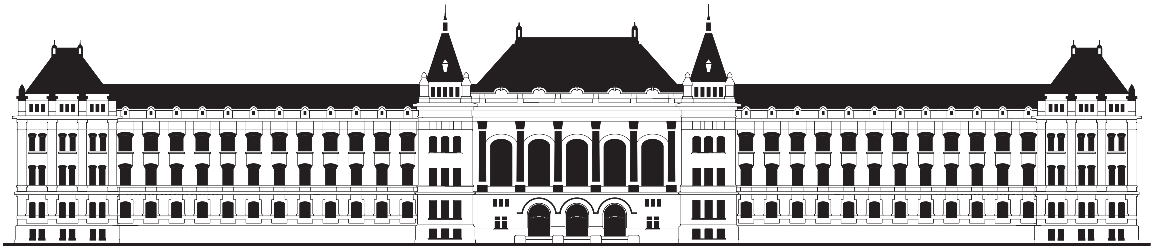
A Survey on Frequent Itemset Mining



A Survey on Frequent Itemset Mining

in preparation

by Ferenc Bodon

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

Budapest
April 28, 2006

Preface

The frequent itemset mining problem first has been formulated in 1993 as the computational relevant step in association rule mining. Given a sequence of itemsets, we have to find itemsets that are contained as a subset in more than a given number of elements of the sequence. More than 180 papers have been published about algorithms to solve this task, most of them declared to be the most efficient. The open-source competition, which was organized ten years after the problem's birth, proved that the truth is far from the claims and the data structure and implementation issues need to be polished even for the basic algorithms.

In this survey we investigate data structure and implementation details of the three most important FIM algorithms Apriori, Eclat, FP-growth, examine their advantages and disadvantages. Besides, we present new techniques to speed-up the basic algorithms.

Contents

Chapter 1

Introduction

Frequent itemset mining (FIM) is a very young research field born in 1993 [3]. It aims to find frequently occurring subsets in a sequence of sets. The FIM problem appears as a subproblem in many other data mining fields like association rule discovery [3], correlations, classification [27], clustering [28], Web mining [55], [34]. The fact that algorithms and techniques developed in FIM are also used successfully in finding frequent patterns of other types (like sequences, episodes, rooted ordered/unordered trees, labeled graphs and boolean formulas) also proves the significance of the field. The frequent sets play an important role in many application such as customers relationship management, improving the efficiency of electronic commerce [48], bioinformatics, DNA and protein analysis, inductive databases [31], query expansion [39], network intrusion detection [29], etc.

After the problem was born, many algorithms were proposed, the authors of each algorithms claimed that their algorithms are the fastest. The efficiency was shown by run-time plots on a few databases and for the comparisons the authors coded the counterpart algorithms as well. Unfortunately, neither the proposed algorithm, nor the implementation of the counterpart algorithm were public available therefore the claimed results were not reproducible.

Those, who wanted to find the real values, the real contributions were not satisfied with lack of reproducibility. The first step toward the quality assurance was the publication of some independent authors who implemented and compared some published algorithms [25] [19] [26] [9] [17]. Unfortunately, these implementations are less effective than the best implementations (if there exists such)of the same algorithms, and often they do not even show the same performance characteristics. This is the reason, we believe that the consequences drawn form the experiments of such implementations are not necessarily attributed to the algorithms themselves, but rather to the non-optimized implementation.

A better and less time-consuming solution was published by Zheng et al. [59], where independent referees collected the implementations from the authors themselves, and run the experiments. The case of Apriori, Eclat and FP-growth has shown the most efficient implementation of an algorithm is not necessarily developed by the inventors of the algorithm, therefore the comparison should be open to everybody. This lead to the public, open-source competitions of FIM implementation [16] in 2003 and 2004. The importance of the FIMI contests is inevitable; it stressed the requirement of reproducibility and provided some baseline implementation and test datasets for the research community. The most important question, however, is not answered. We still don't know the real reasons of the efficiency, the borders and the bottleneck of some implementations. Does the efficiency of best implementation stems from the algorithm itself or from the sophisticated data structure? Or the outstanding run-time is attributed to the brilliant programming technique?

Although we agree with the view "Exposition, criticism, appreciation, is work for second-rate minds."[1] we believe that the publishing of data mining algorithms has reached the point when a theoretical analysis and a comprehensive studies are more useful to the community than new algorithms. It seems that we have moved from the "We are drowning in information, but starving for knowledge" to the "We are drowning in methods, but starving for solutions" era.

Even now, after the numerous publication there are many misbelieves, misunderstanding about the efficiency of certain algorithms. Most reasoning in textbooks are either not true or they are not the real reason of an algorithm's efficiency.

Frequent itemset mining is fortunate compared to other the data mining fields like classification and clustering, because the problem can be easily formulated. It suffers, however, from the lack of evaluation method, and from the fact that it is easy to create a database which is suitable to demonstrate arbitrary inefficient algorithm.

In the beginning this was fortunate, because the opportunity of the easy publication attracted many researchers, and this has raised frequent itemset mining as one of the most popular field of the 90s. Unfortunately the high number of published algorithms, the lack of standard terminology, comparisons and theoretical results led to a chaos and resulted in an unexpressed loss of credibility of the field.

An efficient FIM program is an implementation of a widely known and basic FIM algorithm together with many data structure and implementation technique. The base algorithm alone is not competitive with its counterpart that adapts speed-up techniques. This is the reasons we believe that speed-up techniques and the base algorithm are inseparable and there is no point stating anything about the base algorithm without examine the influence of the statement to the techniques. The FIM world is further complicated by the fact that the techniques are not independent of each other, i.e. one reduces, the other increases the influence of a third technique. Actually, the efficiency of almost any arbitrary simple idea, can be verified by carefully choosing the other

[1] G H Hardy. *A Mathematician's Apology* (London 1941).

techniques, which resulted in the large number of publications. To reveal the truth, we have to examine the base algorithms, the techniques alone and their influence on each other, and the theoretical statements have to be verified with a comprehensive set of experiments. Such publication has not born yet.

Our goal is twofold. First we would like to clarify the three most important algorithms of the field, i.e. Apriori, Eclat and FP-growth. We believe that the implementation of the fastest Apriori, Eclat and FP-growth, the most comprehensive set of experiments and the development of a template FIM library entitles us. Second, e we try to provide a notation and terminology that is as concise, compact and coherent as possible. In the literature, often different names have been used. But as the same entities often got different names and there clearly is no convergence in notation moreover, the historical terminonolgy sometimes obscures more than it helps. So we felt obliged to provide a base for further common understanding. For reference, we always provide the most common historical names in apostrophes.

1.1 The arena of FIM algorithms; a short history

The first FIM algorithm AIS was published in the paper that presented the problem itself [3]. 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 [40], DIC [12], Partition [49] and the sampling algorithm [52] 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. [58] published algorithm Eclat and four years later Han et al. [22] presented FP-growth. Since FP-growth was shown not to perform well on sparse datasets, the authors improved their solution and published H-mine [42]. A very efficient Apriori mutant DCI, which adapts hybrid support count was presented by Orlando et al. [36].

2003 was a milestone in the history of frequent itemset mining. The first open FIM competition was organized [16]. Two FP-growth implementations (FP-growth* by Grahne and Zhu [20] and Patricia by Pietracaprina and Zandolin [43]), a modification of DIC [38] and a highly optimized Eclat called lcm by [53] were standing out from the field of competitors. Lcm was further improved and submitted to the second FIMI competition, where a brand new FP-growth implementation by Rácz [45] overtook the first place from FP-growth* and Patricia.

1.2 Common misbelieves

In this section we list some common believes that are false and led the researches into wrong direction.

- *The efficiency is not primarily determined by the number of scan of the database.* In the early era of the FIM many efforts focused on reducing the number of database scan of Apriori. This led to algorithms DIC [12], Partition [49], etc. Although there exists no public implementation of these algorithms that outperform the most widespread Apriori [11][9], they are quite favored in textbooks.

- *Algorithm FP-growth do generates candidates, furthermore it generates more candidates than Apriori.* An itemset is called candidate if its support is determined, i.e. a space for counter is reserved in the memory. Actually FP-growth determines the supports of only 1-itemsets, but then it does it recursively in the projected database. For example if item B occurs in a transaction that is projected to item A, then the support of itemset AB is determined.

- *The number of candidates is not the primary factor that determines the efficiency of an algorithm.* We will see that Eclat and FP-growth generates more candidate than Apriori, nevertheless they outperform Apriori most of the cases. The same applies to Apriori and DHP, it is easy to present databases where DHP generates fewer candidates, but runs slower. Analysis that use only these numbers (like [24]) have nothing to do with the real performance. To understand the performance, we have take into consideration the number of candidates, the way they are generated and their support are determined.

- *FP-tree (or trie) is not necessarily a condensed representation of the database.* Worst-case the size of the FP-tree is four times as much as the size of the database. In many cases a simple vector that stores the transactions that are deprived of the infrequent items needs less memory than an FP-tree. See Section 7.2.1 for further details.

- *Support count is always the most time-consuming function in Apriori.* In dense, medium-size databases the candidate generation dominates the run-time.

- *Generating candidates of size two is not the bottleneck of algorithm Apriori.* This might have been true in 1997 when the available memories were much smaller and fewer candidates fit into the memory. With todays' memory capacities this restriction no longer lives. See section 7.2 for more details about the bottleneck of Apriori.

- *The numerous database scan is not the bottleneck of algorithm Apriori.* We know that Apriori scans the database as many times as the size of the largest candidates. The time required by the I/O operations is only a small fraction of time required by the support count and almost never dominates the run-time of Apriori.

1.3 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 [54] 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 algorithm required by the new model, the reader is referred to [33]. 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.*

1.3.1 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.

1.3.2 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.

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 generated 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 [46] 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 an other 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.

Chapter 2

The Frequent Itemset Mining Problem

Let $\mathcal{I}$ be a set of uninterpreted symbols called **items**. Any subset $I \subseteq \mathcal{I}$ is called an **itemset**.

Let $\mathcal{T} = \langle t_1, \ldots, t_n \rangle$ be a sequence of itemsets called **data** (also called as **transaction database**). Its elements $t \in \mathcal{T}$ will be called **data itemsets** or **transactions**[1]. For any itemset $I \subseteq \mathcal{I}$ we define the set

$$\mathrm{cover}_{\mathcal{T}}(I) := \{t \in \mathcal{T} \,|\, I \subseteq t\}$$

of data itemsets containing I as the **cover of** I. The size of the cover

$$\mathrm{sup}_{\mathcal{T}}(I) := |\,\mathrm{cover}_{\mathcal{T}}(I)|$$

is called **support**. Given a lower support threshold minsup called **minimum support**, the set

$$F_{\mathcal{T},\mathrm{minsup}} := \{I \subseteq \mathcal{I} \,|\, \mathrm{sup}_{\mathcal{T}}(I) \geq \mathrm{minsup}\}$$

is called the set of **frequent itemsets**.

The **frequent itemset mining (FIM) task** then is, given data $\mathcal{T}$ 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 $\frac{|\,\mathrm{cover}_{\mathcal{T}}(I)|}{|\mathcal{T}|}$ 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 $\frac{\mathrm{sup}_{\mathcal{T}}(I)}{|\mathcal{T}|}$ by $\mathrm{freq}_{\mathcal{T}}(I)$ and $\frac{\mathrm{minsup}}{|\mathcal{T}|}$ by minfreq.

[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 transaction (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 .

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 $I' \in S$ 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 $\mathfrak{I}$ that is frequent. An itemset I is **closed** [41][57] if there exist no proper superset of I that has the same support as I.

Corollary 2.0.1 *All maximal frequent itemsets are closed.*

Definition 2.0.2 *The* ***negative border*** *of a set of itemsets F (denoted by $NB(F)$) contains the itemsets that are not elements of F, but all their proper maximal subsets are in F. Formally*

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

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

Example 2.0.3 *Let $\mathfrak{I} = \{A, B, C, D\}$ and $F = \{\emptyset, A, B, C, AB, AC, \}$. Then $NB(F) = \{BC, D\}$.*

Definition 2.0.4 *Let $\prec$ denote a total order on $\mathfrak{I}$. The ℓ-element* ***prefix*** *of itemset I ($\ell \leq |I|$), which is denoted by P_I^ℓ, is the ℓ-element subset of I that contains the ℓ 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 2.0.5 *Let $\mathfrak{I} = \{A, B, C, D, E\}$ and $\prec$ denote the alphabetic order over $\mathfrak{I}$. Here, $P_{ABC}^2 = AB$ and $P_{BDE}^1 = B$.*

Definition 2.0.6 *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 $\mathfrak{I}$. Formally:*

$$NB^{\prec}(F) := \{I | I \notin F \textit{ and } P_I^{|I|-1} \in F, Q \in F, \textit{ where } P_I^{|I|-1} \prec Q \prec Q' \tag{2.1}$$

$$\textit{for all } Q' \subset I \textit{ such that } |Q'| + 1 = |I|, Q' \neq P_I^{|I|-1}, Q' \neq Q\}. \tag{2.2}$$

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

Example 2.0.7 *Let $\mathcal{I} = \{A, B, C, D, E\}$, $F = \{\emptyset, A, B, C, AB, AC\}$ and for any itemsets of the same size I, J let $I \prec J$ if I lexicographically precedes J. Then $NB^{\prec}(F) = \{ABC, BC, D\}$.*

Corollary 2.0.8 *For any itemset $\mathcal{I}$, $F \subseteq 2^{\mathcal{I}}$ and $\prec$ we have*

$$NB(F) \subseteq NB^{\prec}(F).$$

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

Definition 2.0.9 *Let $\mathcal{T}$ be a transaction database over $\mathcal{I}$. The I-projected database of $\mathcal{T}$ (which is denoted by $\mathcal{T}|I$) consists of the elements of $\mathcal{T}$ that contain I.*

The sequence of transactions that are not contained in the I projected database is denoted by $\mathcal{T}|\overline{I}$ and called the **complement of the projected database**. Obviously, no element of $\mathcal{T}|\overline{I}$ 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 $\langle ABC, AE, BCE\rangle|\{\overline{AE}\} = \langle ABC, BCE\rangle$.

Chapter 3

Base Algorithms

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 by their own 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 a different data structure. Nevertheless, we distinguish them for historical reasons.

3.1 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 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 [30] and CBW [51]). 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 $|I|$.

3.2 Breadth-first, iterative vs. depth-first, recursive algorithms

Apriori is an iterative, breadth-first algorithm. In the iteration step ℓ it determines the frequent itemsets of size ℓ. Eclat and FP-growth, on the contrary, are recursive, depth-first-like algorithms. Given a set of frequent itemsets (denoted by F_P^+) with a common maximal proper prefix P and of size $|P|+1$, it takes the itemsets $I \in F_P^+$ one-by-one and determines the frequent itemsets whose prefix is I. The search is done recursively; initially the emptyset 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 ℓ is equal to the negative border of frequent itemsets found till the iteration step ℓ. In Eclat and FP-growth the set of candidates in the next recursive step belonging to itemset $I \in F_P^+$ is the subset of the order-based negative border of F_P^+ whose element's prefix is I (formally $\{I'|I' \in NB^{\prec}(F_P^+)$ such that $P_{I'}^{|I'|-1} = I\}$). 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 2.0.8 that the number of candidates is never less in Eclat and FP-growth than in Apriori.

> **historical remark:** FP-growth has been viewed as an algorithm operation on the data trie by its inventors [22, 21, 23] that is augmented by so-called

"header lists" that sequentially link nodes with the same item label. From this perspective, FP-growth looks like a depth-first algorithm that is quite different from Eclat. We argue here (and it was also noted by Goethals [19]), that this is a queer view on the algorithm, and that actually the main data structure is the set of prefixes (i.e., the "header lists"), while the data trie is nothing else than a means to compute the relation startsWith efficiently. Then, the effective difference between Eclat and FP-growth is that FP-growth works on prefixes, while Eclat works on single transaction. That means, that FP-growth can take advantage from data that can considerably compressed by a trie, while it has to pay the overhead of a more complex intersection method that has to take into account the relation startsWith.

3.3 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 three most important FIM algorithms. Each algorithm is first described at semantic level, and then we check what kind of data structure supports best the functions of the algorithm. Then we give a comprehensive description of the techniques.

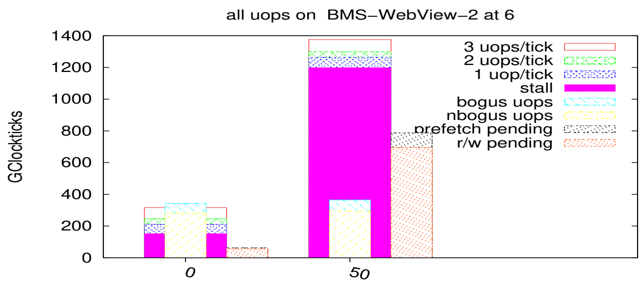
We call a technique **memory safe** if it never increases the memory need of the algorithm significantly (let us say more than 25%). A memory-safe technique is called **strictly memory-safe** if it required the same or less amount of memory than the algorithm without the technique in *all* test databases 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.

3.4 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.



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.

3.5 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 [47] and TreeProjection [1]), we begin with the description this central data structure.

The data structure **trie** was originally introduced by de la Briandais [14] and Fredkin [15] to store and efficiently retrieve words of a dictionary. Mueller [35] 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 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 3.1 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.

A trie that stores all subsets of a given set is quite unbalanced. The following picture 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. For example, in FP-growth the child linked-trie is converted to parent-linked tree after all itemsets are inserted.

3.5.1 The representation of the list of edges

The list of edges can be represented in many ways. The representation used in the algorithms greatly affects both run-time and memory-need. Let us assume that we have

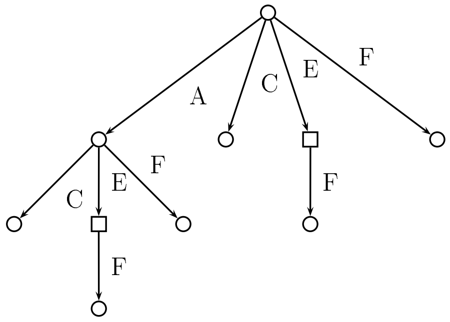


Figure 3.1: Example: a trie that stores sets {A},{C},{F},{AC}, {AF}, {EF}, {AEF}

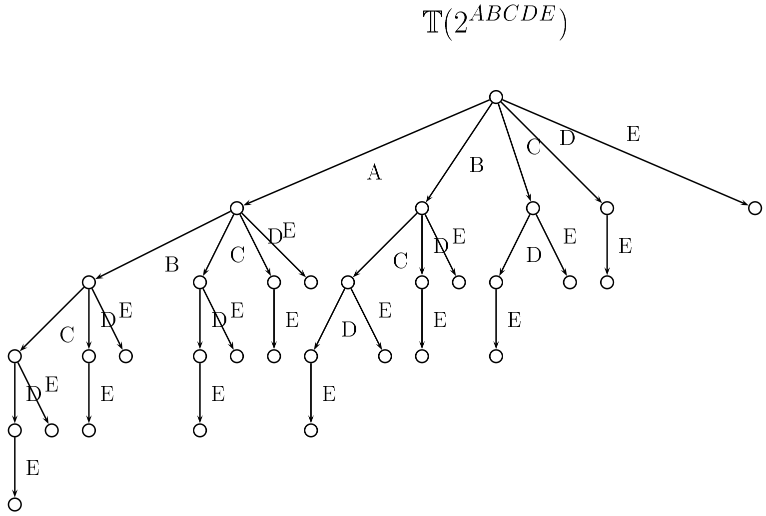


Figure 3.2: Example: a trie that stores all subsets of itemset {ABCDE}

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_{min} and l_{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 list) 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 $2n < l_{max} - l_{min} + 1 + 1$, 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.

3.5.2 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 3.3).

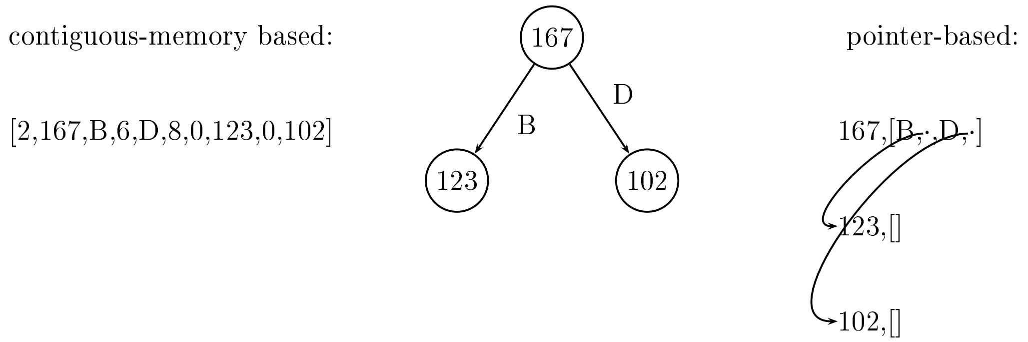


Figure 3.3: different representations of the same trie

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 $ns_i + ns_{ov} + (n-1)s_i + (n-1)s_p$, where n is the number of nodes in the trie, s_{ov} 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 $26n$ bytes in a Pentium 4 and $40n$ 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 $s_{ov} + n(s_i + s_i + 2s_i) \approx 4ns_i$ which is $16n$ in a Pentium and Opteron as well. Note, that we assume that the size of the vector that stores the trie is not greater that 2^{8s_i}, 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.

3.5.3 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.

Chapter 4

Algorithm Apriori

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 [2] and Mannila et al. [32] independently at the same time. Their cooperative work was presented in [4].

The algorithm 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. 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 4.0.1 *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 1 algorithm Apriori

Require: D : database over the set of items $\mathfrak{I}$,
minsup support threshold
Ensure: F : the set of frequent itemsets

```
ℓ ← 1
C_ℓ ← 𝔍
while |C_ℓ| ≠ 0 do
  support_count( D, C_ℓ )
  for all c ∈ C_ℓ do
    if c.support ≥ minsup then
      F_ℓ ← c
    end if
  end for
  C_{ℓ+1} ← candidate_generation( F_ℓ )
  ℓ ← ℓ + 1;
end while
F = ⋃_{j=1}^{ℓ} F_j
```

requirement must be the union of two different ℓ-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 ℓ-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.

Algorithm 2 candidate_generation

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

```
for all {i_1, ... i_{ℓ-1}, i_ℓ}, {i_1, ... i_{ℓ-1}, i'_ℓ} ∈ F_ℓ such that i_ℓ ≺ i'_ℓ do
  c ← {i_1, ... i_{ℓ-1}, i_ℓ, i'_ℓ}
  if all_ℓ_subsets_are_frequent(c, F_ℓ) then
    C_{ℓ+1} ← c
  end if
end for
```

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.

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 [**?**] a hash-tree was proposed for this purpose. The first trie-based Apriori implementation is reported Pasquier et al. [41]. For the sake of correctness we have to mention that a year earlier algorithm DIC [12], which is an extension of Apriori, also used trie to store the candidates. Independent from each other Borgelt, Goethals and Bodon (and maybe several others) published the first open-source Apriori implementations. In [7] 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 [37]. 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.

The fact that trie-based solution provides results in a faster Apriori than prefix-array based solution in all cases, does not imply that trie is the best choice. Prefix-arrays are exploited in the initial phases of DCI, therefore we have to compare the performance of the two data structures at smaller candidates' sizes. Our experiment – in which we terminated the algorithms as soon as the candidates reached a certain size – showed that trie-based solution is always faster than prefix-array based solution at any candidates' sizes.

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

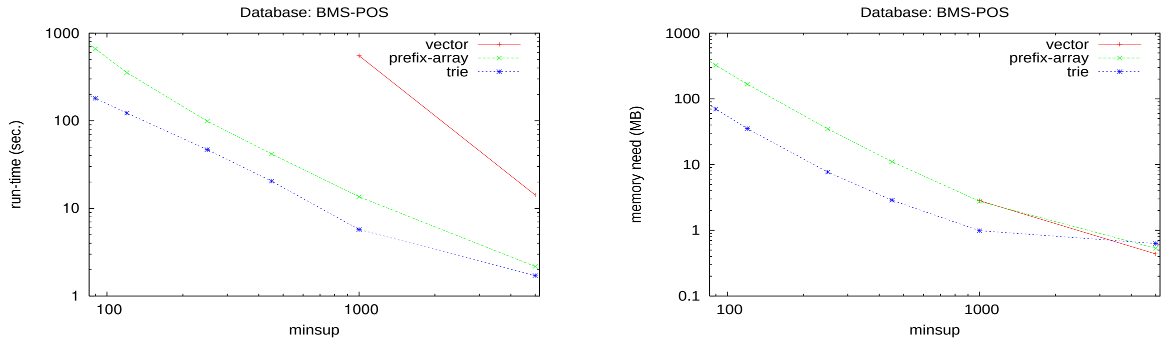


Figure 4.1: Comparison of simple vector, prefix-array and trie-based solution for storing the candidates in Apriori

4.1 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.

4.1.1 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.

4.1.2 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.

4.1.3 Candidate Generation

Here we make use of an other nice feature of tries; ℓ-itemsets, that share the same $(\ell-1)$-prefix, are represented by sibling leaves. 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 ℓ-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.

4.2 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 resources, but more importantly they makes the processor stall, which ruins efficiency. 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. 4.3, 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 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.

We see, that the two implementations perform approximately the same number of

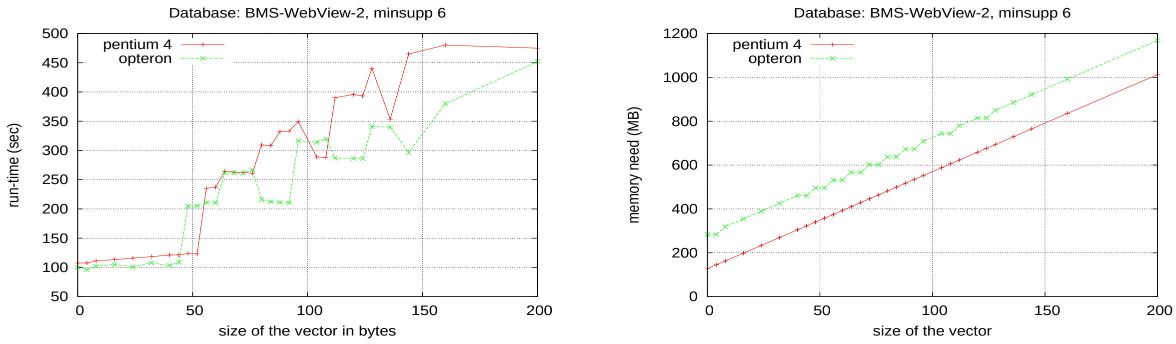


Figure 4.2: The influence of node's size of the trie on run-time and memory need

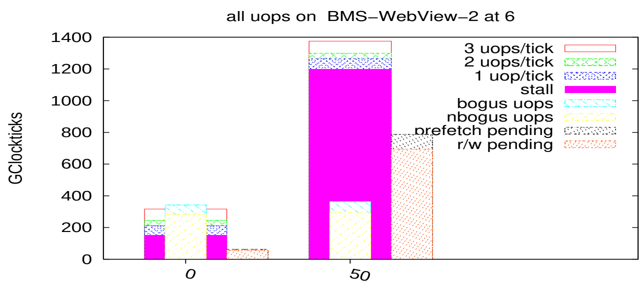


Figure 4.3: Complex hardware-friendliness diagram of two implementations

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.

4.3 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,
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

[1]Actually we used a list of medium-size blocks instead of one big block in our implementation.

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 4.1 shows some experiments concerning this design detail.

database	*minsup*	homogeneous trie	inhomogeneous trie	inhomogeneous trie with block allocator
T40I10D100K	220	670	653	518
pumsb	32600	184	161	133
retail	3	96	208	44
T10I5N1KP5KC0	6	21	21	18
T30I15N1KP5KC0	360	622	557	395

run-time (sec.)

database	*minsup*	homogeneous trie	inhomogeneous trie	inhomogeneous trie with block allocator
T40I10D100K	220	342	128	128
pumsb	32600	19	14	14
retail	3	939	327	327
T10I5N1KP5KC0	6	553	196	196
T30I15N1KP5KC0	360	296	204	203

memory need (MB)

Table 4.1: Inhomogeneous trie and a special block-allocation technique

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

4.4 Removing Dead-end Branches

Frequent itemsets of size ℓ 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 whose all 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 with one exception) 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 4.4 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.

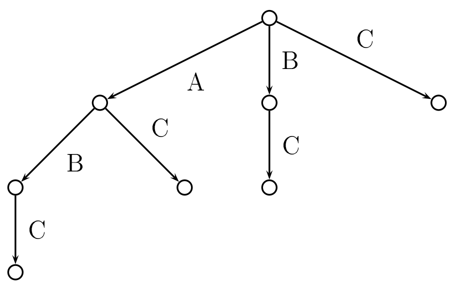


Figure 4.4: Example: removing dead-end branches

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 following property.

Property 4.4.1 *For a given depth d, the depth-first ordering of the nodes' representation at depth d is the same as if we lexicographically order these representations, where the order used in the lexicographical ordering corresponds to the ordering of the trie and the lexicographic ordering of the presentations is based on a global item ordering.*

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 just deteriorate the branch prediction facility of the processor and thus the run-time as well. 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. Figure 4.5 shows the ratio of run-time and memory need of Apriori that does not use the dead-end pruning and the Apriori that does.

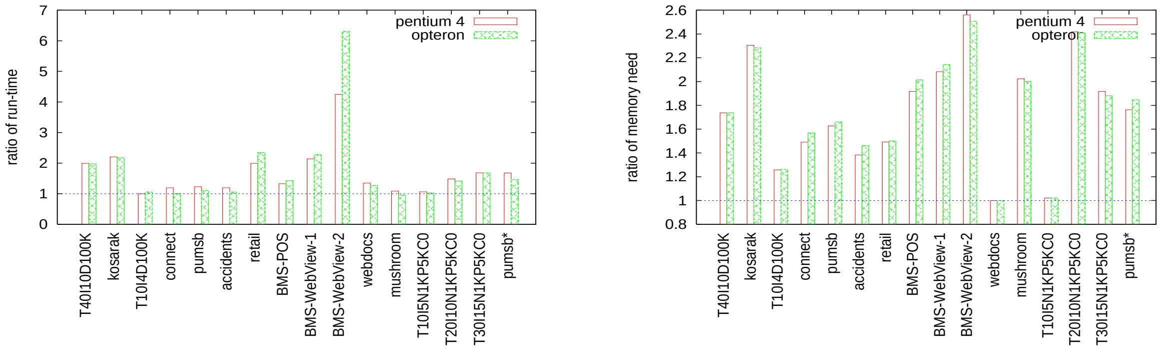


Figure 4.5: Deadend pruning (ratio of run-times and memory-needs)

Some hardware friendliness diagrams is given in Figure 4.6.

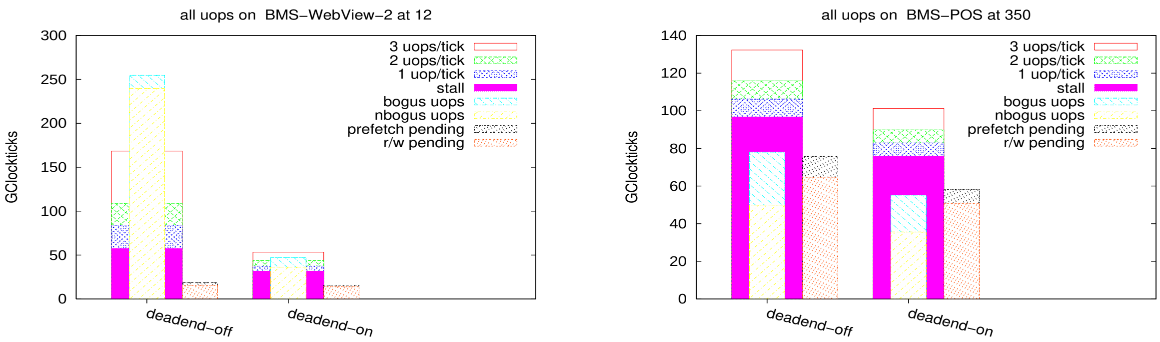


Figure 4.6: Hardware friendliness diagrams of Aprioris with and without dead-end pruning

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

The problem of traversing dead end paths was also considered in [10] as an influence of our earlier paper [6]. The author of [10] has chosen an other 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 33). 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 [10]). 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.

4.5 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, 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 3.5.1). 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.

4.5.1 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 4.7.

Some hardware friendliness diagrams is given in Figure 4.8.

Observations based on **all** the tests are the following:

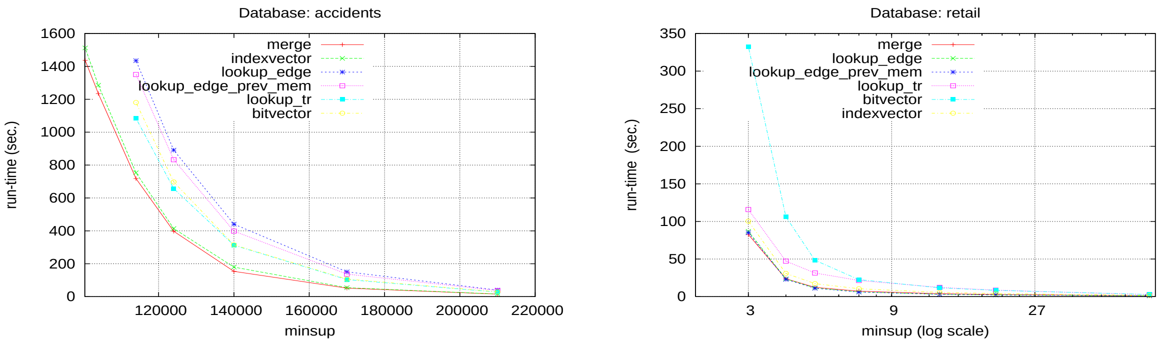


Figure 4.7: Routing strategies in the case of ordered edgelist representation

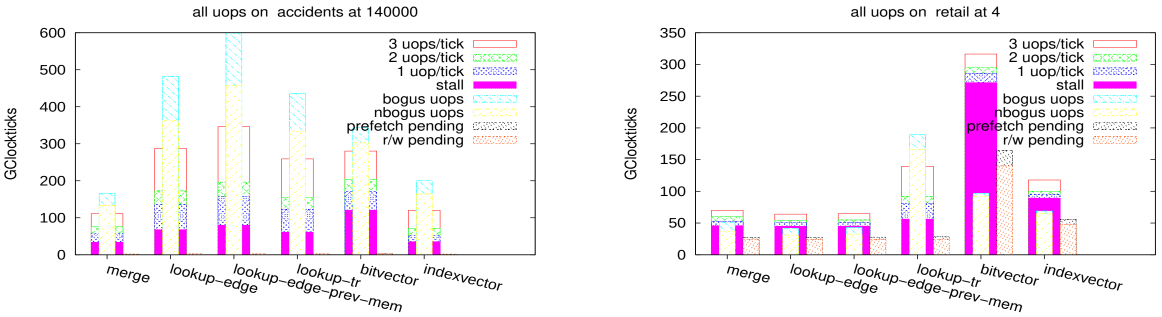


Figure 4.8: Hardware friendliness diagrams of some routing strategies

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.

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.

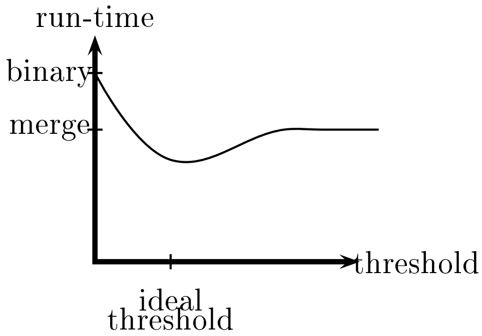
4.5.2 Can we speed up binary search-based routing strategies?

The reasoning about the execution time of the linear and binary search brings up the possibility of improving the performance of binary-search based routing strategies, i.e. `lookup_edge` and `lookup_trans`. We know that under a threshold the linear search is faster, and above this threshold the binary search. The value of this threshold depends on the processor features (cache sizes, prefetching mechanism, length of the pipeline, etc.), the way the binary search is coded and the type of the elements. In our experimental environment (Pentium 4 2.8 Ghz processor – family 15, model 2, stepping 9 –, using `std::lower_bound` for the binary search, the size of a list element is 4 bytes) the threshold is around 14.

The pure binary search-based approaches can be speed up if it is substituted by a hybrid solution which chooses between linear and binary search according to the length of the lists (length of t' in the case of `lookup_trans`).

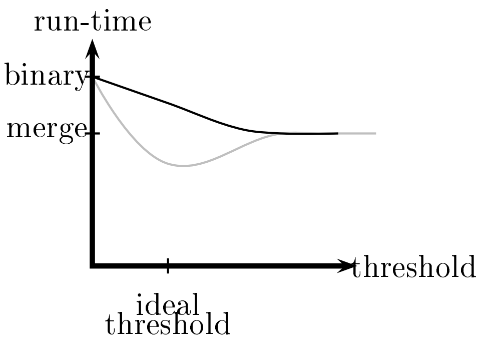
In our implementation the threshold is set by a template parameter. Notice that as soon as a linear search is selected, then the threshold check will prefer linear search in the current node and in the descendants as well. Therefore in our implementation we switch to `merge` routing strategy to avoid the threshold condition check and improve the efficiency of branch prediction. The larger the threshold the sooner we switch to `merge`.

In the next figure we plotted our expectation of run-time in the function of the threshold.



When the threshold is zero, then always binary search is employed, when it is more than the number of frequent item then always linear search is used, which results practically in the merge algorithm. The fastest solution is expected when the threshold equals to the ideal threshold.

In reality we get a totally different characteristic, which applies in **all** databases. This is plotted in the next figure.



The runtime decreases as the threshold increases even if we cross the ideal threshold. It seems that the sooner we switch to `merge` routing strategy the faster algorithm we get.

To resolve the contradiction and understand the observation we have to examine the characteristic of the data. The next two figures show some distributions of the steps between two matches in the transaction. Zero step belongs to the case when the first item in the t' is the same as the label of the first edge. With database `T10I5N1KP5KC0` the `merge` was 2.5 times faster than `look_up_trans` which is not far from the typical case. The smallest advance was just 20% with database `kosarak`.

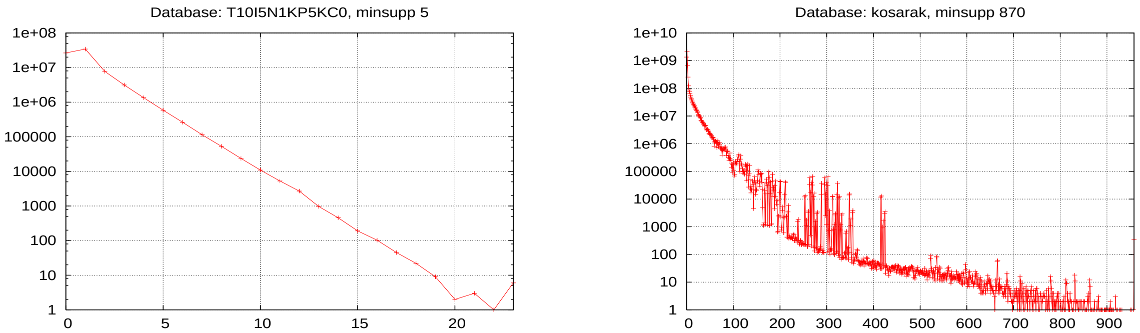


Figure 4.9: distribution of distances between consecutive matches

We can see that the distribution is quite steep (notice the logarithmic scale). The ideal threshold (14 in our environment) is equal to the 0.999989 and 0.645 quantile respectively. This means that although the size of t' might be long the distances between consecutive matches are quite small in most of the cases it is smaller than the advantage of a binary search comes into play. Thus linear search (`merge`) is the fastest most of the case and the extra condition check just ruins the efficiency of branch prediction. The larger threshold we set the sooner we switch to `merge` and the fewer unnecessary conditions are evaluated.

Also notice that the ratio of the number of consecutive steps under 14 to the number of all matches has a strong correlation with the efficiency of speed-up `lookup_trans`. The less this value the more efficient this routing strategy is.

Although in this section we neither presented a new approach neither speeded up the existing routing strategies, we believe that this rationale shows a illuminating example how deep we have to dig down to find the true reasons. To understand the behavior of the routing strategies and their boundaries we have to consider (1.) theoretical possibilities, (2.) hardware friendliness and (3.) the specialties/characteristics of the application domain.

4.5.3 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.

Some hardware friendliness diagrams is given in Figure 4.11.

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.

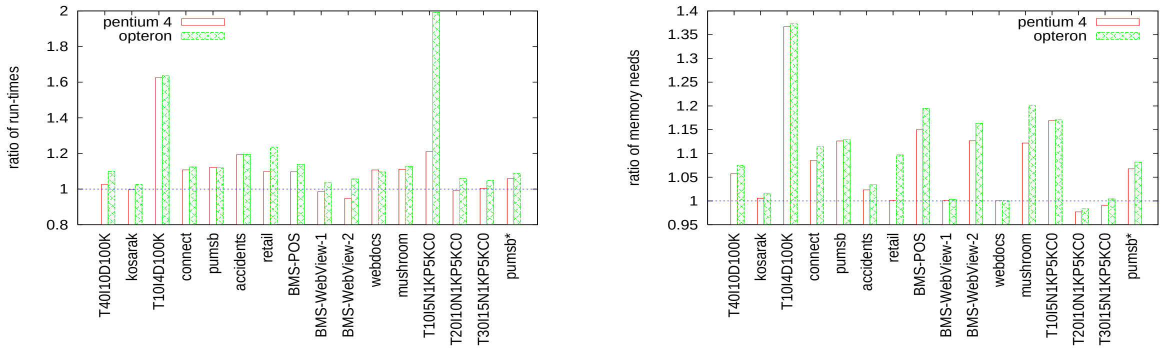


Figure 4.10: Ratio of run-time and memory-need of ordered list-based Apriori compared to hybrid edge representation-based Apriori

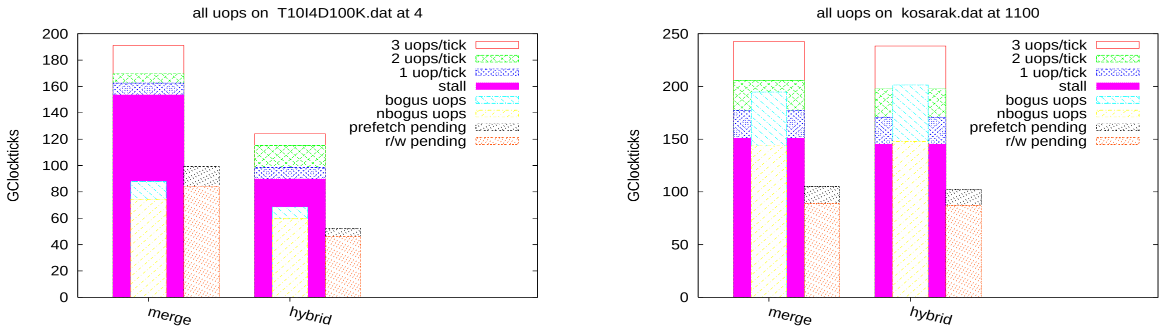


Figure 4.11: Hardware friendliness diagrams of routing strategies `merge` and `hybrid`

4.6 Determining the support of 2-itemset candidates

Using a trie seems unnecessary and complicated when determining the support of 2-itemset candidates [50]. A simple array also does the trick. We know that the elements of each candidate are frequent items coded by 0,1,2... and every pair that consists of two frequent items is a candidate.

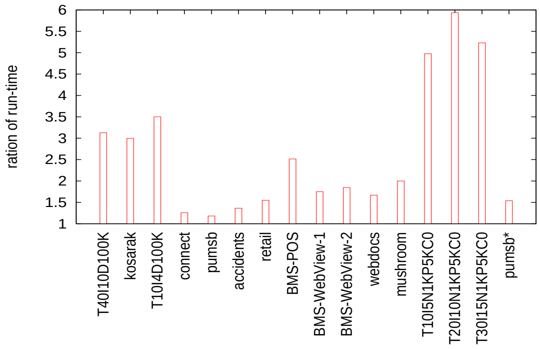
The array stores the counters that are initialized to 0. Counter of itemset $\{i_1, i_2\}$ (we can assume $i_1 < i_2$) is at index $i_1, i_2 - i_1 - 1$ of the array (i.e. we us an upper-triangle array). Notice that theoretically this solution is the same as trie based solution where offset-index representation is used with offset equal to 0. Array-based solution (also called direct count), however, spares the recursive step.

It is not necessary to allocate a counter for each candidate. In *online candidate generation* [19] we allocate a counter only when the pair actually occurs in a transaction. I databases, that contains many frequent items and most 2-element candidates do not even occur, this solution reduces memory need significantly. In this solution the rows of the array are empty at the beginning and item i_2 with counter 1 is added to row i_1 when itemset $\{i_1, i_2\}$ occurs in the first time. So the elements of the array are actually pairs. For the sake of quick insertion the rows are sorted according to the items.

> **historical remark:** Theoretically the same idea with some minor changes was reinvented by Woon et al. [56]. First, they used a trie (called SOTrieT) instead of an array. This is an unnecessary and over-complicated solution, but most importantly it requires more memory, than a simple vector of vectors. Second, the frequent items and the frequent pairs are found in the same iteration. This awkward solution also suffers from a very bad memory usage. All pairs that occur in a transaction require a counter even if they contain infrequent items. For these reasons we use the vector-based on-line candidate generation method in our experiments.
>
> A hash-based technique DHP was proposed by Park et al. [40] in order to reduce the number of candidates in particular the number of candidates pairs. When determining the frequent items an other counter vector is also maintained. Counter at index i belong to the itempairs that has hash-value i. During the first scan at each transaction t the hash-value of all subsets of t of size two are calculated and the corresponding counters are increased. After the first scan, a candidate itempair is generated only if counter determined by the hash-function is greater than minsup.
>
> The problem of this solution is the lack of a universal good hash function. It is easy to find a good hash function if the characteristic of the transaction database is known, but this is not the case. Furthermore a hash-function that works well at a database with a given support threshold performs poorly at

> the same database with an other support threshold. We believe that the sore spot (and actually the applicability) of this technique is the hash-function, which was never analyzed in the literature, i.e. no hash function was proposed that works well at many databases with many support threshold.

The next figure shows ratio of run-time and memory-usage of the online and the triangular array-based support count method.



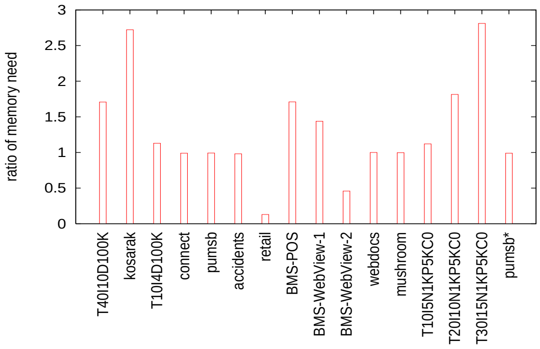


Figure 4.12: Ratio of run-time decrease and memory-need increase of online and static support count of 2-itemsets

The disadvantage of online support count concerning run-time is significant at high support thresholds, especially when the size of the maximal frequent sets is two. As lowering the threshold the difference get insignificant when it is compared to the total run-time.

4.7 Determining the support of 3-itemset candidates

The array-based technique can be naturally generalized to candidates of size ℓ by using an ℓ-dimension array of size $\binom{|L_1|}{\ell}$, where L_1 denotes the set of frequent items . This solution was chosen in the newest implementation of algorithm kDIC [38][44]. The drawback of the array-based solution is straightforward, i.e. it requires $4 \cdot \binom{|L_1|}{3}$ bytes of memory, which can be quite large. For example in the case of database `retail` with support threshold equal to 3 the L_1 is 12889, therefore the array requires 1332 Tbyte! Actually in the case of 9 out of out 16 test databases (with minsup where our Apriori is able to complete FIM task within reasonable time) the array needs more than 2Gbyte of memory. This is not a safe solution.

Nevertheless, the array-based solution for candidates of size three speeds up Apriori in many cases. A hybrid solution that chooses array-based technique if the number of frequent items is small (let say smaller than 700) and trie-based solution otherwise, seems to be a good solution.

Chapter 5

Algorithm Eclat

Chapter 6

Algorithm FPgrowth

Chapter 7

Techniques for improving efficiency

The base algorithms can be greatly improved by algorithmic, data structure and implementation related techniques. The literature is rich in this topic. In this section we investigate the most important technique putting emphasize on the relationship between them.

7.1 Pruning equisupport extensions

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

Property 7.1.1 *Let* $X \subset Y \subseteq \mathfrak{I}$. *If* $\sup(X) = \sup(Y)$, *then* $\sup(Y \cup Z) = \sup(X \cup Z)$ *for any* $Z \subseteq \mathfrak{I}$.

This property holds for all $Z \subseteq \mathfrak{I}$, nevertheless we restrict our attention to itemsets $Z \subseteq \mathfrak{I} \setminus Y$.

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 $\sup(Y)$. 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** [5] if there exist no proper subset of X with the same support.

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 $X \cup Z$ [19].

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 support of a children of itemset P, we check at the infrequent removal phase if their support are equal to $\sup(P)$. 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 $Y \backslash X \prec z$ 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 7.1.2 *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.

7.2 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 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 7.2 and 7.3). 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.

database	*minsup*	counting support	generating candidate	input sort recode	infrequent removal	frequent pair mining
T40I10D100K	3 000	14		53		31.0
kosarak	7 000	21		69		1.9
T10I4D100K	150	68		24		4.1
connect	65 000	73		25		1.4
accidents	210 000	77		21		1.4
pumsb	41 000	97		2.6		
retail	65	64		22		10.6
BMS-POS	5 000	38		56		3.8
BMS-WebView-1	39	67	9.1	21		0.7
BMS-WebView-2	30	56	14.0	23.3	0.5	2.7
webdocs	700 000	1		93		
mushroom	1600	95	1.3	3		
T10I5N1KP5KC0	500	8		67		22.0
T20I10N1KP5KC0	2 000			76		17.8
T30I15N1KP5KC0	1 300			25		73.0
pumsb*	23 000	56		41		2.5

high support threshold

database	*minsup*	counting support	generating candidate	input sort recode	infrequent removal	frequent pair mining
T40I10D100K	220	90	6.5	0.6	0.6	
kosarak	860	94	2.0	2.0		
T10I4D100K	3	33	63	0.7	1.0	
connect	43 100	96	3.1	0.5		
accidents	100 500	98			1.4	
pumsb	32 600	96	1.6		1.9	
retail	3	29	63	1.3	1.8	0.7
BMS-POS	67	84	13	0.8	0.5	
BMS-WebView-1	33	44	54		0.7	
BMS-WebView-2	4	12	83		1.4	
webdocs	200 000	77		21.0		1.3
mushroom	250	86	12.5			
T10I5N1KP5KC0	4	53	39	1.8	0.7	0.8
T20I10N1KP5KC0	90	84	13.0	1.7		0.6
T30I15N1KP5KC0	300	84	12.2	1.6		0.8
pumsb*	13 000	99		0.5		

low support threshold

Table 7.1: The distribution of run-time of Apriori's methods in %

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.

database	number of transactions	number of items	average size of the transactions
mushroom	8 124	119	23.0
pumsb*	49 046	2 088	50.4
pumsb	49 046	2 113	74.0
BMS-WebView-1	59 602	497	2.5
connect	67 557	129	43.0
BMS-WebView-2	77 512	3 340	4.6
retail	88 162	16 470	10.3
T10I4D100K	100 000	870	10.1
T40I10D100K	100 000	942	39.6
T10I5N1KP5KC0	193 373	3 950	10.3
T20I10N1KP5KC0	197 440	4 408	20.2
T30I15N1KP5KC0	199 095	4 599	30.0
accidents	340 183	468	33.8
BMS-POS	515 597	1 657	6.5
kosarak	990 002	41 270	8.1
webdocs	1 692 082	5 267 656	177.2

Table 7.2: Some statistics about the databases

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 (`T10I5N1KP5KC0`, `T10I4D100K`) 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 7.2.4) 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 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.

database	minsup	number of frequent items	number of frequent itempairs	number of frequent itemsets	size of the maximal frequent itemset	average size of the frequent itemsets	average size of the filtered transactions
webdocs	700 000	8	14	34	4	2.0	
T20I10N1KP5KC0	2 000	472	0	473	1	0.99	6.9
kosarak	7 000	93	249	772	6	2.6	3.3
T40I10D100K	3 000	486	307	794	2	1.38	33.5
connect	65 000	15	72	916	7	4.2	14.8
pumsb*	23 000	34	126	1165	8	4.0	20.6
BMS-POS	5 000	145	408	1171	5	2.5	5.3
T10I5N1KP5KC0	500	1494	90	1655	6	1.1	7.7
accidents	210 000	21	125	1685	8	4.17	17.0
T30I15N1KP5KC0	1 300	1667	3	1671	2	1.0	21.9
retail	65	2895	4958	11684	6	2.1	8.2
T10I4D100K	150	767	5549	19127	10	3.39	10.0
pumsb	41 000	25	249	36811	11	5.8	23.5
mushroom	1 600	43	380	53952	15	7.1	19.2
BMS-WebView-1	39	363	3802	69370	12	4.8	2.5
BMS-WebView-2	30	2122	6052	194262	15	6.5	4.4
high support threshold							
webdocs	200 000	195	1 596	58 297	10	5.0	
accidents	100500	32	408	160 874	12	6.7	22.1
pumsb*	13 000	63	900	1 293 829	17	8.8	31.8
T10I5N1KP5KC0	4	3 924	49 0812	1 600 477	14	3.7	10.3
kosarak	860	1 437	11 460	3 578 574	19	8.36	6.0
pumsb	32 600	36	536	6 061 656	20	10.0	31.6
T10I4D100K	3	869	220 988	6 169 854	14	4.43	10.1
mushroom	250	82	1 684	9 944 484	17	8.9	22.6
T40I10D100K	220	901	104 161	10 174 500	20	8.48	39.6
connect	43 100	34	483	11 809 442	19	10.1	30.6
T30I15N1KP5KC0	360	3 489	13 037	15 747 841	20	9.7	29.0
BMS-POS	67	884	37 377	16 037 252	13	6.4	6.5
T20I10N1KP5KC0	90	4 021	86 776	16 964 579	20	8.3	20.1
retail	3	12 889	433 297	20 647 332	20	7.9	10.2
BMS-WebView-2	4	3 185	106 070	60 193 074	23	9.8	4.6
BMS-WebView-1	33	372	5 844	69 417 074	25	11.5	2.5
low support threshold							

Table 7.3: Some statistics about the frequent itemsets

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*.

7.2.1 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 cacher**. 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 cachers. 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 cacher.

The most famous Apriori implementation [11] uses trie and in our previous implementation we have used a red-black tree (denoted by RB-tree). In an RB-tree cacher each node stores a transaction. Due to the success of Patricia-trees in FP-growth based algorithms [43] we also tested this solution.

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

database	*minsup*	ordered-array	RB-tree	trie	patricia
kosarak	48 000	1.74	0.93	0.83	0.84
	840	212.8	2.79	2.26	1.68
accidents	3.96	3.49	1.13	0.90	0.8
	100 500	94.4	1.88	1.48	1.23
BMS-POS	5 000	126.09	1.40	0.89	0.65
	67	153.28	1.55	1.10	0.72
webdocs	700 000	27.05	25.92	25.08	24.98
	200 000	1030.25	38.05	45.20	31.89

run-times

database	*minsup*	ordered-array	RB-tree	trie	patricia
kosarak	48 000	0.69	0.69	0.65	1.93
	840	28.02	32.16	72.55	19.92
accidents	210 000	2.91	2.73	1.45	1.91
	100 500	21.68	19.15	13.00	9.2
BMS-POS	5 000	15.09	17.86	24.84	10.60
	67	23.31	22.87	38.21	13.12
webdocs	700 000	48.66	48.66		
	200 000	278.10	280.98	934.01	264.84

memory need

Table 7.4: Transaction caching with different data structures

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 $5 \cdot 4$ 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 ℓ-itemset transaction in a trie, ℓ 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 7.3.

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 7.1) we have used a Patricia-tree as a transaction cacher.

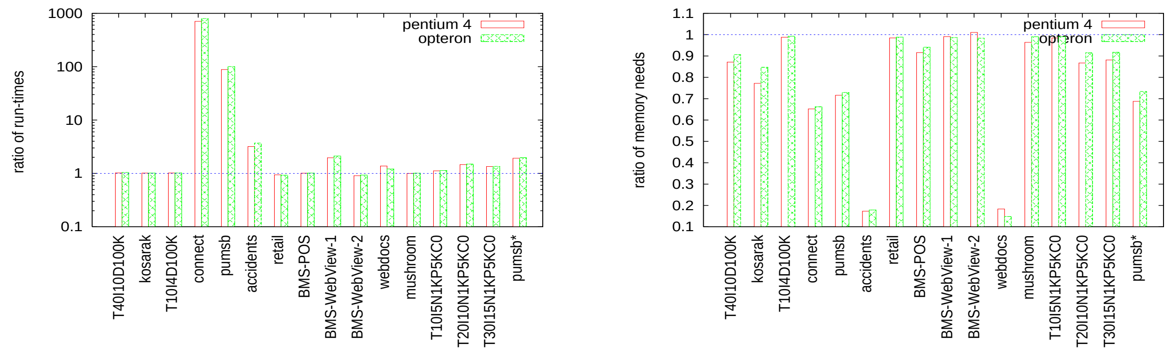


Figure 7.1: Caching the transactions

Some hardware friendliness diagrams are given in Figure 7.2.

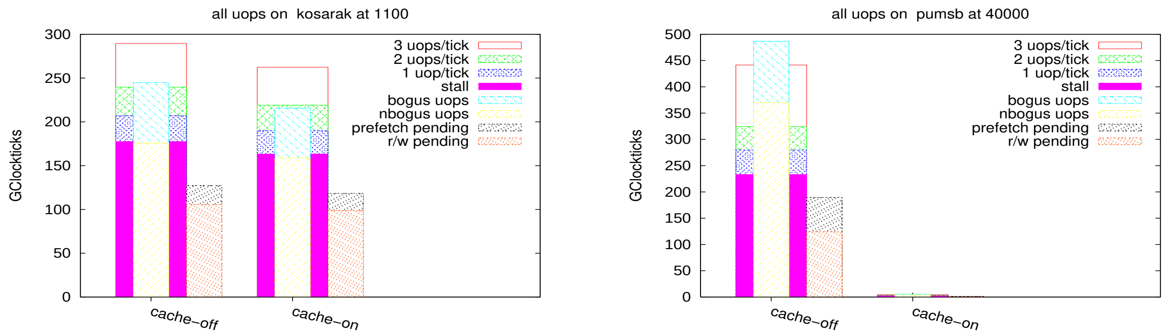


Figure 7.2: Hardware friendliness diagrams of Aprioris with and without transaction caching

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`, `T20I10N1KP5KC0.25D200K`, `pumsb*`). Due to the fast 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.

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.

7.2.2 Support count of Christian Borgelt

When the transactions are stored in a trie or in a Patricia tree then an other support count technique can be applied. This clever idea was already mentioned in [11] and was sketched in [9]. 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 7.3 we will see, that this is a drawback since the two tries prefer different orderings.

Unfortunately, the algorithm is not detailed in [9], 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.

Algorithm 3 BORGELT_SUPPCOUNT

Require: n_c: a node of the candidate trie,
n_t: a node of the transaction trie,
ℓ: number of step from n_c that needs to be done to reach a leaf,
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**
  $n_c.counter \leftarrow n_c.counter + n_t.counter$
else
  for $j = 0$ to n_t.edge_number $- 1$ **do**
    while $i < n_c$.edge_number AND $n_c.edge[i]$.label $< n_t.edge[j]$.label **do**
      $i \leftarrow i + 1$
    end while
    if $i < n_c$.edge_number AND $n_c.edge[i]$.label $\geq n_t.edge[j]$.label **then**
      BORGELT_SUPPCOUNT(n_c, $n_t.edge[j]$.child, ℓ, i)
      if $n_c.edge[i]$.label $= n_t.edge[j]$.label **then**
        BORGELT_SUPPCOUNT($n_c.edge[i]$.child, $n_t.edge[j]$.child, $\ell - 1$, 0)
        $i \leftarrow i + 1$
      end if
    else
      break
    end if
  end for
end if

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 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 [9].

7.2.3 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 ℓ at iteration ℓ.

Further filtering can be applied. To illustrate this imagine that the candidates of size two are AB, AC, BC and DE 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 ℓ if it is not contained in any candidate that occurs in the transaction [9].

A more sophisticated solution was proposed by Park et al. [40]. 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 ℓ candidates of size ℓ 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 ℓ 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 [36] 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 [9] 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.

7.2.4 Equisupport pruning

We have seen that prefix equisupport pruning can be applied in all bottom-up FIM algorithms, where candidates are generated on the basis of prefixes. From a 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 7.2.1 *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 7.3 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.*

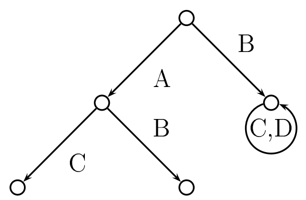


Figure 7.3: Example: removing equisupport leaves

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 depths 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 7.3 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 4.4). 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 7.4. The frequent pairs are AB, AC, AD, BC, BD, CD, CE and $\sup(A) = \sup(AD)$, $\sup(B) = \sup(BD)$, $\sup(C) = \sup(CD)$. 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.

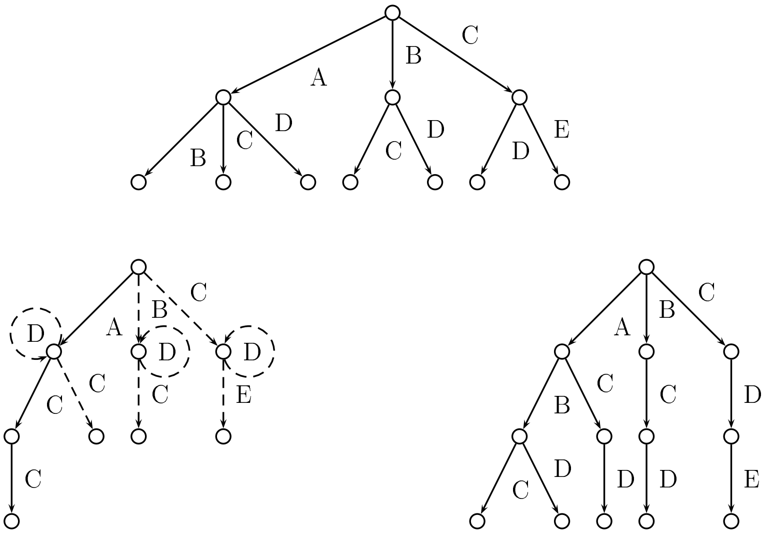


Figure 7.4: Example: distinguishing dead-end edges when equisupport pruning is applied

Notice that when determining which candidates are contained in transaction $\langle ABCDEFGH \rangle$, 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 4.2) 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**.

7.2.5 Level 2 equisupport pruning

It seems contradictory to restrict our equisupport pruning to prefixes in the case of Apriori since all subsets together with the supports are available and the equisupport Property 7.1.1 (see page 47) 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 equisupport 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 $\sup(BC) = \sup(C)$. 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 7.5.

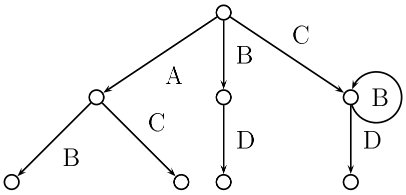


Figure 7.5: Example: nonprefix equisupport pruning ruins ordering

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 7.2.2 *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 $\sup(X \cup z) = \sup(X)$, then $\sup(Y \cup z) = \sup(Y)$.*

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 ℓ-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 7.2.3 *The set of frequent two itemsets are $\{AB, AC, AD, BC, BD\}$ and the only equisupport is $\sup(BC) = \sup(C)$. 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 7.6 depicts the trie before and after the candidate generation. Please keep in mind, that dead-end branch pruning (with the virtual depth modification) is applies during candidate generation.*

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 candidate. Equisupport pruning, however, prevents us from generating ABC as a candidate, and terminates Apriori after the second iteration.

7.2.6 Level 2 equisupport pruning and further dead-end pruning

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

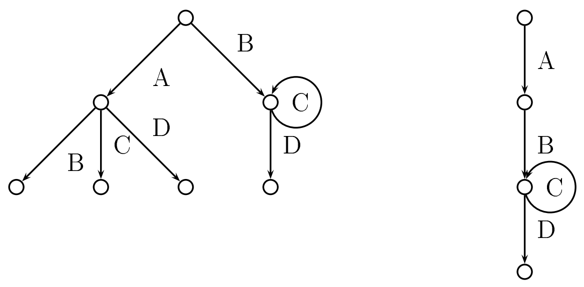


Figure 7.6: Example: special prefix equisupport pruning

Lemma 7.2.4 *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 $Q = \{i_1, i_2, \dots, i_j, \dots i_\ell\}$, is in the equisupport set of itemset $P_Q = \{i_1, i_2, \dots, i_{j-1}\}$. We claim that itemset $Q' = P_Q \cup (P \setminus Q)$ could not have been generated as a candidate at iteration j. If $i_j \prec P \setminus Q$, 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. $\square$

Table 7.5 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 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.

Lemma 7.2.4 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 ℓ-itemset can be a subset of an $(\ell + 1)$-itemset that is generated by the preceding nodes. This corresponds to the property 4.4.1 (see page 31) used in dead-end pruning. We call level 2 pruning together

$P \setminus Q$	Q	i_j			
		A	B	C	D
ABC	D	AD	ABD	ABCD	–
ABD	C	AC	ABC	–	ABCD
ACD	B	AB	–	ABC	ABCD
BCD	A	–	AB	ABC	ABCD

Table 7.5: Illustration of the proof of Lemma 7.2.4

with dead-end pruning presented in this section as *level 3 equisupport pruning.*

Example 7.2.5 *The Figure 7.7 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*

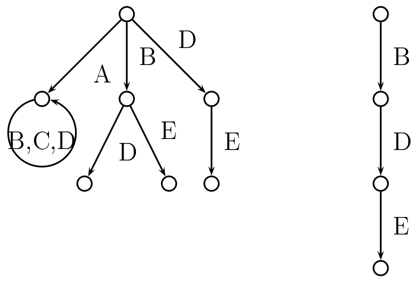


Figure 7.7: Example: removing dead-end branches when level 3 equisupport pruning is applied

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 4^{th} iteration begins.

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 7.8, level 3 equisupport pruning was employed.

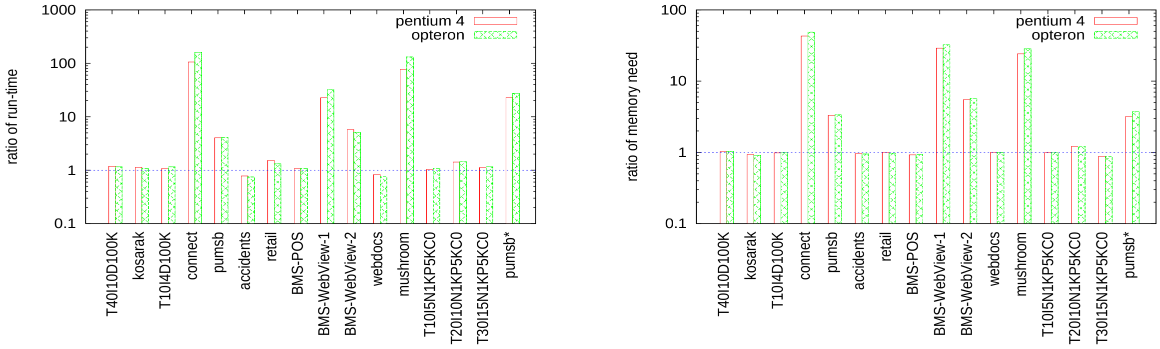


Figure 7.8: Equisupport pruning

Some hardware friendliness diagrams are given in Figure 7.9.

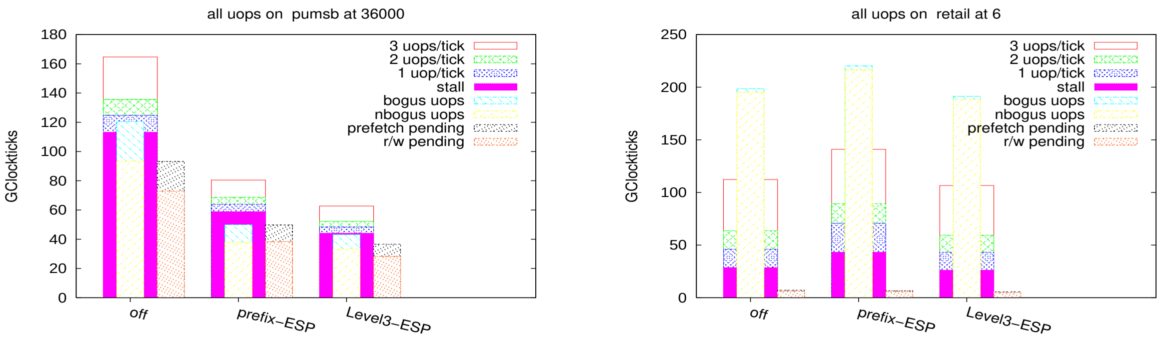


Figure 7.9: Hardware friendliness diagrams of Aprioris with different equisupport pruning techniques

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.

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. 7.10.

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

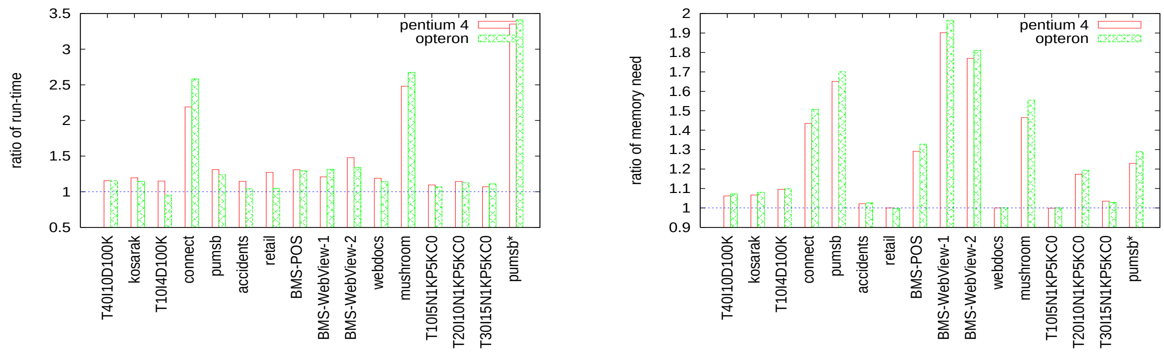


Figure 7.10: prefix equisupport pruning vs. level 3 equisupport pruning

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. [5]. Their solution differs from our in many respective. First of all, they apply *full equisupport pruning*, i.e. they do not calculate the support of any proper superset of itemset I if $\sup(I) = \sup(I')$ 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 equisupport 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 7.8 justifies this argumentation.

7.2.7 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 **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 **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 $ABCDE$, $ABCDF$, $ABCDG$, 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 [8]. 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 subsets. 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 7.2.6 *Let us assume that the trie obtained after removing infrequent itemsets of size 4 and dead-end paths is depicted in Fig. 7.11.*

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

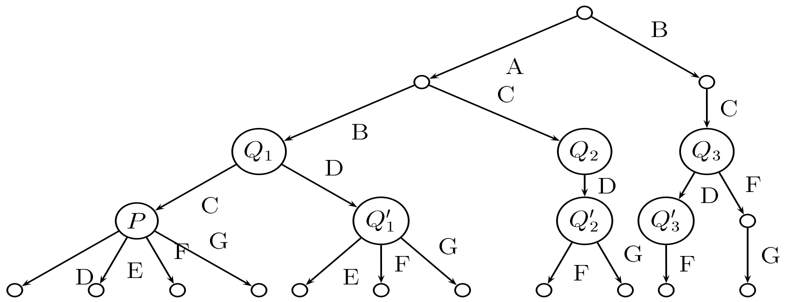


Figure 7.11: Example: intersection-based pruning

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 7.12).

Some hardware friendliness diagrams are given in Figure 7.13.

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.

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.

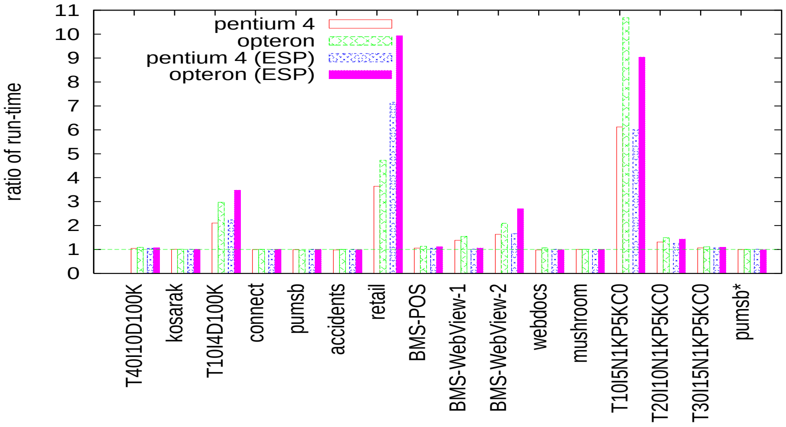


Figure 7.12: Speed-up ratios of intersection-based candidate generation without and with Level 3 equisupport pruning

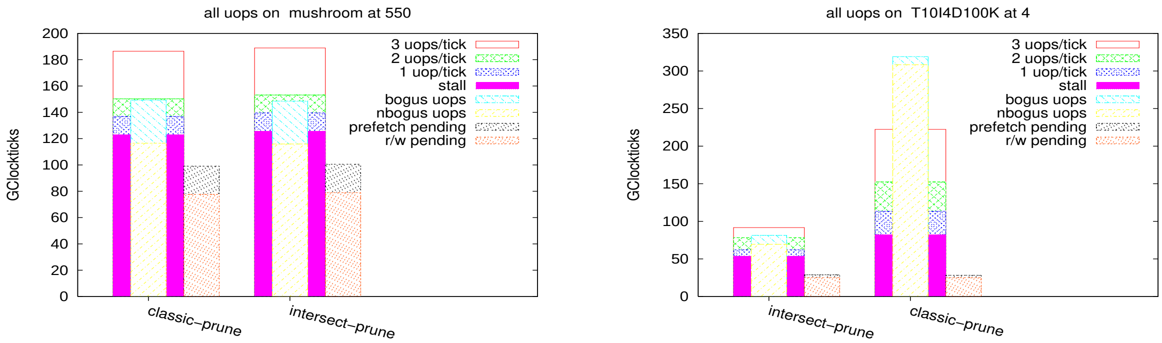


Figure 7.13: Hardware friendliness diagrams of Aprioris with the simple classic and with the intersection-based candidate generation

7.2.8 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 [8] 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-Noprune.

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 $P^I_{|I|-1}$ 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 7.3, 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:

$$|NB^{\prec_A}(F) \setminus NB(F)| \ll |F|.$$

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-Noprune. 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-Noprune 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-Noprune – 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 7.14 shows the performance gain of Apriori-Noprune compared to Apriori with classical pruning. We also check the results when equisupport pruning was turned on. This means full equisupport pruning in the case of classic Apriori and prefix equisupport pruning in Apriori-Noprune.

Some hardware friendliness diagrams are given in Figure 7.15.

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.

7.2.9 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 safeness, 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

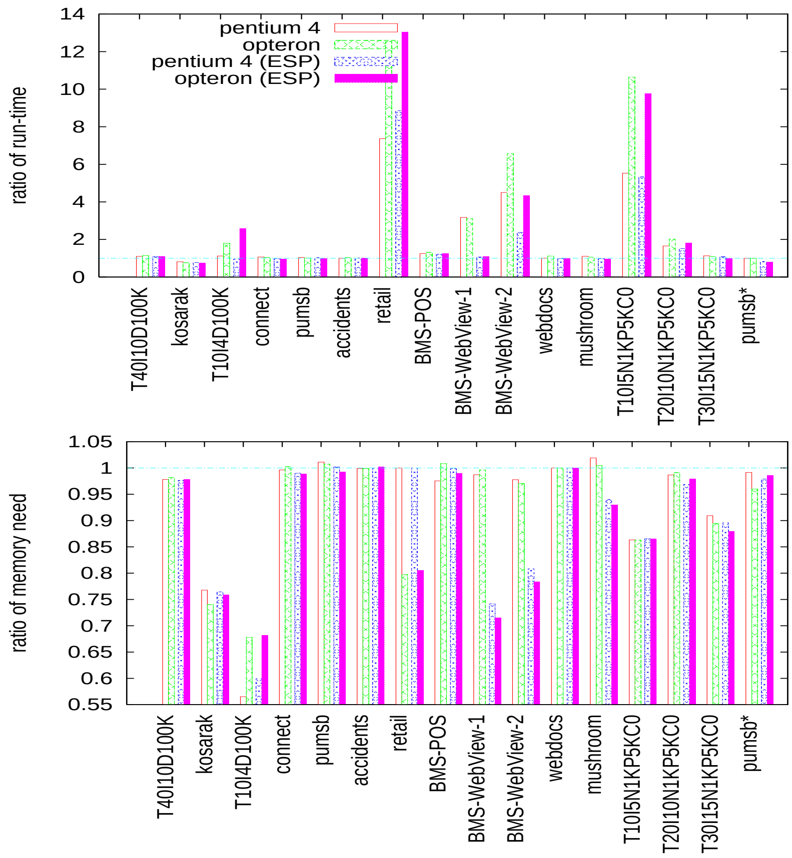


Figure 7.14: Omitting complete pruning

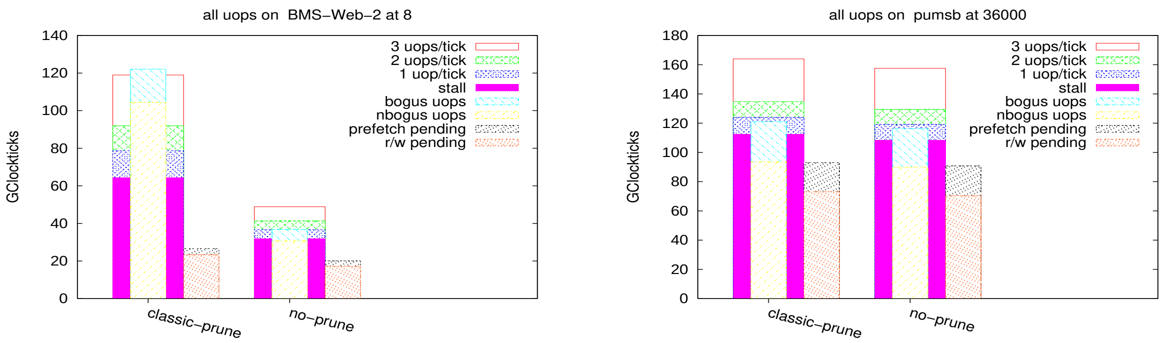


Figure 7.15: Hardware friendliness diagrams of Apriori and Apriori-Noprune

– then the fifth column stores the largest performance degradation (2/3 in this case). The last two columns store the same indicators but for the memory consumption.

technique	run-time	memory need	largest run-time ratio	smallest runtime ratio	largest memory need ratio	smallest memory need ratio
inhomogeneous trie with special block allocator	S	S	2.18		2.86	
dead-end pruning	S	S	4.24		2.56	
hybrid edge representation	✓	S	1.62	0.94	1.37	0.97
transaction caching	✓	–	706	0.94		0.17
level 3 equisupport pruning	✓	✓	105	0.77	42	0.88
prefix vs. level 3 equisupport pruning	S	S	3.35		1.9	
intersection-based pruning	✓		6.12	0.98		
omitting complete pruning	–	–	7.36	0.81	0	0.76

Table 7.6: Summary of the techniques

7.3 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.

7.3.1 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 to our original goal which is discovering unusual, unexpected, correlated patterns in the form of association 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 $\mathrm{freq}(i_1) \geq \mathrm{freq}(i_2) \geq \cdots \geq \mathrm{freq}(i_\ell)$, then the size of the largest itemset would be k where $\mathrm{freq}(i_1)\,\mathrm{freq}(i_2)\cdots\mathrm{freq}(i_k) \geq min\_freq$ but $\mathrm{freq}(i_1)\,\mathrm{freq}(i_2)\cdots\mathrm{freq}(i_{k+1}) < min\_freq$. In general the number of frequent itemsets of size ℓ would be $|\{I = \{i_1, i_2, \ldots i_\ell\} : \mathrm{freq}(i_1)\,\mathrm{freq}(i_2)\cdots\mathrm{freq}(i_\ell) \geq min\_freq\}|$.

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 7.16.

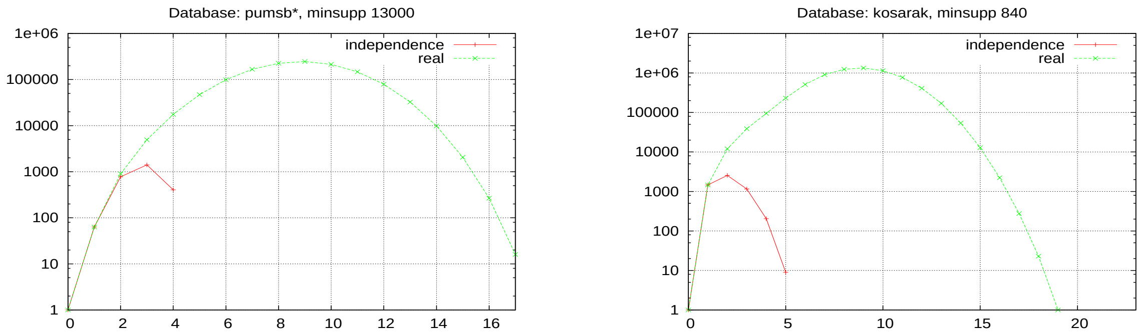


Figure 7.16: Distribution of the size of the frequent itemsets and the distribution of frequent itemsets under independence assumption

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 7.3.1 *The* ***order-preserving assumption*** *requires that* $\sup(X \cup Y) \leq \sup(X \cup Z)$ *holds whenever* $\sup(Y) \leq \sup(Z)$ *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 7.3.2 *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 7.3.3 *Let* $I = \{i_1, i_2, \dots, i_\ell\}$. *If* $\sup(i_1) \leq \sup(i_2) \leq \dots \leq \sup(i_\ell)$, *then* $\sup(\{i_1, i_2, \dots, i_{\ell-1}\}) \leq \sup(I')$ *for all* $I' \subset I$, *with* $|I'| = \ell - 1$. *Also,* $\sup(\{i_2, i_3, \dots, i_\ell\}) \geq \sup(I'')$ *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. $\square$

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

Corollary 7.3.4 *Let* I *be a set of items of size* ℓ. *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 7.3.6), 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 7.3.5 *Let* $\mathcal{T}$ *be a set of itemsets in which the order-preserving assumption holds. Then* $\sup_{\mathcal{T}|X}(Y) \leq \sup_{\mathcal{T}|X}(Z)$ *if and only if* $\sup_{\mathcal{T}}(Y) \leq \sup_{\mathcal{T}}(Z)$ *holds for any disjoint sets* X, Y, Z.

PROOF: Using the fact that the support of $X \cup Y$ in $\mathcal{T}$ equals to the support of Y in $\mathcal{T}|X$ we get the claim, since the definition of order-preserving assumption can be rewritten such as: $\sup_{\mathcal{T}|X}(Y) \leq \sup_{\mathcal{T}|X}(Z)$ holds whenever $\sup_{\mathcal{T}}(Y) < \sup_{\mathcal{T}}(Z)$ for any disjoint sets X, Y, Z. $\square$

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

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 7.3.6 *Let $0 \leq \alpha \leq 1$ be a given constant. The α* order-preserving assumption *requires that $\alpha \cdot \sup(X \cup Y) \leq \sup(X \cup Z)$ holds whenever $\sup(Y) < \sup(Z)$ 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 α order-preserving assumption in a set of itemsets S, in which downward closure property holds, in a sequence of itemset $\mathcal{T}$. We check all different itemset pairs $I, I' \in S$ if their intersection is nonempty. For such itemset pairs we calculate $I_1 = I \backslash I'$, $I_2 = I' \backslash I$. 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 α order-preserving assumption. Table contains the order preserving ratio of the frequent itemsets in our benchmark databases.

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.

7.3.2 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-Noprune. The set of infrequent candidates is equal to the order based negative border of the frequent itemsets. An ℓ-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 7.3.7 *If the order-preserving assumption holds, then the ascending order with respect to the supports results in the smallest order based negative border.*

database	minsup	order-preserving ratio 1	0.95	0.9
T40I10D100	900	0.912	0.998	0.999
kosarak	1 800	0.817	0.980	0.998
T10I4D100K	8	0.690	0.693	0.726
connect	56 000	0.725	1.000	1.000
pumsb	41 000	0.863	0.994	1.000
	38 000	0.219	0.974	0.999
accidents	114 000	0.882	0.960	0.988
retail	11	0.870	0.876	0.909
BMS-POS	400	0.809	0.860	0.901
	350	0.116	0.354	0.544
BMS-WebView-1	37	0.857	0.942	0.984
	36	0.351	0.802	0.961
BMS-WebView-2	30	0.790	0.819	0.853
webdocs	220 000	0.877	0.966	0.990
mushroom	1 600	0.910	0.955	0.990
	900	0.868	0.896	0.913
T10I5N1KP5KC0	100	0.915	0.961	0.967
	10	0.790	0.809	0.819
	8	0.714	0.729	0.739
T20I10N1KP5KC0	400	0.963	0.999	0.999
T30I15N1KP5KC0.25D200K	650	0.999	1.000	1.000
pumsb*	17 000	0.850	0.963	0.986
	15 000	0.434	0.833	0.928

Table 7.7: The order-preserving ratio of the frequent itemsets

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 $i_1 \prec i_2 \prec \cdots \prec i_\ell$. 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 7.3.3 states that $\sup(\{i_1, i_2, \ldots, i_{\ell-1}\}) \leq \sup(I')$ for all $I' \subset I$, where $|I'| = \ell - 1$. 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 $NB^{\prec}(F) = NB(F)$ if $\prec$ denotes the ascending order according to frequencies. By Corollary 2.0.8 (see page 11) no other ordering results in smaller number of candidates, hence the lemma follows. $\square$

Corollary 7.3.8 *If order-preserving assumption holds, then*

$$NB(F) = NB^{\prec_{ASC}}(F),$$

where F denotes the set of frequent itemsets, and $\prec_{ASC}$ denotes the ascending ordering according to the supports.

7.3.3 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 7.17. Both tries stores sets AB and AC. The first trie uses ordering $A \prec B \prec C$ the second uses the reverse.

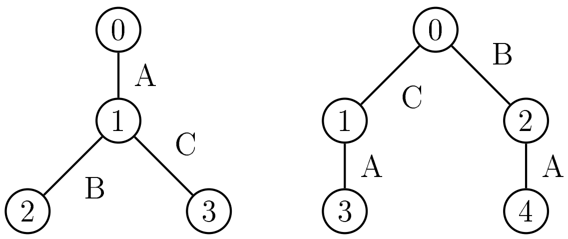


Figure 7.17: tries storing the same sets but using different orderings

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 [13] that the minimal trie problem, i.e., to determine the ordering which gives a minimal trie (denoted by $\mathbb{T}_{OPT}$), 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 tries that store sets A, B, AB, AC and use ordering $A \prec B \prec C$ and $A \prec C \prec A$ are isomorphic and minimal. Furthermore different ordering may result different, non-isomorphic minimal tries. This is shown in Figure 7.18.

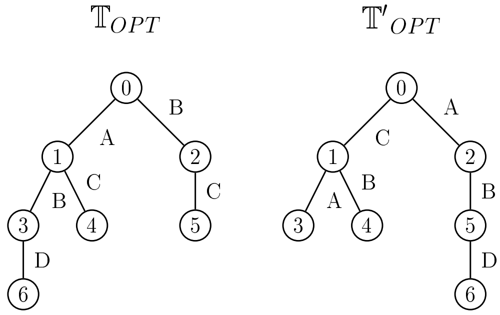


Figure 7.18: Example: minimal non-isomorphic tries

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 ordering based on the two values are not necessary equal. If the elements of the database AB, AC, AD, BC, BC, BC 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 7.3.9 *Let $\mathcal{T}$ be a sequence of itemsets and denote by $\mathcal{T}^*$ the sequence that is obtained from $\mathcal{T}$ by keeping only the different elements (i.e. $\mathcal{T}^*$ contains the same itemsets as $\mathcal{T}$ but with multiplicity exactly one). The* ***unweighted support*** *of itemset I in $\mathcal{T}$ equals to the support of I in $\mathcal{T}^*$, i.e.*

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

Lemma 7.3.10 *Let $\mathcal{T}$ be a sequence of itemsets over $\mathcal{I}$. 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 $\text{sup}(i) \geq supp(i')$ but $\text{uw\_sup}(i) < \text{uw\_sup}(i')$. Let us denote the elements of the cover of i' in $\mathcal{T}^*$ by

$t'_1, t'_2, \ldots, t'_n$ ($i' \in t_j$, $t_j \in \mathcal{T}^*$, $t'_k \neq t'_l$). According to the order-preserving assumption $\sup((t \backslash) i' \cup i) \geq \sup(t)$ for all $t \in cover^*_{\mathcal{T}}(i')$. This is a contradiction, because the number of $(t \setminus i') \cup i$ sets is smaller or equal than $uw\_supp(i')$, they are different and all contain i, therefore the size of uw_sup(i) cannot be less than uw_sup(i'). $\square$

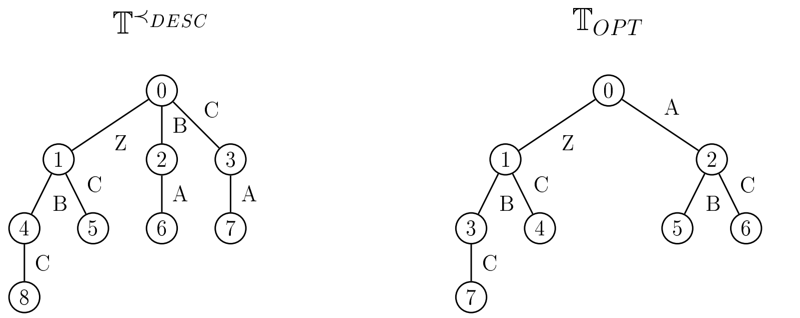


Figure 7.19: Example: descending order does not result in the smallest trie

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 $\sup(Z) > \sup(A)$, but $\sup(ZB) < \sup(AB)$.

Conjecture 7.3.11 *Let $\mathcal{T}$ be a set of itemsets and denote $\prec_{DESC}$ the descending order of items according to the number of occurrences of the items in $\mathcal{T}$. If order-preserving assumption holds then $\mathbb{T}^{\prec_{DESC}}(\mathcal{T})$ is the minimum-size trie among the tries that store $\mathcal{T}$, i.e., there exists no ordering $\prec$ such that $\mathbb{T}^{\prec}(\mathcal{T})$ has fewer nodes than $\mathbb{T}^{\prec_{DESC}}(\mathcal{T})$.*

If the conjecture follows, then we know that the heuristic works fine under ideal circumstances (the order-preserving assumption holds for all sets). Table 7.7 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 α order-preserving ratio of a set of itemset $\mathcal{T}$ and the ratio of $|\mathbb{T}^{\prec_{DESC}}(\mathcal{T})|$ and $|\mathbb{T}_{OPT}(\mathcal{T})|$, where $\mathbb{T}_{OPT}$ denotes a minimum-size trie.

7.3.4 Techniques in Apriori

Support count

One may tend to follow the observation a smaller memory need results in better data locality and hence faster algorithms. Therefore we should use the descending order according to the frequency 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.

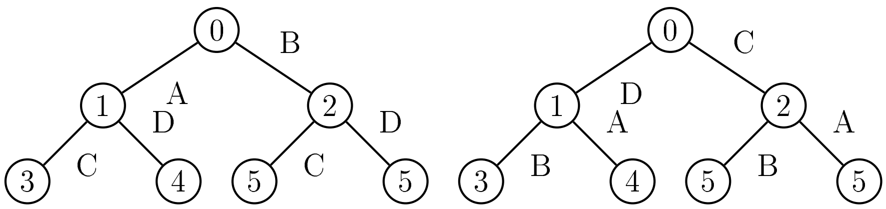


Figure 7.20: Example: Tries with different orders

7.20. 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 cacher. Since the efficient support count (i.e. `merge`) requires the items of the transaction to be stored ascendingly, we simply reverse each transaction when it is retrieved from the cacher.

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 a 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 dichotony, see Figure 7.21.

Values less than one mean that the Apriori that uses descending order according to the frequencies is the faster.

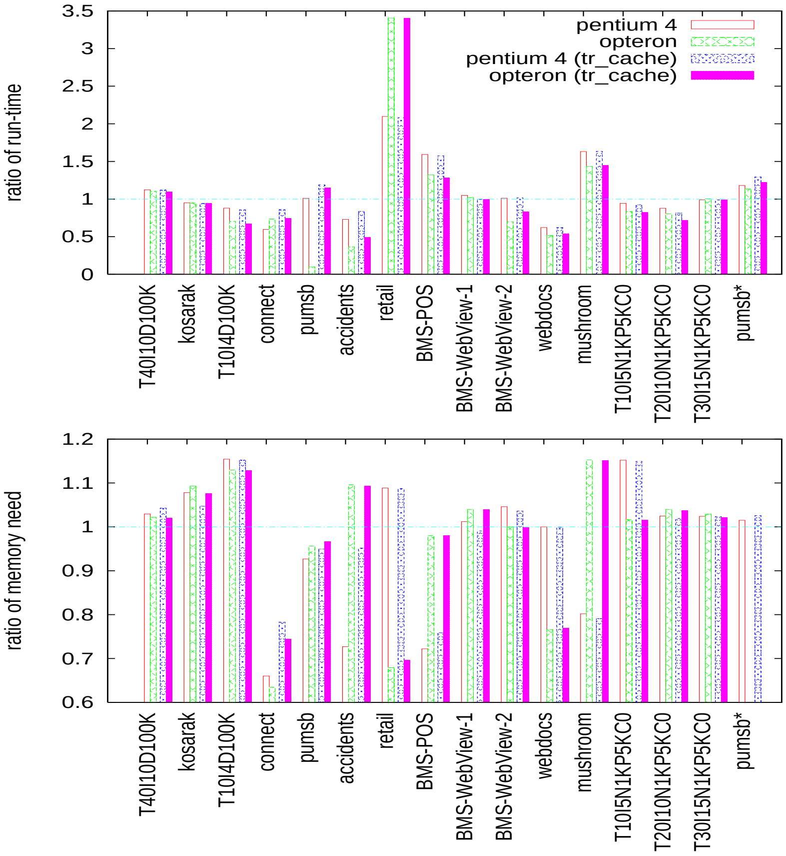


Figure 7.21: Ascending vs. Descending order according to the frequencies

Some hardware friendliness diagrams are given in Figure 7.22.

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 cacher 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.

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-Noprune are the same as candidates in

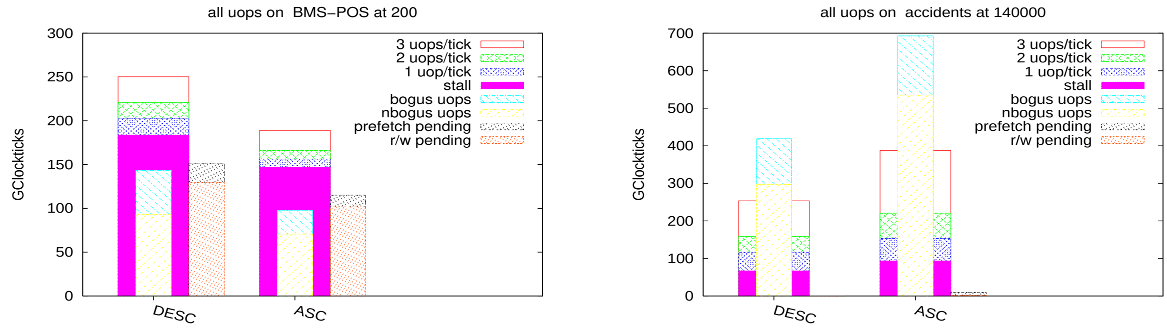


Figure 7.22: Hardware friendliness diagrams of Aprioris with ascending and descending order according to the frequencies

Eclat or FP-growth (see section 7.3.2) therefore the fastest Apriori-noprune is expected when ascending order according to the frequencies is used. Experiments support this conclusion.

Table 7.8 shows the ratio of the number infrequent candidates and the number of frequent itemsets in the case of complete pruning, Apriori-Noprune with ascending and descending order according to the supports.

database	complete prune	NOPRUNE ASC	NOPRUNE DESC	$\frac{\lvert C^{DESC}\setminus F\rvert}{\lvert C^{ASC}\setminus F\rvert}$
T40I10D100K	0.98	1.05	3.20	3.04
kosarak	0.05	0.74	1.61	2.16
T10I4D100K	6.62	15.57	27.98	1.79
connect	0.0001	0.002	2.07	766.83
pumsb	0.008	0.03	0.82	22.71
accidents	0.03	0.03	2.90	86.78
retail	4.82	5.71	578.54	101.15
BMS-POS	0.56	0.59	30.98	52.10
BMS-WebView-1	0.002	0.02	0.09	3.53
BMS-WebView-2	0.05	0.16	3.28	20.18
webdocs	0.21	0.22	12.91	58.41
mushroom	0.001	0.005	2.87	515.85
T10I5N1KP5KC0.25D200	42.31	51.94	194.90	3.75
T20I10N1KP5KC0.25D200K	0.009	0.06	0.24	3.71
T30I15N1KP5KC0.25D200K	0.07	0.26	0.16	0.61
pumsb*	0.002	0.05	0.62	12.42

Table 7.8: Ratio of the number of infrequent candidates and the number of frequent itemsets

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-Noprune in terms of sensitivity of the ordering. Both orderings has their advantage in Apriori, but in Apriori-Noprune the drawback of descending order is dominating.

Chapter 8

Evalutation

8.1 The battle of Apriori implementations

We have enrolled our three selected implementations (`Apriori`, `Apriori-Noprune` and `Apriori-MEMSAFE`) in a competition with three known Apriori codes. `Apriori-MEMSAFE` employs on-line candidate 2-itemset generation [19] and does not use transaction caching. `Apriori-Noprune` omits the complete pruning phase. `Apriori` and `Apriori--MEMSAFE` adapt the intersection-based candidate generation. `Apriori-Noprune` 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 15^{th} 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 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

[1] `http://fuzzy.cs.uni-magdeburg.de/~borgelt/apriori.html`
[2] `http://www.adrem.ua.ac.be/~goethals/software/`
[3] `http://www.cs.ualberta.ca/~tszhu/software.html`

edge representation, and a trie storing the transactions. It adapts a novel support counting method, (the basis of which was described in section 7.2.2) together with the simple unimportant item filtering technique (see section 7.2.3).

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/~bodon/fim/test.html`. Three typical run-time plots are depicted in Figure 8.1.

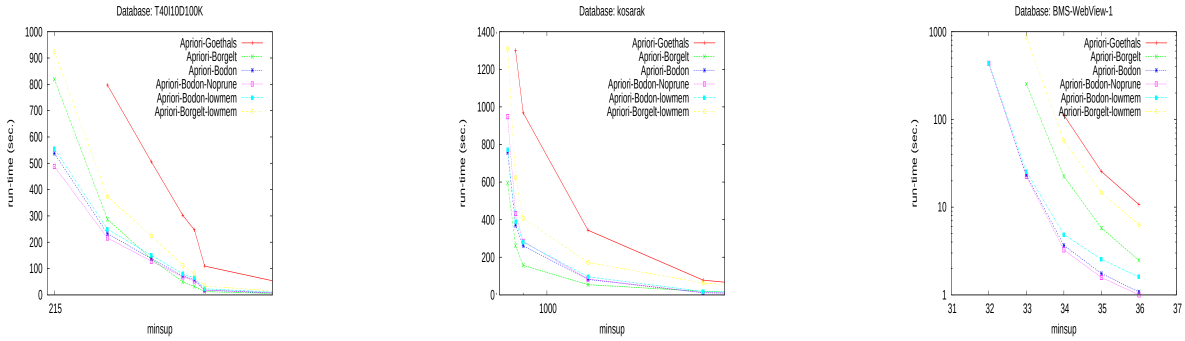


Figure 8.1: Battle of the Apriori implementations, run-times

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`. Positive value means that `Apriori` was faster than `Apriori-Borgelt-FIMI`.

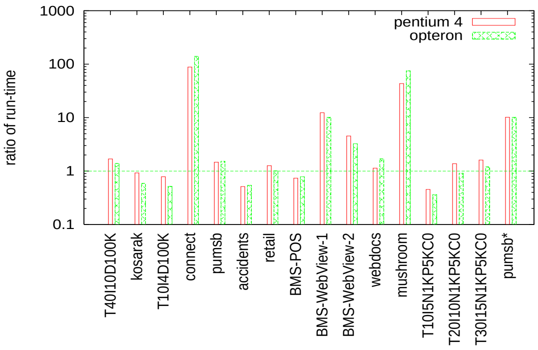


Figure 8.2: Borgelt vs. Bodon (run-times)

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 8.3.

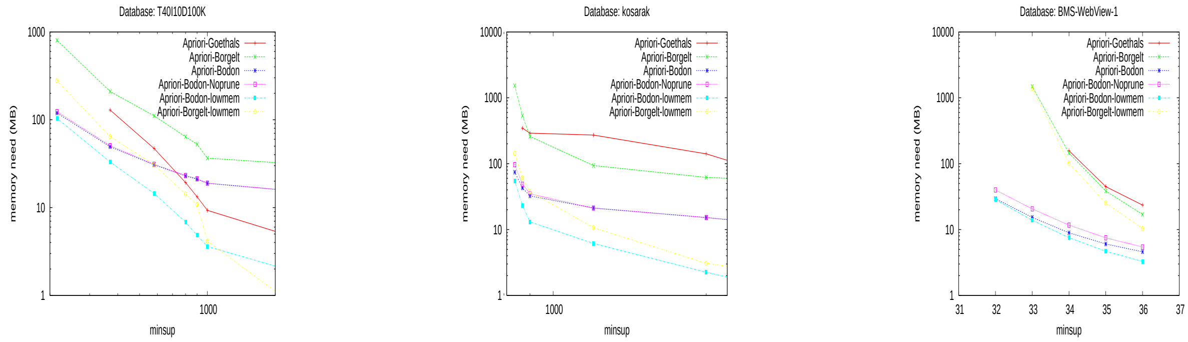


Figure 8.3: Battle of the Apriori implementations, memory need

The comparison of the two main rivals, i.e. `Apriori` and `Apriori-Borgelt-FIMI` is found in Fig. 8.4.

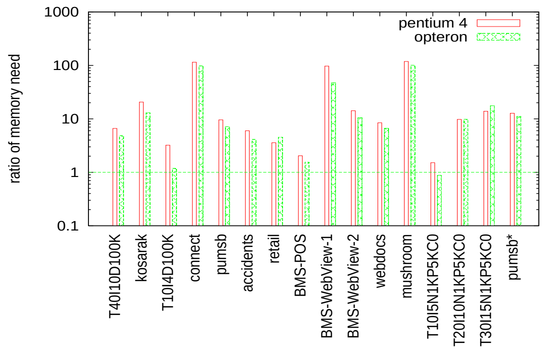


Figure 8.4: Borgelt vs. Bodon (memory needs)

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

8.2 The battle of Eclat implementations

8.3 The battle of FP-growth implementations

8.4 Comparing Aprior, Eclat and FP-growth

In compared the three algorithms we endeavored to be as fair as possible. Common methods (like frequent item mining, input/output operations, coding/decoding subset enumeration) use the same code. We spend many efforts on making these common methods as efficient as possible in order the algorithm specific codes be dominant in run-time and memory need.

We determine the supports of the items by using a simple vector. The element at index i belong to item i. Initially all elements are zero, we take the transactions one-by-one and increase the counter of those elements that occur in the actual transaction.

In input and output routines we use buffering (with a carefully chosen buffer size) manual integer to string (and backward) conversion and low level file operation. To further reduce the output of the result, which is quite dominant in dense datasets with low support threshold (like database `mushroom` with minsup = 30000), we used a depth-first output manager, which caches the string representation of the previously frequent itemset written out. For further information and experiment results of this sophisticated solution the reader is referred to [46].

TEST RESULTS COME HERE!!!

The test results immediately proves that the often cited misbelief *"The numerous database scan is the reason for ineffectiveness of algorithm Apriori"* has nothing to do with the reality. Our Apriori implementation uses transaction caching (see section 7.2.1) thus Apriori scans the entire dataset only twice, the same times as Eclat and FP-growth do. Apriori is still much slower than the counterparts in many cases.

8.5 The bottleneck of Apriori, Eclat and FP-growth

We have seen that there is no single best algorithm that outperforms the other algorithms at every databases with every support thresholds. Each algorithm has its bottleneck.

On the contrary to the believes (see section 1.2), the reason why Apriori falls behind in efficiency from Eclat and Fp-growth is that Apriori *does not utilize the information gained in the previous iteration. Although it determines the cover of all subsets of a candidate in the previous iteration, this information is not used in determining the support of the candidate.* Eclat and FP-growth are smarter in this respect, i.e. only those transactions are considered in determining the support of a candidate that contain the prefix of the candidate.

TO BE CONTINUED!

Chapter 9

The furure: toward hybrid algorithms

The fact that each algorithm has its drawback, opens the gate toward hybrid algorithms. The first hybrid algorithm AprioriHybrid appeared quite early. It is a combination of Apriori and AprioriTid, based on the observation that Apriori performs better in the initial phases while AprioriTid is a better choice in the later phases. The difference between Apriori and AprioriTid lies in the support count method. AprioriTid uses a table, each row of a table belongs to a transaction and a row at iteration ℓ contains the candidates of size ℓ that occurs in the transaction (empty rows are removed from the table). Both the support of a candidate and the table of the next iteration can be counted directly from the table. The reason AprioriTid runs faster in the final iteration is not the always emphasized property that it does not use the input data (IO operations requires insignificant time compared to the other operations in the support count) but the simplified support determination of a candidates.

The switch point depends on the size of the table. If the number of candidates decreases and the size of the table fits into the memory then Apriori switches to AprioriTid. In [18] is was shown that this heuristic does not necessarily works (with our words it is neither memory nor run-time safe), because the number of candidates may grow again, which may prevent the table fitting into the memory. This results in a significant performance deterioration.

The second hybrid solution was proposed in [26] where the authors proposed to use Apriori is the beginning and then switch to Eclat. Unfortunately, the main question, i.e. when to do the switch is not answered and can be simply set by a parameter. In [19] it was shown that the hybrid algorithm that switches to Eclat after the second iteration and uses the array-based technique to determine the support of the pairs outperforms Apriori and Eclat at many databases.Since the efficiency of Apriori and Eclat fall far behind from the efficiency of our Apriori and Eclat, this observation does not necessarily hold. Also to understand this hybrid solution we don't have to know anything about Apriori and its speed-up techniques, hence we do not regard this solution as a hybrid method, but rather a modification of Eclat.

We believe that the first remarkable hybrid solution is algorithm DCI [36] whose improvement kDCI [38] turned out to be one of the most successful FIM implementations in 2003. In the beginning it works as an Apriori that used prefix-array to store the candidates and applies the unimportant item filtering technique in order to reduce the size of the database. As soon as the database fits into the memory it switches to a novel intersection-based counting method. Moreover, it uses a heuristics to decide if the input database is dense or sparse and chooses the counting procedure that is expected to perform better.

9.1 Conclusion

Bibliography

[1] Ramesh C. Agarwal, Charu C. Aggarwal, and V. V. V. Prasad. A tree projection algorithm for generation of frequent item sets. *J. Parallel Distrib. Comput.*, 61(3): 350–371, 2001. ISSN 0743-7315. doi: http://dx.doi.org/10.1006/jpdc.2000.1693.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In J.B. Bocca, M. Jarke, and C. Zaniolo, editors, *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94), Santiago de Chile, September 12-15*, pages 487–499. Morgan Kaufmann, 1994.

[3] R. Agrawal, T. Imielienski, and A. Swami. Mining association rules between sets of items in large databases. In P. Bunemann and S. Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD Conference on Managment of Data*, pages 207–216, New York, 1993. ACM Press.

[4] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo. Fast discovery of association rules. In U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 307–328. MIT Press, 1996.

[5] Yves Bastide, Rafik Taouil, Nicolas Pasquier, Gerd Stumme, and Lotfi Lakhal. Mining frequent patterns with counting inference. *SIGKDD Explor. Newsl.*, 2(2): 66–75, 2000. doi: http://doi.acm.org/10.1145/380995.381017.

[6] Ferenc Bodon. A fast apriori implementation. In Bart Goethals and Mohammed J. Zaki, editors, *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, volume 90 of *CEUR Workshop Proceedings*, Melbourne, Florida, USA, 2003.

[7] Ferenc Bodon and Lajos Rónyai. Trie: an alternative data structure for data mining algorithms. *Hungarian Applied Mathematics and Computer Application*, 38(7-9): 739–751, October 2003.

[8] Ferenc Bodon and Lars Schmidt-Thieme. The relation of closed itemset mining, complete pruning strategies and item ordering in apriori-based fim algorithms. In *Proceedings of the 9th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD'05)*, Porto, Portugal, 2005.

[9] Christian Borgelt. Efficient implementations of apriori and eclat. In Bart Goethals and Mohammed J. Zaki, editors, *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, volume 90 of *CEUR Workshop Proceedings*, Melbourne, Florida, USA, 2003.

[10] Christian Borgelt. Recursion pruning for the apriori algorithm. In Bart Goethals, Mohammed J. Zaki, and Roberto Bayardo, editors, *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'04)*, volume 126 of *CEUR Workshop Proceedings*, Brighton, UK, 2004.

[11] Christian Borgelt and Rudolf Kruse. Induction of association rules: Apriori implementation. In W. Hrdle and B. Rnz, editors, *Proceedings of the 15th Conference on Computational Statistics*, pages 395–400. Physica-Verlag, 2002.

[12] Sergey Brin, Rajeev Motwani, Jeffrey D. Ullman, and Shalom Tsur. Dynamic itemset counting and implication rules for market basket data. In Joan Peckham, editor, *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 255–264. ACM Press, 05 1997.

[13] Douglas Comer and Ravi Sethi. The complexity of trie index construction. *J. ACM*, 24(3):428–440, 1977. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/322017.322023.

[14] R. de la Briandais. File searching using variable-length keys. In *Proceedings of the Western Joint Computer Conference*, pages 295–298, March 1959.

[15] Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/367390.367400.

[16] B. Goethals and M. J. Zaki. Advances in frequent itemset mining implementations: Report of fimi'03. *ACM SIGKDD Explorations*, 6(1):109–117, June 2004.

[17] Bart Goethals. Memory issues in frequent itemset mining. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 530–534, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-812-1. doi: http://doi.acm.org/10.1145/967900.968012.

[18] Bart Goethals. *Efficient Frequent Pattern Mining.* PhD thesis, Transnationale Universiteit Limburg, 2002.

[19] Bart Goethals. Survey on frequent pattern mining. Manuskript, 2002.

[20] Gsta Grahne and Jianfei Zhu. Efficiently using prefix-trees in mining frequent itemsets. In Bart Goethals and Mohammed J. Zaki, editors, *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, volume 90 of *CEUR Workshop Proceedings*, Melbourne, Florida, USA, 2003.

[21] J. Han and J. Pei. Mining frequent patterns by pattern-growth: Methodology and implications. *ACM SIGKDD Explorations*, 2(2):14–20, 2000. Special Issue on Scalable Data Mining Algorithms.

[22] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In W. Chen, J. F. Naughton, and P. A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM Press, 2000.

[23] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation: a frequent-pattern tree approach. *Data Mining and Knowledge Discovery*, erscheint demnchst, 2003.

[24] Jia Liang Han and Ashley W. Plank. Background for association rules and cost estimate of selected mining algorithms. In *CIKM '96, Proceedings of the Fifth International Conference on Information and Knowledge Management, November 12 - 16, 1996, Rockville, Maryland, USA*, pages 73–80. ACM, 1996.

[25] Jochen Hipp, Ulrich Güntzer, and Gholamreza Nakhaeizadeh. Algorithms for association rule mining a general survey and comparison. *SIGKDD Explor. Newsl.*, 2(1):58–64, 2000. doi: http://doi.acm.org/10.1145/360402.360421.

[26] Jochen Hipp, Ulrich Güntzer, and Gholamreza Nakhaeizadeh. Mining association rules: Deriving a superior algorithm by analyzing today's approaches. In *PKDD '00: Proceedings of the 4th European Conference on Principles of Data Mining and Knowledge Discovery*, pages 159–168, London, UK, 2000. Springer-Verlag. ISBN 3-540-41066-X.

[27] Keyun Hu, Yuchang Lu, Lizhu Zhou, and Chunyi Shi. Integrating classification and association rule mining: A concept lattice framework. In *RSFDGrC '99: Proceedings of the 7th International Workshop on New Directions in Rough Sets, Data Mining, and Granular-Soft Computing*, pages 443–447, London, UK, 1999. Springer-Verlag. ISBN 3-540-66645-1.

[28] Walter A. Kosters, Elena Marchiori, and Ard A. J. Oerlemans. Mining clusters with association rules. In *IDA '99: Proceedings of the Third International Symposium on Advances in Intelligent Data Analysis*, pages 39–50, London, UK, 1999. Springer-Verlag. ISBN 3-540-66332-0.

[29] Wenke Lee and Salvatore Stolfo. Data mining approaches for intrusion detection. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, 1998. URL `citeseer.ist.psu.edu/article/lee98data.html`.

[30] Dao-I Lin and Zvi M. Kedem. Pincer search: A new algorithm for discovering the maximum frequent set. In *EDBT '98: Proceedings of the 6th International Conference on Extending Database Technology*, pages 105–119, London, UK, 1998. Springer-Verlag. ISBN 3-540-64264-1.

[31] 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`.

[32] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Efficient algorithms for discovering association rules. In Usama M. Fayyad and Ramasamy Uthurusamy, editors, *AAAI Workshop on Knowledge Discovery in Databases(KDD-94)*, pages 181–192, Seattle, Washington, 1994. AAAI Press. URL `http://citeseer.nj.nec.com/mannila94efficient.html`.

[33] Ulrich Meyer, Peter Sanders, and Jop F. Sibeyn, editors. *Algorithms for Memory Hierarchies, Advanced Lectures [Dagstuhl Research Seminar, March 10-14, 2002]*, volume 2625 of *Lecture Notes in Computer Science*, 2003. Springer. ISBN 3-540-00883-7.

[34] B. Mobasher, N. Jain, E. Han, and J. Srivastava. Web mining: Pattern discovery from world wide web transactions. Technical Report TR-96050, Department of Computer Science, University of Minnesota, 1996.

[35] A. Mueller. Fast sequential and parallel algorithms for association rule mining: A comparison. Technical report, Department of Computer Science, University of Maryland-College Park, 1995. CS-TR-3515.

[36] S. Orlando, P. Palmerini, R. Perego, and F. Silvestri. Adaptive and resource-aware mining of frequent sets. In *ICDM '02: Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM'02)*, page 338, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1754-4.

[37] Salvatore Orlando, Paolo Palmerini, and Raffaele Perego. Enhancing the apriori algorithm for frequent set counting. In *DaWaK '01: Proceedings of the Third*

International Conference on Data Warehousing and Knowledge Discovery, pages 71–82, London, UK, 2001. Springer-Verlag. ISBN 3-540-42553-5.

[38] Salvatore Orlando, Claudio Lucchese, Paolo Palmerini, Raffaele Perego, and Fabrizio Silvestri. kdci: a multi-strategy algorithm for mining frequent sets. In Bart Goethals and Mohammed J. Zaki, editors, *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, volume 90 of *CEUR Workshop Proceedings*, Melbourne, Florida, USA, 2003.

[39] Bruno Pôssas, Nivio Ziviani, Jr. Wagner Meira, and Berthier Ribeiro-Neto. Set-based model: a new approach for information retrieval. In *SIGIR '02: Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 230–237, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-561-0. doi: http://doi.acm.org/10.1145/564376.564417.

[40] Jong Soo Park, Ming-Syan Chen, and Philip S. Yu. An effective hash-based algorithm for mining association rules. *SIGMOD Rec.*, 24(2):175–186, 1995. ISSN 0163-5808. doi: http://doi.acm.org/10.1145/568271.223813.

[41] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Pruning closed itemset lattices for association rules. In *Proceedings of the BDA French Conference on Advanced Databases*, October 1998. URL `http://citeseer.nj.nec.com/pasquier98pruning.html`.

[42] Jian Pei, Jiawei Han, Hongjun Lu, Shojiro Nishio, Shiwei Tang, and Dongqing Yang. H-mine: Hyper-structure mining of frequent patterns in large databases. In *ICDM*, pages 441–448, 2001.

[43] Andrea Pietracaprina and Dario Zandolin. Mining frequent itemsets using patricia tries. In Bart Goethals and Mohammed J. Zaki, editors, *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, volume 90 of *CEUR Workshop Proceedings*, Melbourne, Florida, USA, 19. November 2003.

[44] Adriana Prado, Cristiane Targa, and Alexandre Plastino. Improving direct counting for frequent itemset mining. In *DaWaK*, pages 371–380, 2004.

[45] Balázs Rácz. nonordfp: An FP-growth variation without rebuilding the FP-tree. In Bart Goethals, Mohammed J. Zaki, and Roberto Bayardo, editors, *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'04)*, volume 126 of *CEUR Workshop Proceedings*, Brighton, UK, 2004.

[46] Balázs Rácz, Ferenc Bodon, and Lars Schmidt-Thieme. Benchmarking frequent itemset mining algorithms: from measurement to analysis. In Bart Goethals,

Siegfried Nijssen, and Mohammed J. Zaki, editors, *Proceedings of the ACM SIGKDD Workshop on Open Source Data Mining Workshop (OSDM'05)*, Chicago, IL, USA, August 2005.

[47] Jr. Roberto J. Bayardo. Efficiently mining long patterns from databases. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 85–93, New York, NY, USA, 1998. ACM Press. ISBN 0-89791-995-5. doi: http://doi.acm.org/10.1145/276304.276313.

[48] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Analysis of recommendation algorithms for e-commerce. In *EC '00: Proceedings of the 2nd ACM conference on Electronic commerce*, pages 158–167, New York, NY, USA, 2000. ACM Press. ISBN 1-58113-272-7. doi: http://doi.acm.org/10.1145/352871.352887.

[49] Ashoka Savasere, Edward Omiecinski, and Shamkant B. Navathe. An efficient algorithm for mining association rules in large databases. In *VLDB '95: Proceedings of the 21th International Conference on Very Large Data Bases*, pages 432–444, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc. ISBN 1-55860-379-4.

[50] R. Srikant. *Fast algorithms for mining association rules and sequential patterns.* PhD thesis, Univeristy of Wisconsin, Madison, 1996. Supervisor-Jeffrey F. Naughton.

[51] Ja-Hwung Su and Wen-Yang Lin. Cbw: An efficient algorithm for frequent itemset mining. In *HICSS '04: Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 3*, page 30064.3, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2056-1.

[52] Hannu Toivonen. Sampling large databases for association rules. In *VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases*, pages 134–145, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc. ISBN 1-55860-382-4.

[53] Takeaki Uno, Tatsuya Asai, Yuzo Uchida, and Hiroki Arimura. Lcm: An efficient algorithm for enumerating frequent closed item sets. In Bart Goethals and Mohammed J. Zaki, editors, *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, volume 90 of *CEUR Workshop Proceedings*, Melbourne, Florida, USA, 19. November 2003.

[54] John von Neumann. First draft of a report on the EDVAC. Contract No. W–670–ORD–4926 Between the United States Army Ordnance Department and the University of Pennsylvania, June 1945. URL `http://qss.stanford.edu/~{}godfrey/vonNeumann/vnedvac.pdf`.

[55] Yew Kwong Woon, Wee Keong Ng, and Ee-Peng Lim. Online and incremental mining of separately-grouped web access logs. In *WISE '02: Proceedings of the 3rd International Conference on Web Information Systems Engineering*, pages 53–62, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1766-8.

[56] Yew-Kwong Woon, Wee-Keong Ng, and Ee-Peng Lim. A support-ordered trie for fast frequent itemset discovery. *IEEE Transactions on Knowledge and Data Engineering*, 16(7):875–879, 2004. ISSN 1041-4347. doi: http://dx.doi.org/10.1109/TKDE.2004.1318569.

[57] Mohammed Javeed Zaki and Mitsunori Ogihara. Theoretical foundations of association rules. In *Proceedings of third SIGMOD'98 Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD'98)*, Seattle, Washington, 1998. URL `http://citeseer.nj.nec.com/zaki98theoretical.html`.

[58] Mohammed Javeed Zaki, Srinivasan Parthasarathy, Mitsunori Ogihara, and Wei Li. New algorithms for fast discovery of association rules. In David Heckerman, Heikki Mannila, Daryl Pregibon, Ramasamy Uthurusamy, and Menlo Park, editors, *In 3rd Intl. Conf. on Knowledge Discovery and Data Mining*, pages 283–296. AAAI Press, 12–15 1997. ISBN 1-57735-027-8. URL `citeseer.nj.nec.com/zaki97new.html`.

[59] Z. Zheng, Ronny Kohavi, and L. Mason. Real world performance of association rule algorithms. In Foster Provost and Ramakrishnan Srikant, editors, *Proceedings of the 7th International Conference on Knowledge Discovery and Data Mining (KDD'01), New York*, pages 401–406. ACM Press, 2001.