth-first-like, recursive, projection-based algorithms (DFL-RC).

**Algorithm 5.2.1** algorithm scheme DFL-RC

**Require:** \( D \): a database,  
    \texttt{minsup} : support threshold.

**Ensure:** \( F \): the set of frequent patterns

\[
F^+_\emptyset \leftarrow \text{determine frequent pattern of size one}(D, \texttt{minsup}) \\
\text{return DFL-RC-MAIN}(\emptyset, D, \texttt{minsup}, F^+_\emptyset)
\]

In the \texttt{determine frequent patterns of size plus one} method we determine the frequent patterns with prefix \( p \) of size \(|p| + 1\). For this the algorithms generate candidates and only the support of the candidates are calculated. In both approach the candidates equal to the order-based negative border of those frequent patterns that have prefix \( p \). A pattern \( p \) is an element of the order-based negative border of a set of pattern \( F \), in which the downward closure property holds, if \( p \) is not in \( F \), but its prefix and the subsequent subpattern (both of size \(|p| - 1\) are in \( F \).

The above description is just a pseudo-code scheme, it does not contain the precise steps of the two major functions, i.e., (1) determining the frequent patterns of size \( \ell \) with prefix \( p \) if all frequent set of size \( \ell - 1 \) with prefix \( P_{\ell-1}(p) \) are given, (2) determining the projected database. If we examine Fclat and FP-growth and their extensions to mine frequent sequences and trees we conclude that the algorithms suit to the above schemes. Using software engineering terminology, we would say that we gave the functionality and the interfaces of the main methods.
Algorithm 5.2.2 algorithm scheme DFL-RC-Main

Require: \( P \): the prefix pattern.
\( D_P \): a database projected to \( P \),
\( \text{minsup} \): support threshold,
\( F^+_P \): the set of frequent pattern with prefix \( P \) of size \(|P| + 1\)

Ensure: \( F_P \): the set of frequent patterns with prefix \( P \)

\[
F_P \leftarrow F^+_P \\
\text{for all } p \in F^+_P \text{ do} \\
F^+_P \leftarrow \text{determine frequent patterns of size plus } 1(p, D_P, F^+_P) \\
\text{if } |F^+_P| > 1 \text{ then} \\
D_p \leftarrow \text{project database}(D_P, p) \\
F_P \leftarrow F_P \cup \text{DFL-RC}(p, D_p, \text{minsup}, F^+_P) \\
\text{else if } |F^+_P| = 1 \text{ then} \\
F_P \leftarrow F_P \cup F^+_P \\
\text{return } F_P
\]

The difference between Eclat and FP-growth is how they solve the main tasks, which is determined by the data structure used to store the database. In Eclat the database is represented by the so called TID-lists. The TID-list possess two important properties. First, the support of a pattern can be determined quickly from its TID-list. Second, the TID-list of a candidate can be calculated from the TID-lists of the generators. For example in the pattern context \( PC_{SET} \) the TID list of itemset \( I \) equals to the list of indices of the transactions that contains \( I \) as a subset. See [111] for the definition of TID-list in the case of \( PC_{TREE} \).

FP-growth uses a different approach. It stores the projected transactions in a special tree structure. In \( PC_{SET} \) this tree is a prefix-tree extended by cross links and a header table. The projected database is filtered before it is inserted into the tree. For example in the pattern context \( PC_{SET} \), filtering database \( D_P \) means removing infrequent items from the projected database \( D_P \). Determining the support of itemsets of size \(|P| + 1\) means determining the support of items in the filtered \( D_P \). We omit the details, because our goal was to emphasize the functional equality of FP-growth and Eclat.

5.3 Conclusion

Due to the diversity of frequent pattern mining it is quite important to find the common roots of the techniques and notions. When presenting a new technique or a new concept it is quite important to know its scope, i.e., we would like to know how general this solution is. To this end a unified framework is required that is accepted and used by the FPM community. In this chapter we presented such a framework and showed how
can we clear the theoretical background of some known and wide-spread technique by using our framework.
Chapter 6

Filtering False Alarms

6.1 Introduction

Nowadays it is essential and a basic need to connect the computer network of a company to the World Wide Web. The original purpose of the Internet was to support educational institution with a decentralized network, where the effectiveness (speed, and reliability) was relevant and security was less important. The widespread use of WWW opened the gate for new users and new application domains but also posed new demands and stressed that part of the system that received only modest attention in the beginning. Unfortunately, additional solutions bring up more problem than those that were taken into consideration in the planning phase. This may be the main reason for the security of Internet being a hot topic in the scientific community.

A wide variety of security devices (virus checkers, firewalls, coding and policy methods, Intrusion Detection Systems, etc.) are available on the market, but each of them just attempts to fill in a security gap. They provide partial solutions and, as they communicate with each other in a limited way, their capabilities are limited. A centralized system that watches every part of the system could collect more data, could be more efficient in the chase of intrusions than a standalone system. We call such system a Remote Supervision System.

In the Remote Supervision System data coming from different security devices placed at different points of the network are collected in the center. It is similar to a traditional security guard who is sitting in front of a monitor wall and can see the monitors of the cameras and the signals of all the protection systems at the same time. Theoretically, by increasing the number of the protection systems (and hence the available information) incidents can be handled more effectively. On the other hand we have to face the problem of handling the huge amount of data.

The sensitivity of the typical network monitoring security devices can be set within a wide range. If the parameters are set to high levels, then a device reports an alarm at
every event that is a little bit suspicious. In fact, most (but not all!) of the "little bit suspicious" events are harmless, which is the reason for setting the sensitivity parameters to lower values in the practice. In general overloaded system administrators have capacity only to analyze the dangerous and critical events, however for analyzing and tracing them back, they also need to know the preceding events as well. If the sensitivity is low, then real attacks could be ignored, because none of the security devices find them suspicious enough. In the other extreme (high sensitivity) we have to cope with the huge amount of data and the large number of false alarms (an alarm is false if no real attack can be associated with it). Data mining gives us a helping hand in analyzing big volumes of data to discover frequent rules of false alarms.

In our approach we collect as much information as we can. By using episode mining algorithms frequent patterns that precede an alarm can be analyzed. This makes it possible to automatically discover the reason of frequent false alarms. Our goal is to develop a method that can infer the hidden patterns from the central database. If we can match a known pattern of false alarm to the event sequence preceding an alarm, then we degrade this alarm to false alarm. Of course before accepting a rule for false alarms, the approval of a professional person is needed. This is necessary for adequate human control.

In this thesis the architecture of the system and the technical details are not discussed, we are focusing on the mathematical model and the episode mining algorithm [17, 19, 18]. For further details on the overall system the reader should consult [59].

6.2 Related Work

We shortly review the known episode mining algorithms. For a more comprehensive review the reader is referred to [37]. The first published algorithm that could cope with large datasets of event sequences was APRICRIALL [6]. It introduced the notion of frequent sequential pattern as a generalization of frequent itemsets known from the association rule mining field. Episodes were defined as sequences of itemsets, and the algorithm found those episodes that were contained in many (more than a given support threshold) sequences. The algorithms GSP [101] and SPADE [113] solve the same problem much faster (in addition, they can handle time constraints).

Another algorithm that finds frequently occurring serial and parallel episodes in one given sequence was presented in [71]. Similarly to our approach, it uses fixed size windows to define the containment relation. In its model the events are atomic, hence its method is not adaptable to the context, where events are determined by parameters.

From our point of view the most promising episode mining algorithm that can handle events with attributes was presented in [53, 68]. We mainly adapted the approach of [68] in our mathematical model. However, we are not looking for episodes and their minimal occurrences, but rather for episodes, which occurred in windows that often ended by
alarm. Furthermore, we allow the building blocks of episodes (i.e., the predicates) to be more general by letting them parametric.

The purpose of the Remote Supervision System project was to study the adaptability of data mining techniques to filter false alarms coming from different security devices. Our final goal was to implement a prototype system that proves our hypothesis that data mining is a powerful tool in this field. Our second aim was to construct an efficient and scalable algorithm. It is needless to say, that a system suitable for public use has to be fast. However, in our first approach, we avoided the use of sophisticated data structures and other techniques to speed up the program. We merely wanted to show, that the approach is working and ”tuning” of the prototype was left as a work for the future.

6.3 False Alarms

One cannot give an overall description for the reasons of false alarms. Warning messages usually reflect suspicious situations that might be results of an attack or an attempted attack. But if they are not the consequences of such malicious actions, the cause can be almost anything: a misspelled password, a wrongly executed command, configuration problems of network settings, incompatibility of products, software bugs or even a rarely used – otherwise normal – feature of a program.

A false alarm may be generated for example if someone has a bad IP address configuration on his PC. It will produce several warnings from simple ”no network connection” to ”possible intruder: alien computer in the system”. If the source of this problem is traced, then there is no reason to send repeatedly warning messages that reflect the same problem. We expect the data mining approach to provide rules on events and/or on their attributes, which describe the reason for such frequent unjustified alarms.

In several cases the reasons for false alarms are consequences of the behavior of software or network elements that cannot be modified. For example if a software component regularly wants to connect to its service portal, looking for updates and the company policy prohibits this activity, then there will be a large number of warnings about someone trying to break the regulation. In many cases there is no option to turn this feature off, the only way to filter out this false alarm is based on an appropriate rule. The goal is to create rules that filter out only those warnings that are caused by that specific software component. We definitely do not want to give a chance for an attacker to abuse this rule and hide his activity behind a similar alarm.

The task of the data mining algorithm is hard because it should be ”open” to discover new and weird causes of alarms, i.e., it has to consider every possibly important attributes of events. On the other hand, the attributes that differentiate false alarms from true positives are the most important. Unfortunately this latter requirement cannot be handled by data mining techniques, since in general we have enough number of
samples only for false alarms and not for actual attacks.

6.4 A Formal Definition of Episode Mining

Among the various data mining approaches the episode mining framework seems to be the most suitable for our purposes. Episodes are searched in a sequence of events that are determined by their attributes. Let \( R = \{ A_1, \ldots, A_n \} \) be a set of attributes, where the domain of \( A_i \) is \( D_i \). We denote the set \( D_1 \times D_2 \times \ldots \times D_n \times \mathbb{R} \) by \( \mathcal{E} \).

**Definition 6.4.1** An event over attribute set \( R \) is an element of \( \mathcal{E} \) and we denote it by an \( n + 1 \) tuple \( e = (a_1, \ldots, a_n, t) \), where \( a_i \in D_i \) and \( t \) is a real number, which we call the time of the event.

In the rest of the chapter the time of event \( e \) is referred as \( e.T \) and the attribute \( A \in R \) of \( e \) as \( e.A \). Some examples for attributes used in the common security message format are: Type, Analyzer_Process_Name, Create_Time, Target_Node_Address, Target_Service_Port, Source_User_UserID. To handle the very different messages of various security devices we defined a common, XML based file format (called SMEF, Security Message Exchange Format) and converted all incoming messages to this form.

The **alarm function** \( W : \mathcal{E} \to \mathbb{N} \) plays an important role in our model. If \( W(e) = 0 \), then \( e \) is said to be a normal event, otherwise it is an event that generates alarm of type \( W(e) \).

An **event sequence** is a sequence of events over \( R \), where events are ordered by time. We describe an event sequence of length \( l \) as \( S = \langle e_1, \ldots, e_l \rangle \); here \( e_i \in \mathcal{E} \) and \( e_1.T \leq e_2.T \leq \ldots \leq e_l.T \).

**Filtering the Event Sequence**

The event sequence that is processed by the episode mining algorithm is not the whole raw data coming from the devices. First the list of messages is cleaned and filtered to be more suitable for data mining. This filtering returns an event sequence (or more precisely, several event sequences) that we expect to be smaller than the whole data and we concentrate only on those events that are in relation with the alarms. Consequently, the aim of the filtering is to reduce the complexity. Imagine a user who has a harmless habit that regularly generates alarms. Obviously, we want to discover the pattern of this habit to ignore its alarms in the future. In general, traffic of a network can be so heavy that the elements of the pattern get far from each other, numerous other irrelevant events can be inserted between them. Discovering a pattern whose elements are far from each other needs much more computational capacity than discovering a pattern whose elements are next to (or very close to) each other. Hence patterns that belong to a user are easier to be discovered if we filter the original sequence of messages by a function
that makes selection e.g. according to the IP addresses. In the last section we study formally the complexity-reducing effect of these filter functions.

Episodes

The habits or patterns are defined by episodes. An episode, which describes the preceeding causes of a false alarm can be formalized as a conjunction of several conditions.

**Definition 6.4.2** Let $\mathcal{X} := \{x_1, \ldots, x_k\}$ be variables that can take events as values (event variables). We say that a triple $p(\mathcal{X}, <, \Phi)$ is an episode of size $l$, if $<$ is an order over the time of the event variables, and $\Phi$ is a conjunction of unary predicates, that refer to the attributes of the variables, so

$$\Phi = \bigwedge_{i=1}^{l} \phi_i,$$

where the $\phi_i$ are given predicates applied to an attribute of an event variable.

Without loss of generality, we can presume that for any $i > j$, the inequality $x_i.T \geq x_j.T$ holds. If $<$ is a total order, then $p(\mathcal{X}, <, \Phi)$ is a *serial episode*. If the order is trivial, then the episode is *parallel*. If the episode is neither serial nor parallel, then it is *composite*.

For example the warning about a badly configured IP address we discussed earlier may be filtered by the episode $p(\mathcal{X} = \{x_1, x_2, x_3\}, <, \Phi)$, where

$$\Phi = (x_3.APN = "idslogd") \wedge (x_3.CN = 404) \wedge (x_3.TNAE = 236.182.6.22) \wedge$$

$$(x_2.APN = "suidlogd") \wedge (x_2.CN = 404) \wedge$$

$$(x_2.TNAE = "08:00:07:A9:B2:FC") \wedge (x_2.TNAA = 236.182.6.22) \wedge$$

$$(x_1.APN = "eventlog") \wedge (x_1.SNAA = 236.182.6.22) \wedge (x_1.CN = 206)$$

and $x_1.T < x_2.T < x_3.T$. This episode describes the following situation:

- A message that a network service is started with IP address 236.182.6.22 comes from a PC.
- A gateway sends a message that the network card 08:00:07:A9:B2:FC has an invalid IP address 236.182.6.22.
- A message is sent from the network IDS that a possible alien computer is connected to the network. By the way, this message is an alarm so if $x_3$ is $e_3$, then $W(e_3)$ returns positive value.
Note that this episode filters out only this type of alarms related to this specific computer.

**Definition 6.4.3** The episode \( p'(\mathcal{X}', \prec', \Phi') \) is a subepisode of episode \( p(\mathcal{X}, \prec, \Phi) \) (denoted by \( p' \subseteq p \)), if there exists injection \( f : \mathcal{X}' \to \mathcal{X} \) such that every predicate in \( \Phi' \) that is applied to an \( x \in \mathcal{X}' \), can be found in \( \Phi \) as well applied to \( f(x) \). Furthermore if \( (x_i, x_j) \in \prec' \), then \( (f(x_i), f(x_j)) \in \prec \) is also true. If the size of \( \mathcal{X}' \) is less than the size of \( \mathcal{X} \), then \( p' \) is a proper subepisode of \( p \). We denote this relation by \( p' \subset p \).

It is useful to restrict the episodes that we want to discover. We can presume that an episode \( p(\{x_1, \ldots, x_k\}, \prec, \Phi) \) is always continuous in the sense that at least one predicate applies to each variable. Otherwise there exists an episode \( q(\{x_1, \ldots, x_{k'}\}, \prec, \Phi) \), such that \( k' < k \) and \( p, q \) are isomorphic. Episodes \( p \) and \( q \) are isomorphic if \( p \) is subepisode of \( q \) and \( q \) is subepisode of \( p \). For every episode \( p \) there exists a continuous episode that is isomorphic to \( p \), hence we can restrict our attention to continuous episodes. In the following, every episode is considered to be continuous.

**Definition 6.4.4** Episode \( p' \) is an immediate subepisode of \( p \), if there exist no episode \( p'' \) such that \( p' \subset p'' \) and \( p'' \subset p \).

For example the episodes \( p(\{x_1, x_2\}, \prec, \beta(x_2) \land \alpha(x_1)) \) and \( p'(\{x_1, x_2\}, \prec, \beta(x_1) \land \gamma(x_1)) \) are immediate subepisodes of \( p''(\{x_1, x_2\}, \prec, \beta(x_2) \land \gamma(x_2) \land \alpha(x_1)) \). In the case of the first episode \( f \) may be the identical mapping of the variables, in the second case \( f(x_1) = x_2 \) suffices. Obviously, an immediate subepisode of \( p \) contains all but one predicates of \( p \).

**Invertible Parametric Predicates**

The known algorithms that can handle events with attributes [68], [53] work with pre-defined, given predicates. An episode is a conjunction of such predicates. However, we expect more from our algorithm. It should generate the predicates themselves and then the episodes from these predicates as well. For this we provide "types" of the predicates. The predicate types are defined in the form of parametric invertible predicates. For example, if we think that the predicate that checks the equality of IP address may be important, then we don't want to give \( 2^{32} \) different predicates that check a given IP address, but rather provide only one general predicate.

**Definition 6.4.5** A parametric predicate \( \nu : D \times T \to \{true, false\} \), which applies to the attribute \( A \) (\( A \in R \)) of the variable \( x_i \), is a predicate, whose value depends on the value of the parameter \( q \in T \). The parametric predicate \( \nu \) is invertible, if for every event \( e \) there exists a unique \( q \in T \) such that \( \nu(e, A, q) \) is true.
6.4. A FORMAL DEFINITION OF EPISODE MINING

When we want to discover episodes that contain predicates, which apply to attributes with large domain (for example IP address), then we have to add the parametric predicate

\[ \nu(x.A, q) = \begin{cases} \text{true} & \text{if } x.A = q \\ \text{false} & \text{otherwise} \end{cases} \]

to the given predicates. In the next section we present an algorithm that can handle these parametric predicates. Of course, the parameters of the episodes are set by the algorithm.

Since there are many special events, not all attributes are set or can be interpreted in the actual situation. Regarding the value of a parameter on an event, where an attribute is not applicable, we consider a predicate that applies to a missing attribute as false for any value of its parameter. Please note that with a fixed \( q \) value the predicate \( \nu(x.A, q) \) is regarded as a traditional unary predicate. It is important, that a predicate with different parameter values gives different unary predicates. We also refer to a parametric predicate with a fixed parameter as a \textit{parameter-predicate pair} and a predicate with non-fixed parameter as a \textit{predicate type}.

6.4.1 Support and Alarm Support

\textbf{Definition 6.4.6} The event sequence \( S = \langle e_1, \ldots, e_l \rangle \) contains the episode \( p(\{x_1, \ldots, x_k\}, <, \Phi) \), if there exists different events \( e_{j_1}, \ldots, e_{j_k} \in S \) such that in \( p(\{e_{j_1}, \ldots, e_{j_k}\}, <, \Phi) \) \( \Phi \) is true and \( < \) holds for the time of the events in \( S \).

If the sequence \( S \) contains episode \( p \), then we say "\( p \) occurs in \( S \)" or "\( p \) is true in \( S \).

\textbf{Definition 6.4.7} An \( m \) window of the event sequence \( S = \langle e_1, \ldots, e_l \rangle \) that ends with an alarm of type \( w \) is an event sequence \( S' = \langle e_j, \ldots, e_{j+m-1} \rangle \), where \( 1 \leq j \leq l - m + 1 \) and \( W(e_{j+m-1}) = w \).

The set of windows of \( S \) defined above is denoted by \( auw(S, m, w) \).

\textbf{Definition 6.4.8} The support of episode \( p \) in \( auw(S, m, w) \) is the number of windows that contain \( p \):

\[ \text{supp}_{s,m,w}(p) = |\{S' \in auw(S, m, w) | S' \text{ contains } p\}| \]

An episode is \textit{frequent}, if its support is higher than a given support threshold (\text{minsup}), otherwise it is \textit{infrequent}.

The frequent episodes are not necessarily important in practice. There can be many that have no connection with alarm situation, but they occur in many windows that end with an alarm. Such universally frequent episodes are out of interest in our context.

If episode \( p(\{x_1, \ldots, x_k\}, <, \Phi) \) is contained in a window such that \( x_1 \) generates an alarm, then this episode may be important because the conditions described by the
episode may have been considered improperly to be an attack by some security device. We shall focus on such episodes and their occurrences.

Let us define the term alarm support.

**Definition 6.4.9** The alarm support of a serial episode \( p\{x_1, \ldots, x_k\}, \prec, \Phi \) in the au(S, m, w) is defined by

\[
\text{alarm supp}_{s, m, w}(p\{x_1, \ldots, x_k\}, \prec, \Phi) = |\{S' = \{e_1, \ldots, e_m\} \in \text{au}(S, m, w) | e_{j_1}, \ldots, e_{j_k} \in S' \setminus \{e_m\} \text{ different events such that in } p(\{e_m, e_{j_2}, \ldots, e_{j_k}\}, \prec, \Phi) \Phi \text{ is true } \}|
\]

An episode is alarm frequent, if its alarm support is greater than a given threshold (\( \text{minsup} \)).

**Definition 6.4.10** The expression \( p[m, w] \Rightarrow \{\text{real alarm}, \text{false alarm}\} \) is an episode rule, if \( p \) is an episode, \( m \) is an integer number and \( w \) is an alarm type. The interpretation of the rule \( p[m, w] \Rightarrow \text{false alarm} \) is the following: if \( p \) occurs in an event sequence of \( S \) of width \( m \) that ends at an alarm of type \( w \), then the alarm is false, otherwise it is a real alarm.

Our final goal is to determine episode rules that filter out false alarms. The data mining algorithm discovers alarm frequent episodes and an expert (security specialist) sets the right-hand-side (false alarm or real alarm) of the rules. Obviously this step cannot be automated since it requires domain knowledge (knowledge about the local network, about the security devices, about the users, etc). We expect that determining the alarm frequent episodes can help the security specialists to handle the vast number of alarms effectively.

### 6.4.2 The Aim of Data Mining

After clarifying the basic definitions, we can set the model and define the aim of data mining in the Remote Supervision System. Given is a filtered event sequence \( S \) that has to be processed off-line. The invertible, parametric, unary predicates \( \alpha(x, A, q_a), \beta(x, B, q_b), \ldots \), the window width \( (m) \) and the alarm type \( (w) \) are provided by an expert (system administrator, security specialist). Based on this set of parameters we have to determine the alarm frequent serial episodes in \( S \).

Our first task is to determine the values of the parameters so that from the predicates obtained, frequent episodes can be built. The output of the data mining module (the alarm frequent episodes) is examined exhaustively by the expert, who finally approves the false alarm filtering episode rules. After the episode rules are set, the on-line processing of the network traffic can be started. If an alarm of type \( w \) arrives and its preceding \( m \) wide window contains episode \( p \) (more precisely the episode rules \( p[m, w] \Rightarrow \text{falsealarm} \) exists), then the alarm is determined to be false and automatically filtered out.
6.5 The Algorithm ABAMSEP

We restrict our search to serial episodes, however the model and the algorithm can be extended to handle parallel and a family of composite episodes as well, at the expense of performance degradation. These generalizations are not discussed here. In the following event variables are always referred as $x_1, \ldots, x_k$ and the order on time is $x_k.T < \ldots < x_1.T$. If only continuous, serial episodes are concerned, the conjunction of the predicates unambiguously determines the episode itself. So for the sake of simplicity, an episode is understood to be the conjunction of the predicates (we write $p = \bigwedge_{i=1}^k \phi_i$).

6.5 The Algorithm ABAMSEP

The detailed algorithm ABAMSEP (Apriori-Based Algorithm for Mining Serial Episodes with parametric Predicates) is based on algorithm Apriori (see section 2.8). It discovers frequent serial episodes and handles invertible parametric predicates. The pseudo-code is given in the next page.

The algorithm has two phases. First, the parameters of the interesting predicates are determined, and the windows are found where alarm frequent episodes may occur. Next, these windows are scanned and the frequent episodes are discovered.

So in the beginning we determine those predicate-parameter pairs that can be true on an event that generated alarm of type $u$. From these predicates we can immediately generate alarm frequent episodes consisting of only one condition. The occurrences of these episodes will be the last events of those windows, where alarm frequent episodes can be found. This set of windows is a subset of $aw(S, m, u)$ so let us denote it by $aw'(S, m, w)$ $(aw'(S, m, w) \subseteq aw(S, m, u))$.

In order to determine frequent episodes in $aw'(S, m, w)$, we need some further evaluations. The following property holds for every frequent episode.

**Property 6** If an episode $p$ is frequent in some windows of the event sequence $S$, then all subepisodes of $p$ are also frequent in these windows.

This follows from the fact that if an episode occurs in an event sequence, then the subepisodes occur as well. This property suggests to adapt the scheme of algorithm Apriori.

We scan every event of $aw'(S, m, w)$ one-by-one and determine those predicate-parameter pairs that are true on the actual event. Notice, that a predicate-parameter pair can be regarded as an episode of size 1. Every predicate-parameter pair has a counter, and if the pair is true on an event we increase this counter. The counter can be increased just once in a window although it may be true on more than one event of the window. After reading through the event sequence we select those predicate-parameter pairs that have support higher than minsup. The frequent episodes of size 1 will consist of them. In the following only these frequent predicate-parameter pairs are considered. As we mentioned earlier, these predicates after the parameters are fixed, can be regarded
Algorithm 6.5.1 algorithm ABAMSEP

Require: $S = \langle e_1, \ldots, e_i \rangle$ : event sequence ordered by time,
        $m$ : width of the window,
        minsup : support threshold
        $\alpha, \beta, \ldots$ : parametric predicates
        $w$ : alarm type

Ensure: $P^w$ : set of the alarm frequent episodes

I. PREPROCESSING:
   for all events $e_i \in S : W(e_i) = w$ do
      determine those predicate-parameter pairs that are true on $e_i$
      determine $rep_{\alpha}(S, m, w)$
      generate $C_1$
   $i \leftarrow 1$

II. MAIN CYCLE:
   repeat
      determine the support of elements of $C_i$
      $P_i \leftarrow \{e | e \in C_i, e\text{support} \geq \text{minsup}\}$,
      delete $C_i$
      $C_{i+1} \leftarrow \text{candidate generation}(P_i)$
      for all $p \in P_i$ do
         if $p\text{support} \geq \text{minsup}$ then
            $P_i^w \leftarrow P_i^w \cup p$
         else
            delete $p$
         end
      end
      OPTIONAL STEP: delete nonmaximal episodes from $P_{i-1}^w$
      $i \leftarrow i + 1$
   until $\{|C_i| > 0 \text{ AND } P_{i-1}^w > 0\}$
   $P^w \leftarrow \bigcup_i P_i^w$

as traditional predicates. Without loss of generality, we assume that these predicates are ordered.

The next step is to generate candidate episodes of size 2 from frequent episodes of size 1. An episode can be candidate if all of its subepisodes are frequent. Note that this is just a necessary condition for an episode to be frequent, therefore for each candidate the support should be determined in an additional step. To do this, after candidate generation the support counting method is evoked. In general candidate episodes of size $i + 1$ are generated from the frequent episodes of size $i$. The candidate generation is detailed in section 6.5.1. After the candidate generation, we need only the alarm frequent episodes of size $i$, the others can be deleted. The next step is to determine the
6.5. THE ALGORITHM ABAMSEP

support of the candidates of size \(i + 1\), and delete those that have support less than the support threshold (\(\text{msup}\)). By repeating this process \((i = 1, 2, \ldots)\) we can determine all alarm frequent episodes. The algorithm terminates, if no new candidate is generated.

The output of the algorithm is the set of alarm frequent episodes. The problem with this solution is that too many useless episodes are generated. For example an alarm frequent episode of size 5 and variable number 5 has \(2^5 - 2\) subepisodes that are also alarm frequent. Consequently, it is useful to return to the expert only the maximal (with respect to \(\subseteq\)) alarm frequent episodes.

6.5.1 Candidate Generation

The candidate generation is similar to the method proposed in algorithm Apriori. The differences stem from the fact that Apriori works with itemsets while here we are working with episodes. Candidate generation has two phases: join and prune.

6.5.1.1 Join Phase

A candidate \(c\) of size \(i + 1\) is generated from two frequent episodes \((p_1, p_2)\) of size \(i\). Without loss of generality we can assume that \(p_1\) has \(l\) variables, \(p_2\) has \(k\) variables and \(l \geq k\). We join the two episodes, if by deleting the predicate \((\mu(x_i, A))\) from \(p_1\) that has the largest order among those that apply to the variable \(x_i\), we obtain the same episode as we get if we delete the predicate \((\nu(x_k, B))\) from \(p_2\), which has the largest order among those that apply to the variable \(x_k\). Thus, \(p_1\) and \(p_2\) must have \(i - 1\) common predicates that apply to the variables \(x_1, \ldots, x_k\). Three different cases are possible:

1. \(p_1\) is equal to \(p_2\) (so \(l = k\) and \(\mu = \nu\)). We join an episode with itself if only one predicate applies to \(x_i\) (\(= x_k\)). If this condition holds, then we generate the candidate \(c := p_1 \land \mu(x_{k+1}, B)\).

2. If \(p_1 \neq p_2\) and more than one predicates apply to the variable \(x_k\) in \(p_2\), then \(c := p_1 \land \nu(x_k, B)\) is the candidate. Obviously if predicate \(\nu\) applies to \(x_k\) in \(p_1\) (even with different parameter), then we can immediately delete the candidate. The reason for this is, that if the parameters are the same, then the candidate is not of size \(i + 1\), otherwise it will not occur in any window (invertibility).

3. Otherwise \(l = k\) and only one predicate applies to the variable \(x_i\) in \(p_1\), since the episodes are continuous (each variable is contained in at least one predicate). In this case 3 candidates are generated: \(c' := p_1 \land \nu(x_k, B)\), \(c'' := p_1 \land \nu(x_{k+1}, B)\), \(c^* := p_2 \land \mu(x_{k+1}, A)\). Again, if predicate \(\nu\) applies to \(x_k\) in \(p_1\), then \(c'\) can be deleted.

The episode pair \((p_1, p_2)\) generates the same candidate as the pair \((p_2, p_1)\) does. We suppose that an order on the episodes can be defined (for example lexicographic order
that is defined based on the ordering of the predicates). Two episodes are joined if and only if $p_2$ is larger than $p_1$ with respect to the order.

Let us consider some examples for the join phase (the attributes of the variable are omitted for the sake of simplicity).

<table>
<thead>
<tr>
<th>$p_1$</th>
<th>$p_2$</th>
<th>candidate</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\gamma(x_3) \land \beta(x_2) \land \alpha(x_1)$</td>
<td>$\alpha(x_1) \land \beta(x_1) \land \delta(x_1)$</td>
<td>not joinable</td>
</tr>
<tr>
<td>$\gamma(x_3) \land \beta(x_2) \land \alpha(x_1)$</td>
<td>$\gamma(x_3) \land \delta(x_2) \land \alpha(x_1)$</td>
<td>not joinable</td>
</tr>
<tr>
<td>$\beta(x_2) \land \gamma(x_2) \land \alpha(x_1)$</td>
<td>$\beta(x_2) \land \delta(x_2) \land \alpha(x_1)$</td>
<td>$\gamma(x_3) \land \beta(x_2) \land \delta(x_2) \land \alpha(x_1)$</td>
</tr>
<tr>
<td>$\gamma(x_3) \land \beta(x_2) \land \alpha(x_1)$</td>
<td>$\delta(x_3) \land \beta(x_2) \land \alpha(x_1)$</td>
<td>$\delta(x_4) \land \gamma(x_3) \land \delta(x_2) \land \alpha(x_1)$</td>
</tr>
<tr>
<td>$\gamma(x_3) \land \delta(x_3) \land \beta(x_2) \land \alpha(x_1)$</td>
<td>$\gamma(x_4) \land \delta(x_3) \land \beta(x_2) \land \alpha(x_1)$</td>
<td>$\gamma(x_4) \land \delta(x_3) \land \beta(x_2) \land \alpha(x_1)$</td>
</tr>
</tbody>
</table>

Table 6.1: Example for joining

It is instructive to look at the possible candidates generated from the pair $p_1 = \mu(x_1.A), \ p_2 = \nu(x_1.B)$. Here $l = k = 1$ and if $\mu \neq \nu$, then the candidate pairs are $\mu(x_1.A) \land \nu(x_1.B); \ \mu(x_2.A) \land \nu(x_1.B); \ \nu(x_2.B) \land \mu(x_1.A)$. If $\mu = \nu$, then the first candidate is deleted. We remark that the two episodes are always immediate subepisodes of the candidate generated.

### 6.5.1.2 Prune Phase

The objective of this phase is to prune the candidates that have an immediate subepisode of size $i$ that is infrequent, i.e., it is not among the frequent episodes.

Let us consider some examples with two different frequent episode sets of size 3, which are given in Table 6.2. Frequent episodes are found in the first column, the candidates after the join phase in the second. If the candidate is pruned, then yes can be found in the $3^{rd}$ column, otherwise no. If the candidate is pruned, then its immediate subepisode that is infrequent is shown in the $4^{th}$ column.

### 6.5.2 Determining the Support

To determine the support of the candidate episodes, we present a simple algorithm that is easy to implement. Using tries and other techniques we can further improve the support count procedure. Since the data structure and most of the speed-up techniques presented in chapter 3 and 4 are also applicable in mining frequent sequences of items [16], they can be employed in our episode mining setting.
### 6.5. The Algorithm ABAMSEP

<table>
<thead>
<tr>
<th>frequent episodes of size 3</th>
<th>candidates after join</th>
<th>is pruned</th>
<th>infrequent subepisodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\gamma(x_3) \land \delta(x_2) \land \alpha(x_1)$</td>
<td>$\gamma(x_3) \land \delta(x_2) \land \delta(x_1) \land \alpha(x_1)$</td>
<td>yes</td>
<td>$\gamma(x_2) \land \beta(x_1) \land \delta(x_1)$</td>
</tr>
<tr>
<td>$\beta(x_2) \land \delta(x_2) \land \alpha(x_1)$</td>
<td>$\gamma(x_3) \land \delta(x_2) \land \delta(x_1) \land \alpha(x_1)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\gamma(x_3) \land \delta(x_2) \land \alpha(x_1)$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\delta(x_3) \land \beta(x_2) \land \alpha(x_1)$</td>
<td>$\gamma(x_3) \land \delta(x_2) \land \delta(x_1) \land \alpha(x_1)$</td>
<td>yes</td>
<td>$\gamma(x_2) \land \delta(x_2) \land \alpha(x_1)$</td>
</tr>
<tr>
<td>$\delta(x_3) \land \gamma(x_2) \land \alpha(x_1)$</td>
<td>$\delta(x_4) \land \gamma(x_3) \land \beta(x_2) \land \alpha(x_1)$</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>$\delta(x_3) \land \gamma(x_2) \land \beta(x_1)$</td>
<td>$\gamma(x_4) \land \delta(x_3) \land \beta(x_2) \land \alpha(x_1)$</td>
<td>yes</td>
<td>$\gamma(x_3) \land \delta(x_2) \land \beta(x_1)$</td>
</tr>
</tbody>
</table>

| Table 6.2: Example for pruning |

The support of the candidates has to be calculated so that the frequent ones can be selected, and the infrequent ones are pruned. Each window of $au'(S, m, \omega)$ has to be examined and those candidate episodes have to be found that are true in the window. A candidate episode of size $k$ occurs in a window if there exists $k$ different events such that all predicates that apply to the variable $x_j$ are true on the $j^{th}$ event ($1 \leq j \leq k$).

We use a greedy algorithm to find the episodes of size $k$ that occur in the given event sequence $S = \langle e_1, \ldots, e_m \rangle$. Let us read through the event sequence from the end to the beginning. A finite automaton can represent each episode. All automata are in the initial ($0^{th}$) state before we process the event sequence. When event $e$ is considered, the automaton of the candidate episode $p(x_1, \ldots, x_k)$ jumps to the next state from state $i$, whose predicates that apply to the variable $x_{i+1}$ are true on $e$. Otherwise we don’t jump. The state $k$ is the accepting state.

A boolean variable has to be added to each automaton. It is set when the last (from the end the first) event is processed. Its value is true for those automata that jumped to the first state after this event (the one that generated alarm) is processed. Otherwise the variable is set to false.

When the left end of the window is reached, all instances of the automata are deleted. We increase the support of those candidates, whose automata is in the accepting state. If, in addition, the boolean variable is true, then the alarm support is also incremented.

In order to implement the automaton a reference to the episode and a variable that stores the state index of the automata needs to be handled. These two pieces of data and the boolean variable can be stored in a list. We add a triple to this list only when their imaginary automaton jumps to the first state.

Please observe that the (costly) disk operations are carried out in the first two steps of the preprocessing, and in the first step of the main cycle.
6.5.3 An Optional Step

A partially ordered set can be built from the alarm episodes, where the alarm frequent episodes are under a border. The elements of the border are the maximal alarm frequent episodes. It would be useless to present all the frequent episodes to the user since the border has fewer elements, and any frequent episode can be obtained from the maximal ones.

The alarm frequent episode $p$ is maximal, if there exists no frequent episode $p'$ such that $p$ is a proper subepiside of $p'$. Maximal episodes can be filtered in two ways. First, we can filter the output of algorithm ABAMSEP; second we can weave the filtering process into the algorithm. The advantage of the second solution is that it decreases the memory need, since fewer episodes are stored. We applied this approach in the implementation.

After infrequent candidates of size $i$ are deleted, the frequent episodes of size $i-1$ can also be pruned, which are subsets of some frequent episode of size $i$. Hence, if the user is interested only in the maximal episodes, then the following line have to be inserted into the main cycle (by definition $P_i^{(w)}$ equals to $\emptyset$).

**Algorithm 6.5.2** Removing nonmaximal episodes

```plaintext
for all $p_1 \in P_{i-1}^w$ do
    for all $p_2 \in P_i^w$ do
        if $(p_1 \subset p_2)$ then
            $P_{i-1}^w \leftarrow P_{i-1}^w \setminus p_1$
```

6.5.4 Completeness and nonredundant generation

By the following lemma the greedy algorithm (presented in section 6.5.2) properly counts the support of an episode.

**Lemma 6.5.1** The greedy algorithm finds all episodes that occur in a given event window.

**Proof:** Suppose that there exists a candidate episode $p(x_1, \ldots, x_k)$ that is true in event window $S' = \langle e_1, \ldots, e_m \rangle \in aw(S, m, w')$, but the greedy algorithm did not find it to be true. According to the assumption there exist different events $e_{i_1}, \ldots, e_{i_k} \in S'$ such that $p(e_{i_1}, \ldots, e_{i_k})$ is true. Since the greedy algorithm did not find the episode after processing $S'$, the automaton that represents the episode stopped in a state $(k')$, with $k' < k$. Thus there exist different events $e_{i_1}', \ldots, e_{i_{k'}}' \in S'$, such that all predicates that apply to $x_1$ are true on event $e_{i_1}'$, and all predicates that apply to $x_2$ are true on event $e_{i_2}'$, and so on. The algorithm finds these events starting from the back, and due to the
greedy nature of the method the relations $i_1 \geq i'_1$, $i_2 \geq i'_2$, ..., $i_k \geq i'_k$ must hold. We are searching for occurrences of serial episodes, it follows from the above that there exists an event $(e_{i_{l+1}})$ that is in the window, it is before $e_{i_l}$, and all predicates that apply to variable $x_{k+1}$ are true on it. The greedy algorithm will clearly find this event. This is a contradiction. □

**Theorem 6.5.2** The algorithm ABAMSEP is complete: it finds all alarm frequent episodes.

**Proof:** The proof is based on induction of the size of episodes. In the first step we check each event in windows $aw'(S, m, w)$, and all predicates are found from whom a frequent episode of size 1 can be built. Let us suppose that we found all frequent episodes of size $l \geq 1$, but an episode $p$ of size $l + 1$ and variable number $k$ was not found. According to Lemma 6.5.1, if a frequent episode is generated as a candidate episode, then its support is calculated exactly. If $p$ was not found to be frequent, then it were not generated at all.

3 different cases can occur: (1) at least two predicates apply to $x_k$, (2) one predicate applies to $x_k$, and at least two to $x_{k-1}$, (3) one predicate applies to $x_k$ and one to $x_{k-1}$.

In the first case $p$ is in the form of $\alpha(x_k) \land \alpha'(x_k) \land p'$, where $\alpha \neq \alpha'$. However, $p$ should have been generated by joining episodes $p' \land \alpha(x_k)$ and $p' \land \alpha'(x_k)$.

In the second case $p$ is $\alpha(x_k) \land \alpha'(x_{k-1}) \land p''$, where at least one predicate applies to $x_{k-1}$ in $p''$. In this case $p$ is obtained if episodes $p'' \land \alpha(x_k)$ and $p'' \land \alpha'(x_{k-1})$ are joined.

In the third case $p = \alpha(x_k) \land \alpha'(x_{k-1}) \land p^*$, where the largest variable in $p^*$ is $x_{k-2}$. Here, by joining $p^* \land \alpha(x_{k-1})$ and $p^* \land \alpha'(x_{k-1})$ episodes we obtain $p$. If $\alpha = \alpha'$, then $p_1 = p_2$, hence it is a case of self-join, where the condition holds (i.e., only one predicate applies to the largest variable). Each case leads to contradiction, hence the statement follows. □

**Corollary 8** Each candidate is generated once in algorithm ABAMSEP.

**Proof:** It is immediate from the proof of the Theorem 6.5.2. For any candidate we can uniquely determine the two subepisodes that generated it, hence it cannot happen that two different episode pairs generate the same candidate. □

Candidate generation algorithms that do not generate the same candidates in different ways are called nonredundant in the literature. Nonredundant candidate generation is a requirement of an efficient frequent pattern mining algorithm.
6.5.5 Theoretical Remarks on the Time and Memory need of the method

Let us denote the size of the largest alarm episode by $|p_{\text{max}}|$, and as earlier, the length of the sequence by $l$ and the width of the window by $m$. We analyze the time and memory need with the assumption that the event sequence is on disk.

Operations in the memory are much faster than operations on the disk, therefore disk access is of primary concern. It is easy to determine how many times does the algorithm read through the event sequence. ABAMSEP is a levelwise algorithm, it reads through the database as many times as the size of the largest episode. If the number of the given predicates is $u$, then this is at most $m \cdot u$, because the number of variables cannot be more than the size of the window and a predicate can apply only once to any variable (this is a consequence of invertibility). We infer that the number of disk access is linear in the parameters $l, m$ and $u$.

The candidates and the actual window are stored in the memory. Insufficient space in the main memory slows down very sharply the processing of the candidates. It is impossible to estimate, which episode counter should be increased before processing a window, hence swapping the candidates to and from the disk would take a lot of time. Therefore algorithm ABAMSEP does not possess the "graceful degradation" property, similarly to all other frequent pattern mining algorithms [5, 6, 71, 55, 99, 113].

The procedure that finds the supported candidates in a window is executed as many times during a single reading of the sequence as many elements $aw'(S, m, w)$ has. However, this can be slower than reading in the sequence from the disk. We know that when an event is processed, we have to check the state of all instances of the automata, hence support determination is proportional to the number of the candidates. If $|p_{\text{max}}| < m$, then in the worst case just the number of episodes with $|p_{\text{max}}|$ variables can be $(u \cdot |p_{\text{max}}|)^{l|m|}$, since $u \cdot |p_{\text{max}}|$ different predicates can apply to each variable. If $|p_{\text{max}}| \geq m$, then the number of the candidates can be even more than $u^m$. Consequently, determining the support can be proportional to $l \cdot u^m$.

This exponential growth is not as bad as it seems. Every data mining algorithm, where the aim is to find frequent objects, shows similar characteristic [5, 6, 55, 71, 99, 113]. Fortunately, the theoretical bounds on time and memory complexity and real performance are often far from each other. When the algorithm is slow, then the parameters are probably set improperly and too many episodes are generated. Generating too many episodes should be avoided since these episodes have to be examined one-by-one by an expert. The test results presented in the next section supports the following hypothesis of ours: when the mining yields manageable results, then the algorithm finds the episodes in acceptable time.
6.6 Experimental Results

Implementation of the proposed algorithm was developed within the framework of a research project supported by the Hungarian Ministry of Education in cooperation with ICON Ltd., a Hungarian IT company. During this project a common message format was elaborated. We collected large volume of log files from different security products and the execution experiments come from this work.

In the figures the influence of parameters on the run-time and the candidate number can be seen. The parameters examined were the support threshold, the width of window and the number of invertible predicates. In the previous section we showed that theoretically there may be an exponential growth in the run-time.

In the test system events were generated by 20 different devices. Each event had 11 attributes. The first attribute returned the type of the event, i.e.: entry of a computer to the network, signal from a virus checker, entry of a user to the network, or system event. Other attributes were: process name, creation time, classification name, detection time, source node address, source service port, target node address, target service port, target file and target file path.

The raw database consisted of 2400 events, the filtered sequence was of length 600. 100 alarms were hidden in the data: 50 randomly and 50 were inserted with predefined events. These events were generated so that episodes could be retrieved from them. These episodes had 4, 5 or 6 variables and the size of them varied from 18 to 28. Random events were inserted between the predefined events such that the episodes could be discovered by using windows of size 10.

We implemented the algorithm on a Linux operating system (Red Hat Linux version 7.2). The tests ran on a configuration with Athlon XP 1700+ processor and 256 DDR operative memory.

The ABAMSEP algorithm worked properly. If the window size parameter was set to 10 or more, then each episode was successfully discovered. Obviously if the window size was less than 10, then some episodes were infrequent, hence left unnoticed.

Further test results and the description of the test environment can be found in [61].

Before presenting the results, we would like to draw the reader's attention to a very important feature. In most data mining applications, we experienced low correlation of a large number of items. Even if the size of dataset is very large, the number of frequent items is still manageable. Unfortunately this is not the case with security events. There are a few type of events, only the values of parameters change.

Let us first examine the number of candidates of different sizes (Figure 6.1). We can see that by increasing the number of predicates the number of the candidates rapidly grows. After it reaches the peak the number of candidates decreases. Similar characteristics were observed and analyzed in the case of frequent itemset mining [42]. This is due to the similarity with respect to the inclusion anti-monotonicity, i.e., if a set/episode is a candidate, then all its subsets/subepisodes are candidates as well.
Figure 6.1: Number of candidates of different size

The Figures 6.2, 6.3, 6.4 show how the window width, the predicate number and the support threshold affect to the run-time. Please note, that the exponential growth of the run-time is a property of the problem itself and not of the solution. The search space is exponential in the number of predicates, thus if we set minsup to zero, then all possible episodes become frequent and simply outputting the result requires exponential time.

We can see that the retrieval speed is getting lower if we increase the width of the window or the number of the predicate types, or decrease the support threshold. One may find the run-times too high. However, we have to emphasize that this primary implementation did not include any accelerating techniques. Simple data structures (like lists) were used where not even ordering and binary search was implemented. Our primary aim to demonstrate the viability of a frequent pattern mining solution for the problem.

6.7 Consequences and Future Research

The aim of this research was study the improvement possibilities of a Remote Supervision System. Such systems seem to be the most effective in computer security, therefore this kind of research is of importance. We intended to battle and defeat the most dangerous enemy, the large number of possibly false or unimportant alarms. We have won the battle, however, the end of the war is still far away. In our work we proved that data mining is a powerful weapon. An efficient and scalable algorithm was proposed that makes it possible to automatically filter many false alarms.
Figure 6.2: Run-time as the function of the window size

Figure 6.3: Run-time as the function of the number of predicates
Several simplifications have been made in our model in order to keep the complexity of the algorithm acceptable even when large event sequences are processed. Our solution can be improved in many ways. Episodes can be generalized so that more complex patterns can be found. The efficiency of our existing algorithm can also be improved. Here we shortly discuss some avenues of further research.

- Algorithm ABAMSEP is searching for serial episodes only. However, parallel and more complex episodes are also of interest. Candidate generation and support count can be easily extended to handle parallel episodes. The time complexity increases as soon as more general episodes are searched for. We suggest that a middle way solution i.e., serial episode that is made of small parallel episodes could still be manageable.

- Episodes were defined as sets of conditions where the conditions were given by unary predicates. When higher order predicates (for example binary) are allowed in the conditions, then neither the candidate generation nor the support count could be solved so easily.

- We have proposed that a filtered sequence and not the raw data should be processed by the data mining algorithm. Filters could be efficiently implemented in the system, and produce the filtered sequence very fast. We know that some binary predicates can be substituted if a proper filter is used. For example the binary predicate \( x_1.IP = x_2.IP \) is implicitly included in the episodes if we filter the raw data according to the same IP addresses. However there are binary predicates that cannot be substituted by any reasonable filter. A theoretical and practical research on the limitations and realizations of the filters is still ahead.
The prototype implementation of our algorithm ABAMSEP does not include any advanced methods for fast operation. Obviously, many of the speed-up techniques discussed in the preceding chapters may be of value here.

We see that many interesting and important open problems can be posed. We believe that we have proved here that data mining algorithms can be applied in the security supervision of IT systems by discovering the sources of false alarms. If the number of false alarms can be decreased thanks to the rules determined, then the sensitivity parameters of security devices can be set to high and the number of recognized attacks will increase as well.
Chapter 7

References


[29] David Wai-Lok Cheung, Jiawei Han, Vincent Ng, and C. Y. Wong. Maintenance of discovered association rules in large databases: An incremental updating technique. In Proceedings of the International Conference on Data Engineering (ICDE'96), pages 106–114, 1996. URL citeseer.nj.nec.com/cheung96maintenance.html. 2


[33] Yun Chi, Yirong Yang, and Richard R. Muntz. HybridTreeMiner: An efficient algorithm for mining frequent rooted trees and free trees using canonical forms. In The 16th International Conference on Scientific and Statistical Database Management (SSDBM'04), June 2004. URL citeseer.ist.psu.edu/chi04hybridtreeaminer.html. 2


[49] Eui-Hong Han, George Karypis, Vipin Kumar, and Bamshad Mobasher. Clustering based on association rule hypergraphs. In Research Issues on Data Mining and Knowledge Discovery, pages 0–, 1997. URL citeseer.ist.psu.edu/11747.html.

[50] Jiawei Han and Yongjian Fu. Discovery of multiple-level association rules from large databases. In Proc. of the 21st International Conference on Very Large Databases (VLDB), Zurich, Switzerland, 1995.


[69] Heikki Mannila and Hannu Toivonen. Multiple uses of frequent sets and condensed representations (extended abstract). In Knowledge Discovery and Data Mining, pages 189-194, 1996. URL citeseer.ist.psu.edu/mannila96multiple.html.


[75] Raymond T. Ng, Laks V. S. Lakshmanan, Jiawei Han, and Alex Pang, Exploratory mining and pruning optimizations of constrained associations rules. In In Proceedings of 1998 ACM SIGMOD International Conference Management of Data, pages 13–24, 1998. URL citeseeer.ist.psu.edu/ng98exploratory.html. 2


[85] Jian Pei, Jiawei Han, and Runying Mao. CLOSET: An efficient algorithm for mining frequent closed itemsets. In ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, pages 21–30, 2000. URL citeebook.ist.psu.edu/pei00closedet.html.

[86] Jian Pei, Jiawei Han, and Laks V. S. Lakshmanan. Mining frequent item sets with convertible constraints. In Proceedings of the International Conference on Data Engineering (ICDE'01), pages 433-442, 2001. URL citeebook.ist.psu.edu/383962.html.


[108] Ke Wang, Yu He, and Jiawei Han. Mining frequent itemsets using support constraints. In The VLDB Journal, pages 43–52, 2000. URL citeseer.ist.psu.edu/wang00mining.html. 2


