Frequent Itemset Mining on Multiprocessor Systems

Dissertation
zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)
vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik
eingereicht von
Dipl.-Inf. Benjamin Schlegel
geboren am 13. Dezember 1980 in Meerane

Gutachter: Prof. Dr.-Ing. Wolfgang Lehner
Technische Universität Dresden
Fakultät Informatik, Institut für Systemarchitektur
Lehrstuhl für Datenbanken
01062 Dresden

Prof. Ph. D. Johann-Christoph Freytag
Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät, Institut für Informatik
Lehrstuhl für Datenbanken und Informationssysteme
10099 Berlin

Tag der Verteidigung: 30. Mai 2013

Dresden im April 2014
Abstract

Frequent itemset mining is an important building block in many data mining applications like market basket analysis, recommendation, web-mining, fraud detection, and gene expression analysis. In many of them, the datasets being mined can easily grow up to hundreds of gigabytes or even terabytes of data. Hence, efficient algorithms are required to process such large amounts of data. In recent years, there have been many frequent-itemset mining algorithms proposed, which however (1) often have high memory requirements and (2) do not exploit the large degrees of parallelism provided by modern multiprocessor systems. The high memory requirements arise mainly from inefficient data structures that have only been shown to be sufficient for small datasets. For large datasets, however, the use of these data structures force the algorithms to go out-of-core, i.e., they have to access secondary memory, which leads to serious performance degradations. Exploiting available parallelism is further required to mine large datasets because the serial performance of processors almost stopped increasing. Algorithms should therefore exploit the large number of available threads and also the other kinds of parallelism (e.g., vector instruction sets) besides thread-level parallelism.

In this work, we tackle the high memory requirements of frequent itemset mining twofold: we (1) compress the datasets being mined because they must be kept in main memory during several mining invocations and (2) improve existing mining algorithms with memory-efficient data structures. For compressing the datasets, we employ efficient encodings that show a good compression performance on a wide variety of realistic datasets, i.e., the size of the datasets is reduced by up to 6.4x. The encodings can further be applied directly while loading the dataset from disk or network. Since encoding and decoding is repeatedly required for loading and mining the datasets, we reduce its costs by providing parallel encodings that achieve high throughputs for both tasks. For a memory-efficient representation of the mining algorithms’ intermediate data, we propose compact data structures and even employ explicit compression. Both methods together reduce the intermediate data’s size by up to 25x. The smaller memory requirements avoid or delay expensive out-of-core computation when large datasets are mined.

For coping with the high parallelism provided by current multiprocessor systems, we identify the performance hot spots and scalability issues of existing frequent-itemset mining algorithms. The hot spots, which form basic building blocks of these algorithms, cover (1) counting the frequency of fixed-length strings, (2) building prefix trees, (3) compressing integer values, and (4) intersecting lists of sorted integer values or bitmaps. For all of them, we discuss how to exploit available parallelism and provide scalable solutions. Furthermore, almost all components of the mining algorithms must be parallelized to keep the sequential fraction of the algorithms as small as possible. We integrate the parallelized building blocks and components into three well-known mining algorithms and further analyze the impact of certain existing optimizations. Our algorithms are already single-threaded often up an order of magnitude faster than existing highly optimized algorithms and further scale almost linear on a large 32-core multiprocessor system. Although our optimizations are intended for frequent-itemset mining algorithms, they can be applied with only minor changes to algorithms that are used for mining of other types of itemsets.
Acknowledgments

First of all, I would like to thank my advisor Wolfgang Lehner. He told me almost everything I know about data management and awakens my interest in data mining algorithms. I want to thank him for the many things I learned in from him, his patience, and his advice. Despite of his often overfilled appointment calendar, Wolfgang always took the time for discussions or answering my questions. I always felt welcome when I entered his office and after the years I'm convinced that “Sure, I have always time for you, Benjamin” is not an empty phrase. Summing up, I'm deeply grateful for being supervised by you.

I further want to thank the many people who either directly or indirectly contributed to this thesis: Rainer Gemulla for providing useful feedback and many valuable hints, for coauthoring and proofreading my papers, and the deep technical discussions. Matthias Böhm for proofreading, suggestions, the countless discussions and helping me in many situations, Ulrike Fischer for the many off-topic discussions. All three of you became friends I always can count on. Further thanks to Philipp Rösch for the jokes and pranks we played each other, Dirk Habich and Martin Hahmann who helped me facing challenges in the IBIT project, Thomas Kissinger for discussions and productive cooperation, Laurynas Siksnys for improving my English skills, Tim Kiefer for deep details about multiprocessor systems, Tomas Karnagel for giving me insights and access to not yet available hardware, Thomas Legler for invaluable comments about itemset mining, Thomas Willhalm for the productive cooperation and coauthoring, Bernhard Jäcksch for being a soulmate, Julian Eberius and Maik Thiele for providing hints to the thesis, Ines Funke for supporting me dealing with the university’s bureaucracy, and Sven Schmidt for supervising my diploma thesis and introducing me into the group.

I want to thank Johann-Christoph Freytag for co-refereeing this thesis and for providing many valuable comments to it. I’m also grateful to all proofreaders reading parts of this thesis: Matthias, Rainer, Stefanie, Tim, and Thomas. I further would like to express my gratitude to the remaining co-worker at the database technology group: Bernd, Frank, Gunnar, Hannes, Katrin, Lars, Markus, Robert, Steffen, and Tobias. All of you provide an excellent working atmosphere in our group. I cannot remember a single day, when I was not happy to go to work. I thank you all.

Finally, I want to thank my family and my friends for constantly supporting and motivating me. Especially, my wife Stefanie was always there for me, encouraged me so much over the years, and shored me up when I doubted myself. I truly would not made it without your endless support. Thank you!

Benjamin Schlegel
Dresden, March 19, 2013

1In case he exists, I also want to thank Matthias2.
# Contents

## I. Introduction and State-of-the-Art

1. Introduction

2. Frequent-itemset mining
   2.1. Itemset Mining
   2.1.1. Application areas
   2.1.2. Mining for frequent-itemsets
   2.1.3. Mining for infrequent-itemsets
   2.2. Sequential Algorithms
   2.2.1. Apriori
   2.2.2. Eclat
   2.2.3. FP-growth
   2.3. Parallel algorithms
   2.3.1. Shared-memory systems
   2.3.2. Distributed systems
   2.3.3. Graphic processors
   2.4. Out-of-core and Sampling-based Algorithms
   2.4.1. Out-of-core algorithms
   2.4.2. Sampling-based algorithms
   2.5. Parallelization opportunities and parallel programming
   2.5.1. Instruction-level parallelism
   2.5.2. Data-level parallelism
   2.5.3. Thread-level parallelism
   2.6. Summary

## II. Basic algorithms

3. Sorted-set intersection
   3.1. Background and Related Work
   3.1.1. Related Work
   3.1.2. List intersection
   3.1.3. Bitmap intersection
   3.2. List intersection using speculative comparisons
   3.2.1. Basic intersection
   3.2.2. Hierarchical intersection
   3.3. Experiments
   3.3.1. Setup
## Contents

3.3.2. Results ............................................. 66
3.4. Summary and Discussion ..................................... 69

4. Compression 71
4.1. Background and Related Work .................................. 72
   4.1.1. Application of compression in itemset mining ............... 72
   4.1.2. Classification of integer compression techniques ............ 73
   4.1.3. Static Compression Techniques ................................. 75
   4.1.4. Delta Encoding ........................................... 77
   4.1.5. Parallelization opportunities ................................ 78
4.2. Vectorized Integer Compression ................................. 78
   4.2.1. Layout of Compressed Data ................................. 79
   4.2.2. k-Wise Null Suppression ................................ 80
   4.2.3. k-gamma Encoding ........................................ 81
   4.2.4. Delta encoding .......................................... 83
4.3. Experiments ................................................... 84
   4.3.1. Compression results ..................................... 85
   4.3.2. Performance results ..................................... 88
4.4. Summary and Discussion ...................................... 94

III. Frequent-itemset mining algorithms 97
5. cApriori 99
   5.1. Data representation ....................................... 100
      5.1.1. Page-wise layout .................................... 100
      5.1.2. Memory management .................................. 102
   5.2. Support counting .......................................... 103
      5.2.1. Parallel direct counting ................................ 103
      5.2.2. Lookup-based subset counting ......................... 107
      5.2.3. Parallel trie-based subset counting .................... 112
   5.3. Candidate and Database Pruning ............................. 113
      5.3.1. Candidate pruning .................................... 113
      5.3.2. Database pruning and trimming ........................ 114
   5.4. Experiments .................................................. 116
      5.4.1. Setup ............................................... 117
      5.4.2. Single-threaded ...................................... 119
      5.4.3. Multi-threaded ....................................... 127
   5.5. Summary and Discussion .................................... 132

6. cEclat 135
   6.1. Data representation ....................................... 135
      6.1.1. Logical representation ................................ 136
      6.1.2. Physical representation ............................... 138
      6.1.3. Conversion into the vertical layout .................... 138
   6.2. Support counting .......................................... 140
Part I.

Introduction and State-of-the-Art
1. Introduction

“I am terrified by terabytes”
– Anonymous

Data mining is an important part of computer science; it aims on finding information that is not already known by users in datasets of various application areas. The ever-growing amounts of data make mining in most of them indispensable since the datasets often contain valuable information that cannot be obtained manually. Frequent-itemset mining is a popular part in the area of data mining with the goal of finding values or items that co-occur frequently together in a dataset. It has many application areas like market-basket analysis [Agrawal et al., 1993], bioinformatics [Creighton and Hanash, 2003], recommender systems [Fu et al., 2000], web-mining [Mobasher et al., 1996], astronomy [Borne, 2001], medicine [Ohkunle and Ehikioya, 2002], weather prediction [Zhang et al., 2004], fraud detection [Brause et al., 1999], network intrusion detection [Lee and Stolfo, 1998], geography [Koperski and Han, 1995], and others. In many of these applications (e.g., network intrusion and fraud detection), the response time of the mining process is important because the process is often executed and the results are required as soon as possible. In other application areas (e.g., in bioinformatics and mining of astronomical data) the processed data volumes easily reach several terabytes of data. Thus, there is a need for efficient algorithms.

In general, there exists a large number of well-designed itemset mining algorithms that are very efficient on small datasets. However, most of them cannot be applied on large datasets because they either or both (1) have high memory requirements and (2) do not take advantage of the full spectrum of available parallelism provided by modern processors. Both issues need to be addressed to cope with current hardware trends.

The high memory requirement of most frequent itemset mining algorithms is problematic because accessing secondary memory, i.e., magnetic or solid-state disks, is getting more and more expensive compared to accessing main memory. A single processor, for example, could load about 30GB/s from main memory whereas the bandwidth of a typical solid-state disc is often below 0.3GB/s.¹ This gap of two orders of magnitude is believed even to be widen in future. Furthermore, not only the bandwidth is an issue; also the access latency differs significantly between main memory and disk. For example, a single disk access is about one million times slower than a single access to main memory. For this reason, the complete working set of an algorithm should fit in main memory for an efficient mining. Many mining algorithms, however, have large space requirements so that disk access is often required.

Modern processors provide a high degree of parallelism. Former processor generations got faster by increasing the processors core frequency; however, because of power and thermal constraints it reached the physical limits so that speedup nowadays is achieved by adding more parallel units. Thread-level parallelism allows to run multiple threads in parallel; it is often employed by parallel frequent-itemset mining algorithms. However, most of these algorithms were designed and evaluated for only a small number of threads, i.e., often only up to eight threads. Today’s available processors like Intel’s Xeon Westmere EX and AMD’s Interlagos consist already of 10 and 16 cores, respectively. These processors provide fast interconnect ca-

¹Considering processor and solid-state disk in the same price class.
1. Introduction

pabilities, e.g., Intel’s QuickPath Interconnect [2009] or AMD’s HyperTransport [2004], which allow to build systems with multiple processors that provide a single shared memory with a common address space. For example, an 8-socket system equipped with 10-core processors can have up to 80 cores. With simultaneous multithreading enabled, this allows 160 parallel threads. Moreover, research directions like Intel Terascale or the Single-chip cloud computer\(^2\) scale single processors to 100 cores and more. Hence, many parallel mining algorithms do not scale on current or future systems. Besides thread-level parallelism, modern processors also provide instruction- and data-level parallelism. Instruction-level parallelism allows to process multiple instructions in parallel but can only be exploited by code without or with only few control and data dependencies. Data-level parallelism allows to process multiple data elements in parallel. Compilers can identify and automatically parallelize code to take advantage of instruction- or data-level parallelism, however, in most cases this works only well for simple code fragments. In all other cases, the complete algorithm as well as the data layout has to be changed to benefit from such types of parallelism. Currently, only a few mining algorithms employ such changes to the algorithm and data layout.

The first challenge that is faced in this thesis is reducing the large memory requirements of mining algorithms to allow mining of very large datasets without accessing secondary memory. The datasets being mined and the algorithms’ intermediate state must fit in main memory at all times because out-of-core processing is often orders of magnitudes slower than processing in main memory. The datasets are often accessed multiple times during several mining invocations and must thus be kept in main memory because a repeated loading from disks or over a network incurs high costs. Compression can be used to reduce the datasets’ size and besides allows to keep datasets that are uncompresses even larger than the available memory. Compression itself, however, incurs overhead that clearly should not outweigh the costs for accessing a file from network or disks. Hence, efficient encodings must be found that provide a high compression rate but incur only small overhead for encoding and decoding at the same time. Especially decoding should be fast because it is repeatedly required. Besides reducing the size of the datasets, the memory required by the algorithms should be reduced. Many existing algorithms use rather inefficient data structures so that they require even more space than the dataset that is mined by them. As we will discuss in this thesis, there is not a single frequent-itemset mining algorithm that superior over all other algorithms. For this reason, not only the core data structures of a single algorithm should be improved.

The second challenge is to exploit available parallelism that is provided by current multi-processor systems in frequent-itemset mining algorithms. For that purpose, all performance hot spots and scalability issues of existing algorithms must be identified. The known hot spots of most of them are (1) counting the frequency of fixed-length strings, (2) building prefix trees, (3) compressing integer values, and (4) intersecting lists of sorted integer values or bitmaps. For all of them, it must be carefully examined which types of the available types of parallelism can be exploited. For exploiting instruction-level and data level-parallelism, the data layouts and algorithms often must be changed. For exploiting thread-level parallelism, synchronization between the threads must be minimized and good load balancing is crucial to scale to a large number of threads. Furthermore, almost all components of the mining algorithms (e.g., dataset parsing and conversion, counting frequent items) must be parallelized to keep the sequential fraction of the algorithms as small as possible.

\(^2\)The current prototype has 48 cores on a single chip.
Contributions

The key contributions of this thesis are:

1. We survey and discuss application areas as well as different popular itemset mining variants. Since the focus in this thesis is on finding frequent-itemsets, we review three well-known frequent-itemset mining algorithms and respective optimizations. We survey parallel algorithms for shared-memory systems, distributed systems, and graphic processors; approximate algorithms and algorithms for out-of-core algorithm datasets. We further provide an overview about the available parallelism in modern processors and how it can be exploited.

2. We propose efficient parallel sorted-set intersection algorithms for integer values of different precision, which can be exploited in all frequent itemset mining algorithms that are based on repeatedly intersecting sets. The proposed algorithms can be used for intersecting sets of integer values as well as bitmaps. All of the parallel algorithms exploit the three types of parallelism and achieve high speedups compared to highly efficient sequential algorithms.

3. We propose highly efficient parallel versions of well-known static encodings. This includes parallel versions of null suppression, gamma encoding, and delta encoding, which are used for compressing large sets of integer values, i.e., datasets in frequent itemset mining. The parallelized encodings are tailor-made for exploiting instruction-level and data-level parallelism so that they achieve high bandwidths for compression as well as for decompression.

4. We introduce a parallel frequent-itemset mining algorithm that relies on counting fixed-length strings. CAPRIORI is based on highly optimized data structures for counting, techniques for reducing its internal state, and an efficient memory layout. For parallel counting, we propose a parallel processing model that scales very well on systems providing a large number of threads.

5. We propose a parallel version of an existing frequent-itemset mining algorithm that is based on the intersection of sets. ECLAT relies on a memory-efficient set representation that combines the advantages of lists and bitmaps. We discuss the parallelization of all its key components, e.g., the conversion of the input dataset into the sets and intersecting the sets, and evaluate it using large datasets.

6. We propose two novel data structures for frequent-itemset mining algorithms that use prefix trees to represent the datasets being mined. Both data structures require about an order of magnitude less memory compared to their original data structure and can be seamlessly integrated in such algorithms. The resulting smaller memory footprint allows to delay expensive out-of-core computation so that large datasets can be processed in main memory. Finally, we present an effective parallelization of the prefix-tree building process. While previous parallel tree-building approaches scale only to a small number of threads, our algorithm relies on a tree-specific partitioning and allows almost linear speedups with an increasing number of threads. The parallel tree-building algorithm is furthermore well-suited for sorting large mining datasets in parallel.
1. Introduction

Outline

The thesis is divided—as illustrated in Figure 1.1—into four major parts, which contain in total 8 chapters. Each of our key contributions forms an own chapter.

| Part I: Introduction and State-of-the-Art | 1. Introduction |
| Part II: Basic algorithms | 2. Frequent-itemset mining |
| Part IV: Summary | 5. cApriori | 6. cEclat | 7. CFP-growth |
| | 8. Conclusion |

Figure 1.1.: Outline of this thesis

The first part contains this introduction (Chapter 1) and Chapter 2, which introduces the necessary background for this thesis. This includes reviewing the association rule mining process, typical application areas, itemset types, and different sequential as well as parallel frequent-itemset mining algorithms. The remainder of Chapter 2 includes a discussion of parallelization opportunities and respective programming techniques. Part II contains two chapters and comprises algorithms that are essential for frequent-itemset mining but are not restricted to this application area and thus have their related work mainly outside of the mining community. This includes parallel sorted-set intersection algorithms (Chapter 3) and parallel compression algorithms (Chapter 4). In Part III, we present three parallel frequent-itemset mining algorithms for which we parallelized each of their performance critical phases. We propose cAPRIORI, cECLAT, and CFP-GROWTH, which are based on the existing algorithms APRIORI, ECLAT, and FP-GROWTH, in the Chapter 5, 6, and 7, respectively. Finally in Part IV, we conclude this thesis (Chapter 8).
2. Frequent-itemset mining

This chapter presents the necessary background of this work. We start in Section 2.1 with an overview of the association rule mining process, typical application areas, and different types of itemsets. As we will show, mining for frequent itemsets forms the foundation for mining of other types of itemsets (e.g., closed-frequent and rare itemsets) so that optimizations applied to frequent-itemset mining algorithms can also—with only small changes—be applied on their respective algorithms. As there is no best algorithm for frequent-itemset mining in general, we review three sequential algorithms for mining frequent-itemsets, which are the ancestors of almost all other frequent-itemset mining algorithms, in Section 2.2. Thereby, we reveal the algorithms performance-critical parts that we address in this thesis. We further review and discuss existing optimizations for each one of the three algorithms. In Section 2.3, we give an overview about existing parallel frequent-itemset mining algorithms. We focus mainly on algorithms for “shared-memory” and distributed systems but also look at approaches that exploit graphic processors. We discuss algorithms for very large datasets in Section 2.4. This includes “precise” algorithms that have an improved disk access pattern as well as sample-based algorithms with approximate results. In the remainder of this chapter (Section 2.5), we discuss parallelization opportunities and parallel programming techniques that are exploited in this thesis.

2.1. Itemset Mining

Association rule mining is part of the well-known knowledge discovery process, which aims to automatically discover useful information in large datasets that would otherwise remain unknown. More specifically, association rule mining is used to discover hidden relationships in data like associations [Agrawal et al., 1993], correlations [Brin et al., 1997a], and causal structures [Silverstein et al., 2000]. The typical association rule mining process, which is based on the general knowledge discovery process [Tan et al., 2005], is illustrated in Figure 2.1. It consists of three stages: (1) data preparation, (2) itemset mining, and (3) rule generation. The process may iteratively repeated until the user is satisfied with the obtained rules.

Data preparation is done to clean up the data that should be analyzed. This includes the removal of outliers, coping with missing or empty values, and the transformation of the data into a layout that is amenable for the mining stage. Furthermore, also the amount of data is sometimes reduced (e.g., via sampling, aggregation, etc.) based on user knowledge to lower the costs of the following stages. This, however, may or may not lead to an information loss.
2. Frequent-itemset mining

so that certain interesting rules cannot be found anymore. It is therefore crucial to optimize
the following two stages to avoid data reductions caused by performance reasons. For more
details on data preparation, the interested reader is referred to Pyle [1999].

The actual itemset mining is done after the data preparation. Let \( \mathcal{I} = \{ a_1, \ldots, a_m \} \) be a set
of items and \( \mathcal{D} = \{ T_1, \ldots, T_n \} \) be a database of transactions, where each transaction \( T_i \subseteq \mathcal{I} \)
consists of a set of items. The relative support of an itemset \( I \subseteq \mathcal{I} \) denotes the percentage
of transactions that contain the itemset \( I \). The goal of itemset mining is to find all itemsets
that satisfy a certain minimum relative support \( \xi \). This value—or also multiple values for
some itemset mining variants—is either defined by the user or obtained during processing.
The chosen \( \xi \) value thereby influences the effort for mining; it becomes more expensive as \( \xi \)
decreases because more frequent itemsets are found. In what follows, we partially use also the
minimum absolute support \( \hat{\xi} \), which denotes the absolute number of transactions in which an
itemset must occur at least to be considered as frequent. Thereby, \( \hat{\xi} \) is given by \( \hat{\xi} = \xi \cdot |\mathcal{D}| \).
The output of the itemset mining phase is an often large set of itemset–support pairs.

The third stage—the rule generation—takes the discovered patterns of the mining process
as input and creates rules of the form \( X \Rightarrow Y \) where \( X \) and \( Y \) are disjoint itemsets. Thereby,
the number of rules is minimized using certain techniques and measures to obtain interesting
rules [Piatetsky-Shapiro, 1991, Klemettinen et al., 1994]. There exists a huge number of
different measures (or metrics) that allow to determine the strength or interestingness of a
single rule. For example, a well-known measure is the confidence of a rule; it indicates how
often an itemset \( x \) co-occurs with an itemset \( y \) in transactions that contain \( x \). The impact of
the applied metrics depends on the application and thus requires user knowledge. Tan et al.
[2005] provides an overview and a detailed description of different interestingness measures.

In this work, we focus solely on speeding up the itemset mining stage so that very large
datasets can be processed. We consider a prepared dataset as input and return itemsets that
could be used for the rule generation. In the following, we provide details about application
areas in which itemset mining plays an important role. We also discuss the characteristics of
their typical datasets (i.e., their size) to further motivate our work.

2.1. Application areas

There are many application areas in which itemset mining is used. Each of these areas has
typical datasets that can be classified according to different characteristics like the number
of transactions \( |\mathcal{D}| \), average transaction cardinality \( \hat{\mathcal{C}} \), and total number of distinct items \( |\mathcal{I}| \).
These characteristics determine the dataset size and thus have a large impact on the runtime
of the algorithms, i.e., the higher the values are, the larger are the datasets and the more
expensive is the mining. If the datasets are stored in ASCII format, then a single item is
usually encoded using 5 to 8 bytes (depending on the number of distinct items \( |\mathcal{I}| \)). A dataset
with \( |\mathcal{D}| = 10,000 \) transactions and an average cardinality of \( \hat{\mathcal{C}} = 20 \) items per transaction
would consume about 1MB. In the following, we focus on the major application areas of
itemset mining that received a lot of attention so far and take a closer look on the potential
sizes of their respective datasets.

Retail business

The perhaps best-known application area for itemset mining is the market-basket analysis
[Agrawal et al., 1993, Agrawal and Srikant, 1994, Brin et al., 1997b, Liu et al., 1999] used for
2.1. Itemset Mining

retail business. The goal is to determine which products are frequently bought together to support certain business decisions; for example, which item should be put on sale or how to better place products within the supermarket. In this application area, the number of distinct items \( I \) is given by the number of different products; each transaction consists of items that are bought together by a customer at the same point in time or over a period of time.

Typical datasets of the market-basket analysis have many transactions but only a small number of distinct items and only few items per transaction. A typical supermarket chain has less than 100,000 products, sells in average less than 30 products per customer but has billions of transactions per year. Hence, the data obtained every year for such a retail company would be about 140GB. The datasets would even be larger for larger retail companies like Walmart; it has 7.2 billion transaction per year and about 500,000 products.\(^1\)

**Bioinformatics and Medicine**

Gene-expression analysis in bioinformatics [Wang et al., 2005] as well as mining medical image data in medicine [Olukunle and Ehikioya, 2002] are further important application areas of itemset mining. With more sensors in both areas—generating huge amounts of data containing valuable information—the analysis cannot be done by humans alone anymore.

DNA-microarray technology [Stoughton, 2005] allows to monitor the expression level of a set of genes within a biological experiment. This data is then used for gene-expression analysis, which relies on itemset mining [Creighton and Hanash, 2003, Pan et al., 2003, Carmona-Saez et al., 2006, Borgelt et al., 2011]; the task is to discover which genes are frequently activated together for certain experimental settings and conditions. The co-occurrence of the same activation level of some genes allows to explore and validate hypotheses of an underlying biological process. The gene-expression analysis data itself is often mapped to itemset mining data as follows: Genes are considered as items while the experiments represent transactions. Each transaction contains only those items (genes) with a certain activation level.

Itemset mining in medicine is often used as part of a medical image classification process to support diagnoses of physicians [Antonie and luiza Antonie, 2001, Olukunle and Ehikioya, 2002, Zaiane et al., 2002]. An example for such a classification process is brain tumor detection using computed tomography scan images [Rajendran and Madheswaran, 2010]. This is often done as follows: preprocessing is used to remove over-brightness, over-darkness, and noise in the background of the image. Based on the clean images, a number of different features (color, structure, etc.) are extracted with the help of image processing techniques. These features are considered as distinct items \( I \) whereas the images itself represent the transactions. Finally, obtained rules by itemset mining are considered as input for the classifier.

Datasets of both application areas, gene-expression analysis and mining medical image data, have similar data characteristics. The datasets have a large number of distinct items—represented by the number of genes or image features—and transactions with a very high cardinality; there are many genes per test or many present features per image. The number of transactions is typically lower compared to other application areas because it depends on the number of tests or images, which is usually low. Nevertheless, the datasets could easily exceed several hundred gigabytes considering, for example, 1GB for a full genome of a person and 4GB per 3D computed tomography scan [Gibbons, 2008].

2. Frequent-itemset mining

Web-mining and Recommender systems

Itemset mining is also part of increasingly popular tools that improve the layout of web-sites and support web-shop customers in finding products of interest.

The goal of web-usage mining [Mobasher et al., 1996, Cooley et al., 1997, Punin et al., 2001] is to improve design and organization of corporate web-pages. For that, web-log data (user navigation histories) is analyzed to obtain correlations among users accesses on the web-pages. For itemset mining—as one way to analyze the web-log data—a visit of an user in a certain time span represents a transaction while the user’s requested URLs or clicked objects are considered as items of this transaction.

Online stores like Amazon.com, Netflix.com, or Expedia.com increasingly employ recommendation systems to provide suggestions like (“customers who like this item also like . . .”). This web-shop personalization is based on similarities and dissimilarities of customers preferences that are captured with their product ratings—and also buyings. Itemset mining is a popular way to obtain recommendations based on these ratings [Lin et al., 2002, Sandvig et al., 2007]. There are many ways to map ratings to transactions. A simple way is to categorize ratings with a defined threshold value into two classes: like and dislike. Based on that, transactions consist of users that like an item while there is a transaction for each item of the shop. Finally, rules are observed online whereas the number of rules is strongly reduced since only rules that contain the current customer are of interest.

Web-mining datasets (i.e., web-log data) could be quite large. These datasets have many transactions with high cardinality (thousands of items) and many distinct items. A web-site with 200,000 users can produce more than 1GB per day. Datasets used for recommendation systems have the same characteristics. For example, Netflix had over 1.9 billion ratings in 2007 and receives 2 millions ratings per day [Bennett et al., 2007].

Fraud detection and Network intrusion detection

Fraud detection and network intrusion detection are application areas for itemset mining in which mining is very often executed. In both application areas, itemset mining is a key component of the overall process.

The goal of credit card fraud detection [Brause et al., 1999] is to reduce misuse of credit cards; this is done by revising or aborting credit card transactions that are probably fraudulent. The main idea is to build general rules from confirmed fraud transactions using association rule mining. Credit card transactions are considered as transactions and features (like merchant code, account number, and client name) are used as items; the resulting rules are used to classify legal transactions and fraud transactions.

Itemset mining is further used for fraud detection in the setting of Internet advertising commissioners [Metwally et al., 2005]. Since website publishers earn revenue for visiting users that click on advertisements, there is an incentive to increase this numbers with certain tricks. These tricks can be exposed by analyzing parts of the HTTP request stream made by customers while visiting the websites of potentially dishonest website publishers. The required stream could be provided by Internet Service Providers in form of anonymous log files of HTTP requests. The actual itemset mining is similar to web-usage mining (see above) but executed on unlimited streams with windows of small size (e.g., 100 elements).

Network intrusion detection [Lee and Stolfo, 1998, Lee et al., 2000] is used to detect intruders after a break-in in a computer system. Many solutions for this highly relevant problem
2.1. Itemset Mining

analyze audit data (resource usage, certain commands with arguments, network packages, etc.) produced by the users of the computer system. A deviation from regular users behavior could indicate an intrusion. Similar to credit card fraud detection, itemset mining is used to build classifiers whereas these classifiers detect anomalies and report intrusions. Without intrusion, the rules are continuously updated with new audit data of a session. The mapping from audit data to transactions depends on the audit source. For network audit data, for example, a complete TCP connection represents a transaction while attributes like the number of source bytes, the service, duration, and timestamps are considered as items.

The characteristics of fraud detection as well as network intrusion detection are quite similar. The transaction cardinality is often low while the number of transactions could be large. For example, capturing the TCP connections easily produces several hundred gigabytes of data. Furthermore, there is a large number of different items and mining is very often executed either on the datasets to keep the classifier’s rules up-to-date or in time windows on transaction streams.

Summary

There are many application areas in which itemset mining plays an important role. In many of them itemset mining is repeatedly executed on ever-growing very large datasets with several hundreds of gigabytes of data. Mining such datasets requires either an effective pre-processing that reduces the size of the datasets at the cost of losing potential valuable information or very efficient itemset mining algorithms. Our goal is to provide such algorithms.

Figure 2.2.: Dataset characteristics of different application areas

The discussed characteristics of a dataset influence—besides of its size—also which algorithm is best suited for processing it. In Section 2.2, we provide a simple heuristic that is used for selecting appropriate datasets for our experiments. For that purpose, we classify the datasets of our example application areas as shown in Figure 2.2. The optimal algorithm assignment was discussed by Veloso et al. [2002] and HooshSadat et al. [2011]. Both automatically select the best algorithm for a certain dataset based on its characteristics—although they use a larger number of dataset characteristics as we use. Interestingly, HooshSadat et al. [2011] chose the same three algorithms that we chose as starting basis for our optimizations for their optimal assignment algorithm. In the following, we will review different types of itemsets that influence the costs of mining and differ in the size of the result set.

2.1.2. Mining for frequent-itemsets

There are several types of itemsets for which mining algorithms exist; the most well-known are frequent itemsets [Agrawal et al., 1993], which include all possible itemsets \( I \in \mathcal{I} \) that
2. Frequent-itemset mining

are more frequent than the given \( \xi \) value; i.e., an itemset \( I \) is frequent if it occurs at least \( \hat{\xi} \) times in the transaction database. The \( \xi \) value is crucial for frequent-itemset mining. For low \( \xi \) values, the number of output itemsets may be very large while—on the opposite side—for high \( \xi \) values, the result set might be empty or only few itemsets are found.

Finding frequent itemsets within a transaction database \( D \) is subject of many research papers. The brute-force approach works by generating all possible itemsets based on the distinct items of \( D \) and count their occurrence within it. This, however, requires to count an exponential number of itemsets, i.e., for \( n \) distinct items there are \( 2^n - 1 \) possible non-empty itemsets. Hence, for even few distinct items like 500 different products in a retail business dataset, the support of \( 2^{500} - 1 \) itemsets need to be obtained; this makes the brute-force approach infeasible. A more efficient method constitutes the AIS algorithm [Agrawal et al., 1993]. Its main idea is to grow itemsets starting from the distinct items. Whenever an itemset is found to be frequent, then candidate itemsets are derived from it; these are created by combining any item from all transactions in which such a frequent item occurs. The candidates are then counted within the database to decide which of them are frequent and which are not. The algorithm proceeds until no new candidates can be derived from newly obtained frequent itemsets. AIS is much more efficient than the brute-force approach but still generates too many candidates. For this reason, all state-of-the-art frequent-itemset mining algorithms rely on the Apriori principle to minimize the number of candidates.

**Definition 1** (Apriori Principle). A \( k \)-itemset is only frequent if all of its \( 2^k - 2 \) non-empty subsets are also frequent.

Basically, the Apriori principle allows to prune all candidates that have at least one infrequent subset. In Section 2.2, we review sequential frequent-itemset mining algorithms that rely—either implicit or explicit—on it. The Apriori principle, however, can also be exploited to reduce the size of the result set. For frequent-itemset mining, the result set usually contains a lot of redundancy because there exists an exponential number of subsets for single long itemset. For example, if the itemset \( \beta = abc \) is frequent, then also its subsets \( \alpha_1 = ab \), \( \alpha_2 = ac \), and \( \alpha_3 = bc \) must be frequent and thus they are in the results set. In general, an itemset of length \( k \) consists of \( 2^k - 2 \) subsets. For this reason, there are several techniques that deliver a smaller result set and further often reduce the mining effort. In the following, we consider that \( \alpha_1, \alpha_2, \alpha_3, \) and \( \beta \) occur 60, 50, 70, and 50 times within the transaction database, respectively.

**Closed frequent-itemsets** The set of closed frequent itemsets [Pasquier et al., 1999, Pei et al., 2000, Zaki and Hsiao, 2002] forms a condensed representation of the set of full frequent itemsets. An itemset \( I \) is a closed frequent itemset only if it is frequent and none of its supersets has the same support as \( I \). In our example, only the itemsets \( \alpha_1, \alpha_3, \) and \( \beta \) must be retained in the result set, because \( \alpha_2 \) has the same support as \( \beta \). In general, the set of closed itemsets is often orders of magnitudes smaller than the set of frequent itemsets but provides all necessary information to recreate it.

There are several further condensed representations for frequent itemsets like *free sets* [Boulicaut et al., 2000], *non-derivable itemsets* [Calders and Goethals, 2002], and *disjunction-free sets* [Bykowski and Rigotti, 2001]. Calders and Goethals [2007] give an excellent overview about the relationships between these condensed representations.
Maximal frequent-itemsets A subset of the closed frequent itemsets constitute the maximal frequent itemsets [Burdick et al., 2001]. Each frequent itemset that has no frequent superset is denoted as maximal frequent itemset. In our example, only the itemset $\beta$ and its support would be retained. The number of maximal frequent-itemsets is often much smaller—orders of magnitudes—than the number of closed frequent itemsets. Hence, it is only a small fraction of the set of frequent itemsets. However, this comes at the cost of losing information; the support values of the full set of all frequent itemsets cannot be reconstructed. It is only known that the support of the itemsets $\alpha_1$, $\alpha_2$, and $\alpha_3$ must be greater than 50. Nevertheless, in some application areas this is acceptable.

Top-$k$ itemsets A hybrid between maximal and closed frequent itemsets are the top-$k$ frequent itemsets [Han et al., 2002]. They consist of only the $k$ closed frequent itemsets that have an user-specified minimum length and are more frequent than all other itemsets of this length. For example, if there are no further frequent 2-itemsets with a support greater or equal than 50, then the top-3 frequent itemsets with a minimum length of two are $\alpha_3$, $\alpha_1$ and $\beta$. The itemset $\beta$ would be preferred over $\alpha_2$ because of its length. The top-$k$ itemsets are—depending on $k$—a strict subset of the closed frequent itemsets. However, the main advantage of top-$k$ frequent itemsets is that no $\xi$ value must be set before mining. Only $k$ and the minimum length need to be set; this is often easier for non-experts because no knowledge about the dataset is required.

To summarize, all other frequent-itemsets representations lead to—for a given dataset and minimum support—often much smaller results sets compared to full frequent-itemset mining. Furthermore, respective algorithms that are tailor-made for these representations like CLOSET [Pei et al., 2000], MAFIA [Burdick et al., 2001], CHARMO [Zaki and Hsiao, 2002], and TFP [Han et al., 2002] are usually faster than pure frequent-itemset mining algorithms because they can employ better search and pruning techniques. Nevertheless, all of them rely on almost the same techniques for obtaining the support of itemsets and use the same data representations as used for full frequent-itemset mining. In this thesis, we focus mainly on optimizing these two basic building blocks of itemset mining algorithms. For this reason, we restrict our attention to full frequent-itemset mining since we believe that almost all of our improvements on support counting and the internal data representation can also be applied to the other algorithms with only minor changes. In the following, we discuss rare-itemset mining for which similar arguments hold.

2.1.3. Mining for infrequent-itemsets

Rare-itemset mining [Liu et al., 1999, Weiss, 2004] focus on finding association rules of items that occur only few times in the dataset. Since a rare item can frequently co-occur with other rare items, such rules might be interesting in scenarios where the confidence is more important than the support. A typical example of a rare association rule in the retail business is “frying pan” ⇒ “slotted turner”. Both items are rarely purchased in a supermarket but may be bought often together.

In fact, rare itemset mining could be achieved with full frequent-itemset mining and a very low minimum support value $\xi$. However, this would lead to a combinatorial explosion of the output rules; most of them would be considered as uninteresting. Besides, the mining process would be very expensive or even not feasible for large datasets with many items.

13
2. Frequent-itemset mining

There are several algorithms for efficient rare itemset mining that rely on an extension of the traditional association rule model [Liu et al., 1999]. Instead of defining only a single $\xi$ value for all distinct items, each item $\gamma \in \mathcal{I}$ has its own assigned value $\xi_\gamma$. A rule inherits the lowest minimum support value of all items in that rule. The mining algorithms itself are based on \textsc{Apriori} [Liu et al., 1999, Szathmary et al., 2007] and \textsc{FP-growth} [Hu and Chen, 2006, Kiran and Reddy, 2011]. In this thesis, we optimize both base algorithms; we believe that most of our improvements are also applicable to these extended versions.

2.2. Sequential Algorithms

In this section, we review existing sequential frequent-itemset mining algorithms. As stated by several authors [Veloso et al., 2002, Goethals and Zaki, 2004, HooshSadat et al., 2011], there is no “best” algorithm for frequent-itemset mining in general; each algorithm has its bottleneck so that there is always a dataset on which other algorithms perform better. However, algorithms can be divided into classes according to their underlying base data layout (or transaction representation), search strategy, and intermediate data structures; in each of these classes usually there exists a single algorithm that is superior to all other algorithms of this class. In this thesis, we take three—well-known—algorithms that differ in the underlying base data layout. The algorithms \textsc{Apriori} [Agrawal and Srikant, 1994], \textsc{Eclat} [Zaki et al., 1997c], and \textsc{FP-growth} [Han et al., 2000] use the horizontal, vertical, and prefix-tree based layout, respectively, for representing the transaction base. In the following, we will review the sequential versions of these algorithms and respective optimizations.

2.2.1. Apriori

The \textsc{Apriori} algorithm [Agrawal and Srikant, 1994] is considered as the first frequent-itemset mining algorithm that can cope with large datasets. The algorithm iteratively generates and tests candidate itemsets to obtain frequent-itemsets. We will now discuss the used data layout, search strategy, and support counting step in more detail.

As mentioned before, \textsc{Apriori} represents the transaction base using the \textit{horizontal layout}. In this layout, the \textit{filtered transactions}, which do not contain infrequent items anymore, are stored in a list where each of them has an unique identifier and contains its respective items, i.e., items are assigned to transactions. This is usually realized by storing all items that fulfill $\xi$ of a transaction in a simple array and connect these arrays using a simple linked list.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{breath_first_search_space_traversal.png}
\caption{Breath-first search space traversal}
\end{figure}
2.2. Sequential Algorithms

During mining, APRIORI repeatedly scans the transaction database to build frequent itemsets of increasing cardinality. It traverses the candidate search space, which can be described using a lattice that contains all combinations of itemsets, in a breath-first manner. The frequent items $F_1$ are obtained in the first scan. Based on these items, the candidate itemsets $C_2$ of length 2 are generated. For example, if the items $a$, $b$, $c$, and $d$ are frequent, then the itemsets $\{ab, ac, ad, bc, bd, cd\}$ might be frequent. The frequency of these candidates is obtained within the second scan. Infrequent itemsets are removed whereas the frequent itemsets of $C_2$ are kept and used to generate candidate itemsets $C_3$ of length 3. This procedure is repeated until all of the candidates $C_k$ in the $k$-th iteration are found to be infrequent. To reduce the number of candidates in each iteration, the candidate generation process exploits the Apriori principle (cf. Definition 1). Recall that candidates that can never be frequent can be pruned. For example, consider the frequent itemsets $L_2 = \{ab, bc, bd, cd\}$. Based on these itemsets, only $bcd$ might be frequent since all of this its subsets are frequent. Contrary, the itemsets $abc$, $abd$, or $acd$ cannot be frequent because neither $ac$ nor $ad$ is frequent. A lattice describing the search space for the items $I = \{a, b, c, d\}$ is illustrated in Figure 2.3. Frequent itemsets are illustrated as white nodes whereas infrequent itemsets are gray; candidate itemsets have dashed lines. In general, the costs for generation and storage of candidate itemsets depends on the number of frequent items. Usually, the lower the number of frequent items is, the fewer candidates are generated during processing.

Within each scan, support counting is performed. It is used to extract the frequent itemsets from the candidates (i.e., to prune infrequent candidates) and to obtain the precise frequencies of the itemsets. The procedure is always the same: all transactions are scanned and for each of them \( \binom{n}{k} \) subsets of size $k$ are generated where $n$ denotes the length of the transaction and $k$ denotes the number of the scan. For each of these subsets, a lookup within the indexed candidates is performed. If the subset is found, then the support value of the candidate is increased. The employed candidate index depends on which scan is performed. Within the first scan, a simple lookup table is sufficient where the $i$-th entry holds the support of the item $i$. For the second and all later scans ($k > 1$), APRIORI employs a hash tree, which has $k$ levels. The tree is rebuild before each scan and stores the candidates at its leaf nodes. Each candidate has further a variable assigned that holds the candidate’s support.

The complete algorithm works as follows. The dataset is parsed to obtain the frequent items $F_1$. Based on these items, the dataset is transformed into the horizontal layout. Thereby, infrequent items are removed from the transactions. After the dataset has been fully converted, the algorithm executes two phases repeatedly until $C_k$ is empty:

**Candidate generation and pruning:** Generate the candidates $C_k$ within the $k$-th iteration using the frequent itemsets $F_{k-1}$. For that purpose, merge all itemsets in $F_{k-1}$ with all other itemsets in $F_{k-1}$ that share the same first $k-2$ items. Prune all candidates in $C_k$ of which at least one of its $k$ subsets is not frequent.

**Support counting:** Scan the transaction database and generate for each transaction all of its $k$-subsets. Search each of these subsets within the candidate hash tree. If it is found, then increase the support value of the respective candidate. After the scan is complete, the frequent itemsets in $C_k$ form $F_k$. Increase $k$ by one and goto the first phase.

APRIORI requires $k$ iterations (i.e., database scans) to obtain frequent itemsets of length $k$. If $k$ denotes the size of the largest found frequent itemset, then the output of the algorithm is given by $\bigcup_{i=1}^{k} F_i$. 

15
2. Frequent-itemset mining

APRIORI is often considered to be inferior compared to other frequent-mining algorithms because it requires many database scans during mining. This, however, is only an issue if the data is stored on disk and must be loaded within each scan. If the base data is located in main memory, APRIORI is usually superior over other algorithms when the datasets being mined have many small transactions (e.g., market-basket analysis datasets). Large transactions lead to high costs for support counting because the effort for subset generation grows polynomial with the transaction size so that other algorithms perform better. Nevertheless, even for datasets with many, mainly small transactions, support counting still forms the performance hot spot of APRIORI. Especially if the datasets are large, the runtime for candidate generation is negligible; only parsing and transforming the dataset then contributes to the overall runtime besides support counting.

Optimizations

A large number of optimizations were published for APRIORI that focus on (1) faster support counting, (2) a smaller physical database representation, (3) reducing the number of candidates, and (4) fewer scans during mining.

Optimizations for support counting have a strong impact because it amounts to a large fraction of APRIORI’s runtime. Zaki et al. [1996] propose short-circuited subset checking that avoids checking leaf nodes in the candidate hash-tree twice. More specifically, after reaching a leaf node in the hash-tree, all of its itemsets are checked whether these are available in the current transaction. Checked nodes are marked in the tree to avoid processing these nodes for other subsets of the same transaction again. Besides optimizing the hash-tree, also other data structures were employed to represent and index the candidates. Some implementations [Mueller, 1995, Borgelt, 2003] use prefix-trees for that. Each node in such a tree forms an array of entries, where each entry consists of an item and count field and a pointer to a successor node. All nodes in the path from the root node to a specific node represent an itemset; the node’s count value denotes the support of this itemset. Prefix trees can easily be extended with new nodes while inserting longer itemsets. Thus, rebuilding them during candidate generation is not necessary as for hash-trees. Finally, the direct count technique [Perego et al., 2001] uses an array to represent candidate 2-itemsets. The main idea is to transform a 2-itemset into an integer value and use this value as index in the array. For example, if there are 10 frequent items, the items a and d are mapped to the values 1 and 4, respectively, then the itemset ad could be mapped to $1 \cdot 10 + 4 = 14$. Hence, whenever the subset ad occurs in a transaction, the value in the array at position 14 is incremented. The hash function itself is more elaborate to avoid unused entries like aa or da, which cannot occur as subsets. Direct counting works well because only a single access per 2-itemset into the array is required. Perego et al. [2001] and Prado et al. [2004] use this technique only to count k-itemsets with $k = 1$ or $k = 2$. For larger itemsets, they combine it with a prefix-tree; the first $k - 2$ items of a k-itemset form a path in the tree while the last two items constitute an array, which is connected at the end of the path. Lucchese et al. [2004a] evaluated using the direct count technique up to the third iteration ($k = 3$).

There exist also several optimizations for reducing the memory requirements of the horizontal layout. Some algorithms [Park et al., 1995a, Perego et al., 2001] employ pruning techniques that remove unnecessary transactions and items in the database to reduce the effort of later scans. For example, all transactions that consist of only three frequent items can be removed before the 4-th scan because they cannot contribute to any frequent 4-itemset. Orlando et al.
[2003a] uses dynamic data type selection to reduce the size of the physical representation of the database. If the database consists of only 256 different frequent items, a single byte is sufficient to store an item. Hence, storing a transaction with five items requires only 6 bytes: one byte for the length information and 5 bytes for the items. Finally, Borgelt [2003] proposed a prefix-tree representation of the database. This tree is similar to the previously described candidate tree and summarizes all database transactions. Mining is employed directly on the tree by counting the subsets in all tree paths. Depending on the dataset, the tree might be smaller than the transaction database, however, building the tree is more expensive than building the regular horizontal representation.

Kosters and Pijls [2003] proposed a depth-first implementation of Apriori. The algorithm partitions candidates based on their first frequent item; each partition is process independently with the Apriori-typical multiple scans. It starts with the frequent item that has the smallest support. After processing all candidates in a partition, it uses the obtained frequent itemsets to reduce the number of candidates for mining the next partition. The algorithm requires less space to store the candidates compared to Apriori, however, it requires that the complete database fits into main memory since it is scanned very often.

There are several other algorithms that are based on Apriori or at least are very similar to it. The direct hashing and pruning (DHP) algorithm [Park et al., 1995a] approximately counts all pairs of items—using a hash-table—already in the first pass over the transaction database. These count values are then used for reducing the number of candidates in the second and later iterations; and for pruning transactions of the database that do not contribute on a frequent-itemset. However, Agrawal and Shafer [1996] showed in their experiments that this technique may even hurt the performance instead of improving it.

Nevertheless, the main idea of DHP was reused by other algorithms. They are all based on the assumption that a single scan of the complete database is prohibitively expensive and thus should be avoided. This holds true if the database does not fit into main memory. The number of scans can be reduced with pass-bundling: candidates of later scans are speculatively generated and their support is obtained within the earlier scans; e.g., some 3-itemsets are counted already within the second scan. The dynamic itemset counting (DIC) algorithm [Brin et al., 1997b] uses dynamic candidate generation instead of the level-wise approach of Apriori. It divides the transaction database in multiple partitions. After processing a partition, it employs candidate generation based on the so far obtained support values. Sear [Mueller, 1995] uses the same technique but maintains all candidates in a prefix-tree that allows to easily add new candidates. In general, speculative candidate generation leads to unnecessary overhead that usually cannot offset the fewer scans. This is especially true for skewed databases or if the database fits into main memory. So as stated by Bodon [2006], there is no public implementation of these algorithms available that outperforms plain Apriori.

Finally, Spear [Mueller, 1995] is an out-of-core algorithm (see Section 2.4 for more details) that divides the database into multiple partitions. Each partition fits into main memory and Apriori is employed to obtain local frequent itemsets. All of these itemsets are then combined and verified using a full scan over the complete database.

2.2.2. Eclat

Eclat [Zaki et al., 1997c] employs also candidate generation and testing but uses the vertical data layout. In this layout, transactions are assigned to items instead of assigning items to transactions like in the horizontal layout. Each frequent item has a tid-set assigned that
contains the ids of the transactions (tids) in which the item occurs. ECLAT represents tid-sets using lists (tid-lists), which are ascending sorted. Intersecting the tid-lists of two items reveals the transactions that contain both items. In general, the support of a $k$-itemset can be easily determined by intersecting the tid-lists of two arbitrary subsets of length $(k - 1)$ and counting the elements in the result tid-list. For example, if the itemsets $\alpha_1 = ab$ and $\alpha_2 = ac$ occur in the transactions $T(\alpha_1) = \{1, 3, 4, 6\}$ and $T(\alpha_2) = \{1, 2, 4, 6, 7\}$, respectively, then the itemset $\beta = abc$ has a support of 3 since it occurs in the transactions $T(\beta) = \{1, 4, 6\}$, which is the intersection result of $T(\alpha_1)$ and $T(\alpha_2)$.

Both, the horizontal and the vertical data layout, have roughly the same memory requirements since they represent the same number of items. However, the intermediate results during mining are much larger for the vertical data layout. Many tid-lists of the frequent $k$-itemsets must be maintained for mining of $(k + 1)$-itemsets. For example, if the itemsets $ab$, $ac$, and $bc$ are frequent, then two of the three tid-lists must be maintained for creating the tid-list of the itemset $abc$. Hence, the Apriori-like bottom-up candidate generation and testing, i.e., counting the support of all $k$-itemsets within the $k$-th scan, cannot be applied because almost all tid-lists of frequent $k$-itemsets would be required in the $(k + 1)$-th iteration. Indeed, the size of the tid-list of an itemset is usually smaller than or at most equal to the size of the tid-lists of its subsets, however, the number of frequent $(k + 1)$-itemsets is usually much larger—at least in the first iterations—than the number of frequent $k$-itemsets. For example, 100 frequent items can lead up to 4950 possible frequent 2-itemsets of which many could be required to build frequent 3-itemsets. Thus, depending on the cardinality of the 2-itemsets tid-lists, the intermediate tid-lists would easily exceed the size of the transaction database.

To reduce the size of the intermediate results, ECLAT employs equivalence class clustering to break the search space into smaller sub-problems. All candidate $k$-itemsets that share a prefix of length $(k - 1)$ and thus differ only in their last item form an equivalence class. For example, the itemsets $\{ab, ac, ad\}$ and $\{bc, bd\}$ would be in the equivalence classes $[a]$ and $[b]$, respectively. All itemsets produced by an equivalence class are independent from itemsets produced by other classes, i.e., for obtaining the support of any itemset in a class no itemset of any other class is required. This effectively reduces the number of intermediate tid-lists, which have to be maintained, and also allows an effective parallelization. Figure 2.4 illustrates the mapping of itemsets to the four equivalence classes $[a]$, $[b]$, $[c]$, and $[d]$. 

![Figure 2.4: Itemsets in the equivalence classes $[a]$, $[b]$, $[c]$, and $[d]$](image-url)
2.2. Sequential Algorithms

The algorithm itself uses bottom-up candidate generation and testing within each class. However, there is no distinct candidate generation; it is done simultaneous while mining the sets. The complete algorithm works as follows:

**Preparation** Similar to **Apriori**, the database is scanned twice. The first scan reveals the frequent items $F_1$ while the second scan is used to transform the database into the vertical layout. For each item $\gamma \in F_1$, a tid-list $T(\gamma)$ is created, which contains all transactions the item $\gamma$ is part of.

The set of frequent items $F_1$ is added to the final result set $F$ and bottom-up mining is performed with $F_1$ as parameter.

**Bottom-up mining** A set $\hat{F}_k$ of frequent $k$-itemsets is used as input. The equivalence class $[\alpha_i]$ for each itemset $\alpha_i$ of $\hat{F}_k$ is recursively mined as follows.

For each $\alpha_i \in \hat{F}_k$ with $i = 1, 2, \ldots, |\hat{F}_k|$ do:

1. Initialize the frequent $(k+1)$-itemsets $\hat{F}_{k+1}$, which are within the equivalence class $[\alpha_i]$, with $\hat{F}_{k+1} = \emptyset$.
2. For each $\alpha_j \in \hat{F}_k$ with $j = i + 1, \ldots, |\hat{F}_k|$ do:
   a) Intersect the tid-list $T(\alpha_i)$ with the tid-lists $T(\alpha_j)$ to obtain the tid-list $T(\beta)$ for the itemset $\beta$ where $\beta = \alpha_i \cup \alpha_j$.
   b) If $\beta$ fulfills $\xi$ with $|T(\beta)| \geq \hat{\xi}$, then add $\beta$ to $\hat{F}_{k+1}$.
3. If $\hat{F}_{k+1}$ is not empty, then add $\hat{F}_{k+1}$ to $F$ and perform bottom-up mining recursively with $\hat{F}_{k+1}$ as parameter.

After mining is finished, $F$ holds the obtained frequent itemsets.

Contrary to **Apriori**, **Eclat** is well suited for mining large frequent itemsets because as they grow during mining, their respective tid-lists get smaller. Hence, intersecting the small tid-lists of large itemsets is much faster than intersecting the long tid-lists of small itemsets or even frequent items. For a similar reason, **Eclat** performs well on dense datasets with many frequent items and mainly long transactions, which are basically the datasets on which **Apriori** performs worse.

**Optimizations**

There are several optimization techniques that can be employed for **Eclat**. Many of them aim on reducing the time spend during bottom-up mining, since it often amounts to a large fraction of **Eclat**’s overall runtime. Other optimizations reduce **Eclat**’s memory footprint to avoid expensive out-of-core processing for large datasets.

The mining time itself is dominated by the tid-set intersections. Hence speeding up these intersections is very beneficial. The *short-circuited intersection* [Zaki and Gouda, 2003] stops intersecting two tid-lists whenever the result tid-list cannot fulfill the given $\xi$ value anymore. For example, if the tid-lists for two itemsets $\alpha$ and $\beta$ both consist of 100 tids and $\xi = 80$, then the intersection can be stopped after 21 mismatches. This optimization works typically well when the tid-lists being intersected have a size that is close to the $\xi$ value.

An other popular technique is to represent tid-sets using tid-bitmaps; each frequent itemset (or item) has an assigned tid-bitmap where the $i$-th bit represents whether the itemset occurs in the $i$-th transaction or not. Many algorithms [Shenoy et al., 2000, Burdick et al., 2001]...
2. Frequent-itemset mining

2001, Borgelt, 2003] employ this technique because it allows fast intersections based on simple “AND” operations and tid-bitmaps often require less space compared to tid-lists. However, Burdick et al. [2001] and Borgelt [2003] mention that tid-lists composed of 32-bit integers are more space efficient if the support of an item or itemset is below than 1/32 (3%). Unfortunately, intersecting a tid-bitmap and a tid-list is expensive so that only one of both representations is used at a time.

Wei et al. [2007] speed up the counting of the number of “ones” (i.e., population count) in a tid-bitmap. Instead of using a lookup-based population count implementation, they use a “calculation”-based implementation\(^2\) and parallelize it using SIMD instructions.

Borgelt employs range list intersections within its Eclat implementation [Borgelt, 2003]. For that purpose, each tid-set is represented using a list of ranges (or intervals); a single range summarizes a continuous set of tids in which the tid-set’s respective item occurs. The tid-set \(\{1, 2, 5, 6, 7\}\), for example, would be represented as the range list \(\{[1, 2], [5, 7]\}\). To minimize the number of intervals, the filtered transactions, whose items are ordered with respect to their global frequency, are lexicographically sorted before they are converted. The intersection of two range lists itself resembles to the simple list intersection. Although the range intersection is usually superior over the tid-list intersection, it has large initial overhead for sorting the transactions, which might pay off only for low \(\xi\) values.

Viper [Shenoy et al., 2000] and Mafia [Burdick et al., 2001] employ bitmap compression techniques to reduce the space requirements of the tid-bitmap representation. Viper compresses consecutive runs of “ones” and “zeros” using run-length encoding [Bassiouni, 1985] combined with colomb codes [Golomb, 1966]. Both, compression and decompression, are expensive because many bit-arithmetic instructions are required. In experiments, this compression scheme reduces the tid-list representation of the transaction database to 1/3 of its size. This allows to process large datasets in-memory. Mafia aims solely on maximal-itemset mining, however, it uses a “compression” scheme for projected bitmaps, which can also be applied for frequent-itemset mining. The main idea is to remove tids in projected bitmaps that are not relevant for finding larger itemsets. For example, all transactions that do not contain the item \(a\) cannot contain any of its supersets within the \([a]\) equivalence class. Hence, after building the sets \(ab, ac,\) and \(ad\) from \([a]\), all bits in their bitmaps corresponding to “zero” bits in the bitmask of \(a\) can be removed because they provide no information for obtaining larger itemsets; these bits were always “zero” in all itemsets that have \(a\) in their prefix. Clearly, removing the bits in the bitmask of all direct supersets is expensive. Thus, a rebuilding threshold is used to decide when such a bitmap rebuilding is reasonable.

The dEclat algorithm [Zaki and Gouda, 2003] uses diffsets for storing intermediate results during the mining process. A diffset contains only the difference between the itemset’s tid-list and the tid-list of its subset. For example, if the tid-list of \(a\) is \(T(a) = \{1, 3, 7, 9\}\) and the tid-list of \(ab\) is \(T(ab) = \{1, 3, 9\}\), then the diffset would be only \(T(ab) = \{7\}\). Diffsets are sufficient for generating larger itemsets and counting their support. They are often much smaller than corresponding tid-lists and thus require less space and are faster “intersected”.

DCI [Orlando et al., 2002a] combines the advantages of Apriori and Eclat. In the early iterations—the counting-based phase—it uses an Apriori-like candidate generation and support counting. It switches to Eclat—the intersection-based phase—as soon as a bitmap representation (i.e., the vertical layout) of the transaction database fits into main memory. This works very well, since Apriori requires often less memory in the early phases and Eclat

\(^2\)An example implementation is available in AMD’s software optimization guide [Manual, 2005] on page 179.
Figure 2.5.: An FP-tree

is faster for finding long patterns in the later phases. kDCI [Orlando et al., 2003a] provides
heuristic to distinguish between sparse and dense datasets; depending on this density several
optimizations are employed in the intersection-based phase.

Finally, Schmidt-Thieme [2004] proposed new and discussed existing optimizations for
Eclat. For example, perfect extension pruning reduces the number of candidates during
mining. Whenever two itemsets \( \alpha \) and \( \beta \) occur in the same transactions with \( T(\alpha) = T(\beta) \),
then both itemsets are “extended” with exactly the same items within the deeper recursion
steps. Hence, it is sufficient to use only \( \alpha \) for further mining and calculate the extensions of \( \beta \)
based on the extensions of \( \alpha \). This optimization may significantly reduce the mining effort—
especially if some eligible itemsets are found in the early recursion steps. Schmidt-Thieme
further evaluated the impact of the Eclat optimizations on various datasets. Similar to the
pros and cons of the mining algorithms, also most of the optimizations are beneficial on some
datasets and on others not. The diffset optimization, for example, is superior over bitmaps
on sparse datasets whereas on dense datasets it is the other way around.

2.2.3. FP-growth

Han et al. [2000] proposed FP-GROWTH, which is usually considered one of the fastest available
frequent-itemset mining algorithms. FP-GROWTH is a divide-and-conquer algorithm that
consists of two phases: build and mine. The initial build phase transforms the transaction
database into a representation that is well-suited for mining: the FP-tree (for frequent-pattern
tree). It represents all filtered transactions of the complete database and is usually much
smaller than the database itself. The second phase extracts the frequent itemsets from this
tree without further access to the transaction database. Thus, the algorithm is very I/O
efficient if the FP-tree fits into main memory while the original database does not.

The FP-tree is a compact data representation of the transaction database. Figure 2.5 shows
an FP-tree built from a FIMI dataset [FIMI, 2004] with minimum support of \( \xi = 80,000 \). The
FP-tree is constructed in two passes over the database. The first pass counts the support
of each individual item; only frequent items and their support (count values) are retained. In
the example, items \( f \), \( a \), \( b \), and \( d \) are frequent and have counts \( \phi_f = 308,656 \), \( \phi_a = 116,102 \),
\( \phi_b = 115,691 \), and \( \phi_d = 89,331 \), respectively. The database is then scanned a second time
2. Frequent-itemset mining

and the FP-tree is built. The items in each transaction are sorted in descending order of their support; the FP-tree is simply a prefix tree on the sorted transactions enriched with some additional information. Apart from the item itself, this information includes the number of times each prefix has been encountered (count) and links for efficient navigation (node-links). For example, leaf node \((d | 13490)\) represents prefix \(fad\); this prefix occurred 13,490 times in the database. The node-links are used to connect all nodes that have the same item, i.e., they form a linked list (dotted lines in Figure 2.5). A header table holds all frequent items—sorted with regard to their frequency—and provides the starting points for the linked list of each frequent item.

Based on the initial FP-tree, FP-growth recursively builds smaller “conditional” FP-trees that are eventually used to obtain the actual frequent itemsets. The conditional tree for an itemset \(\alpha\) represents all transactions that contain this itemset. It is denoted as \(\alpha\)-conditional tree and \(\alpha\) is the prefix of this tree. Mining starts recursively from the initial FP-tree, which can be seen as \(\emptyset\)-conditional tree since it contains all frequent items without any restriction. Recursion proceeds in a depth-first manner for all frequent items in all conditional trees; in each recursion step, the prefix of a new generated conditional tree is expanded by one (pattern-growth). In our example from Figure 2.5, the \(d\)-conditional is build from the \(\emptyset\)-conditional tree, then the \(db\)-conditional tree is build from the \(d\)-conditional tree, and so on.

Building the \(\beta\)-conditional tree for a frequent item \(\gamma\) from the \(\alpha\)-conditional with \(\beta = \gamma \cup \alpha\) is done in two steps. In the first step, each path from all nodes containing \(\gamma\), which are connected via the \(\gamma\) node-links, is traversed to the root node of the tree. All these paths represent itemsets that co-occur with \(\beta\) in the \(\alpha\)-conditional tree; they form the conditional pattern base of the \(\beta\)-conditional tree and can be seen as compact representation of all transactions that contain \(\beta\). The frequency of each itemset is given by the count value of the respective \(\gamma\) node. In our example, the \(b\)-conditional pattern base consist of \(af (39,770), f (50,369),\) and \(a (369)\).

In the second step, all infrequent items are removed from the \(\beta\)-conditional pattern base and the \(\beta\)-conditional tree is build from it. All remaining frequent items are returned with \(\beta\) as frequent itemsets. In our example, only \(f (\hat{\xi} = 90,139)\) is frequent (\(\hat{\xi} = 80,000\)) in the \(b\)-conditional pattern base so that the \(b\)-conditional tree contains only one node with item set to \(f\) and count set to 90,139. Hence, \(bf (90,139)\) is returned as frequent itemset.

The full algorithm then works as follows:

**Build phase** The database is scanned twice. The first scan reveals the frequent items of the database that form the header table of the subsequent generated FP-tree.

The second scan is used to transform the database into an FP-tree. For that purpose, infrequent items are removed from each transaction; the remaining (frequent) items are sorted with regard to their frequency and inserted into the FP-tree.

**Mine phase** The FP-tree is recursively mined by calling the \(FP\text{-growth}(\ )\) function. This function takes two parameter, an FP-tree \(tree\) and a prefix \(\alpha\).

Whenever it is called, \(FP\text{-growth}(\ )\) generates the pattern \(\beta = \gamma \cup \alpha\) for each frequent item \(\gamma\) in the header table of \(tree\). Then, the \(\beta\)-conditional tree is build from the \(\alpha\)-conditional tree. If the new generated tree contains at least one frequent item, \(FP\text{-growth}(\ )\) is recursively called and the \(\beta\)-conditional tree and \(\beta\) are passed as parameters. The pattern \(\beta\) is returned as frequent itemset with the frequency of the item \(\gamma\).

If \(FP\text{-growth}(\ )\) is called and \(tree\) contains only a single path \(\lambda\), then an optimization can be employed. Instead of calling \(FP\text{-growth}(\ )\) for each item in \(\lambda\), all combinations of \(\lambda\)
2.2. Sequential Algorithms

(the power set of $\lambda$) are concatenated with $\alpha$ and returned as frequent items. Thereby, always the least frequent item in a combination of the items in $\lambda$ denotes the frequency of the returned frequent itemset.

As already mentioned, FP-GROWTH works well if the FP-tree representation is much smaller than the original dataset. Thus, as also stated by Tan et al. [2005], the runtime of FP-GROWTH depends on the compaction factor of the dataset. Very bushy trees lead to high memory requirements (compared to the requirements of the original datasets) and reduce the algorithm’s performance. In the worst case, if the $\xi$ values are very low (e.g., similar to rare-itemset mining), the FP-tree may even exceed the size of the original database. Furthermore, the FP-tree is an irregular data structure and requires many costly pointer de-references during building and mining the FP-trees. Thus, if only small itemsets from many transactions are expected, APRIORI is usually superior. ECLAT might perform better, if the datasets consist of only few transactions and the average transaction cardinality is rather high.

Optimizations and variants

There are several optimizations for FP-GROWTH and also variants that are based on or are very similar to FP-GROWTH. Besides the parallel versions of FP-GROWTH (see Section 2.3), all optimizations focus on either reducing the memory requirements to allow mining larger datasets in memory or adapting FP-GROWTH to the characteristics of modern hardware.

To reduce the memory requirements of FP-GROWTH, FP-GROWTH-Tiny [Özkural and Aykanat, 2004] builds conditional FP-trees without creating a conditional pattern base. Each path read from the $\alpha$-conditional tree is immediately inserted into the $\beta$-conditional tree. This avoids the need for storing the conditional pattern base. Both, CT-ITL [Sucihyo and Gopalan, 2003] and CT-PRO [Sucihyo and Gopalan, 2004], use a “compressed” FP-tree structure that avoids repeated storage of similar subtrees. Additional arrays are used to store the differences (i.e., count values) between the shared subtrees. Pietracaprina and Zandolin [2003] use a Patricia trie [Knuth, 1973] to represent the transaction database. All nodes within a single path that have the same count value are coalesced into a “single” node. Thus, pointer between the nodes can be omitted; this leads to a space efficient representation of the FP-tree and works very well on sparse datasets. Finally, NONORDFP [Rácz, 2004] transforms the FP-tree before the mine phase into two separate arrays consisting of only count and parent fields of all nodes. Conditional trees are also represented by such arrays; the $\beta$-conditional arrays are created by iterating multiple times over the complete $\alpha$-conditional arrays. NONORDFP has much lower memory requirements for representing an FP-tree, however, in its mining phase it has high memory requirements because for each projection the arrays representing an FP-tree must be duplicated.

To reduce the costly dereferences while mining the FP-tree, cache-conscious representations of the FP-tree were proposed. The main idea is to unroll all paths from each leaf node to the root so that the nodes of each path are stored continuously in memory. This procedure improves cache locality but increases the size of the FP-tree. Examples of such approaches include the CC-tree [Ghoting et al., 2005] and the FP-array [Li and Liu, 2007]. Although these optimization significantly speed up the mining process, in case of dense datasets or low $\xi$ values both may require much more memory compared to regular FP-GROWTH.

Furthermore, there are algorithms that resemble FP-GROWTH like the TREE PROJECTION algorithm [Agarwal et al., 2001]. As for FP-GROWTH, it does not require an explicit candidate generation. TREE PROJECTION uses a lexicographical tree to represent so far obtained
frequent-itemsets. All paths hold sub-databases that are recursively projected to obtain larger frequent-itemsets. For example, the active sub-database \{A, B\} summarizes all transactions that contain the items A and B. During mining, two-dimensional matrices are created from the respective sub-databases using the direct-count technique; each cell of the matrix contains the frequency of a combination of two frequent items in this sub-database. The main disadvantages—as mentioned by Han et al. [2000]—are a high memory consumption of the matrices for a large number of frequent items and also high costs for generating the frequent 2-itemsets in each sub-database.

H-MINE [Pei et al., 2001] is a further algorithm that is based on the pattern-growth method. Similar to FP-GROWTH, H-MINE scans the transaction database once to obtain the frequent items and removes infrequent items from the transactions. However, instead building an FP-tree, H-MINE sorts the transactions and transforms them into an array-based data structure; each transaction is represented by an array of items and hyperlinks. The hyperlinks—each item per transaction has an assigned hyperlink—are used to connect transactions that have an equal prefix. For example, the connected \{A\}-transactions are traversed to obtain the \{A, B\}-transactions. The disadvantage of H-MINE compared to FP-GROWTH is its excessive memory consumption for dense datasets. Also for low support values, the “transformed” database is often larger than the original dataset. However, for sparse datasets it may be faster than FP-GROWTH, so Liu et al. [2002] proposed a combination between H-MINE and FP-GROWTH.

2.3. Parallel algorithms

So far, we reviewed sequential frequent-itemset mining algorithms as well as respective optimizations. In this section, we will give an overview about existing parallel versions of these algorithms. For that, we classify the algorithms according the platforms for which they are intended. There are algorithms for (1) shared-memory multiprocessing systems, (2) distributed systems, and (3) graphic processors. We will now review each class in more detail.

2.3.1. Shared-memory systems

Symmetric multiprocessing (SMP) or non-uniform memory access (NUMA) multiprocessing systems consist of multiple processors that provide a shared main memory with a single address space. Parallel algorithms executed on such systems scale well and since the processors are tightly coupled, communication and synchronization is quite inexpensive compared to distributed systems. The main objective for algorithms tailor-made for SMP or NUMA systems are an effective work partitioning, achieving a high memory locality, and the avoidance of false sharing. The latter occurs whenever two or more threads write into non-overlapping memory regions that, however, rely in a common cache line. Further details regarding such systems and their programming can be found in Section 2.5.3 starting on page 42.

Apriori

The first parallel algorithms for SMP systems were based on APRIORI. All of these algorithms perform candidate generation and support counting in parallel. For that purpose, the transaction database is partitioned into equal-sized chunks, which are divided among the threads. In general, the algorithms differ mainly in the data structures used for storing the candidates.
2.3. Parallel algorithms

The algorithm **Common Candidate Partitioned Database** (CCPD) was proposed by Zaki et al. [1996]. It parallelizes the candidate generation based on partitioned equivalence classes; the classes are divided using bitonic partitioning [Cierniak et al., 1995] that minimizes work imbalance. The algorithm uses a single candidate hash tree that is shared among the processors; it is built in parallel using locks at the leaf nodes. During support counting, locks are used per itemset in the tree. In their experiments, they found that locking was sufficient on a system with 12 processors. Parthasarathy et al. [2001] improve the CCPD algorithm by increasing data locality and avoiding false sharing that is introduced by the pointer-based nature of the candidate hash-tree. They provide three memory allocation policies that address both challenges.

The **Adaptive Parallel Mining** (APM) algorithm [Cheung et al., 1998], which is based on the **DIC** algorithm [Brin et al., 1997b], uses a shared prefix tree (trie) for support counting. The “DIC-specific” dynamic candidate generation is performed by all threads in parallel.

Jin et al. [2005] evaluated five different synchronization techniques applied on a shared hash-tree for support counting. They found that **full replication** (the candidate tree is replicated for each thread) and **full locking** (all candidates have an assigned lock within a single candidate tree) impose a large overhead because of the high additional memory requirements and lock contention, respectively. The other three techniques form a tradeoff between full replication and full locking also considering cache effects and false sharing. However, as shown in their experiments, full replication is always superior over the locking techniques at the cost of a high memory consumption.

To summarize, CCPD and APM are parallel frequent-itemset mining algorithms that are tailor-made for SMP systems. Both are based on **Apriori** and focus solely on thread-level parallelism. Their major issue is the shared data structure used for support counting; if it is based on locking, then it will not permit scaling to a high number of threads because of lock contention and the locking overhead. Otherwise, if it is based on full replication, then it will not scale to a high number of threads because of the high memory consumption.

**Eclat**

There are no parallel versions of **Eclat** that are tailor-made only for SMP or NUMA systems. Zaki et al. [1997b,d] proposed two frequent-itemset mining algorithms that are based on **Eclat** for distributed SMP systems. They exploit intra-node level parallelism on SMP systems and inter-node level parallelism on distributed systems; we defer the discussion for the latter systems to Section 2.3.2. **PAR**Eclat and **PAR**Clique differ only in the equivalence class clustering strategy they employ. The former uses prefix-based classes while the latter uses clique-based classes. The steps of both algorithms can be summarized in three phases. The **initialization phase** is used for obtaining the frequent 2-itemsets (including the frequent items), creating the equivalence classes, and converting the dataset. Each class has an assigned weight—based on its cardinality—which is used for a greedy-based partitioning of the classes. The parallel conversion of the transaction database into the vertical layout is performed by dividing the dataset into equal-sized partitions; each thread converts a different
range of tids, i.e., the tids for the transactions that are in its assigned partition. Knowledge
about these tids, the partial count values, and global count values are exploited to obtain the
global tid-lists in which the tids are lexicographically ordered. Each thread basically writes
“its” tids starting from certain offsets in the final tid-lists. In the *asynchronous phase*—denoted
as asynchronous because there is no synchronization between the processors required—each
processor generates frequent itemsets using its assigned equivalence classes; this is done via
tid-list intersection as in the sequential algorithm. In the last phase, the *reduction phase*, all
processors are synchronized and the results are combined.

**ParDCI** [Orlando et al., 2002b] is a parallel version of the DCI algorithm. Similar to
**ParEclat** and **ParClique**, it is intended for distributed SMP systems. **ParDCI** starts—as
its sequential version—with an *Apriori*-like counting-based phase and migrates to an
*Eclat*-like intersection-based phase. Within the former phase, a single thread distributes
parts of dataset to worker threads, which perform the subset counting. The counters are
replicated to minimize synchronization between the threads; after each iteration, they are
merged to obtain the global frequencies of the candidates. In the intersection-based phase,
the filtered transactions are converted into tid-bitmaps, which can be accessed by all threads.
Distributing the candidates to the threads is performed using a work queue. Whenever, a
thread finished processing an equivalence class, it gathers the next class from this queue.

**FP-growth**

More recent research has focused on parallel versions of **FP-growth** for SMP systems. Both,
building the initial FP-tree and its subsequent mining, can be parallelized. Of both phases,
the building step is more complicated since all threads work together on a common data
structure. The mining afterwards is more or less already partitioned by the frequent items.

In general, all parallel versions of **FP-growth** share the same two phases:

**Build phase** The transaction database is partitioned into equal-sized chunks. Each thread
gathers the frequency of all items in its partition; the global frequent items are obtained
by aggregating these frequencies. Then, a “global” FP-tree is build in parallel. This
step is handled by each of the following algorithms differently.

**Mining phase** Conditional FP-trees are recursively build like in the original algorithm. Based
on a partitioning of the frequent items located in the “global” header table, each thread
recursively processes the conditional trees of its assigned frequent items.

The first parallel **FP-growth** version for SMP systems was proposed by Zaïane et al.
[2001]. They use **Multiple Local Parallel Trees** instead of a global FP-tree. Each thread
builds a local FP-tree from its assigned chunks so that locking could be completely avoided.
Clearly, this leads to a high memory consumption because the multiple FP-trees often have
a similar number of nodes compared to a “single” FP-tree, however, with only lower count
values. Furthermore, to build the conditional pattern base for a single frequent-item, the
corresponding tree paths in all local trees must be visited. This effort is necessary for the
pattern base of each frequent item; the recursive generation of further pattern bases is the same
as for the sequential algorithm. In summary, the Multiple Local Parallel Trees approach does
not scale well with an increasing number of threads because of its higher memory consumption
and the additional overhead for generating conditional pattern bases.

Chen et al. [2006] propose a **lock-free single tree** approach. The basic idea is to divide
the transactions into non-overlapping partitions based on their items they consist of. Each
parallel algorithms

partition forms an independent subtree of the global FP-tree; thus a thread can create nodes or increase their count values without locking because it processes the subtree alone. The $n$ most-frequent items are used for the static partitioning so that $2^n$ partitions are created. For example, consider $n = 2$ and the two most frequent items are $f$ and $a$; the transactions are then inserted into one of four partitions based on which of both items they contain: $f$ and $a$ $\rightarrow P_1$, $f$ but not $a$ $\rightarrow P_2$, not $f$ but $a$ $\rightarrow P_3$, and neither $f$ nor $a$ $\rightarrow P_4$. In the following, we refer to this partitioning as hot-item partitioning. For load balancing, a heuristic is applied to form groups of partitions so that each group has a similar number of transactions. Each group is then assigned to a thread. The transaction partitioning and subsequent insertion in the tree are executed together using a master/slave processing model. A master thread reads all transactions and puts them—based on their partition and thus group membership—into queues, which are assigned to the slave threads. Each slave thread reads from its queue and inserts the transaction into the FP-tree. The mining afterwards is parallelized based on a partitioning of the frequent items; dynamic scheduling is employed to balance the work to multiple threads. Li and Liu [2007] avoid the queues of the master/slave processing model because these queues must be synchronized. They instead separate the partitioning process from the actual tree building process, i.e., a scan is used to partition the transactions and afterwards these are processed in parallel. Hence, the full dataset is almost replicated before the tree is built.

Ghoting et al. [2005] use symmetric multi-threading (SMT) to speed up the mining phase. Since SMT threads compete for the same execution units of a core (see Section 2.5.3 for further details) and the mining process is usually limited by occurring memory stalls, they propose to reuse fetched data in the cache by the threads. This works well whenever multiple threads traverse backwards the same path in a conditional or the initial FP-tree. In their experiments they achieve a speedup up to 1.45x using two SMT threads.

Summary

The parallel versions of all three algorithms exploit different kinds of parallelism and usually scale well on SMP systems with a small number of processors. The main focus is on the avoidance of locks in the data structures and of false sharing. Load balancing was not considered as an issue since the algorithms use only a rather small number of threads and often employ dynamic load balancing techniques. Furthermore, none of the algorithms is adapted to the specific characteristics of NUMA systems that are gaining an increasing popularity.

2.3.2. Distributed systems

One of the main contributions in this work are highly scalable parallel versions of Apriori, ECLAT, and FP-Growth that are tailor-made for NUMA multiprocessing systems. These systems share some characteristics with distributed systems, i.e., a large number of available threads and higher communication costs for memory transfers between processors compared to multi-core processors or SMP systems. For this reason, we will now survey frequent-itemset mining algorithms for distributed multiprocessing systems.

Distributed systems—also denoted as shared-nothing architectures—consist of multiple machines that have their own processors, main memory, and hard disks. Communication, i.e., memory transfers, is done via a network connecting all machines. Depending on the network

---

4The optimal combination of the partitions is an NP-Complete problem.
2. Frequent-itemset mining

hardware and topology, communication is one of the major bottlenecks of such systems. For example, the average latency and bandwidth of a single network access (Gigabit Ethernet) is about 1000x higher and roughly 300x lower, respectively, compared to main memory access of a current processor. For this reason, communication minimization is one of the major optimization objectives for designing efficient distributed algorithms. Furthermore, since distributed systems can consist of a large number of machines, a further optimization objective is an efficient load balancing to avoid underutilized machines. In the following, we survey distributed versions of Apriori, Eclat, and FP-growth; we focus on their communication pattern and load balancing techniques, which are partially reused in this thesis.

Apriori

The first frequent-itemset mining algorithms for distributed systems were based on Apriori. All these algorithms have in common that either or both candidate generation or support counting is done in parallel. For the candidate generation, the algorithms employ either “replicated” or distributed candidate generation. Replicated candidate generation is based on synchronized support values where each machine generates the same set candidates; this reduces communication effort because only the support values after each iteration must be exchanged. Distributed candidate generation is based on the local or global support values of the machines; each machine generates a set of “local” candidates that must be synchronized to obtain the global candidates. This may introduce a large overhead if only few of the local candidates are also global candidates. For parallel support counting, the dataset is usually partitioned into equal-sized chunks.

One of the first distributed frequent-itemset mining algorithms—denoted as Parallel Data Mining (PDM)—was proposed by Park et al. [1995b]. It is based on DHP and performs candidate generation as well as support counting in parallel. More precisely, PDM uses replicated candidate generation based on synchronized support values; the synchronization of the values is done via an all-to-all broadcast. To reduce the communication overhead, PDM uses a method clue and poll where each machine exchanges only support values of the items that have a high change of reaching the global $\xi$ value. However, this requires a second round of communication to avoid false negatives, i.e., for items that are on some machines identified as infrequent but on other machines identified as frequent and therefore might be globally frequent. Thus—as also mentioned by Zaki [1999]—the algorithm has considerably high communication costs.

Mueller [1995] proposed for its sequential algorithms SEAR and SPEAR distributed versions PEAR and PPAR, respectively. PEAR performs support counting in parallel and uses replicated candidate generation; each machine builds—in each iteration—a candidate prefix tree based on the synchronized global support values. The synchronization of the support values is the only communication required by the algorithm. It is achieved via an MPI reduction operation [MPI Manual, 2009], which requires $\log n$ messages for synchronizing $n$ machines. Nevertheless, the algorithm has high communication costs since the local support values of all candidates are exchanged in each iteration. PPAR requires only two rounds of communication. Before the first round, each machine obtains—similar to SPEAR—all candidates for its assigned part of the “global” transaction database. These local candidates are then reduced into global candidates using communication; this is done in a pipelined fashion to reduce the number of messages. Each machine has a successor that merges the candidates of both machines. After all candidates are merged, the last machine broadcasts the global candidates to
all other machines. Furthermore, the prefix tree that represents the candidates is linearized so that each node is represented by a single integer value. Based on the global candidates, all machines count the support of the candidates in their partition in parallel. The second round of communication is used to obtain the global support of all candidates. Similar as SPEAR, PPAR also suffers of a very large number of candidates that are eventually not frequent.

Agrawal and Shafer [1996] proposed three algorithms: COUNT DISTRIBUTION (CD), DATA DISTRIBUTION (DD), and CANDIDATE DISTRIBUTION (CAD). In CD, support counting is done in parallel on distributed parts of the transaction database. It uses replicated candidate generation to avoid communication. The support values are synchronized using a recursive merge tree in \( \log n \) steps—similar to the MPI reduction operation. DD divides the candidates into equal-sized partitions. Each machine counts only the support for the candidates in its partition and hence must scan the complete dataset in each iteration. This allows to count a larger set of candidates as it would be possible on a single machine. However, the algorithm has high communication costs since the complete dataset must be exchanged in each iteration between all machines. The exchange itself is done via an all-to-all broadcast. CAD uses in the first \( l \) iterations either count or data distribution. After the \( l \)-th iteration, the candidates are distributed based on their equivalence classes. Each machine then processes—without synchronization—its assigned candidates on a partially replicated transaction database; it contains all transactions that are necessary for mining the equivalence class. Clearly, the value \( l \) and the partitioning of the equivalence classes have a large impact on load balancing.

The algorithms INTELLIGENT DATA DISTRIBUTION (IDD) and HYBRID DISTRIBUTION (HD) were proposed by Han et al. [1997]. IDD is based on DD but exchanges the transaction database in a “ring-based” manner. Each machine receives and sends parts of the transaction database from a predecessor machine and to a successor machine, respectively. This greatly reduces the communication overhead of DD. Furthermore, the candidates are partitioned based on their equivalence classes to avoid redundant work during parallel support counting. As soon as the candidates fit in main memory, CD is used for the remaining iterations. HD forms a combination of IDD and CD. The transaction database is partitioned into groups where each group itself is processed using IDD and CD “combines” the results of the groups.

There are several further algorithms that are based on CD and improve it using pruning and communication reduction techniques. For example, FAST DISTRIBUTED MINING [Cheung et al., 1996] reduces communication effort by generating a smaller set of candidates that must be exchanged. For that purpose, it employs local and global pruning techniques and has an improved communication pattern; it requires only \( n \) messages to synchronize \( n \) machines. The DISTRIBUTED DECISION MINER [Schuster and Wolff, 2001] extends this work and provides theoretical bounds. FAST PARALLEL MINING [Cheung and Xiao, 1998] uses distributed pruning and global pruning to reduce the number of exchanged candidates. Both techniques are based on the pruning techniques of FAST DISTRIBUTED MINING but instead of two they require only one round of communication.

To summarize, all distributed algorithms that are based on APRIORI suffer despite applied optimizations from high communication overhead or redundant work. For this reason, distributed versions of ECLAT or FP-GROWTH are usually considered as superior over them.

**Eclat**

Distributed versions of ECLAT work similar to their sequential or SMP counterparts. Work partitioning is achieved by distributing the equivalence classes to different machines. This
2. Frequent-itemset mining

usually requires replication of the tid-lists so that each machine can independently process its assigned equivalence classes. Basically, the following algorithms differ only in (1) which tid-lists are replicated on the machines and (2) the load-balancing strategy they rely on.

Par-Eclat and Par-Clique [Zaki et al., 1997b,d] differ only in the employed equivalence clustering strategy and can be used on SMP systems (cf. Section 2.3.1) as well as on distributed systems. Both versions selectively replicate the tid-lists before the asynchronous mining step, i.e., each machine has all required tid-lists to mine its assigned equivalence classes. For example, consider a machine with the assigned equivalence class $S_a = \{a\}$ and the candidates $C_3 = \{abc, abd, acd\}$. Before mining, this machine thus requires the tid-lists $F_2 = \{ab, ac, ad, bc, bd, cd\}$. Depending on the assigned classes, some machines may require the tid-lists of all frequent items; this leads to high memory requirements on these machines. The exchange of the tid-list itself is achieved via an all-to-all communication. Each machine sends parts or complete tid-lists to all machines that require them. Load balancing is done as before in the SMP version; the equivalence classes have assigned weight values and these are used in a greedy algorithm that equally assigns the classes to the available machines.

As mentioned earlier, ParDCI [Orlando et al., 2003b] runs also on distributed systems. The counting-based phase is basically the same as for CD; each machine has a replicated set of the candidates and processes parts of the distributed dataset. In the intersection-based phase, each machine converts its part of the dataset into tid-bitmaps, which are then distributed to all machines. Hence, all tid-bitmaps are stored on each machine. This limits scalability for mining larger datasets on an increasing number of machines (weak scalability). ParDCI uses further a static scheduling of the equivalence classes; these are distributed in a round-robin manner to the available machines. This may or may not lead to load imbalance depending on dataset and the number of machines.

**FP-growth**

Distributed versions of FP-GROWTH are often considered to be superior over other distributed frequent-itemset mining algorithms because they require less communication between the machines, i.e., only a fixed number of synchronization points are required. Furthermore, also smaller amounts of data are transferred because of the “compressed” nature of the FP-tree. Basically, all distributed FP-GROWTH versions share the same four steps in which communication is required:

**Dataset distribution** The dataset is divided into $n$ non-overlapping partitions; each of the $n$ machines transfers one of these partitions into its local memory, i.e., main memory or disk. In some cases, the dataset is already partitioned and distributed.

**Obtaining frequent items** Similar to the sequential algorithm, each machine scans its local partition to obtain “local” frequent items. After synchronization with all other machines, the “global” frequent items are identified and distributed to all of them.

**Distributing conditional pattern bases** Each machine builds a local FP-tree based on the transactions in its data partition. After the local trees are build on all machines, they are used to obtain the “global” conditional pattern bases. For that, synchronization with all other machines is required. The pattern bases are then partitioned—for the following mining process—and distributed to the machines.

5Each machine applies pruning techniques so that the actual bit-vectors hold by the machines are different.
Asynchronously mining Each machine recursively builds and mines only the conditional trees based on its assigned frequent items. Except for dynamic load balancing in some algorithms, this mining can be completely done without communication. Finally, the results are exchanged and returned.

The distributed FP-GROWTH versions differ only in how they build the global pattern bases, the employed load balancing techniques, and their communication pattern. Furthermore, some of them use certain optimization techniques to reduce the communication overhead.

Pramudiono and Kitsuregawa [2003] proposed a distributed FP-GROWTH version where each machine generates local frequent-pattern bases for all frequent items from their local FP-tree. These pattern bases are then transferred to the machines that are assigned for their processing. This leads to a high communication overhead since each machine sends—up to the number of frequent items—local pattern bases to other machines. The algorithm employs static as well as dynamic load balancing at the asynchronous mining step. Initially, the frequent-pattern bases are randomly assigned to available machines. To avoid load imbalance, dynamic load balancing is employed based on the path depth of the frequent items; it denotes the number of frequent items that occur in the conditional tree of this item. Whenever the path depth of a pattern base is smaller than a given threshold, the machine recursively mines this pattern base itself. Otherwise, its subsequent pattern bases—all conditional trees based on this pattern base—may be distributed to idle machines. Clearly, this introduces additional communication overhead.

The PFP-tree algorithm [Javed and Khokhar, 2004] reduces communication by maintaining a “partial global” FP-tree on all machines. Each machine has only the global versions for the tree-paths that are required for mining its assigned frequent items. To achieve that, each machine identifies—using a depth-first search—all paths that are required by other machines in its local FP-tree. These paths are then packed into an array; the size of the array is reduced by storing only the item, parent, and support field from a tree-node per array element. These fields are sufficient to reconstruct the paths. The communication itself is done via a recursive merge tree. Thus, each machine has its “partial global” FP-tree after log n rounds.

Cong et al. [2005] aim on improving load balancing of distributed FP-GROWTH algorithms; they provide an efficient static load balancing that is based on sampling (cf. Section 2.4.2 for other sampling techniques). The key idea is to mine a sample of the complete transaction database and record the execution times. These times are then used to equally partition the mining tasks to the participating machines.

Distributed FP-GROWTH (DFP) [Buehrer et al., 2007] builds a global FP-tree from all local FP-trees before the asynchronous mining phase. For that purpose, the machines communicate the local trees in a ring-based manner. Thus after n rounds each machine has the global FP-tree. To reduce the communication effort for exchanging the trees, DFP transforms—similar to the PFP-tree algorithm—all tree paths into an array-based representation. Here, however, each tree-node is represented by only two integer values, the item and count field, within such an array. Load balancing is achieved using the sampling techniques of Cong et al. [2005].

PFP [Li et al., 2008] is a distributed FP-GROWTH implementation based on MapReduce. After obtaining the global frequent items, mapper machines generate “group-dependent transactions” for each transaction of their database partition, which can be aggregated into independent conditional pattern bases. For example, the group-dependent transactions for the transaction $t = \{f, c, a, b, d\}$ are $t_d = \{f, c, a, b\}$, $t_b = \{f, c, a\}$, $t_a = \{f, c\}$, and $t_c = \{f\}$. All
transactions that are in the same group form a conditional tree, which can be asynchronously mined. The algorithm, however, has thus high communication costs. For a transaction with \( n \) items, \( n - 1 \) group dependent transactions with an average cardinality of \( n/2 \) items per transaction are created. These must be transferred to the machines (i.e., reducer machines) that are assigned to mine them.

**Summary**

The main optimization objectives for distributed frequent-itemset mining algorithms are load balancing and communication minimization. Most distributed algorithms employ static load balancing to avoid communication in the mining phase. The partitioned mine tasks either get an estimated weight value or their relative mining time is measure based on a mined sample. Dynamic load balancing redistributes tasks, however, it requires communication for task assignment and potential transfer of intermediate results (candidates, tid-lists, and conditional pattern bases).

For communication reduction usually recursive merge trees or ring-based communication pattern are superior over all-to-all communication. Furthermore, several pruning techniques or implicit compression techniques are employed to reduce the size of the data being transferred. In general, most communication can be avoided in the mine-phase if the machines hold all data that is required for an asynchronous mining. For this reason, distributed algorithms would greatly benefit from our proposed compression techniques in Chapter 4 and 7. More in detail, static compression techniques (Chapter 4) reduce the costs for transferring transactions, candidates, and “array-converted” FP-trees while our FP-tree compression technique (Chapter 7) allows to hold larger replicated trees within the machines main memory.

### 2.3.3. Graphic processors

Recently, various authors proposed parallel frequent-itemset mining algorithms tailor-made for graphics processing units (GPUs). These processors can be seen as massively multi-threaded many-core processors, which provide higher memory bandwidths and computational “horsepower” compared to traditional CPUs. For example, NVIDIA’s Fermi architecture [Whitepaper, 2009] has 16 streaming multiprocessors—each with 32 CUDA cores—and achieves a bandwidth of 177GB/s and up to 665 Gigaflops within a Tesla M2090 board [TESLA M-Class Whitepaper, 2011]. However, utilizing this potential is difficult because the executed task must be equally partitioned to a large number of threads. Furthermore, these threads are partitioned into groups (called warps) in which all threads execute the same code—similar to SIMD. For this reason, GPUs are usually well suited for number crunching tasks with lots of inherent parallelism like matrix multiplication [Larsen and McAllister, 2001] or sorting [Govindaraju et al., 2006]; processing irregular data structures or complex code with many branches as usual for most efficient sequential frequent-itemset mining algorithms is challenging. One way to cope with that is to use the CPU and GPU in combination, i.e., the CPU executes the algorithm’s sequential “complex” parts while the GPU is used for its highly parallel parts.

Fang et al. [2009] proposed two GPU implementations of ECLAT—although, they denote it as APRIORI. In both implementations, the transaction database is represented using a two-dimensional bit-matrix; each row corresponds to a single frequent item and for all transactions in which this item occurs the respective bit is set to “one”. Support counting is achieved via intersection of these rows; a lookup-based population count implementation is used for
2.4. Out-of-core and Sampling-based Algorithms

counting the number of “ones” in a row. The pure bitmap-based implementation works solely on the GPU. Candidates are represented using bitmaps so that candidate generation and pruning are performed using bitwise “or” operations and binary search in this bitmaps, respectively. For this reason, almost no data transfers between a host CPU and the GPU are necessary. The trie-based implementation requires such data transfers. It represents candidates using a trie and—since a trie is an irregular data structure—it is maintained on a CPU. Thus, candidate generation and pruning are done on CPU-side while support counting is afterwards done on GPU-side. As already mentioned, the matrix representation has usually high memory requirements and does not work well for sparse datasets. For this reason, both algorithms do not scale well for an increasing number of frequent items and transactions. Furthermore, as mentioned by the authors itself, they fail to compete with a fast sequential implementation of FP-growth on their tested datasets.

A further GPU version of Eclat was proposed by Amossen and Pagh [2011]. Also this version is based on fast intersection of transactions represented in the vertical layout. For this representation, they propose a new data structure called batmap; its main idea—borrowed from distributed data structures—is to redundantly store the elements of sets to allow more parallel operation compared to traditional data structures. Storing itself is achieved via hashing; the elements of a set are mapped to multiple locations of an array using multiple random hash functions. Furthermore, compression is employed to reduce the size of a batmap element to 1 byte.

Teodoro et al. [2010] proposed a GPU version of the tree projection algorithm [Agarwal et al., 2001]. Their algorithm inserts in parallel transactions into the lexicographic tree, i.e., each thread generates 2-itemsets and increases the corresponding count values in the matrices of the active nodes. Synchronization of this concurrent support counting is achieved using atomic add instructions. Further, the transaction database is represented using a simple array-based data structure on the GPU. Also in this algorithm, the support counting is done on GPU-side while the tree-projection (node management) is done on CPU-side.

To summarize, frequent-itemset mining algorithms for GPUs are adapted to the specific properties of this hardware—i.e., their massive thread-level parallelism and high bandwidth. However, all of these algorithms share two issues. First, transfers from and to the host-memory—and thus from and to the CPU—are expensive in terms of latency and throughput. This implies that the algorithms must cope with the small available global memory of the GPU, which is currently restricted to at most 6GB. For comparison, current multiprocessor systems allow up to 2TB of main memory. The second issue is that the most efficient algorithms are based on irregular data structures. Thus, GPUs usually are limited to support counting on simple data structures or to speed up inefficient sequential algorithms. Therefore, the GPU-based parallel implementations are—in most cases—slower than the best-performing sequential implementations of elaborate algorithms running on CPUs.

2.4. Out-of-core and Sampling-based Algorithms

In the following, we will discuss algorithms that can be used for coping with out-of-core datasets, i.e., datasets that do not fit into main memory. Firstly, there are out-of-core algorithms that store intermediate results of the mining process on disk and use several techniques to increase the temporal and spatial locality of their data structures to reduce costly trans-
fners to the disks. Secondly, sampling-based algorithms are used for speeding up mining by processing only a small sample of the transaction database.

### 2.4.1. Out-of-core algorithms

Most algorithms presented so far have only reasonable execution times if their data structures (e.g., candidates, tid-lists, FP-trees) fit into main memory. They cannot cope with out-of-core datasets that are both, “producing” intermediate results and itself, larger than the available main memory. To mine such datasets, disk access during mining is required, which has a much higher latency compared to main memory. For example, the latency of current available server disks ($\approx 10$ms per access) is about one million times higher than the latency of main memory ($\approx 10$ns per access); even the access times of the fastest solid-state PCIe-based disks (e.g., ioDrive Duo card [Whitepaper, 2010] with $30\mu$s access time) are more than thousand times higher compared to main memory. Furthermore, the main memory bandwidth of a single CPU is usually around 30GB/s while even the fastest PCIe-based disks achieve a read bandwidth that is far below 2GB/s. For this reason, all of the following algorithms try to improve spatial and temporal locality to avoid accessing disks as much as possible.

Early approaches utilize the divide-and-conquer paradigm like the Apriori-based SPEAR algorithm [Mueller, 1995] and the Eclat-based Partition algorithm [Savasere et al., 1995]. Both algorithms divide the transaction database into non-overlapping partitions that can be processed in main memory. For each partition, the local frequent-itemsets are obtained using Apriori and Eclat for SPEAR and Partition, respectively. These itemsets are then combined into global candidates and a full scan of the complete database is used to obtain their actual support values. However, as found by Buehrer et al. [2006], the number of candidates is often very large—most of them are false positives, i.e., candidates that are not frequent—making the subsequent support counting infeasible.

Another divide-and-conquer approach is Disk-mine [Grahne and Zhu, 2004], which is based on FP-growth. In general, the algorithm differs from FP-growth only in how the conditional pattern bases are generated. Whenever the FP-tree, of which a conditional pattern base is generated, does not fit into main memory, the pattern base is generated using a full scan of the transaction database. For example, if the initial FP-tree and the $d$-conditional tree are both larger than the available main memory, then—assuming the items $a$, $b$, $c$, and $d$ are frequent—the pattern bases for $b$, $c$, $dc$, and $db$ are obtained each with a full database scan. Thus often much more than the two scans of FP-growth are required for mining out-of-core datasets. Furthermore, an FP-tree has to be build until it is finished or it hits the main memory border, because there is no way to estimate its size in advance. For this reason, the algorithm does a lot of unnecessary work.

Medic [Goethals, 2004] is based on ECLAT but requires less memory during mining. The main idea is to sort the transaction database in lexicographic order using external sorting [Knuth, 1973] and load only the transactions into main memory that were necessary to mine an equivalence class. For example, all transactions starting with the item $a$ are stored in a continuous region on disk. Hence, the tid-lists are much smaller while mining this class so that no further disk access is required.

OoC [Buehrer et al., 2006] is a further out-of-core algorithm based on FP-growth. Basically, it uses virtual memory to allow FP-trees that are larger than the available physical main

---

*Recall that FP-growth obtains all these pattern bases from the initial FP-tree.*
2.4. Out-of-core and Sampling-based Algorithms

memory. To avoid trashing\(^7\) approximate sorting is employed to split the dataset into multiple files before building the FP-tree; each of these files contains logically a subtree that fits into main memory. A file after file processing thus avoids many expensive page faults since for each file only one subtree must remain in main memory. Furthermore, OoC employs the same spatial and temporal improvements that were also used in their cache-conscious FP-growth algorithm [Ghoting et al., 2005] (cf. Section 2.2.3).

Despite scaling up to very large datasets, out-of-core processing is usually several orders of magnitude slower then in-core processing because the disk accesses lead to a significant performance degradation. Indeed, high latencies of secondary memory can be masked with prefetching, asynchronous load operations, and better temporal and spatial locality. Nevertheless, one of the main bottlenecks is still the low bandwidth of disks that is—depending on the system—often more than two orders of magnitudes lower than main memory bandwidth. Thus it is beneficial to avoid out-of-core processing to the extent possible.

Finally, another way to cope with out-of-core datasets and without relying on external memory is to employ distributed processing. The basic idea is to increase the main memory of a distributed system by adding further machines. As already discussed in Section 2.3.2, the algorithms DD [Agrawal and Shafer, 1996] and IDD [Han et al., 1997] distribute candidates to the available machines and process the dataset in a stream-based manner. Pramudiono and Kitsuregawa [2003] also proposes a path distribution technique for distributed FP-growth; if local FP-trees do not fit in the memory of the machines, this technique partitions and distributes the global FP-tree—while it is created—to different machines. However, as also mentioned before, if the datasets (or intermediate results) do not fit into the memory of the single machines, usually high communication is required—limiting the scalability of the distributed algorithms.

2.4.2. Sampling-based algorithms

A promising way to reduce the computational and memory requirements of frequent-itemset mining algorithms is to process only a sample of the transaction database. For example, if only ten percent of a dataset’s transactions are used, then the algorithms usually require less than one tenth of the runtime and memory\(^8\) they would require as for processing the complete dataset. Hence, sampling can efficiently support out-of-core mining if the sample fits in main memory while the original dataset does not. Sampling, however, leads to approximate results in which some frequent itemsets might be missing.

The basic idea of sample-based algorithms is always the same; a small number of transactions is selected from the database and a regular mining algorithm is employed on this subset. Zaki et al. [1997a] were the first that evaluated the potential of sampling for frequent-itemset mining; based on three different datasets, they found that often a sample size of about 10% is sufficient to provide reasonable results with a high accuracy. Chen et al. [2002] proposed a further sample-based algorithm that works in two phases. In the first phase it picks a larger random sample of the database to estimate the support of items while in the second phase this sample is refined by removing “outlier” transactions and keeping “representative” transactions. The refined sample is then processed with a regular mining algorithm.

---

\(^7\)Trashing occurs if an application requires more than the available physical main memory and the operating system continuously swaps pages from main memory to disk and back.

\(^8\)The memory consumption of FP-growth may be same for a sample as for the full dataset because the FP-tree “summarizes” the transactions and thus its size does not depend on the number of inserted transactions.
Besides providing approximate results, Toivonen [1996] used sampling to speed up the “precise” mining process. In a first step, its algorithm draws a sample from the transaction database, which is used to generate candidates. A single full scan is required to obtain the support of all candidates and thus the frequent itemsets. If some frequent itemsets were missed in the sample—i.e., false negatives—a second scan is required. Schuster et al. [2005] provides a distributed version of Toivonen’s work.

Sampling can also be used to improve static load balancing of distributed algorithms. Cong et al. [2005] proposed a framework for distributed frequent-itemset mining based on FP-growth. The framework records the relative execution times for processing projected trees by mining a small sample of the complete transaction database. They statically partition the frequent items—used for building and mining the projected trees—based on these execution times. Buehrer et al. [2007] reused the same idea in their distributed version of FP-growth.

To summarize, sampling can significantly reduce the effort for mining large datasets. If approximate results are returned, some important or critical itemsets can be missing in the result set. This may acceptable in some application areas, e.g., for users that want quick responses (i.e., interactive mining) or for refining mining parameters like minimum support and confidence to obtain “interesting” rules (i.e., iterative mining). In the latter, sampling is used before an actual “precise” mining. However, the main disadvantage of sampling is that it works only for frequent-itemset mining. Other popular mining variants like rare-itemset and multi-support mining cannot benefit from sampling while most of the techniques presented so far can be applied to them.

2.5. Parallelization opportunities and parallel programming

This section gives an overview about parallelization opportunities of modern processors to provide the necessary background for our parallel algorithms in this work. We will explain instruction-level (ILP), data-level (DLP), and thread-level parallelism (TLP) that is available in multi-core processors. In case of thread-level parallelism, we focus solely on shared-memory systems (i.e., shared-memory multiprocessing) since distributed algorithms are beyond the scope of this thesis. For all three types of parallelism, we will present techniques, tools, and libraries that allow to exploit the respective parallelization opportunities efficiently. We consider only processors based on the x86 architecture, however, similar tools and libraries are also available for other processor architectures (e.g., PowerPC, ARM, or Itanium).

2.5.1. Instruction-level parallelism

Modern processors execute instructions using multiple subsequent stages in which each stage is completed after a single cycle. To increase the instruction throughput (number of processed instructions within a certain time) pipelined processors execute multiple instruction in parallel. Instead of waiting for the complete execution of all stages, the next instruction is issued as soon as the first stage of the previous instruction is completed. Thus, instructions are processed in a pipelined fashion allowing in the best-case to complete one instruction per cycle (IPC). Even more than one instruction per cycle can be processed using super-scalar processors. These processors have multiple execution units, which allow to process multiple instructions

---

9 The length of the pipeline (i.e., number of stages) usually influences the processor’s clock frequency. Roughly speaking, a higher number of stages allows higher clock frequencies.
2.5. Parallelization opportunities and parallel programming

in parallel as long as these instructions are independent of each other. For example, a single core of a processor that is based on Intel’s Nehalem architecture [Casazza, 2009] has six execution units, allowing to complete six instructions per cycle.\(^\text{10}\) However, the real number of instructions per cycle is often much lower than the maximum reachable. The reason for that are data and control dependencies between the instructions being executed.

Data dependencies between instructions occur when one instruction operates on the result of an other instruction; this is problematic whenever the two instructions are issued too close in the pipeline so that the result of the first instruction is not yet processed and therefore not available for the second “depending” instruction. There are several ways to deal with data dependencies in such cases. The naïve way works by inserting no-operation instructions between the two respective instructions. This reduces, however, the amount of useful executed instructions per cycle so, in general, other ways are preferred. Reordering of “non-depending” instructions—whether done during compile time using in-order optimization techniques or at runtime using out-of-order function units—is similar to inserting no-operation instructions but fills the pipeline with useful instructions. This works well if there are enough non-depending instructions. Another optimization in hardware are store-and-forward buffers within the execution units. These buffers are used to store the result of an instruction so that the next instruction can use this result earlier and without loading it from the registers. Because of all of these effective techniques, data dependencies are usually less problematic compared to control dependencies. Nevertheless, because of data dependencies there is often not enough work to feed the many execution units and therefore fully exploit super-scalar processors.

Control dependencies occur whenever the program flow is influenced by instructions. This happens at if-then-else branches within program code. In other words, instructions—within the then- or else-block—may or may not be executed depending on the result of other instructions that influence the branch condition. Control dependencies can also reduce the effect of pipelining because the next instruction of one of the two blocks can be issued after the branch condition is evaluated. Similar to data dependencies, no-operation instruction may be inserted between the last stage of the branch evaluation instruction (i.e., the conditional jump) and the first instruction of the selected block. To avoid these useless instructions, modern processors employ speculative execution in which the block with the higher probability to be taken is executed; branch prediction is used for calculating probabilities based on previous decisions or hints given by the compiler or programmer. Whenever the wrong block was selected, the instructions of the other branch must be executed and compensation for the already and wrongly executed instructions must be done. This includes restoring changed variables, registers, and flags of the processor. Nevertheless, this approach works very well as long as the branch prediction accuracy is high. For hardly predictable or unpredictable branches some processors employ predication where both blocks, the then- and else-block, are executed in parallel and the correct result is gathered from the block that satisfies the branch condition. However, this technique is not always beneficial and also available. Thus, in most cases where branches cannot be predicted, the IPC score gets low and the performance strongly degrades.

Although, there are many hardware features in processors that can cope with data and control dependencies in programs—and thus increase their IPC score—they are not always

---

\(^{10}\) Three memory operations (load, store, and address store) and three computational operations can be issued per cycle [Intel Optimization Manual, 2011].
sufficient. Often, the programmer must tune the algorithms manually or rearrange the code so that the compiler can automatically apply certain optimizations.

Programming techniques

There are several ways to increase the IPC value of an application. The main objective is in most cases to remove hardly predictable branches and transform them into data dependencies; also loop unrolling, data prefetching, and code inlining are useful. Often these techniques are applied automatically by the compiler itself. However, in some cases the programmer needs to optimize the code manually because a change of an improper data layout or even of the complete algorithm is necessary. Currently, there are no tools available that support a developer for writing code that efficiently utilizes ILP. Profiler like Intel’s VTune or the freely available Valgrind can help to detect hot-spots with low IPC scores. Both tools profile the algorithm’s execution and provide performance values like branch mispredictions, memory access behavior, and IPC scores for the functions of the algorithm. The programmer, however, must rewrite the code based on its own knowledge. An overview and detailed description about techniques that improve the IPC score can be found in Intel’s Architecture Optimization Guide [Manual, 2011] and AMD’s Software Optimization Guide [Manual, 2005]. In the following, we will explain only techniques that are used in this thesis.

For removing unpredictable branches, branch predication—as mentioned before as a hardware feature—can also be applied in software. The basic idea is the same: calculate the result both branches and select the right one based on the branch condition. To achieve this, the setting of certain status flags after evaluating the branch condition are used as a regular variable; this condition variable is set to 1 if the branch condition is evaluated to “true” while otherwise it is set to 0. It then used with the cmov instruction (for conditional move) for selecting the right branch result. The cmov instruction allows to overwrite a variable with one of two other variables controlled by a third variable, i.e., the variable that should hold the final branch result is overwritten with one of the branch results controlled by the condition variable.\footnote{A similar instruction that works on SIMD registers is the blend instruction of SSE4.1 that selects bytes from two input variables based on a third variable.}

For another form of software predication, the condition variable is used for integer arithmetic. For example, it can be used for a conditional increment operation where a variable is incremented whenever the branch condition is “true”. Combined with other arithmetic instructions (e.g., multiplication or shift) arbitrary values can be “conditionally” added.

```c
int o = 0;
for(int i = 0; i < 1000; i++)
    if(input[i] > constant) {
        output[o] = input[i];
        o++;
    }
```

**Figure 2.6.** Branch in main loop

```c
int o = 0;
for(int i = 0; i < 1000; i++) {
    output[o] = input[i];
    o = o + (input[i] > constant);
}
```

**Figure 2.7.** Branchless version

Figure 2.6 shows an example where all values that are greater than a variable constant are copied from an array input into an array output. Whenever the branch condition evaluates to “true”, the i-th value of the first array is written to the o-th position in the second array.

11
In this example, software predication could be applied and may increase the IPC score. The “predicated” code—illustrated in Figure 2.7—writes in the i-th iteration the i-th value of the input array at the o-th position of the output array. The branch condition is used as a variable and is increased only for a branch condition evaluated to “true”. Thus, copied values that do not satisfy the condition are overwritten. The compiler itself cannot perform this rewriting because the “predicated” code is not semantically equivalent to the original one, i.e., their results differ if the last value of the array input is not copied. The performance of both code fragments depends on the predictability of the branch. For a good predictability—almost all or almost none of the values satisfy the condition—the original code would be superior over the branchless code. Otherwise, if this condition is hard to predict and both branches are selected with equal probabilities, then the “predicated” code usually performs better. Thus, predication is only useful if it is applied to branches that are hard to predict.

Loop unrolling (also loop unwinding) is applied on loops to reduce the costs for evaluating loop conditions as well as managing associated loop variables. The main idea is to replicate a loop’s body n times and increment the loop counter by n after each iteration. Loop unrolling increases the code size—and may be contra-productive if the code does not fit in the instruction cache anymore—but enables the compiler to apply various other optimizations on the code [Intel Optimization Manual, 2011]. One of them is software pipelining [Allan et al., 1995]. It avoids data dependent latencies by reordering the unrolled instructions; they are shuffled in a round-robin manner. Out-of-order execution units perform this reordering without compiler support but this works only when there are no dependencies between instructions of different iterations (loop dependencies). For example, both code fragments (Figure 2.6 and 2.7) cannot benefit of software pipelining because there is a loop dependency for the assignment of the variable o. Loop splitting and peeling [Callahan and Kennedy, 1988] can be employed if such dependencies cannot be avoided but relaxed in a certain way. The basic idea is that each replicated loop body processes a different continuous part of the loop iterations. Hence, data dependent latencies are avoided because the instructions within a replicated loop body have no dependencies to the instructions of an other loop body and can be shuffled. In the context of our conditional copy example this means to conditionally copy multiple arrays of the same length to different locations.

To summarize, the programmer can apply certain optimizations to remove control dependencies and avoid data dependent latencies, which are caused by data and loop dependencies, within performance-critical loops. This can be done with thoughtful programming or by applying certain application knowledge that allows to rearrange the code.

2.5.2. Data-level parallelism

Apart from their regular instruction sets, modern x86-processors provide SIMD instruction sets (for single instruction, multiple data). Initially introduced with the Multi Media Extension (MMX) for the Intel Pentium processor, there exist a large number of such instruction sets like 3DNow, different revisions of the Streaming SIMD Extensions (SSE), and the Advanced Vector Extensions (AVX). Compared to traditional scalar instructions that process only a single data element at once, a SIMD instruction can process p elements at once. The value of p depends on the capabilities of the instruction set as well as on the data type.

---

12In this case, the o-th value in the output array is undefined for the original code while it is overwritten with the last value of the array input for the “predicated” code. The variable o has the same value in both cases so that this behavior may be acceptable but must be considered when the “predicated” code is used.
2. Frequent-itemset mining

For example, most SSE instructions can process four 32-bit or eight 16-bit values at once, while AVX instructions [AVX Manual, 2011] allow to process eight 32-bit values. Because of the many different instruction sets, we will now classify and briefly review available instructions.\textsuperscript{13} We restrict our attention to SIMD instructions that work on integer values because we use only these in the algorithms of the following chapters. For this reason, we review only SSE instructions, which work with 128-bit SIMD registers, since currently available AVX instructions are restricted to floating point operations.

**Instruction classification**

The SIMD instructions exploited in the algorithms that are proposed in this thesis each belong to one of the following four classes: (1) load and store instructions, (2) element-wise instructions, (3) conversion instructions, and (4) horizontal instructions.

*Load and store instructions* copy data from memory into a SIMD register and vice versa. Currently, all of them can only copy continuous chunks of memory. For example, if four 32-bit values should be loaded into a SIMD register with one instruction, all four values must reside in a continuous block of 128 bit. For this reason, $p$ load instructions are required to load $p$ values from non-continuous memory locations into a single SIMD register. Processor architectures with gather-and-scatter instruction support alleviate this problem by allowing loading from and storing to $p$ non-contiguous addresses; the upcoming Haswell architecture will provide gather instructions (see the AVX Manual [2011]). Note that loading continuous chunks will be faster than “gathered” loading because almost always only a single cacheline needs to be fetched (rarely two cachelines) while gather-and-scatter instructions often require to fetch $p$ cachelines. For each operation that works on chunks there exist two instructions; one that can cope with only aligned memory access and one without alignment restrictions. For current processors, there is no performance difference whether aligned or unaligned memory load/store instructions are used. However, unaligned loads can cause cacheline splits so that two cachelines need to be loaded. *Set instructions* constitute a special case of store instructions. These instructions, which are available for different data types, take as input a scalar value and fill a SIMD register with $p$ copies of that scalar value.

*Element-wise instructions* execute an operation element-wise on one or more SIMD registers, i.e., the $n$-th element of the output register only depends on the $n$-th element of each input register. We make use of element-wise versions of arithmetic integer functions ($+,-,\cdot,/$), bit-shifting, logical functions (AND, OR, NOT), and comparison functions ($<,=,>$). All functions store their results in another register, which can be treated as a vector of integers; comparison functions also use integer values to represent their result. The comparison result is either 0 for “false” (all bits are set to zero) or $-1$ for “true” (all bits are set to one). Notice that, this is different from the scalar “true” comparison result that is 1.

*Conversion instructions* transform the datatype of values into another datatype—e.g., floating point to integer or 32-bit integers into 16-bit integers. The respective instructions use one or two vector registers as input and write the “converted” values into a single vector register or a scalar value. Similar to scalar instructions, converting values of higher precision into values with lower precision may lead to an information loss. In this thesis, we solely require the instructions **PUNPCKHWD** and **PUNPCKLWD**, which take two vectors as input and mix

\textsuperscript{13}Parts of the following description are taken from the background description of previous published works [Schlegel et al., 2009, 2010, 2011b].
the eight upper (PUNPCKHWDM) or eight lower (PUNPCKLWDM) 16-bit elements of both vectors and write them into an output vector.

Finally, horizontal instructions act across the elements of a SIMD register. Here, the \( n \)-th element of the output vector can possibly depend on all elements of the input vectors. In this thesis, we use one of the four STTNI instructions of SSE 4.2. Originally intended for string comparison, the four instructions differ only in the output format (index or bitmask) and with respect to the length specification of the processed strings is given explicitly or implicitly (using a ‘0’-termination symbol). The execution procedure of all four instructions consist of three stages: (1) full comparison of two input vectors both consisting of either sixteen 8-bit values or eight 16-bit values. (2) aggregation of the intermediate comparison result using one of four different aggregation functions. For our algorithms, we use the “equal any” aggregation function; this function XORs the comparison results per element for the second input vector. In other words, if there is at least one element in the first input vector that is equal to the \( n \)-th element of the second input vector, then the \( n \)-th bit in the aggregated bitmask is set to one. (3) the final output is returned. The STTNI mask instructions return the aggregated bitmask while the STTNI index instructions return the position of the first or last bit in the aggregated bitmap. In our algorithms, we use the STTNI mask instruction PCMPESTRM that requires an explicit length specification.

Further required horizontal instructions are the PMOVMSKB and PSHUFB instruction. The former instruction is applied on a 128-bit SIMD register and computes a 16-bit mask having the \( i \)-th bit set to 1 if the most significant bit of the \( i \)-th byte within the register is set to 1. The latter instruction—denoted as byte-shuffle instruction—is part of the SSSE3 instruction set and performs arbitrary in-place shuffles of the bytes in a SIMD register. The byte permutation itself is controlled by a shuffle control mask. It stores for each byte of the output vector which byte of the input vector should be copied to it.

All discussed instructions are supported by Intel and AMD processors starting with the Nehalem and Bulldozer architecture, respectively. Further details about these instructions are available in the Intel’s Basic Architecture Guide [Manual, 2011].

Programming techniques

Utilization of SIMD capabilities in algorithms is ideally achieved through automatic vectorizing by compilers or by inserting SIMD instructions manually. The basic idea is always the same: vectorize loops by exchanging scalar instructions with corresponding SIMD instructions and thus reduce the number of iterations of the loop, i.e., instead of \( n \) iterations only \( n/p \) iterations are required. However, this works only if (1) there are no loop dependencies, (2) the data being processed is stored in continuous arrays from which \( p \) values can be loaded with a single load instruction, and (3) there is a substitute SIMD instruction for each scalar instruction that has to be exchanged. In all other cases, a change of the algorithm or its underlying data layout is necessary so that compilers cannot apply automatic vectorization, i.e., the programmer has to vectorize the code manually.

Currently, there are no tools available providing help for an effective vectorization of complex code. The programmer itself has to restructure and parallelize the code based on standard optimization techniques. As mentioned before, loop dependencies can often be removed by exploiting certain application knowledge. For example, by using loop splitting and peeling as explained in Section 2.5.1. Storing the data in continuous arrays so that \( p \) values can be loaded with a single load instruction is more involved. This often requires a change of
2. Frequent-itemset mining

The algorithm and its data layout. We explain such changes for our algorithms in Chapter 3 and 4. This includes aligning the data being processed on 16-byte boundaries, i.e., the data is stored starting at a memory address that is a multiple of 16, to avoid cache line splits while loading the data into the registers. Finally, not all scalar instructions have a SIMD counterpart. In some cases, a desired SIMD instruction must be “emulated” from multiple available SIMD instructions executed sequentially or avoided by changing the algorithm. The former usually reduces the speedup of vectorization. For example, there are no greater-than SIMD comparison instructions that work on unsigned data types. Hence, either such an instruction is emulated using two available instructions or the processed data type must be transformed into signed integer. However, there are also SIMD instructions that have no scalar counterpart like the PSHUFB instruction; it allows to substitute multiple scalar instructions and thus can be used for optimizations that would not be possible with scalar instructions.

Inserting SIMD instructions using assembly is usually done via asm-blocks within the C-code. Since assembly is not portable and furthermore difficult to program and maintain, SIMD intrinsics are preferable and, therefore, are used in the algorithms of this thesis. To use intrinsics, specific header files must be included. For example, xmmmintrin.h, emmintrin.h, pmmintrin.h, smmintrin.h, and xmmmintrin.h contain intrinsics for SSE, SSE2, SSE3, SSE4.1 and SSE4.2, respectively. Besides the intrinsics for instructions, also new data types are available that represent 128-bit registers: __m128 for floating point values and __m128i for integer values. The precision of data type itself is controlled by the programmer, i.e., the programmer must choose the right instruction if two vectors each with sixteen 8-bit values (= PADDB), eight 16-bit values (= PADDW), four 32-bit values (= PADDD), or two 64-bit values (= PADDQ) should be added. A good overview about vectorization techniques can be found in Chapter 4 and 5 of Intel’s Optimization Guide [Manual, 2011] and Chapter 10 of AMD’s Optimization Guide [Manual, 2005].

2.5.3. Thread-level parallelism

TLP is the third type of parallelism that is supported by modern processors. Instead of executing only a single thread at a time, multiple threads can be executed in parallel. The number of parallel threads depends on the capabilities of the underlying hardware; these can be classified into simultaneous multi-threading, multi-core, and multiprocessors. In the following, we will review these capabilities and discuss their speedup potential.

Simultaneous multi-threading (SMT) permits to run multiple independent threads on a single core. The main idea is to better utilize the execution units of a super-scalar processor (cf. Section 2.5.1) by sharing them between multiple threads. Whenever a single thread does not require all execution units, instructions of other threads are executed on them to increase their utilization. This works well since the concurrent threads have no dependencies between each other. Thus, the ILP is increased and therefore higher IPC scores are achieved. To support SMT, a processor has to maintain multiple thread contexts; this is realized by making a full set of registers available for each of the threads. Scheduling of the instructions of the concurrently executed threads is usually done via an out-of-order engine. Intel’s SMT realization is denoted as Hyperthreading [Marr et al., 2002]; it allows to execute two threads simultaneously and achieves in average a 15 to 30 percent better performance compared to an

14 A greater-than SIMD comparison instruction for unsigned data types can be emulated using an element-wise max instruction (PMAXUx) that works on unsigned data types and an equal-to SIMD comparison instruction (PCMPEx).
2.5. Parallelization opportunities and parallel programming

execution without SMT. More details to Intel’s Hyperthreading can be found in Intel’s Basic Architecture guide [Manual, 2011]. The AMD Bulldozer architecture [Butler, 2010] provides some “extended” SMT; each Bulldozer core is denoted as module and allows to execute two threads simultaneously. A module, however, has twice as much integer execution units compared to a regular core so that the two threads share only the floating point execution units. Therefore, this technique can be seen as a hybrid between SMT and multi-core. Although SMT allows to increase the utilization of the processor’s execution units, it allows only a limited speedup when certain execution units are required by both threads.

Multi-core processors—also denoted as chip-multiprocessors—provide multiple homogeneous cores integrated in the same chip; $p$ cores allow to process $p$ times the number of threads in parallel compared to a single core. Currently, the Intel and AMD server processors have up to 10 cores (“Westmere EX” [Nagaraj and Kottapalli, 2010]) and 16 cores (“Interlagos” [White, 2011]), respectively. In theory, a multi-core processor with $p$ cores would allow a speedup of $p$ times when executing “perfect” parallel algorithms. However, this is rarely the case since each algorithm has a sequential part (cf. Amdahl’s Law [Amdahl, 1967]), where parallelism cannot be exploited; synchronization between the threads introduces overhead that would not be necessary in sequential algorithms; and components like the caches and memory bandwidth must be shared among the threads. Nevertheless, for well parallelized algorithms high speedups close to $p$ are possible.

Multiprocessors or multi-processing systems consist of multiple processors that are connected with each other to allow processing a common problem. Two important classes of such systems are symmetric multiprocessing (SMP) and non-uniform memory access (NUMA) multiprocessing systems. Systems of both classes have a “shared” main memory with a single address space—i.e., all processors can access the complete memory—but differ in the way they realize the connection of the processors to this memory. SMP systems use a common bus for this connection, which, however, strongly limits the scalability in terms of the number of connected processors. For this reason, the trend goes to “point-to-point” connections that can be found in NUMA systems. Figure 2.8 gives a schematic overview of such a system with four processors, which also describes the larger of both test systems used in this thesis. Each processor has multiple cores, which have their own L1- and L2-cache but share a common L3-cache; an own integrated memory controller (IMC); and dedicated memory. Access to this “local” memory is always faster than accessing memory that is dedicated to other processors. This makes the placement of the data important. Point-to-point connections are used to connect the dedicated memory (also the I/O chipset) as well as other processors. Processors are connected on Intel systems via Quickpath Interconnect (QPI) [Whitepaper, 2008] and

![Figure 2.8: Multiprocessor system with 4 processors](image)
2. Frequent-itemset mining

on AMD systems via HyperTransport [Whitepaper, 2004]. Current AMD and Intel systems allow to connect up to four and eight processors, respectively, within in a single system.

To summarize, TLP is available via SMT per core, multiple “cache-connected” cores per multi-core processor, and multiple point-to-point connected processors per NUMA multiprocessing system. Hence, systems can process up to 160 threads in parallel, if it has eight 10-core processors and SMT is enabled. Such a high number of threads can only be exploited efficiently if almost no thread synchronization is required and the load can be evenly distributed. For this reason, we will now give a short overview about decomposition, synchronization, and load balancing techniques that are required in this thesis. Thereafter, we discuss libraries that ease writing multi-threaded applications and allow to influence the placement of data.

Principles for multi-threaded algorithms

The first step for exploiting TLP is to decompose work into smaller portions of work that can be processed in parallel. For that purpose, there exist several decomposition strategies [Kumar, 2002] of which we employ two in this thesis. Data decomposition divides the data being processed in multiple partitions. Thereby, the chosen partition granularity has a large impact on the performance and scalability of the multi-threaded algorithm. If the partitions are too large, then work imbalance might occur where threads are being idle. On the contrary, if the partitions are too small, then the distribution of the work introduces serious overhead. Task decomposition divides the task itself into possibly different subtasks. These subtasks could then be run in parallel when certain dependencies between them are fulfilled. After each iteration in Apriori, for example, the subtask of writing out the results could be performed in parallel to the subtask of generating the candidates for the next iteration.

TLP works best when the data or task being processed can be decomposed into independent partitions or subtasks (like for parallel matrix multiplication [Dekel et al., 1981]) where synchronization is not required during processing. For frequent-itemset mining algorithms, however, it is often required—for example, if multiple threads obtain the frequencies of item-sets in parallel, then communication is needed to get the final result. For such cases, there exists a large amount of synchronization primitives. The most important of them are mutexes and futexes, which are used to circumvent multiple threads entering a critical section of a program. The critical section itself contains either system calls that are only permitted to be invoked by a single thread at a time (e.g., most memory allocation calls) or secures multiple variables that might be written by multiple threads at the same time, i.e., to avoid race conditions where a variable is updated by two threads and one of the updates is lost. Both, mutexes and futexes, usually lead to lock contention; whenever a single thread attempts to enter a critical section that is already occupied by another thread, the thread outside the critical section is put to sleep. This is quite expensive because the operating system scheduler is invoked. Spinlocks avoid this scheduling overhead by continuously polling the lock variable until the other thread leaves the critical region. This works well for short critical regions while otherwise the “polling” thread wastes a large amount of CPU cycles. Atomic instructions like fetch-and-add and compare-and-swap allow to execute multiple operations on a single variable without being interrupted by an other thread. They are often considered to be superior over other synchronization primitives because they do not rely on a lock variable that must be transferred to the core on which the thread currently runs on before a shared variable can be changed.

15Futexes are handled in userspace while mutexes require to invoke kernelspace.
be safely overwritten. Nevertheless, also atomic instructions should be avoided because the cache line being updated must be transferred whenever a thread updates it and it is in the cache of another core. Hence, all of these synchronization primitives should be avoided or at least their use should be minimized because they lead to expensive communication (e.g., bus snooping, request for ownership changes of cachelines) between the cores and processors of a system. More details about synchronization primitives and their overhead can be found in Intel’s Optimization Guide [Manual, 2011].

Parallel algorithm models (or parallel programming patterns) are blueprints for parallel algorithms with minimal synchronization. They combine decomposition techniques with a certain strategy that minimizes communication between the threads. Our algorithms in this thesis basically rely on two different models. The data-parallel model resembles to data level parallelism, but with threads. All threads perform the same or similar instructions on different parts of the data being processed; communication is typically only required to distribute the “unprocessed” parts to the threads and merge the final results. Figure 2.9a illustrates an example for the data-parallel pattern; a single dataset is decomposed in three equally sized partitions, which are processed by three threads. The producer–consumer (or pipeline) model processes the data in two or more stages where—possibly multiple—threads apply on each stage different operations on the data. Thereby, the data flows like a stream through the stages so that ideally many threads can work in parallel. Figure 2.9b depicts an example for this pattern with three stages. The first stage is a single producer, which reads and parses a dataset; the parsed transactions are passed to the second stage where infrequent items are removed; finally, the filtered transactions are used in the third stage to count frequent pairs within them. Clearly, the producer–consumer model works only well when there are enough stages or many threads can work in parallel on single stages. Kumar [2002] provides more details to these two and also other parallel programming pattern.

Finally, load balancing is used to keep all threads busy during processing, i.e., to reduce the time they are being idle. Static load balancing is employed in all cases when sufficient knowledge about the load is available. The data or tasks are distributed to the threads before the actual processing, i.e., communication is not required during processing. In Figure 2.9a, for example, each thread could get one third of the dataset assigned before processing if each part “produces” a similar load. Dynamic load balancing is used in the remaining cases where the load cannot be predicted or calculating an optimal load distribution is too expensive. The threads have always one partition or subtask assigned and gather new work whenever they are finished with their current work. Although more communication is required, idle times are often better avoided than for static load balancing.
2. Frequent-itemset mining

Programming techniques

There are several ways for exploiting TLP in programs. As for the other types of parallelism, compilers itself support automatic parallelization. For example, the Intel ICC compiler has a parameter `-parallel` that enables the generation of multi-threaded code by distributing the iterations of certain loops to different threads. Eligible loops should have many iterations and almost no dependencies. Nevertheless, as before, automatic parallelization often produces code that is inferior compared to hand-written parallel code—especially if the sequential code is complex or a change of the algorithm is required for an effective parallelization. In the following, we will thus review different ways for creating multi-threaded code by hand.

The parallelization using POSIX threads (also Pthreads) is a common way to exploit TLP on a low abstraction level. The programmer has a fine-grained control over the parallelism. This, however, requires management of the threads by hand—including creation, synchronization, and assignment of tasks—and thus largely increases the complexity of the code. An extensive overview about programming with Pthreads is available in Butenhof [1997].

```c
int sum = 0;
#pragma omp parallel for reduction(+:sum)
for ( int i = 0 ; i < 1048576 ; i++ ) {
    sum += a[i];
}
```

**Figure 2.10.** Parallelization of a loop using OpenMP

The OpenMP framework allows an effective parallelization on a higher level compared to Pthreads. Here, only a few lines of code—often only a single line—are required for parallelizing certain blocks of a program. The framework consists of libraries, compiler directives, and environment variables. The libraries contain functions for interacting with the runtime environment like obtaining a thread’s id via `omp_get_thread_num()`. The compiler directives—inserted as preprocessor `#pragma` constructs—are used to annotate OpenMP constructs to single instructions or blocks of instructions. Among others this includes constructs for parallel execution, worksharing, tasking, and synchronization. Figure 2.10 illustrates an example where a loop is parallelized using the `parallel for` worksharing construct. This construct is a realization of the data-parallel model and distributes the loop’s iterations equally among the available threads. Each thread further works on an own copy of the variable `sum`; all of its copies are afterwards “reduced” into a single variable by summing up their values. Since the constructs are inserted only using preprocessor directives, the code can also be compiled as a sequential program. The environment variables are used to control the parallelism before runtime; this includes the number of available threads or global load balancing policies. More details to OpenMP can be found in Chapman et al. [2007] and the OpenMP Reference Manual [2008].

Besides OpenMP, there exist also other frameworks that ease writing multi-threaded programs. One of them is Intel’s Threading Building Blocks (TBB), which is similar to OpenMP but is solely available for C++ because it is based on template programming. The programmer specifies tasks using several available templates; TBB maps these tasks to threads. The templates include among others range partitioning of loops, pipelining, and parallel reduce. As for OpenMP, several synchronization primitives—like mutexes, spin-locks, and atomic instructions—are available. A detailed description can be found in Reinders [2007] and the
Intel TBB Manual [2011]. *OpenCL* is another popular framework for parallel programming; its main focus is on heterogeneous computing. Target platforms are mainly CPUs and GPUs, but also digital signal processors and inherent heterogeneous processors like the Cell Processor [Kahle et al., 2005] and AMD Fusion processors [Brookwood, 2010]. In general, OpenCL constitutes an abstraction of different parallel programming frameworks. Stone et al. [2010] and the OpenCL Specification [Munshi, 2011] provide further details for OpenCL. Despite both TBB and OpenCL are well suited for parallelizing many algorithms, we found them unsuitable for our algorithms. TBB is often too abstract, e.g., threads are hidden from the programmer, while OpenCL has certain problematic restrictions, e.g., it does not support recursion. For this reason, we employ OpenMP within our parallel algorithms.

Additionally to the OpenMP framework, we use the *numalib* library for controlling the data placement on our NUMA multiprocessing system. As already mentioned, the latency of a single memory access within such systems depends on the location of the data, i.e., in which of the processors local memory the data resides. If not controlled explicitly, the first-touch allocation policy takes place whenever allocated memory is mapped to physical pages, i.e., pages were taken from that processor the current thread is running on. Only if the processor has not sufficient pages, they were taken from an arbitrary processor. The first-touch policy usually works well as long as threads do not migrate often to other processors. Nevertheless, it could be problematic if, for example, the data structures of an algorithm are initialized by only a single thread because then all of the algorithms pages could be located in the dedicated memory of a single processor. The processor’s memory bandwidth then could be quickly saturated when all threads read from its memory. To avoid such problems, the *numalib* library provides several functions like `numa_alloc_onnode()` that allow an explicit placement of the data. It further facilitates to bind threads on processors or even certain cores.

### 2.6. Summary

In this chapter, we gave the necessary background for this thesis; we started with a discussion of the overall association rule mining process in which itemset-mining—as an essential part—amounts for a large part of the overall runtime. We illustrated several application areas and discussed the potential size of their respective datasets. Since these datasets can easily grow to tens, hundreds, or even thousands of gigabytes and itemset mining itself is a computational and memory expensive process, it is meaningful to optimize it. Although our focus is primarily on frequent-itemset mining, we illustrated algorithms for mining other types of itemsets. These algorithms also benefit of such optimizations, which makes them even more useful.

We reviewed a large quantity of existing algorithms and classified them into APRIORI-based, ECLAT-based, and FP-GROWTH-based algorithms. As shown by various authors, there is no best itemset mining algorithm and each class has an algorithm that is for a certain dataset and ξ value superior over all other algorithms. For this reason, we aim to improve algorithms of all three classes by speeding up their “base” algorithms APRIORI, ECLAT, and FP-GROWTH. We achieve this by employing different kinds of parallelism to the most expensive parts of these algorithms. For APRIORI and ECLAT, which both rely on candidate generation and testing, the support counting is usually the most expensive step when mining large datasets. FP-GROWTH grows pattern by repeatedly building smaller FP-trees from a large initial FP-
2. Frequent-itemset mining

tree that summarizes the transaction database; building this tree and the recursive mining of it are here the most expensive steps.

There are already numerous parallel mining algorithms available; we classified them according to the system architecture they are intended for: “shared-memory” systems, distributed systems, and graphic processors. Thereby, we discussed workload partitioning and load balancing techniques; most of them do not work for a large number of threads and need to be improved. Although our algorithms in this thesis are intended for “shared memory” systems, we apply some ideas from algorithms for distributed systems. Algorithms for graphic processors, however, are mainly included to illustrate that these are still not applicable for mining very large datasets; this is because of the (current) limitations of this architecture. As mentioned before, mining of large datasets works only well, if the algorithm’s working set fits into main memory. We thus reviewed out-of-core algorithms that show a much better performance when main memory is limited. Nevertheless, they rely on expensive disk access and can only soften the performance degradation when main memory is not sufficient. We further reviewed sampling algorithms which seem to be—at a first glance—a suitable solution for mining large datasets but they are not for several reasons.

In the remainder of the chapter, we reviewed the three types of parallelism that are available in modern processors. Both, ILP and DLP, are usually exploited automatically by the compiler for simple code fragments. However, in many cases the programmer has to tune manually performance-critical parts of its code to benefit from these types of parallelism. For ILP this means to avoid dependencies within the code, transform control dependencies into data dependencies at unpredictable branches, and apply certain techniques for hiding data dependent latencies. Optimizing for DLP means to partition the task into \( p \) tasks where \( p \) denotes the vector width, changing the data layout, and exchanging sequential instructions with SIMD counterparts. Finally, parallelism on thread-level requires always annotations in the program code like OpenMP directives at loops and at blocks that need to be synchronized. In general, the programmer has to partition the data or tasks into independent partitions or subtasks, avoid race conditions and false sharing, and decide on using a static or dynamic load balancing algorithm to balance the load of an algorithm. In Chapter 3 and 4 we improve the algorithms, whose application area is not restricted to frequent itemset mining, mainly using ILP and DLP. Exploiting TLP depends on the higher-level algorithms that integrate them and is thus not the primary focus in these chapters. Contrary, our three parallel frequent-itemset mining algorithms, which are proposed in the Chapters 5–7, mainly exploit TLP and are optimized for a large number of threads.
Part II.

Basic algorithms
3. Sorted-set intersection

In this chapter\(^1\), we deal with the key performance hotspot of Eclat-based algorithms: the intersection of sets performed during their support counting phase. Recall from Chapter 2 that Eclat [Zaki et al., 1997c] transforms the transaction database into a set of tid-sets—represented either as tid-lists or tid-bitmaps—which are repeatedly intersected to obtain the frequencies of certain itemsets. The number of tid-sets depends on the number of frequent items and each tid-set of an equivalence class is intersected with all other tid-sets of the same equivalence class. Hence, the number of tid-sets is usually high and there are many independent intersections performed. The goal of this chapter is therefore to provide efficient algorithms for a fast intersection of multiple tid-lists and tid-bitmaps. Thereby, the main focus is on exploiting ILP and DLP since the independent intersections can be easily distributed to the available threads of a system, i.e., there is no need for speeding up a single intersection using multiple threads.

In Section 3.1, we survey existing set intersection algorithms that are eligible for support counting in Eclat-based algorithms. We first review existing merge-based list intersection algorithms, e.g., adaptive, hierarchical, and parallel list intersection algorithms. Based on that, we discuss the state-of-the-art merge-based list intersection algorithm and its parallelization opportunities in more detail; we provide an ILP-optimized version of it and explain why it is not amenable for DLP. In the remainder of this section, we discuss an efficient bitmap intersection algorithm, which is eventually used for intersecting tid-bitmaps.

We propose intersection algorithms that exploit DLP in Section 3.2. These algorithms perform a large amount of comparisons speculatively, i.e., much more than the necessary comparisons as done by the scalar list intersection algorithms. To do this efficiently, we exploit the string and text processing instructions (STTNI) offered by SSE4.2. Basically, the speculative comparison algorithms differ only in the precision of the integer values they process. We propose versions for 8-bit, 16-bit, and 32-bit integer values; each of them uses uncompressed integer values as input and output. Integer values with a precision larger than 16 bits require a preprocessing before the actual intersection. For this reason, we present a hierarchical data layout for such integer values that avoids this preprocessing.

Finally, we evaluate the performance of the discussed intersection algorithms in Section 3.3. We implemented highly efficient list-based and bitmap-based intersection algorithms and evaluated their performance on synthetic datasets; the list-based algorithms include traditional scalar and our speculative comparison algorithms. We evaluate the influence of the different kinds of parallelism as well as the point at which merge-based approaches are superior over bitmap-based approaches and vice versa. Our intersection algorithms are up to 5.5x faster than existing state-of-the-art intersection algorithms, which translates into a better overall performance of Eclat-based algorithms. This makes them also useful for other application areas like for query processing in databases [Raman et al., 2007] and information retrieval [Witten et al., 1999].

\(^1\)The material in this chapter has been developed jointly with Thomas Willhalm and Wolfgang Lehner. The chapter forms an extended version of prior published work [Schlegel et al., 2011b].
3. Sorted-set intersection

3.1. Background and Related Work

In this section, we survey related work in the area of set-intersection algorithms. We start by classifying set intersection algorithms that are eligible for Eclat. Based on that, we discuss the state-of-the-art merge-based list intersection algorithm and its parallelization potential. In the remainder, we present an efficient bitmap intersection algorithm.

3.1.1. Related Work

There exists a large variety of sorted-set intersection algorithms, which can be roughly classified into search-based and merge-based algorithms. The former algorithms [Hwang and Lin, 1972] are employed, if the cardinality of the two sets being intersected differs greatly; e.g., one set is hundred times larger than the other set. The basic idea is to search each value of the smaller set within the larger set using binary or interpolation search. In our application area, the sets (i.e., the tid-sets) have usually a similar cardinality. Indeed, there are some large sets but most sets have a cardinality that is close to the minimum support count \( \xi \). For this reason, we focus in the following on the merge-based intersection algorithms, which are well suited for intersecting sets that have a similar cardinality. The ancestor of these algorithms is the simple merge intersection algorithm—often also denoted as zipper. It iterates stepwise through both sets similar to the merge step of merge-sort [Knuth, 1973]. All other merge-based intersection algorithms are variations of this algorithm and differ from it in how they iterate through and how they represent the sets. Basically, these algorithms itself can be divided into adaptive intersection and hierarchical intersection algorithms.

Adaptive algorithms [Demaine et al., 2000, Baeza-Yates, 2004, Barbay et al., 2006, 2010] focus on reducing the number of comparisons during the intersection; this is achieved by combining the merge algorithm with various search techniques and works well when applied on skewed data. However, adaptivity sometimes introduces a large overhead that offsets the benefit of the fewer comparisons [Demaine et al., 2001]. Ding and König [2011] even showed recently that adaptive intersection algorithms are often much slower on modern processors than the simple merge algorithm. For this reason, we do not consider to employ such algorithms for support counting in Eclat-based algorithms.

Besides the pure merge approaches that work on sets represented as lists or arrays, there are also hierarchical approaches that represent sets using various data structures and perform the intersection on these representations. Thereby, the chosen data structure always fulfills a certain purpose. Representing the sets using tree-like data structures, e.g., AVL trees [Brown and Tarjan, 1979] or treaps [Blelloch and Reid-Miller, 1998], improves the updateability of the sorted sets. Partitioning the sets being intersected into smaller subsets [Tsirogiannis et al., 2009] or the combination of hashing with bitmap intersection [Ding and König, 2011] speed up the intersection if the output set has a small cardinality. Both, a better updateability or optimizations for small result set cardinalities, however, are not required in our application because (1) all tid-sets generated during support counting are created once and are never changed and (2) the cardinality of an output set is typically high, i.e., it is larger than the absolute minimum support \( \xi \). Nevertheless, hierarchical approaches can also be used for reducing the amount of memory required for representing the sets [Sanders and Transier, 2007], which is indeed useful for reducing the intermediate state of Eclat-based algorithms. Therefore, we reuse such techniques within our list intersection algorithms that process integer values with precisions equal to or larger than 32 bit.
3.1. Background and Related Work

Based on the hierarchical intersection algorithms, there further exist parallel set-intersection algorithms [Tatikonda et al., 2009, Tsioigiannis et al., 2009], which focus on speeding up a single intersection using multiple threads. For example, Tsioigiannis et al. [2009] propose a set partitioning that allows an efficient load balancing for multi-core processors. As already mentioned, ECLAT-based algorithms perform many independent intersections during support counting so that exploiting TLP for a single set intersection is usually not necessary.

To summarize, the simple merge intersection algorithm is best suited for intersecting tid-sets within frequent itemset mining algorithms based on ECLAT. Hence, we use this algorithm as the foundation of our speculative intersection algorithms and therefore discuss it in the following in more detail.

3.1.2. List intersection

Tid-list intersection uses two sorted, duplicate-free sorting of tid-lists as input and returns a sorted tid-list that contains the common values of both input lists. Let \( A \) and \( B \) be two sorted tid-lists of length \( l_a \) and \( l_b \), respectively, and \( C \) the result list. The merge algorithm iteratively compares two values taken from each list; it starts with the first value of both lists. Whenever the two compared values are equal, the value is written into the result list and a counter—that counts the number of common values—is incremented. Then, the next value of each list is loaded for the next comparison. If the compared values are unequal, only the next value of the sorted list with the smaller value is loaded and compared with the current value of the other list. The algorithm finishes when there is no next value in one of the two lists.

\[
\begin{align*}
1 & \quad 2 \quad 3 \quad 9 \quad 11 \quad 12 \quad 16 \quad 17 \quad 18 \quad 21 \quad 22 \quad 26 \quad 27 \quad 32 \quad 34 \quad 35 \\
1 & \quad 6 \quad 7 \quad 8 \quad 16 \quad 17 \quad 18 \quad 19 \quad 23 \quad 26 \quad 28 \quad 30 \quad 31 \quad 32 \quad 33 \quad 34
\end{align*}
\]

**Figure 3.1.**: Scalar merge intersection of two sorted sets with 22 comparisons; the resulting intersection set consists of 7 elements (gray shaded).

The worst-case number of comparisons for intersecting two tid-lists of length \( l_a \) and \( l_b \) is given by \( l_a + l_b - 1 \); in this case, the lists have no common value. The best-case requires only \( \min(l_a, l_b) \) comparisons. Figure 3.1 illustrates the scalar merge intersection using an example. Here, 22 comparisons are necessary to find the 7 values that occur in both sets.

Figure 3.2 shows C-code for an implementation of the merge algorithm that works on lists consisting of 16-bit integer values (short). Basically, it consists of a loop, which is repeated as long as the indices \( i_a \) and \( i_b \) refer to the current list values being compared, are smaller than the length of lists \( l_a \) and \( l_b \), respectively. The loop body itself consists of two branches; they are used to maintain the indices. The first branch checks if two values are equal; if it is taken, both list indices are increased and the common value is written into the result list \( C \). Otherwise, if both values are unequal, the second branch is used to decide which list has the smaller value; only the indice of this list is increased.

---

2 We focus solely on list intersection of integer values without any duplicates. This is sufficient for frequent-itemset mining because there are no duplicates within a tid-list. Nonetheless, the proposed list intersection algorithms within this and the following section could be extended to allow duplicates by storing the frequencies of the integer values.
3. Sorted-set intersection

```c
int intersect(short *A, short *B, int l_a, int l_b, short* C) {
    // initialize variables
    int count = 0, i_a = 0, i_b = 0;

    // repeat the loop body until the end of an input tid-list is reached
    while(i_a < l_a && i_b < l_b) {
        // compare the current values of both lists
        if(A[i_a] == B[i_b]) {
            // both values are equal - increase both list indices
            // and write the current value into the output list
            C[count++] = A[i_a];
            i_a++;
            i_b++;
        } else if (A[i_a] > B[i_b]) {
            // the current value in list B is smaller
            i_b++;
        } else {
            // the current value in list A is smaller
            i_a++;
        }
    }
    // return the number of values values that occur in both lists
    return count;
}
```

Figure 3.2.: Code snippet for a list intersection implementation with branches

Parallel list intersection

Despite of its simplicity, the merge algorithm is difficult to parallelize. In general, parallelism could be exploited for speeding up (1) a single intersection or (2) multiple intersections. The former increases the “intersection throughput” indirectly—i.e., if the time of a single intersection is reduced, more intersections can be issued faster one after another—whereas the latter increases it directly.

Exploiting parallelism to speed up a single intersection is difficult because the merge intersection algorithm has many data dependencies. First, there are dependencies within each iteration. The variables i_a and i_b are increased depending on the result of the comparisons. Second, there are flow dependencies: all variables in each iteration depend on the variables of the previous iteration, i.e., the indices of two values being compared depend on the comparison of the previous iteration. Furthermore, two branches are required that are—depending on the intersection’s selectivity3—hard to predict. For both, a low or a high selectivity, the branch prediction works well, however, in all other cases there are many branch mispredictions.

A way to avoid the branch mispredictions and thus increasing ILP is to eliminate both branches within the main loop using predication (cf. Chapter 2 on page 38). Recall that this increases the amount of issued instructions but control dependencies are removed so that the

---
3The selectivity denotes the fraction of the number of common values of both sets and cardinality of the smaller set.
3.1. Background and Related Work

```c
int intersect(short *A, short *B, int l_a, int l_b, short* C) {
    // initialize the variables
    int count = 0, i_a = 0, i_b = 0;

    // repeat the loop body until the end of an input tid-list is reached
    while(i_a < l_a && i_b < l_b) {
        // store the current value of each input list in a temporary variable
        short tmpA = A[i_a];
        short tmpB = B[i_b];
        // overwrite the last element of the output list
        C[count] = tmpA;
        // increase the indices using predication
        count += (tmpA == tmpB);
        i_a += tmpA <= tmpB;
        i_b += tmpA >= tmpB;
    }
    // return the number of values that occur in both lists
    return count;
}
```

**Figure 3.3.** Code snippet for a *branchless* list intersection implementation

The C-code for a *branchless* implementation of the merge algorithm for 16-bit values (*short*) is shown in Figure 3.3. Each value of A is written into the result set at position count. However, only if the current values of both lists are equal, then the count value is increased by one. In all other cases, the value in the result set is eventually overwritten by the next value of A. The indices are altered by adding the result of certain boolean expressions. The variable i_a, for example, is increased by one whenever the expression tmpA <= tmpB evaluates to “true”.

DLP cannot be exploited to speed up a single intersection of the merge algorithm because of the data dependencies in its main loop. In each iteration only two values need to be compared; all subsequent comparisons depend on the outcome of the current comparison. Hence, a completely different algorithm is required for efficiently exploiting DLP. We propose such an algorithm in Section 3.2.

TLP can be exploited by dividing both lists into partitions that can be intersected independently. However, it is always more efficient to use TLP to speed up multiple intersections because less communication is required and load balancing is easier. Only if not sufficient “intersections” are available, multiple threads could process large intersections together.

For speeding up *multiple intersections* only ILP and TLP can be exploited. On processors that support simultaneous multi-threading, ILP is automatically utilized if two or more intersections are processed on the same core. Clearly, this works only if the core’s pipelines are not already fully utilized by a single intersection. Exploiting TLP is trivial because the problem is already partitioned; each intersection can be processed independently.

Despite well partitioned independent tasks that are provided by intersecting multiple lists in parallel, DLP cannot be exploited. Even though all scalar comparison, arithmetic, and load/store instructions could be exchanged with SIMD counterparts, eventually scatter and gather instructions would be necessary for becoming efficient. The reason for that is the access pattern of the list intersection algorithm. In each step, the algorithm requires to load one value
3. Sorted-set intersection

from each list; for \(k\) parallel intersections this implies loading \(k\) values from the \(k\) upper lists and loading \(k\) values from the \(k\) lower lists. Since as mentioned before, increasing of the list’s indices for a single intersection depends on the values being intersected, the indices of the \(k\) upper lists as well as the indices of the \(k\) lower lists are increased independently of each other. For this reason, it is not possible to load the \(k\) values from \(k\) lists from a continuous chunk of memory using the available SIMD load instructions. Without scatter and gather instructions, the only way is to fill up the vector registers with \(2 \cdot k\) load instructions—introducing a high sequential fraction to the algorithm.

To summarize, the list intersection algorithm can exploit ILP and TLP for both, a single intersection and multiple intersections, but DLP cannot be utilized for neither one of them. In Section 3.2 we propose a merge-based list intersection algorithm that is based on speculative comparisons and efficiently exploits DLP.

3.1.3. Bitmap intersection

The intersection of tid-bitmaps forms a fast alternative to the list intersection algorithm under certain conditions. The basic idea is to transform the tid-lists into tid-bitmaps; the \(i\)-th bit in a tid-bitmap is set to “1” if the item \(i\) is part of the corresponding tid-list. For example, if a tid-list consists of the items 1, 3, and 9, then only the first, third, and ninth bit in the corresponding tid-bitmap are set to “1” whereas all other bits are set to “0”. The length of a tid-bitmap thereby depends on the domain of the values being encoded. For example, if the domain consists of 8,000 values, then a tid-bitmap requires 1,000 bytes regardless of the number of values in the represented set.

Two bitmaps can be intersected with only bitwise AND instructions when they have the same size and all bits at the same positions represent the same items. Figure 3.4 illustrates the intersection of two bitmaps, A and B. The bitmap C—as the result of the intersection—has only the bits set to “1” that are also set to “1” in both input bitmaps. Notice that the bitmaps A and B represent the same lists that were used in Figure 3.1.

![Figure 3.4: Intersection of bitmaps consisting of 40 bits (set bits are gray shaded).](image)

Obtaining the number of common values can be done via counting the number of “1” bits in the result bitmask. Without appropriate hardware support, this task is performed using either lookup-tables [Burdick et al., 2001] or multiple bit-arithmetic instructions [Wei et al., 2007]. With hardware support, i.e., an available POPCNT instruction\(^4\), both approaches render meaningless because an instruction that is directly realized in hardware is always faster than an emulated instruction consisting of multiple instructions. In our algorithms, we therefore use the POPCNT instruction, which counts the number of “1” bits within either 4 or 8 bytes.

\(^4\)The POPCNT instruction is available in Intel and AMD processors beginning with the Nehalem and Barcelona architecture, respectively.
3.1. Background and Related Work

```c
int intersect(long long *A, long long *B, int words, long long *C) {
    // initialize the variables
    unsigned count = 0;

    // repeat the loop body until all words are intersected
    for(int i = 0 ; i < words ; i++) {
        // intersect two words using bitwise AND and write back the result word
        c[i] = a[i] & b[i];
        // count the number of "1" bits that are within the result word
        count += _mm_popcnt_u64( c[i] );
    }
    // return the number of values that occur in both input tid-bitmaps
    return count;
}
```

**Figure 3.5.:** Code snippet for a bitmap intersection implementation

Figure 3.5 shows the C-code for an implementation of the tid-bitmask intersection algorithm. The two bitmasks A and B are stored in an array of 64-bit integer values (*long long*). Both arrays hold the same number of integer values, which is stored in *words*. Each of these integer values holds 64 tids. The loop body is repeated until all integer values are intersected and consists of only three instructions. The actual intersection is done via a bitwise AND. The number of common values is obtained using the *_mm_popcnt_u64* instruction, which works for 64-bit integers. We also implemented a 32-bit version of this algorithm using the *_mm_popcnt_u32* instruction but it was clearly always slower than the 64-bit version.

Compared to the list intersection, the bitmap intersection has several advantages. First of all, the time required for intersecting two tid-bitmaps does not depend on the respective sets’ cardinality, i.e., the number of “1” bits. It only depends on the length of the bitmaps. Second, the bitmap intersection requires only few instructions, which aside leads to high memory bandwidth requirements since only few instructions are issued per loaded word. Finally, a tid-bitmap has lower memory requirements than a tid-list as long as the domain is small and the cardinality is rather high. As mentioned before, a tid-bitmap requires 1000 bytes for a domain with 8000 values. Integer values in a tid-list could be encoded using two bytes for this domain. Hence, if a set’s cardinality is larger than 500, then the bitmap representation is more memory efficient. Nevertheless, tid-lists can also be compressed to reduce their memory requirements—thus shifting this border to a higher number of values.

There exists also a wide variety of bitmap compression techniques [Witten et al., 1999], which could be used to reduce the size of the bitmaps. Almost all of these techniques transform the bitmaps into lists and compress them using static codes; the lists itself consist of integer values that encode the distance to the next “1” bit. For sparse bitmaps these compression techniques work very well but—as also mentioned by Witten et al. [1999]—there is no difference between a compressed bitmap and a compressed list anymore. Thus, it is more efficient to decompress the list and perform a list intersection instead of transforming it back to the bitmap representation and do a bitmap intersection. Therefore, the main advantages of bitmaps disappear if they must be compressed.

Besides the pure list and pure bitmap representation there exist also hybrids. A simple approach is to represent all bitmap words that have at least a single “1” bit as bitmaps while
3. Sorted-set intersection

Gaps of “zero” words in between are represented as integer values; these values encode how many “zero” words are omitted. Wu et al. [2001] proposes and evaluates several of such hybrid representations. Similar to compressed lists, they require less memory than pure bitmaps if there are long runs of “zero” bits. Their achieved compression ratio, however, is usually smaller than for regular list(bitmap compression techniques because if only a single bit set to “1” in a word, then the compression does not work. Nevertheless, such representations could be improved if they consider the characteristics of frequent itemset mining, i.e., the frequencies of the items are known and the items can be arbitrary mapped to tids. For this reason, we present a hybrid representation in Chapter 6 (starting on page 135) that is tailor-made for frequent-itemset mining and is based on pure list and bitmap intersection. Hence, we do not further consider hybrid representations in this section.

To summarize, the main loop of the bitmap intersection algorithm consists of only few instructions. Most of them are load and store instructions so that the algorithm has a low arithmetic intensity and thus requires a high memory bandwidth. Furthermore, the bitmap representation works only well, if the fraction between the sets’ cardinality and the respective domain of the values that are within the sets is not to small. Otherwise, the list intersection algorithms are usually superior, which is also indicated by our experiments in Section 3.3.2.

Parallel bitmap intersection

One of the main advantages of the bitmap intersection algorithm is that it is easy to parallelize. ILP can be fully exploited because there are no loop dependencies in the main loop. For the same reason, the iterations of a single intersection could be split among multiple threads—although this is not required in our application area. Only DLP cannot be fully exploited because there is no SIMD population count instruction available in current x86-processors.\(^5\) The main problem of the bitmap intersection, however, is the high memory bandwidth that is already required by a single thread. All threads of a processor use the same connection to main memory and depending on their number the bandwidth may not be sufficient to “feed” all threads with data. Thus, the speedup of the multi-threaded bitmap intersection is usually limited by the available memory bandwidth of the system.

3.2. List intersection using speculative comparisons

As discussed earlier, DLP cannot be exploited for multiple intersections in parallel. For this reason, we exploit it to speed up the execution of a single list intersection. In what follows, we discuss the basic idea of speculative comparisons and propose an intersection algorithm for lists that consist of 8-bit or 16-bit integer values. Thereafter, we propose a hierarchical algorithm that is based on this speculative list intersection algorithm and can process lists with integers of arbitrary precision.

3.2.1. Basic intersection

The parallel merge algorithm is based on the speculative execution of comparisons. While the scalar merge compares in each iteration only the two values indexed by \(i_a\) and \(i_b\) of both input lists, the main idea of the parallel merge is to compare also values that are beyond

\(^5\)Currently, only the Cell processor [Handbook, 2008] implements such an instruction. It allows to count the number of bits within a byte for sixteen bytes in parallel.
3.2. List intersection using speculative comparisons

the current indexed values. Usually, such an approach would not be beneficial because of the additional overhead for the large percentage of unnecessary comparisons. The STTNI instructions (cf. Section 2.5.2 on page 41), however, allow a large number of comparisons in almost the same amount of cycles that a single comparison requires such that the many unnecessary comparisons are almost for free and payoff with less iterations. Figure 3.6 illustrates such a STTNI comparison using the `PCMPESTRM` instruction on two vectors \( v_a \) and \( v_b \)—each with eight 16-bit values. An intermediate result matrix contains 64 cells; it is aggregated using the “equal any” method (i.e., combining all bits in a column using logical OR) to obtain a single bitmap with 8 bits. This bitmap indicates that the first, seventh, and eighth value of vector \( A \) are also found in vector \( B \).

The STTNI instructions are restricted to 8-bit and 16-bit integer values so that the basic parallel merge can only process integer values of such a precision. In what follows, we focus on the parallel merge of two sets with 16-bit values; the version for 8-bit values works similarly.

As for the scalar version, the input of the parallel algorithm are two sorted arrays \( A \) and \( B \) with lengths \( l_a \) and \( l_b \), respectively. Both arrays are 16-byte aligned and, for the ease of explanation, we assume that the length of each vector is a multiple of 8. Furthermore, two variables \( i_a = 0 \) and \( i_b = 0 \) are required to indicate already processed values in the arrays \( A \) and \( B \), respectively. The output consisting of the common values of \( A \) and \( B \) is written back to an array \( C \) and the number of common values is stored in \( \text{count} \). The major steps of the algorithm are: (1) load values of both lists into the vector registers, (2) perform a full-comparison of both vectors, and (3) write back and count the common values.

The full algorithm works as follows: The main loop is repeated as long as there are unprocessed values in both input sets, i.e., \( i_a < l_a \) and \( i_b < l_b \).

1. **Load both vectors:** Eight consecutive values of the array \( A \) starting at position \( i_a \) are loaded into the vector register \( v_a \). Similarly, the vector register \( v_b \) is filled with eight consecutive values of \( B \) starting at \( i_b \). Both loads are performed using two aligned load instructions.

2. **Fully compare both vectors:** We perform the full comparison of \( v_a \) and \( v_b \) using the `PCMPESTRM` instruction; the 8-bit result bitmask is stored in a variable \( r \). Each one bit in \( r \) indicates a common value in \( v_a \) and \( v_b \), i.e., a bit at position \( n \) in \( r \) is set to one if there is a value in \( v_b \) equal to the \( n \)-th value in \( v_a \).

A scalar comparison of the respective last values of \( v_a \) and \( v_b \) is used to select the values of \( A \) and \( B \) for the next full comparison. If the last value of \( v_a \) is smaller than the last
int intersect(short *A, short *B, int l_a, int l_b, short* C) {
    // initialize the variables
    int count = 0, i_a = 0, i_b = 0;

    while(i_a < l_a && i_b < l_b) {
        // load 8 values from each input list into a vector variable
        __m128i v_a = _mm_load_si128((__m128i*)&A[i_a]);
        __m128i v_b = _mm_load_si128((__m128i*)&B[i_b]);

        // perform the full comparison of both vectors
        __m128i res_v = _mm_cmpestrm(v_b, 8, v_a, 8,
            _SIDD_UWORD_OPS|_SIDD_CMP_EQUAL_ANY|_SIDD_BIT_MASK);
        // store the result bitmask in a temporary variable
        int r = _mm_extract_epi32(res_v, 0);
        // store the 8-th value of each vector in a temporary variable
        unsigned short a7 = _mm_extract_epi32(v_a, 7);
        unsigned short b7 = _mm_extract_epi32(v_b, 7);
        // increase the indices using predication by 8 values
        i_a += (a7 <= b7) * 8;
        i_b += (a7 >= b7) * 8;

        // shuffle the common values to the front of a temporary vector
        __m128i p = _mm_shuffle_epi8(v_a, sh_mask[r]);
        // write back the "shuffled" common values into the output list
        _mm_storeu_si128(&C[count], p);
        // count the number of "1" bits in the bitmask
        // and increase the index of the output list
        count += _mm_popcnt_u32(r);
    }
    // return the number of common values
    return count;
}

Figure 3.7.: Code snippet for list intersection using speculative comparisons and SIMD instructions.

value of \(v_b\), then \(i_a\) is increased by 8. Otherwise, \(v_b\) is increased by 8. In case of the last values of \(v_a\) and \(v_b\) are equal, both values \(i_a\) and \(i_b\) are increased by 8.

3. **Write back common values:** Finally, the common values stored in \(v_a\) are collapsed and stored in \(C\). Collapsing is performed using the byte permutation (PUSHUFB) instruction; it shuffles all common values to the front of a temporary variable \(p\). The required permutation mask is obtained from a lookup table that contains permutation masks for all 256 combinations of common values in \(v_a\). The bitmask \(r\) is used as an index to select the mask. Finally, an unaligned store instruction is used to write back the packed common values stored in \(p\). The number of common values is calculated using a population count instruction on the bitmask \(r\).
3.2. List intersection using speculative comparisons

The C-code of the algorithm is shown in Figure 3.7. After initializing the indice variables, the loop body is repeated until a list indice equals the length of the respective list. In each iteration of the main loop, 8 values of each list are loaded into vector variables using two SIMD load instructions. The full comparison is performed using the PCMPESTRM instruction that has five parameters: the two vectors that should be compared, the length of each vector, and a flag variable for controlling the output behavior. The flags _SIDD_UWORD_OPS, _SIDD_CMP_EQUAL_ANY, and _SIDD_BIT_MASK indicate that 16-bit values are used, the “equal-any” aggregation should be applied, and a bitmask should be returned, respectively. Increasing the index variables is done using predication; they are increased by using the result of comparison of the last values of each vector v_a and v_b. After “shuffling” the common values to the front of the vector variable p using PSHUFb and the r-th entry in the sh_mask lookup table, the common values are written back into the output list C. Finally, the variable count holding the number of common values is increased with the numbers of “1”s in the bitmask r.

Figure 3.8.: Parallel merge of two sorted sets with only three full comparisons and three end-pair comparisons. The latter comparisons on the particular last value of each vector (bold) are used to decide which 8 values are compared next.

Figure 3.8 illustrates the parallel merge on the same sorted sets as used before (cf. Figure 3.1). Each set consists of 16 values. The first step is the full comparison of the first eight values of both sets. After this first comparison, the common values 1, 16, and 17 are found and written back to the result set (not shown). The comparison of each set’s 8-th value, 17 for the upper set and 19 for the lower set, indicates that all values smaller or equal than 17 are already merged. However, since the values 18 and 19 from the lower set might occur in the upper set, only the index of the upper set is increased by 8. After the second full comparison, the value 18 is found. The last full comparison (second 8 values of both sets) reveals the common values 26, 32, and 34. After only three full comparisons and three scalar comparisons all common values are found and the intersection is finished.

The parallel merge performs more comparisons than the scalar merge. For the 16-bit parallel merge, the best-case occurs if both sets are identical and the length of both sets is a multiple of 8. The number of comparisons is then given by $8 \cdot l_a$ because after each full comparison the algorithm proceeds in both sets by 8 values. The worst-case number of comparisons is given by $16 \cdot l_a – 64$. It occurs when both sets have the same cardinality and the last values of all fully compared vectors are always unequal; thus the algorithm proceeds after a full comparison only in one set by 8 values. Under the unrealistic assumption that the STTNI instructions process the 64 comparisons in the same time like a single scalar comparison instruction, the speedup for the parallel merge is in a range of 4x to 8x. Similar considerations lead to a potential speedup of 8x to 16x for the parallel merge of 8-bit values.

The code snippet has been adapted for better readability and differs from the source code used for the experiments.
Finally, the parallel merge algorithm is the foundation of a parallel set difference algorithm, which could be employed in \textsc{Eclat} when the \textsc{difset} optimization [Zaki and Gouda, 2003] is enabled. Recall that the difference set \( \{ A - B \} \) contains only the values of \( A \) that do not occur in \( B \). The sequential difference algorithm is basically the same as the sequential intersection algorithm. Instead of writing back the \( A \) set’s values in the \( (A[i_a] == B[i_b]) \) branch, however, they are written back in the \( (A[i_a] < B[i_b]) \) branch. Furthermore, possibly remaining values of \( A \) are written back when the main loop is left. The parallel difference algorithm requires similar changes on the write back operation. An additional bitmask is used to indicate all values that do not occur in the set \( B \) for the current vector \( v_a \). After a full comparison, a bit in this bitmap is set to “zero” if the corresponding value in \( v_a \) occurs in \( v_b \). The bitmap itself is used whenever the index \( i_a \) is incremented. All values not found in \( v_a \) are written back and all bits in the bitmap are reset to “1”. More details for the parallel difference algorithm are available in Appendix B.

### 3.2.2. Hierarchical intersection

So far, we presented a parallel sorted-set intersection algorithm for 8-bit and 16-bit integer values. Since the STTNI instructions can only be applied on such integer values, we propose a hierarchical intersection approach that allows us to process integers of higher precision (> 16-bit). The basic idea is to partition all integers of a given domain \( D \) based on their \( h \) upper bits where \( h + 16 \) is the precision of the integer; e.g., \( h = 16 \) for 32-bit integer values. A set \( A \) is thus partitioned into disjoint subsets \( A_1, A_2, \ldots, A_m \) where each subset \( A_i \) consists of integer values that share the same upper \( h \) bits. The set \( B \) is partitioned into \( B_1, B_2, \ldots, B_n \) in the same way. The hierarchical intersection works then as follows:

**Upper-level intersection** is done by merging the subsets \( A_1, A_2, \ldots, A_m \) and \( B_1, B_2, \ldots, B_n \) based on the shared \( h \) bits of each subset. For each pair \( A_i \) and \( B_j \) that share the same \( h \) bits, the lower-level intersection is performed.

**Lower-level intersection** merges two subsets \( A_i \) and \( B_j \) based on the lower 16-bits of each value. This is done using our parallel 16-bit merge algorithm.

To summarize, we have a single upper-level intersection and multiple lower-level intersections. The upper-level intersection could be performed using the scalar merge (cf. Section 3.1.2) or also using an intersection based on the STTNI instructions. We observed, however, that in most cases the upper-level intersection is responsible only for a small fraction of the execution time of the complete intersection process. In general, the fraction of upper-level and lower-level execution time depends on the average cardinality of the subsets. Roughly speaking, the higher the cardinality of the subsets, the more of the execution time is spend for the intersection of the subsets. For this reason, we use a scalar merge for the upper-level intersection and our parallel 16-bit merge algorithm for the lower-level intersections.

We provide two different hierarchical parallel merge algorithms. The first algorithm processes uncompressed integer values and requires a pre- and post-processing to partition these values into subsets during processing. The second algorithm uses a tailor-made data structure to avoid the overhead of such processing during the intersection. We will explain both algorithms for 32-bit values only; the steps are similar for integers of higher precision.

In the first algorithm, we execute pre- and post-processing during the intersection to avoid large intermediate results that would be necessary if the phases are separated from the complete intersection process. Instead, we only maintain two buffers each with 65.536 elements.
3.2. List intersection using speculative comparisons

Before each lower-level intersection, we copy the lower 16-bit values of both subsets being intersected into the buffers. This copying can be performed for each 16 values of a subset using only four instructions: Two PSHUFB instructions are used to twice move the lower 16-bit of 8 values to the lower part of a SIMD register; a PUNPCKHW instruction combines the lower parts of two SIMD registers; and a single store writes the collapsed values in the buffer. After all values of both subsets are copied into their assigned buffers, the pre-processing for this lower-level intersection is finished and the actual parallel 16-bit intersection is performed.\(^7\)

The separation of the complete copying of the 16-bit values and their intersection afterwards avoids complex code that would require many branches and would probably be slow. However, post-processing is executed without buffering. It is used to concatenate the upper 16 bits with each 16-bit value of the lower-level intersection result. We exploit the instructions PUNPCKHW and PUNPCKLWD for this concatenation. The upper 16-bit are stored replicated in the first register while the second register contains the intersection results. After each intersection iteration, we execute both conversion instructions followed by two unaligned store instructions.

The second algorithm avoids the pre- and post-processing overhead by using a tailor-made data layout for the input sets. The main idea is to store the values of a set already partitioned using two levels; all data of a set is stored in a continuous memory area representing an array of 16-bit values. The subsets \(A_1, A_2, \ldots, A_m\) of \(A\) are then stored as follows: starting at the first element of the array, the shared 16 bits of the values in \(A_1\) are stored once using a single cell. The next cell contains the cardinality of \(A_1\). A single cell for the cardinality is sufficient since the maximum number of values in a subset never exceeds 65,536.\(^8\) The next \(|A_1|\) cells comprise the lower 16-bit of the values of \(A_1\). All following subsets \(A_2\) to \(A_m\) are stored in a similar fashion. Figure 3.9 illustrates the data layout for 8 values that are partitioned in three subsets. The subsets contain 3, 3, and 2 values and the values in them share the prefix 0x0000, 0x0001, and 0x0002 (in that order).

The two-level data layout has often lower memory requirements than the original representation of the uncompressed integer values it consists of, because the upper bits of all values of a subset are stored only once. Only in the worst-case—each subset contains only one element—we have additional overhead of 16 bit for the subset’s length field per integer value. However, this case is very unusual in our application area because ECLAT uses a sequential numbering

\(^7\)Each 8 consecutive values in the buffers are not ordered anymore. The full comparison still works but we have to adapt the shuffle control masks that are required for packing the result in each iteration of the intersection.

\(^8\)We store the length minus one since 65,535 is the largest possible 16-bit integer value and an empty subset is never stored.
3. Sorted-set intersection

for the tids. Furthermore, already for an average cardinality of 2, the memory requirements are the same as for storing uncompressed 32-bit values (cf. the example in Figure 3.9). For higher average cardinalities, a set’s memory requirements approach 16 bit per value.

Conversion of the two-level data layout into 32-bit values can be performed with almost no overhead during the intersection using the post-processing of the first hierarchical algorithm. As indicated by experiments, there is only a small difference in the execution times whether the intersection result is written back in the two-level data layout or as uncompressed 32-bit integer values.

3.3. Experiments

In this section, we discuss the outcome of our performance evaluation of the set intersection algorithms discussed in this chapter. We first give an overview of the experimental setup before presenting the results of our experiments.

3.3.1. Setup

We implemented several intersection algorithms using C as programming language; this includes the three algorithms of Section 3.1 and the two speculative intersection algorithms of Section 3.2. The first two algorithms BRANCH and BRANCHLESS intersect lists of uncompressed integers; BRANCH uses two branches in the main loop for increasing the indices while BRANCHLESS uses predication and has no branches in the main loop. Both algorithms were instrumented with C preprocessor directives to obtain versions tailor-made for different integer precisions, e.g., BRANCH8 and BRANCHLESS8 process 8-bit integer values. The BITMAP algorithm is used for intersecting uncompressed bitmaps while the algorithms SPEC8, SPEC16, and SPEC32 were based on speculative comparisons and differ only in the precision of the integers they process. Finally, SPEC32-H is similar to SPEC32 but intersects lists that were represented using the proposed hierarchical data layout. The speculative comparison algorithms utilize the STTNI instructions, which were inserted using intrinsics. To ensure fair comparison, all implementations are hand-tuned.

We further implemented a framework for evaluating the set intersection algorithms. It generates synthetic datasets for a given cardinality, domain, selectivity, and number of sets. Before each run within an experiment, the respective lists were generated from scratch; this generation is performed in parallel to reduce the time required for running the experiments. We then measured the wall-clock time—using gettimeofday—for the intersection process only. After each run, the intersection result set is verified using a result set that is produced by the BRANCH algorithm. Our framework allows us to execute multiple intersections in parallel utilizing TLP. Both, the dataset generation and the intersection process itself, were parallelized using OpenMP.

We conducted experiments to evaluate the influence of selectivity, domain coverage, and number of threads to the intersection algorithms. For the selectivity experiments, we intersect in each run in total 10 million values with another 10 million values and vary the selectivity from 0% to 100%. Since the domain of 8-bit and 16-bit integers is restricted to 256 and 65,536 values, respectively, we executed multiple intersections while evaluating the algorithms for this precision: we perform 156,250 intersections of sets each with 64 elements for the 8-bit versions and 1000 intersections of sets each with 10,000 elements for the 16-bit versions. The domain is restricted to 256, 40,000, and 40,000,000 items for evaluating the intersection of 8-bit, 16-bit,
3.3. Experiments

<table>
<thead>
<tr>
<th>intersections</th>
<th>sets</th>
<th>values per set</th>
<th>domain</th>
<th>selectivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selectivity (8-bit)</td>
<td>156,250</td>
<td>312,500</td>
<td>64</td>
<td>256</td>
</tr>
<tr>
<td>Selectivity (16-bit)</td>
<td>1,000</td>
<td>2,000</td>
<td>10,000</td>
<td>40,000</td>
</tr>
<tr>
<td>Selectivity (32-bit)</td>
<td>1</td>
<td>2</td>
<td>10,000,000</td>
<td>40,000,000</td>
</tr>
<tr>
<td>Domain coverage</td>
<td>60/3,600</td>
<td>120</td>
<td>32,768–524,288</td>
<td>4,000,000</td>
</tr>
<tr>
<td>Scalability</td>
<td>60/3,600</td>
<td>120</td>
<td>1,000,000</td>
<td>4,000,000</td>
</tr>
</tbody>
</table>

Table 3.1.: Setup overview for the set-intersection experiments

and 32-bit integer values, respectively. Thus, the available domain for each set is four times larger than the number of elements that are within it. In the domain coverage experiments, we only evaluate the 32-bit intersection algorithms. We intersect 60 sets with another 60 sets in two configurations. In the one-to-one setup, we perform 60 intersections where the \(i\)-th set of the first 60 sets is intersected with the \(i\)-th set of the second 60 sets. In the nested-loop setup, we perform 3600 intersections where all combinations of the first 60 sets and the second 60 sets are intersected. We fix the domain to 4,000,000 values, the selectivity to 50%, and varied the size of the sets from 32,768 to 524,288 values. For the scalability experiments, we use the same parameter as used for the domain coverage experiments but fix the size of the sets to 1,000,000 values. Table 3.1 summarizes the parameter for the different experiments.

Hardware and software setup

In this thesis, we use two different systems for evaluating our algorithms. The single-socket system has a high core frequency and memory bandwidth and is well suited for the experiments in this and the next chapter. The system comprises a single Intel Core i7-3960X processor with a core frequency of 3.3GHz. The processor consists of six cores with Hyperthreading enabled, i.e., it can process twelve threads in parallel. It supports all required SIMD instruction sets (up to AVX\(^9\)) and provides a turbo-mode that allows a core frequency of 3.9GHz if an internal thermal budget is not exceeded. We disable this turbo-mode in all experiments to obtain more consistent results. For the same reason, we turn dynamic frequency scaling off; the cores are always running at a core frequency of 3.3GHz. The system has 32GB of main memory consisting of eight 4GB DDR3-1600 modules. The modules support a frequency of up to 1600MHz and have a CAS latency of 7 cycles. The processor’s memory controller has a quad-channel interface and allows for 1600MHz modules a theoretical memory bandwidth of 51.2GB/s. The real memory bandwidth, however, is about 30GB/s. The four-socket system consists of four processors that have a lower core frequency but provide a much higher thread count. The system is used for our experiments in Chapters 5–7 and is explained in more detail in Section 5.4.1 on page 117.

On the single-socket system, we use Linux (Ubuntu 2.6.38-8) as operating system. All algorithms were compiled using Intel Parallel Composer 2011. We also tried the GCC compiler but the generated code was in almost all experiments slower. For code optimization, we used the compiler flag `-fast`. The input datasets are main memory resident as it is also expected

\(^9\)We only make use of the different versions of the SSE instruction sets (up to SSE4.2) because AVX can only process floating point values.
3. Sorted-set intersection

![Diagram](image.png)

(a) 156,250 intersections with 64 values per set (8-bit values)
(b) 1000 intersections with 10,000 values per set (16-bit values)

Figure 3.10.: Varying the selectivity for intersections in small domains

for frequent-itemset mining. The output is written on preallocated memory so that there is no time required for retrieving the memory (e.g., for malloc calls or page mapping).

3.3.2. Results

In the first two experiments, we compare the single-threaded execution time for intersecting 8-bit and 16-bit sets for a varying selectivity. We compared all algorithms except for SPEC32-H because the hierarchical data layout is not required for such integer precisions. The 8-bit and 16-bit list intersection algorithms require—without the metadata—about 20MB and 40MB to represent all sets, respectively. In contrast, 10MB are required to represent the bitmaps used for the BITMAP intersection algorithm.

Figure 3.10a illustrates the results for the 8-bit intersection. Independently of the selectivity, BITMAP outperforms all other algorithms; it has a constant execution time of only 1.2ms. SPEC8 is the fastest of the list intersection algorithms and requires 8ms for performing 156,250 intersections. It proceeds in almost each iteration by 16 elements in one of both sets. When the sets fully overlap (100% selectivity), it iteratively proceeds in both sets by 16 elements, so its runtime decreases to 5.1ms. When comparing both regular list intersection algorithms, BRANCHLESS performs better up to a selectivity of roughly 85%. At this point, BRANCH performs faster because of the nearly perfect branch prediction. Furthermore, BRANCHLESS as well as BRANCH benefit from an increasing selectivity because the index of both sets is increased for each pair found. If only the list intersection algorithms are considered, SPEC8 achieves speedups from 2.9x to 9x compared to the best performing of the two considered other implementations at each measuring point.

We obtain similar results for the 16-bit intersection (see Figure 3.10b). Again, BITMAP is the fastest of all four algorithms and requires only 1.05ms for the 1000 intersections. SPEC16 is about 12x slower than BITMAP but is always faster than BRANCH and BRANCHLESS. The average execution time of SPEC16 is 12.2ms and only 6.8ms for 100% selectivity. Since SPEC16 proceeds in almost each iteration by 8 elements, it is slower than SPEC8. Nevertheless, it is 1.9x up to 5.45x faster than BRANCH and BRANCHLESS (always considering the better runtime of both algorithms). In summary, the speedups of both speculative algorithms differ from the

---

10 BITMAP is within the second experiment 0.15ms faster than in the first experiment because for this configuration less intersections need to be issued.
3.3. Experiments

(a) Varying the selectivity for a single intersection with 10 million values per set

(b) Varying the sets’ cardinality within a fixed domain of 4 million values.

Figure 3.11.: Varying the selectivity and domain coverage for intersections in large domains (32-bit values only)

Theoretical speedups (cf. Section 3.2.1) because of the additional overhead for the result permutation and higher latency of the STTNI instruction compared to a traditional compare instruction.

In the following experiments, we compare the performance of the five algorithms—including spec32-H—for sets consisting of 32-bit integer values. We again vary the selectivity from 0% to 100%. Bitmap still requires 10MB; spec32-H requires 20MB because of the hierarchical layout; and all other algorithms require 40MB for representing the sets. The results, which are shown in Figure 3.11a, are similar to the results of the previous experiments. Bitmap is the fastest algorithm and the speculative algorithms outperform the regular list intersection algorithms. However, branch has the same execution time as spec32 for 100% selectivity. The execution time of the speculative comparison algorithms differs in average by 7ms, because of the pre- and post-processing overhead of spec32. In other words, spec32-H is 1.6x faster than spec32. Furthermore, spec32-H is less than 1ms slower compared to spec16, which indicates that the overhead for the two-level data layout is negligible.

In the last single-threaded experiment, we vary the number of integer values that are within each set from 32,768 to 524,288 values and fix the domain to 4 million values. Hence, the domain is covered from about 0.1% to 10%. Figure 3.11b illustrates the results for the nested-loop setup (note the log scale). The bitmap algorithm has a constant execution time of 340ms because it processes—indepenently of the sets’ cardinalities—for each single intersection two 0.5MB bitmaps and writes an 0.5MB result bitmap back to main memory. The runtime of the lists intersection algorithms, however, decreases as the cardinality of the sets decreases because fewer integers need to be intersected. Both regular list intersection algorithms, branch and branchless, outperform bitmap for sets that cover less than 0.3% of the given domain. Spec32 and spec32-H are faster than bitmap for sets that cover less than 1% and 1.5% of a given domain, respectively. As we will show in Chapter 5, such small domain coverage values are not uncommon when mining realistic datasets with low ξ values so that the list intersection algorithms should be employed in such cases.

Multi-threaded execution

We run several experiments to analyze the multi-threaded performance of the algorithms. In the scalability experiments, the domain is restricted to 4 million 32-bit integer values, the sets
3. Sorted-set intersection

contain 1 million values, and the selectivity is set to 50%. We vary the number of threads from one to twelve and further use the one-to-one and the nested-loop intersection setup with 60 and 3600 intersections, respectively.

Figure 3.12a shows the speedup of the set intersection algorithms for a varying number of threads and the one-to-one setup. The highest speedup is reached by \textsc{branch} and \textsc{branchless}. They scale \textit{perfectly} up to six threads; even for 12 threads with Hyperthreading enabled, they still scale well because the cores’ execution units are not fully utilized. \textsc{spec32-H} and \textsc{spec32} nearly scale perfectly up to six threads but do not benefit of Hyperthreading as much as \textsc{branch} or \textsc{branchless}. They achieve at most a speedup of 7.5x. Nevertheless, \textsc{spec32-H} is single-threaded as fast as \textsc{branch} with 6 threads or \textsc{branchless} with 4 threads. Of all algorithms, \textsc{bitmap} scales worst; it achieves a maximum speedup of 4x with five threads and even slows down for more threads because it early saturates the system’s memory bandwidth.

In the nested-loop intersection setup—see Figure 3.12b—there is less memory pressure because the threads partly access the same sets. For this reason, \textsc{bitmap} scales better compared to the one-to-one setup; it achieves a maximum speedup of about 5.75x for 7 threads but again slows down as the number of threads is further increased. All other algorithms behave as for the one-to-one setup because they all do not saturate the system’s memory bandwidth in both setups. Notice that the better scaling for 7, 8, 9, and 11 threads results from a better load distribution, i.e., 60 intersections cannot be as good partitioned as 3600 intersections among such a number of threads.

Finally, we repeated the domain coverage experiment with multiple threads. We used the one-to-one and the nested-loop setup and measured the algorithms’ performance for 12
threads. For bitmap, we measured also the performance for 6 threads because it performs best for such a number of threads. The results for the one-to-one setup are illustrated in Figure 3.12c. Since the list intersection algorithms scale better than bitmap, the threshold for switching between the list and bitmap representation shifts. For example, spec32-H is already more efficient than bitmap if the sets cover less than 5% of the domain and 12 threads are used. Hence, it is earlier more efficient to switch to the list representation. This effect is reduced for the nested-loop intersection setup (Figure 3.12) but the threshold is still shifted towards the list algorithms compared to the single-threaded execution.

We obtain similar results also on other systems. Bitmap is always limited by the memory bandwidth and therefore does not scale well. Especially for the one-to-one setup, which is more realistic within Eclat-based algorithms because the threads usually process independent tid-sets, bitmap scales only well to a fraction of the available hardware threads. On some systems, we further observed that spec32 and spec32-H saturate the available memory bandwidth for this setup. In such cases, the memory requirements of the tid-set representation determine which intersection algorithm is faster. Hence, for domain coverage values smaller than 6.25%, tid-lists are most space-efficient so that the list intersection algorithms are best-suited. Otherwise, tid-bitmaps are most space-efficient and bitmap is the algorithm of choice.

3.4. Summary and Discussion

In this chapter, we discussed set intersection algorithms that could be used to speed up the support counting step of Eclat-based algorithms. We first reviewed several algorithms that are mainly used for query processing in information retrieval and databases. Within the context of frequent-itemset mining, most of the sets have a similar length so that we restricted our attention to merge-based list and bitmap intersection algorithms. We discussed parallelization opportunities for these algorithms and described highly efficient implementations of them. The bitmap algorithm is efficient for sets that cover large parts of the domain. It already exploits all three kinds of parallelism so that there is hardly room for improvements. The list intersection algorithms, however, do not fully utilize ILP and cannot exploit DLP. For this reason, we proposed merge-based list intersection algorithms that compare more values per iteration than the traditional list intersection algorithms and thus require less iterations than them. Indeed, our algorithms perform more comparisons, i.e., speculative comparisons for which it is previously unclear whether they are useful or not, but these comparisons can be performed in parallel using DLP.

The proposed speculative comparison intersection algorithms utilize the STTNI instructions of SSE4.2, which were originally intended to speed up text processing operations. The algorithms spec8, spec16, and spec32 use sorted sets consisting of uncompressed integer values as input and output; they differ only in the precision (8-bit, 16-bit, and 32-bit) of the integers they process. Because spec32 requires pre- and post-processing of the integer values being intersected, we provide a hierarchical data layout, which avoids such overhead during the intersection process and is used by spec32-H. Within our experiments, we showed that bitmap is the algorithm of choice if the sets cover more than 5% of the given domain. In all other cases, the speculative comparison algorithms are superior over all other algorithms.

In Chapter 6, we will show that the fast intersection of the speculative algorithms translates into a better overall performance of cEclat, which is an frequent-itemset mining algorithm based on Eclat. In cEclat, we only employ spec32-H—and thus spec16 as underlying
3. Sorted-set intersection

algorithm—because the size of the tid-sets often exceeds 65,536 values. We do not employ
SPEC32 because SPEC32-H is always faster than it; SPEC32-H's initial overhead for transform-
ing integer lists into the hierarchical data layout is amortized very fast. Nevertheless, SPEC32
(and also SPEC8) could be employed in other application areas because they are always faster
that the regular list intersection algorithms. Additionally, we propose in Chapter 6 a hy-
brid data layout that combines the advantages of bitmaps and lists; it is tailor-made for the
cECLAT algorithm.
4. Compression

In the previous chapter, our main concern was on directly speeding up frequent-itemset mining algorithms that rely on intersecting sets. In this chapter\(^1\), we restrict our attention to integer compression techniques that can be used (1) to lower the memory footprint of mining algorithms and (2) to reduce the size of the dataset being mined. Reducing the memory footprint of the mining algorithms allows them to process larger datasets with low \(\xi\) values in main memory and thus delays expensive out-of-core processing. Suitable intermediate data structures for compression are thereby, for example, the tid-lists of ECLAT or the FP-trees of FP-GROWTH. Reducing the size of the datasets using compression allows to keep larger datasets in main memory. This is especially useful because all mining algorithms scan a dataset being mined at least twice and furthermore in iterative mining [Cong and Liu, 2002], it is often accessed repeatedly. Hence, expensive loading from disks or remote nodes in a network is required only once even if a dataset is accessed multiple times. Also loading the dataset the first time is faster when it is stored already compressed on disk or it is compressed before sending it over the network.

Independently of the use case, i.e., either the mining algorithms’ intermediate data structures or the datasets are compressed, the employed compression techniques must fulfill certain requirements. In Section 4.1, we discuss such requirements including properties that might be exploited for compression. Based on that, we survey and review existing integer compression techniques. We find static encodings particularly suitable for our application area because they do not require knowledge (e.g., value distribution) about the data being compressed, which is one of the core requirements. For this reason, we review three different static encodings in more detail and discuss their compression potential and parallelization opportunities. We further discuss delta encoding because it increases the effectiveness of static encodings.

Although static encodings usually induce smaller costs for encoding and decoding compared to other integer compression techniques, the costs are still much higher than when working on uncompressed data. Hence, if the compressed data is often accessed (e.g., when a large compressed dataset is read multiple times), then decoding and partially also encoding may account for a large fraction of a mining algorithm’s overall runtime. For this reason, we propose—in Section 4.2—two vectorized static encodings that achieve much higher throughputs for encoding and decoding compared to their scalar counterparts. We describe their in-memory data structures and underlying data layouts, which can also be used for deriving vectorized versions of other static encodings. Furthermore, we propose vectorized delta encoding, which act in concert with the vectorized static encodings.

We evaluate both, compression ratios and performance, of the presented encodings and provide the respective results in Section 4.3. For evaluating the compression ratios, we use several realistic datasets. We transform them into a binary representation, compress them, and measure their space requirements. We observe that the selected encodings are well suited for compressing large mining datasets. Within the performance experiments, we measure

\(^1\)The material in this chapter has been developed jointly with Rainer Gemulla and Wolfgang Lehner. The chapter forms an extended version of prior published work [Schlegel et al., 2010].
4. Compression

the throughput of the sequential and parallel compression algorithms when compressing large synthetic datasets. The parallel encodings achieve throughputs of up to 25 billion values per second on our test system comprising six cores.

4.1. Background and Related Work

In this section, we discuss the application of compression in itemset mining. Based on this discussion, we classify and review integer compression techniques and discuss three of them in more detail. In the remainder of this section, we analyze the parallelization opportunities of these techniques.

4.1.1. Application of compression in itemset mining

There are two possible applications for compression in itemset mining. It can be used to lower the memory footprint of the algorithms or the size of the datasets being mined. In both cases, the main intention is to avoid expensive disk accesses (or even more expensive communication within a network). Hence, there must be a balance between size reduction and the overhead for compression, i.e., compression is meaningless if encoding or decoding is more expensive than disk access.

Reducing the memory footprint of the mining algorithms is done by compressing the instances of their core data structures. For example, the filtered transactions in \textsc{Apriori}, the tid-lists in \textsc{Eclat}, and the FP-trees in \textsc{FP-growth} could be compressed to reduce the memory footprint of the respective algorithm. Independently of the data structure, the creation of and access to instances is always the same. After a single instance (e.g., a tid-list) is created, it is not changed anymore so that it must be compressed only once. During mining, however, the instances are often repeatedly accessed. Hence, decompression is performed often and therefore a high decompression throughput is required.

The compression potential depends mainly on the algorithms’ core data structures and the distribution of the values that are within the instances of them; only the data structures can be influenced because the value distribution depends on the dataset being mined. In \textsc{Apriori} and \textsc{Eclat}—proposed in Chapter 5 and 6—we use already very efficient core data structures so that the gain of compression is low. However, in \textsc{FP-growth}, which is proposed in Chapter 7, we heavily employ compression because its data structures provide a large potential for it.

Reducing the size of the datasets is often similarly important as reducing the mining algorithms’ memory footprint. The datasets often occupy more space than the algorithms that mine them and must be kept in main memory for multiple runs because mining is often performed iteratively to find the optimal mining parameter (e.g., $\xi$ value). In such cases, the repeated loading of a dataset might be prohibitively expensive. Independently of the file format of the datasets, they can be considered as sequences of integer values. A single sequence represents a transaction and contains non-negative values and a length field. Hence, they form a single large integer sequence when they are stored one after another. A requirement for compression algorithms is now to compress and decompress such a large integer sequence as fast as possible. Thereby, compression should be done in a stream-based manner because multiple scans might not be feasible if the sequence is already too large to fit in main memory.

The datasets have further some properties that can be exploited for compression. First of all, a single sequence (i.e., transaction) is not necessarily ascending or descending ordered. The
4.1. Background and Related Work

order of the items is typically given by the application area but, fortunately, it can be changed while the dataset is compressed; i.e., the order of the items within the transactions or even the order of the transactions within a dataset does not influence the results of frequent-itemset mining when mining the dataset.\(^2\) Hence, changing the order can be exploited for compression; for example, sorted sequences are essential for delta encoding (cf. Section 4.1.4). Although there are no restrictions on sorting the items within the transactions or the transactions itself, the overhead for sorting may be the limiting factor. Sorting each transaction can be performed in a stream-based manner and is even for large transactions with thousands of items quite cheap. Sorting the transactions of the full dataset lexicographically, however, is too expensive because external sorting might be necessary for very large datasets.

Summing up, in this thesis, we employ compression only for reducing the size of the FP-trees used in FP-GROWTH and for compressing large mining datasets. As we will see in Chapter 7, we compress and decompress in CFP-GROWTH, which is a memory-efficient variant of FP-GROWTH, only single integer values at a time. Any optimizations employed for speeding up the compression or decompression of sequences of integers are thus not beneficial. Hence, we reuse only the efficient scalar compression techniques that are discussed in this chapter within CFP-GROWTH. Datasets, however, form long sequences of integer values and can be partially sorted. Since datasets are repeatedly accessed, a high decompression performance is key for a good overall performance of the mining algorithms. Furthermore, compression should take place using only a single run over the full dataset.

In what follows, we denote as compression rate as the ratio between the size of the uncompressed and compressed representation. Hence, the higher it is, the less space is required for the compressed representation. The compression ratio is the inverse of the compression rate and denotes to what size an uncompressed representation is compressed. For example, a compression ratio of 20 percent means that the compressed representation requires only 1/5 of the size of the original uncompressed representation.

4.1.2. Classification of integer compression techniques

We divide integer compression techniques into logical and physical compression techniques. Logical compression techniques work independently of the physical representation of the integer values. For example, run-length encoding [Bassiouni, 1985] replaces a sequence of equal integers with only a single identifier and a count value that denotes the length of the sequence. Delta encoding (see Section 4.1.4) is a further representative of a logical compression technique. Contrary, physical compression techniques change the physical representation of integer values. They map machine-word integer values, which have typically a size of 32 or 64 bit in modern processors, to smaller codewords. In general, physical integer compression techniques can be roughly classified as follows:

**Bit packing** The probably simplest physical compression technique is bit packing. The main idea is to represent all integer values of a set with only as many bits as required to represent the largest integer of the set. If \( n \) denotes the maximum integer value, then only \( \lceil \log_2 n \rceil \) bit are required per integer value. For example, the sequence 3, 1, 2, 1, 1 is mapped to the binary codewords 11, 01, 10, 01, 01.

\(^2\)Only for mining techniques like sequence mining [Agrawal and Srikant, 1995], where the order of the transaction items is important, the transaction items must not be sorted.
4. Compression

Bit packing achieves high compression rates as long as the largest integer value is small. Because of its simplicity, it requires only few instructions to compress and decompress a single integer value and is therefore quite fast. Moreover, Willhalm et al. [2009] exploits DLP to facilitate even faster bit unpacking. Nevertheless, the main disadvantage of bit packing is that the largest value must be known in advance.

**Entropy encodings** Codes that consider the data distribution of the integer values being compressed lead to even better compression rates. The main idea is to map *more* frequent values to *smaller* codewords and *less* frequent values to *larger* codewords. For example, a possible minimum redundancy code of Huffman [1952] maps the integer sequence 3, 1, 2, 1, 1 to the binary codewords 11, 0, 10, 0, 0.

The high compression rates come at the cost of a higher overhead for compression and decompression. Many instructions are required to compress and decompress a single integer value. Furthermore, the exact data distribution must be known in advance and must be stable. Otherwise, the compression rates degrade significantly. There are also adaptive coding algorithms [Moffat and Turpin, 2002] than can cope with unstable data distributions but they introduce even more overhead.

**Static encodings** If neither an upper bound nor the exact distribution of the integer values is available, then static codes can be used for compression. These codes map *small* values to *short* codewords and *large* values to *long* codewords. For example, the sequence 3, 1, 2, 1, 1 is represented using an unary code [Elias, 1975] as 001, 1, 01, 1, 1. The value *three* requires three bits while the value *one* is represented by only a single bit.

Static encodings work well if smaller integer are more frequent than larger integer values. Thus, their compression potential can be improved if they are combined with techniques like delta encoding that produce many small integers.

In this thesis, we solely use static encodings because the upper bound in almost all itemset mining datasets is (1) usually high and (2) often first known after the dataset is fully scanned. This limits the applicability of bit packing. Also entropy encodings are not the best choice because they—except for certain adaptive encodings like dynamic huffman codes [Vitter, 1987]—cannot be applied in a stream-based manner, which makes them unsuitable when a dataset should be compressed during loading from disk or network. Furthermore, entropy encodings usually achieve only low bandwidths for compression and decompression.

To support this assumption, we performed a small experiment on our test system (cf. Section 3.3.1 on page 65) where we compressed typical mining datasets using popular linux compression tools like `bzip2` and `gzip`. The `bzip2` tool employs the Burrows-Wheeler block sorting text compression algorithm [Burrows et al., 1994] combined with Huffman coding [Huffman, 1952]. The `gzip` tool employs Lempel-Ziv coding [Ziv and Lempel, 1977], which is a logical compression technique based on dictionaries. Both tools allow to trade compression ratio with performance; encoding and decoding is faster when the fast flag is set whereas the compression ratio is better when the best flag is set.

Table 4.1 illustrates the average single-threaded encoding/decoding bandwidths and compression ratios for `bzip2` and `gzip`. On all tested datasets, `bzip2` achieves in the fast configuration an average encoding bandwidth of only 25.4MB/s (at best 41.3MB/s) and decoding bandwidth is 24.6MB/s (at best 33.2MB/s). Interestingly, the average compression ratio does not decrease when the best flag is set but the encoding bandwidth decreases significantly.
4.1. Background and Related Work

<table>
<thead>
<tr>
<th></th>
<th>encoding (MB/s)</th>
<th>decoding (MB/s)</th>
<th>compression ratio (in percent)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2 (fast)</td>
<td>25.4</td>
<td>24.6</td>
<td>26.2</td>
</tr>
<tr>
<td>bzip2 (best)</td>
<td>9.0</td>
<td>24.6</td>
<td>26.2</td>
</tr>
<tr>
<td>gzip (fast)</td>
<td>55.1</td>
<td>91.1</td>
<td>42.2</td>
</tr>
<tr>
<td>gzip (best)</td>
<td>8.8</td>
<td>114.3</td>
<td>34.6</td>
</tr>
</tbody>
</table>

Table 4.1.: Average encoding/decoding bandwidths and compression ratios for popular Linux compression tools on realistic mining datasets

The gzip tool achieves in the fast configuration higher average bandwidths for encoding and decoding but has worse compression ratios compared to bzip2; the ratios are even still worse in the best configuration where gzip’s average encoding bandwidth strongly decreases. To summarize, although both gzip and bzip2 are highly optimized tools, they achieve only small bandwidths for encoding and decoding because the underlying compression techniques are computational expensive. For this reason, they would have a large impact on the overall performance of the mining algorithms when they are employed for compressing the datasets or the algorithms’ intermediate state.

As we will show in this chapter, static encodings achieve—within our application area—slightly worse compression ratios compared to entropy encodings (i.e., worse than bzip2 but better than gzip), but they achieve encoding and decoding bandwidths with several GB per second even when only a single thread is used. Hence, the overhead for compressing datasets decreases to only a small fraction of the mining algorithms’ overall runtime.

4.1.3. Static Compression Techniques

The main observation exploited by static encodings of small integers is as follows: For all small integers, most of the space is used for storing leading zero bits. For example, consider the binary representation of the number 100 as a 32-bit integer value:

```
00000000 00000000 00000000 01100100.
```

There are 25 leading zero bits and only 7 bits that carry data. If only full bytes are considered, there are 3 leading zero bytes and only 1 byte actually contains data. The basic idea of static encodings is to store—depending on the granularity—only those bits or bytes that carry data.

There exists a wide variety of static encodings. In the following, we review the most important ones that are best suited for compressing the mining algorithms’ data structures or datasets. More specifically, we will explain the compression techniques null suppression, variable byte coding, and gamma encoding in more detail. We furthermore explain delta encoding, which is used to improve the compression potential of integer sequences.

Null Suppression

The fundamental idea behind null suppression [Bassiouni, 1985, Roth and Horn, 1993] is to substitute null values or successive zeros with a description of how many zeros are omitted. There are many variants of null suppression that differ mainly in the granularity and the precision of the integers being compressed. On the one hand, complete words (32-bit) could
4. Compression

be omitted if all four bytes are zero; a single bit is sufficient to tag if the word is removed or not. Reghabati [1981] uses such a form of null suppression; it uses presence bits to tag missing fields in a database record. On the other hand, single zero bytes could be omitted. Clearly, choosing the right granularity and precision depends on the data distribution of the values being compressed.

The null suppression variant that was proposed by Westmann et al. [2000] removes leading zero bytes of an integer value and instead stores the number of eliminated bytes. Given an integer \( x \), we can partition the binary representation of \( x \) into a (potentially empty) sequence of leading zero bytes and a (non-empty) sequence of effective bytes. At least a single effective byte is stored because \( x = 0 \) occurs infrequently. For \( x = 100 \) as above, we have three leading zero bytes and one effective byte. Define the effective byte length of \( x \) as the number \( l(x) = \lceil \log_{256}(x + 1) \rceil \) of its effective bytes. For a 32-bit integer, the number of leading zero bytes is given by \( 4 - l(x) \) and lies in the interval \([0, 3]\). It thus can be encoded using a 2-bit compression mask. For example, the encoded value of 100 is

\[
1101100100,
\]

where \( 11 \) denotes the compression mask and 01100100 denotes the effective byte. The compression ratio of null suppression is quite low when compared to entropy encoding. Integers of effective byte length 1/2/3/4 are stored using 10/18/26/34 bits, respectively. At best, the compression ratio is therefore \( \frac{10}{32} \approx 31\% \). For 16-bit values, the best-case compression ratio decreases to \( \frac{9}{16} \approx 56\% \). Nevertheless, such a decrease triples (32-bit) or almost doubles (16-bit) the available memory bandwidth if compressed instead of uncompressed data is read.

A variation of Westmann’s null suppression variant uses three bits to encode the number of effective bytes. This extends the interval of leading zero bytes to \([0, 4]\) and is thus beneficial when \( x = 0 \) often occurs.

Implementing leading zero byte suppression requires only few instructions for the compression and decompression function. Only logical and shift instructions, the count leading zero instruction (i.e., the `bsr` instruction), and lookups into small tables are required. The compression function basically counts only the number of leading zeros bytes using `bsr` and a bitshift instruction and writes back the compression mask and the non-zero bytes to memory. Thereby always four compression masks are stored in a single byte. The corresponding decompression function exploits blending words to insert the leading zero bytes in the compressed word. For that purpose, a small lookup table with different blend words is maintained. After loading the compressed 32-bit word, it is blended using a bitwise AND and one of the words of the lookup table, which is indexed by the compressed word’s respective compression mask.

Variable byte encoding

Another static encoding is variable byte encoding, which is also known as varint128 or 7-bit encoding. It works by splitting an \( n \)-bit integer into a sequence of \( \lceil n/7 \rceil \) 7-bit blocks. This sequence is stored in \( \lceil n/7 \rceil \) successive bytes, in which the lower 7 bits are used to store a block and the highest bit is a continuation bit, which indicates whether or not a further block follows (i.e., it is set to 0 if the sequence ends). For example, the value 100 is encoded using only a single byte:

\[
01100100.
\]
4.1. Background and Related Work

Compared to Westermann’s leading zero-byte suppression, variable byte encoding achieves better compression ratios—at best $\frac{8}{32} = 25\%$—for small values ($< 128$) and avoids a separate compression mask for storing the number of suppressed zeros. However, variable byte encoding has usually higher decompression costs for large values ($\geq 128$), and a lookup of the length of a compressed value is not possible without decompression. Indeed, compression as well as decompression can be implemented—similarly to null suppression—only using shift and logical instructions and the $\text{bsr}$ instruction, however, branches are required in the main loop. Predication cannot be applied so that only low bandwidths can be achieved if sequences are decompressed (see the experiments in Section 4.3.2).

### Gamma Encodings

Elias gamma encoding [Elias, 1975] also partitions the binary representation of the integer of interest, but the partitioning is based on bits instead of bytes. Denote by $b(x) = \lceil \log_2(x+1) \rceil$ the effective bit length of integer $x > 0$. The compressed code consists of a prefix of $b(x) - 1$ zero bits followed by the sequence of effective bits of $x$. For example, 100 is encoded as

```
000000 1100100.
```

The length of the codeword is $2b(x) - 1$ bits. When compared to null suppression, Elias gamma encoding provides a better compression ratio for small numbers (up to $\frac{1}{32} \approx 3\%$) but a worse compression ratio for large numbers.

The good compression performance for small numbers makes Elias gamma encoding useful for run-length encoding: To reduce the negative impact of short runs, the length of each run can be encoded using Elias gamma encoding. The frequent-itemset mining algorithm vortex [Shenoy et al., 2000] uses this combination to compress bitmaps.

An efficient sequential implementation of the encoding process consists of the following two steps: (1) Determine the number of effective bits and (2) store the compression mask and the sequence of effective bits. Step 1 can be implemented using the $\text{bsr}$ instruction on the value that should be compressed and a subtraction instruction, i.e., the number of effective bits is given by subtracting the number of leading zero bits from 32—assuming a 32-bit $\text{bsr}$ instruction is used. Before storing the effective bits, they must be shifted in the right position. For that purpose, we employ 64-bit shift instructions to avoid multiple 32-bit shift instructions when the effective bits overlap over a 32-bit boundary. Decompression works similar: Load the 64-bit word that contains the compression mask of a value being decompressed, shift the mask, and use the $\text{bsr}$ instruction to obtain the number of effective bits. The value itself is decompressed by loading its effective bits and shifting in zero bits in front of them. Both, the compression and decompression function, can be implemented without any branch in their core loop.

### 4.1.4. Delta Encoding

Delta encoding is a popular technique to increase the compressability of integer sequences. It is not a compression technique itself because it alone does not reduce the space requirements of encoded values. Its main idea is to encode the difference or delta of a value to its predecessor value instead of encoding the value itself. For example, the integer sequence 1, 3, 4, 8, 9 is encoded as 1, 2, 1, 4, 1. Note that the first value is encoded using the distance to zero.

---

3Elias gamma cannot encode 0. If the data contains zeros, one may encode $x + 1$ instead of $x$.  

77
4. Compression

Delta encoding applied on increasing (or sorted) integer sequences usually leads to many small values. Hence, it efficiently increases the compression rates of static encodings, which rely on many small integer values. It is further easy to implement with only few instructions, i.e., encoding and decoding is realized using simple subtraction and addition, respectively.

4.1.5. Parallelization opportunities

Parallelism could be exploited for speeding up (de)compression of (1) a single integer sequence or (2) multiple integer sequences because both enable a partitioning of the task. In fact, it is already implicitly partitioned when multiple integer sequences should be (de)compressed: each sequence could be processed independently. Also, if a single integer sequence should be processed in parallel, the sequence itself could be divided into independent partitions. This works well for our dataset compression application where the goal is to (de)compress the complete sequence as fast as possible. Hence, TLP could be easily exploited; each thread processes independently a single part of the integer sequence.

Exploiting ILP and DLP is more challenging. All so far presented static encodings share similar characteristics regarding their implementation. Most important, all of the algorithms have loop dependencies because the predecessor of a value must be decompressed before the value itself could be decompressed. This is because the position of a codeword is first known after the length and position of the previous codeword are obtained, i.e., if the n-th value of an integer sequence should be decompressed, its n − 1 predecessors must be decompressed first. This also holds true for compressing values.

There are two possible ways to cope with the loop dependencies limiting the opportunities for ILP. Loop splitting (cf. Section 2.5.1 on page 39) could be used to avoid data dependent latencies. Instead of a single part, each thread processes multiple parts of the dataset being decompressed at the same time; the data dependent latencies then disappear since there are enough instructions that have no dependencies between each other and thus can be issued by the out-of-order scheduler in parallel. The second way is to compress and decompress always multiple integer of a single stream at once, i.e., they are processed in batches. This may have the same effect as loop splitting since more independent instructions occur. Besides avoiding the data dependent latencies introduced by the loop dependencies, it is also crucial to avoid control dependencies within the main loop. Except for variable byte encoding, all algorithms can be implemented without any branches so that they achieve a high IPC score. Nevertheless, also variable byte encoding might perform well; the negative impact of the control dependencies (i.e., mispredicted branches) depends on the data being compressed. If there are almost only small values (< 128), then the influence of the branches is negligible.

DLP, however, cannot be exploited for speeding up compression of either a single sequence or multiple sequence. Indeed, the sequences could be divided into independent partitions but scatter and gather instructions would be necessary to load from multiple partitions of uncompressed values and write to multiple partitions of compressed values. The same holds for decompression. In the following section, we will show how to change the algorithms and their underlying data layout for eventually exploiting DLP.

4.2. Vectorized Integer Compression

In this section, we present vectorized versions of null suppression and Elias gamma encoding. We first discuss alternatives for data layout and then describe the algorithms in detail.
4.2. Vectorized Integer Compression

4.2.1. Layout of Compressed Data

The layout of the compressed data plays an important role in our compression techniques. In contrast to uncompressed data, codewords have a variable length and thus require storage of length information. This poses challenges for vectorized decompression (of \( k \) codewords). First, we have to make sure that all \( k \) codewords can be loaded efficiently into a single SIMD register, i.e., with only a single load instead of multiple load instructions. Second, the length information of the \( k \) codewords has to be organized in a way amenable to vectorization.

There are two alternatives for storing codewords: a horizontal data layout and a vertical data layout, see Figure 4.1. For both layouts, we describe how to load \( k \) codewords from memory and distribute them across the elements of a single SIMD register for further processing. In a horizontal data layout, the \( k \) codewords are stored successively in memory. This facilitates loading (and storing) compressed data using a single unaligned load/store instruction, but distributing the \( k \) codewords into the \( k \) elements of the SIMD register requires data permutation. In contrast, in a vertical data layout, each of the \( k \) codewords is stored in a different memory word. Thus, after loading, the data is already distributed across the \( k \) elements of the SIMD register and data permutation is avoided. However, successive codewords may reside in non-successive memory locations, therefore up to \( k \) load/store instructions are required to access all \( k \) compressed codewords.

Parallel decompression requires knowledge of the length information (effective byte length or effective bit length) of all \( k \) codewords. In a horizontal data layout, the length information of \( k \) codewords should be combined and stored before the \( k \) codewords themselves. Otherwise, parallel decompression is not possible because the location of the \((i+1)\)-th codeword is known only after the \( i \)-th codeword has been decompressed. For a vertical data layout, the length information could be stored in front of each codeword. In general, however, it is advantageous to separate length information and codewords.
4. Compression

4.2.2. k-Wise Null Suppression

Recall that in null suppression, a 32-bit integer is compressed by storing both a 2-bit compression mask and the effective bytes of the binary representation of the integer. To parallelize this procedure, we are working on \( k \) integers at once. Suppose that the \( k \) integers are stored in a single SIMD register. Treating the register content as a single sequence of bytes, compression amounts to removing leading zero bytes, while decompression amounts to reinserting leading zero bytes. Our algorithm is based on the observation that both operations can be implemented with the byte permutation instruction (PSHUFB).

Denote by \( z_1, z_2, \ldots, z_k \) the number of leading zero bytes of each of the \( k \) integers, respectively, where \( 0 \leq z_i \leq 3 \) for 32-bit integers and 128-bit SIMD registers. There are four possible values for each \( z_i \) and thus \( 4^k \) possible cases for the location of the leading zero bytes in the entire SIMD register. The removal of these bytes can be seen as a permutation that moves the effective bytes to the front and the leading zero bytes to the end.

To efficiently perform this permutation, we make use of the PSHUFB instruction. Recall that the instruction takes as input two SIMD registers: one containing the bytes to be permuted and one containing a permutation mask. Computation of the permutation mask is expensive, but since there are only \( 4^k \) different permutation masks of interest (256 for \( k = 4 \)), we can precompute and store them in a permutation lookup table. The table is indexed by the combined compression mask, which is formed by concatenating the 2-bit binary representations of the \( z_i \). To determine the total length of the compressed values, we make use of a separate length lookup table, which is indexed in the same way. The compressed data is stored using a horizontal data layout so that a single store instruction suffices for writing the compressed codewords to memory.

The compression process is illustrated in Figure 4.2a for four 32-bit integers (shown in a hexadecimal representation). Effective bytes are shaded gray. Note that the least-significant byte of the second integer is considered as an effective byte—even though it is zero—because at

![Figure 4.2a: Null suppression using the PSHUFB instruction (k = 4)](image-url)

(a) Compression

(b) Decompression

To efficiently perform this permutation, we make use of the PSHUFB instruction. Recall that the instruction takes as input two SIMD registers: one containing the bytes to be permuted and one containing a permutation mask. Computation of the permutation mask is expensive, but since there are only \( 4^k \) different permutation masks of interest (256 for \( k = 4 \)), we can precompute and store them in a permutation lookup table. The table is indexed by the combined compression mask, which is formed by concatenating the 2-bit binary representations of the \( z_i \). To determine the total length of the compressed values, we make use of a separate length lookup table, which is indexed in the same way. The compressed data is stored using a horizontal data layout so that a single store instruction suffices for writing the compressed codewords to memory.

The compression process is illustrated in Figure 4.2a for four 32-bit integers (shown in a hexadecimal representation). Effective bytes are shaded gray. Note that the least-significant byte of the second integer is considered as an effective byte—even though it is zero—because at
most three leading zero bytes are removed. The combined compression mask B7 (hexadecimal) results from concatenating the binary representations of $z_1 = 2$, $z_2 = 3$, $z_3 = 1$, and $z_4 = 3$. The corresponding permutation mask is shown at the top of Figure 4.2a and the result of the permutation is shown below the uncompressed values. Note that permutation of the zero bytes is not necessary because the "PSHUFB" instruction writes zero bytes into the result vector if the corresponding entry in the permutation mask is set to 80 (hexadecimal).

```c
void decode(unsigned char * input, int* output, int length) {
    int input_byte = 0;
    for(int i = 0 ; i < length/4 ; i++) {
        // load the compression mask
        unsigned char compression_mask = input[input_byte];
        // jump over the compression mask byte
        input_byte++;
        // load the vector that contains the compressed values
        __m128i d = _mm_loadu_si128((__m128i*)&input[input_byte]);
        // decompress the vector using the shuffle instruction
        d = _mm_shuffle_epi8(d,decompress_mask[compression_mask]);
        // jump over the compressed values
        input_byte += size_lookuptable[compression_mask];
        // store the decompressed vector
        _mm_store_si128(((__m128i*)output)+i,d);
    }
}
```

**Figure 4.3.:** Code snippet for vector null suppression (decompression only).

The entire compression algorithm consists of the following three steps: (1) use $k$ count-leading-zeros instructions (bsr) on the $k$ uncompressed values, then shift and combine the results to obtain the combined compression mask; (2) perform the corresponding permutation; and (3) store the combined compression mask followed by the compressed integers using two unaligned store instructions. Decompression is performed in a similar fashion using a permutation lookup table that contains permutation masks for reinserting zero bytes. Figure 4.2b illustrates this process for the same four integers that are compressed in Figure 4.2a.

Finally, Figure 4.3 illustrates a code snippet for the decompression function of $k$-wise null suppression. The function starts with initializing the variable `input_byte`, which is used to store the position of the next four compressed values in the compressed integer sequence provided by the variable `input`. The loop body is repeated until all values are decompressed: Within each iteration, the compressed mask and the compressed values are loaded. The values are decompressed using the "PSHUFB" instruction and written back into the output array. Lastly, the variable `input_byte` is increased by the size (i.e., number of bytes) of the four compressed values being decompressed in the current iteration.

### 4.2.3. k-gamma Encoding

Elias gamma requires a more substantial change of the algorithm to make vectorization effective. This is because neither a horizontal nor a vertical data layout can efficiently handle variable-length codewords preceded by their individual length information; see the discussion in Section 4.2.1. To fix this, we modify Elias gamma encoding by using *shared* length infor-
4. Compression

Figure 4.4.: Memory representation of Elias gamma and \( k \)-gamma encoding of four example codewords

The shared codeword length, called shared prefix, can lead to either an increase or decrease of the compression ratio. To see this, set \( k = 2 \) and consider the following two blocks of integers: \((7, 1)\) and \((7, 6)\). The effective bit lengths of the integers are \((3, 1)\) and \((3, 3)\), respectively. Elias gamma encoding uses 6 bits for the first block and 10 bits for the second block. In contrast, 2-gamma encoding uses the same codeword length for each integer within a block; this codeword length is given by the maximum effective bit length of the integers in the block. In our example, both blocks have a codeword length of three bits. The shared prefix (2 bits) needs to be stored only once so that each block is encoded using 8 bits. The relative compression ratio of Elias gamma and \( k \)-gamma encoding thus depends on the data.

The vectorized \( k \)-gamma algorithm is divided into two parts: (1) obtain the shared length information and (2) construct the \( k \) codewords. The implementation of both parts can be done without any branches or loops; only parallel shift, load, and store instructions are necessary. The shared codeword length is determined using a scalar \texttt{bsr} instruction on the result of a logical OR of the \( k \) integers. The codewords are stored in a vertical layout, but the shared prefixes are stored in a separate memory area. This has the advantage that all memory accesses are aligned.

Each shared prefix consists of a (possibly empty) sequence of zero bits followed by an one bit; this bit serves as separator from the next shared prefix. In contrast to Elias gamma, usage of a separator bit allows us to encode 0. There are two different ways to do this. The first way is used by \( k \)-gamma encoding: The length of the shared prefix denotes the maximum number of effective bits of the \( k \) values. When \( k \) 0/1 values are encoded, the shared prefix and the \( k \) codewords together require \( k + 1 \) bits (recall that the 0 value has one effective bit). An alternative way, which we refer to as \( k \)-gamma\(_0 \) encoding, is to let the length of the zero bit sequence of the shared prefix denote the maximum number of effective bits of only the non-zero values. If all values are 0, only the separator bit has to be stored and the zero bit

---

<table>
<thead>
<tr>
<th>Elias gamma encoding</th>
<th>Effective bits</th>
<th>32-bit word</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( v_1 ) ( v_2 ) ( v_3 ) ( v_4 )</td>
<td>( 0x00 ) ( 0x04 ) ( 0x08 ) ( 0x08 ) ( 0x10 )</td>
<td>( 0x0B ) ( 0x08 ) ( 0x10 ) ( 0x0B ) ( 0x08 ) ( 0x10 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1-gamma encoding</th>
<th>Effective bits</th>
<th>Shared prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v_1 ) ( v_2 ) ( v_3 ) ( v_4 )</td>
<td>( 0x00 ) ( 0x04 ) ( 0x08 ) ( 0x08 ) ( 0x10 )</td>
<td>( 0x0B ) ( 0x08 ) ( 0x10 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2-gamma encoding</th>
<th>Effective bits</th>
<th>Shared prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v_1 ) ( v_2 ) ( v_3 ) ( v_4 )</td>
<td>( 0x00 ) ( 0x04 ) ( 0x08 ) ( 0x08 ) ( 0x10 )</td>
<td>( 0x0B ) ( 0x08 ) ( 0x10 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4-gamma encoding</th>
<th>Effective bits</th>
<th>Shared prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v_1 ) ( v_2 ) ( v_3 ) ( v_4 )</td>
<td>( 0x00 ) ( 0x04 ) ( 0x08 ) ( 0x08 ) ( 0x10 )</td>
<td>( 0x0B ) ( 0x08 ) ( 0x10 )</td>
</tr>
</tbody>
</table>
4.2. Vectorized Integer Compression

sequence of the shared prefix has length zero. Example codewords for \( k \)-gamma as well as \( k \)-gamma\(_0 \) encoding can be found in the Appendix A.3 on page 191.

Figure 4.4 illustrates the memory layout of an example with 4 codewords for Elias gamma, 1-gamma, 2-gamma, and 4-gamma encoding. Elias gamma encoding—shown at the top of Figure 4.4—stores prefix and value of each codeword together. \( k \)-gamma encoding stores shared prefix and the values separately. While each value has its own prefix for 1-gamma encoding, two and four values share a prefix when 2-gamma and 4-gamma encoding is used, respectively. Furthermore, each of the \( k \) values of a block starts at the same relative bit address in their memory word (e.g., \( v_3 \) and \( v_4 \) in words 0x00 and 0x04). In our actual implementation of \( k \)-gamma encoding, we split the memory into smaller chunks. The codewords grow forward from the start of each chunk and the shared prefixes grow backwards from the end of the chunk.

4.2.4. Delta encoding

As mentioned before, delta encoding is used for increasing the compressibility of static encodings. Thus, it must be considered when it is combined with our vectorized compression techniques to minimize the sequential fraction within the complete compression algorithm. As before, vectorization can only be applied if there are no loop dependencies while encoding or decoding the integer values. The encoding step of delta compression has no loop dependencies; to encode a value \( x \), only its predecessor must be subtracted from \( x \). This can be done independently for all values in an integer sequence. Hence, it can be vectorized. The decoding step, however, has loop dependencies. Before decoding an integer value \( x \), all of its predecessors must be decoded, i.e., to decode the \( n \)-th value in an integer sequence its \( n \) predecessors need to be summed up.

The encoding step of delta encoding is vectorized as follows. One vector register is filled with \( k \) values starting at the \( i \)-th value of the integer sequence; another register is also filled with \( k \) values but starting at the \( (i-1) \)-th value of the sequence.\(^4\) The actual delta values are obtained after subtracting the second register from the first one. Thus, two load instructions and a single arithmetic instruction are required to encode \( k \) values. Decoding is performed in a sequential manner because of the loop dependencies; it thus may introduce a high sequential fraction when its executed during the decompression step of a vectorized compression algorithm. Nevertheless, the main benefit of vectorized delta encoding is that its output sequence is the same as for scalar delta encoding.

One way to avoid the sequential decoding step is to reorder the values within the integer sequence. Whenever the order within a sequence is not important, multiple-streams delta encoding can be applied to facilitate a vectorized decoding. The main idea is to divide the original integer sequence into \( k \) equal-sized partitions and to merge the values of the partitions so that for each value \( i \) within the sequence, the \( k \)-th successor contains the successor of the value \( i \) from the original sequence. For example, consider the six values 1, 3, 4, 6, 8, 9. For \( k = 2 \), the merged sequence would be 1, 6, 3, 8, 4, 9. The delta values can be obtained by subtracting the \( (i-k) \)-th value from the \( i \)-th value; this can be done independently for \( k \) succeeding values and thus allows vectorized encoding and decoding. Only the initial reordering before the encoding step requires sequential load instructions as long as scatter and gather instructions are not available. If the sequence is more frequently be decompressed, this

\(^4\)The 0-th value is set to zero.
4. Compression

```c
void decode(__m128i * input, __m128i * output, int length) {
    // create a permutation mask for the replication a vector's last value
    __m128i perm_last = _mm_set1_epi8(0x0C0D0E0F);
    // initialize the shared base with zero
    __m128i shared_base = _mm_set1_epi32(0);
    // iterate over all values of an array with 'length' integer values
    for( int i = 0 ; i < length/4 ; i++ ) {
        // decode the vector by adding the shared base to all of its values
        output[i] = _mm_add_epi32(input[i], shared_base);
        // load the next shared base, i.e., shuffle the last value of the decoded vector
        shared_base = _mm_shuffle_epi8(output[i], perm_last);
    }
}
```

Figure 4.5.: Code snippet for shared-base delta encoding (decode only).

may be acceptable. For our dataset compression application, however, multiple-streams delta encoding cannot be applied because the values in the sequence must remain in the original order.

Finally, shared-base delta encoding allows vectorized encoding and decoding at the cost of higher delta values. Here, the differences to a common value for each \( k \) consecutive values are encoded instead of encoding the values differences to their predecessors. For example, the delta values for \( k = 2 \) and the sequence 1, 3, 4, 6, 8, 9 are 1, 3, 1, 3, 2, 3. The first two values are encoded by their difference to 0, the second two by their difference to 3 while the last two values are encoded by their difference to 6. Thus, shared-base delta encoding leads to higher delta values compared to delta encoding, which encodes the sequence as 1, 2, 1, 2, 2, 1.\(^5\) The vectorized decoding of shared-base delta encoding replicates the common value (i.e., shared base) and adds the \( k \) delta values using a single SIMD instruction to obtain the real values; encoding works analogously.

Figure 4.5 shows a code snippet for vectorized decoding of values that were encoded using shared-base delta encoding. The array `input` holds the encoded values while the decoded values should be written into the array `output`. The initial shared base is set to zero. Within each iteration, the shared base is added to \( k \) delta values; the result is written back into the array `output`. The last value of the written decoded values is the new shared base; it is replicated using the `PSHUFB` instruction and a predefined permutation mask `perm_last`.

4.3. Experiments

This section contains the results of our experimental evaluation. We conducted a large variety of experiments to gauge the performance and compression rates of the discussed static integer compression algorithms. We used realistic datasets for the evaluation of the compression ratios and found that static encodings are suitable for compressing datasets used in frequent-itemset mining. Furthermore, our vectorized algorithms achieve a throughput for decompression up to 24.6B (16-bit) and 18.4B (32-bit) values/second when twelve threads are used. This makes them in particular useful, when the datasets are repeatedly read like in iterative mining.

\(^5\)If the integer sequence has no duplicates, then the delta values could be decreased by subtracting the values 1, 2, \ldots, \( k \) from each \( k \) values that have a shared base.
4.3. Experiments

We implemented all compression algorithms that were presented in this chapter; this includes the sequential versions of null suppression (NULL), variable byte encoding (VARINT), and elias gamma encoding (GAMMA); as well as the vectorized versions k-wise null suppression (K-NUL) and k-gamma encoding (K-GAMMA). For each of the algorithms there exists a 16-bit and a 32-bit version, e.g., NULL16 and NULL32 denote the 16-bit and 32-bit version of null suppression, respectively. Furthermore, we integrated delta encoding in all algorithms. It can be enabled and disabled using a preprocessor argument given via a compiler flag. The values are delta encoded and decoded before compression and after decompression, respectively. We integrated both, sequential as well as shared-base delta encoding, into the vectorized algorithms. All algorithms were implemented in C; we used intrinsics for the SIMD instructions within the vectorized algorithms. On our current hardware, the vectorized 16-bit versions can process eight values at once while the vectorized 32-bit versions can process four values at once.

4.3.1. Compression results

In the first set of experiments, we evaluated the compression ratios of the static encodings that were discussed in this chapter. Recall that the overall goal is to compress datasets used for frequent-itemset mining to reduce communication costs and allow very large files to fit in main memory. For this reason, we want to show that (1) static encodings achieve high compression ratios on itemset mining datasets and (2) our vectorized encodings achieve the same compression performance like their sequential counterparts or similar techniques; NULL—as its compression performance equals K-NUL—should be competitive with VARINT and 4-GAMMA should be competitive with GAMMA.

We used several realistic datasets to evaluate the compression performance of the discussed encodings—and partially to evaluate the algorithms in the following three chapters. Most of these datasets were taken from the FIMI repository [FIMI, 2004]. The FIMI repository has been created for the purpose of evaluating different frequent-itemset mining algorithms; it contains both data and implementations. Provided datasets include retail [Brijs et al., 1999], kosarak [FIMI, 2004], webdocs [Lucchese et al., 2004b], BMS-POS [Zheng et al., 2001], and accidents [Geurts et al., 2003]. Besides these datasets, we also created three real-world datasets, wikilinks, netflix, and netflix2, using publicly available mining datasets that were not original intended for frequent-itemset mining. All datasets have different characteristics (number of transactions, average transaction cardinality, distinct items, ...) and could be assigned to one of the application areas that are discussed in Chapter 2. The size of the raw data varies from 335kB to 1.4GB. More details about the used datasets can be found in the appendix (see Section A.1).

Although the realistic datasets are small, the compression performance of the static encodings can be assumed to be similar for larger real-world datasets because a dataset’s compressibility is mainly affected by the number of distinct items and average transaction cardinality. The number of transactions has only a small impact and it is often the only characteristic that differs between our small realistic datasets and large real-world datasets.

The datasets of the FIMI repository as well as our created datasets are originally encoded in a specific ASCII format, which is used as input for the itemset mining algorithms. The items of a transaction are separated using whitespaces (one whitespace also after the last item) whereas the transactions are separated using carriage returns, i.e., a single line of a dataset’s respective file constitutes a single transaction. For a large number of distinct items,
4. Compression

the ASCII format tends to be inefficient because large values require more characters than small values. This can be avoided by using a binary representation for the datasets. For this reason, we use a simple binary format that encodes a transaction using 32-bit integer values. An encoded transaction resembles an array where the first value contains the length of the transaction and the following entries contain the transaction items. Consecutive transactions are stored directly one after another, i.e., the length field of a subsequent transaction starts directly after the last item of a transaction. Although some datasets could be encoded using 16-bit values, we use for simplicity only 32-bit values.

We vary the employed version of delta encoding and how it is applied on the dataset. Basically, we test four different configurations: (1) we disable delta encoding, (2) we apply “regular” delta encoding and (3) shared-based delta encoding directly on the binary representation, and (4) we apply “regular” delta encoding only on the transactions of the dataset. The file-wise delta encoding in configuration 2 and 3 treats the binary representation as a large sequence of integer values. Hence, the distance to the predecessor value is encoded regardless of it is another transaction item or a transaction length value, which leads to negative distance values. Such values are avoided if delta encoding is only applied on the transactions like in configuration 4. Here, the length values are directly encoded and only for the transaction items their distance between them is encoded. This leads to more smaller values but the throughput for encoding and decoding is lower than if delta encoding is file-wise applied because the binary representation needs to be “parsed” and the code is slightly more complex.

A prerequisite of the configurations 2, 3, and 4 is that the datasets’ transactions need to be sorted in ascending order according their item values. This holds true for all of our realistic datasets except for kosarak. Hence, we only have to sort the transactions of this dataset.

Figure 4.6 illustrates the compression ratios that are achieved by the discussed static encodings on the realistic datasets and the different configurations. As baseline serves the datasets’ uncompressed ASCII representation (ASCII). We also plot the compression ratio of the binary representation (BINARY), which does not depend on the values being encoded; i.e., each value is encoded using 32 bits regardless of it is small or large. Hence, BINARY shows the same compression performance on all four configurations. It is much more efficient than ASCII on large datasets; e.g., on netflix and wikilinks it achieves compression ratios of 52% and 53%, respectively. On the small datasets, however, BINARY does not work well because of the many small values within these datasets. These values require sometimes less than four bytes per value; e.g., a value smaller than 100 requires two bytes for the value and one byte for a separating whitespace.

Unlike BINARY, the compression ratios of the static encodings depend on the delta encoding configuration. Figure 4.6a illustrates them when delta encoding is disabled. In this configuration, the integer values being compressed are in average larger than with enabled delta encoding. As can be seen, VARINT, NULL, and 4-GAMMA perform similar and achieve compression ratios between 35% and 50%. GAMMA, however, achieves on the larger datasets only ratios of 60% to 70% because of the large values; it thus performs worse than BINARY on netflix and wikilinks. The better performance of 4-GAMMA compared to GAMMA mainly results from storing the large prefix of 4 values only once whereas GAMMA stores it for each value. Hence, the compression ratio of K-GAMMA might be even better for larger k.

The small datasets, retail, kosarak, and accidents, could be represented solely using 16-bit values because they all have less than 65,536 distinct values. In this case, BINARY would be more efficient than ASCII.
4.3. Experiments

The results for the second configuration, i.e., when delta encoding is applied on the binary file directly, are depicted in Figure 4.6b. Again, VARPINT and NULL perform similar but they now achieve compression ratios close to 15%. Only on the small datasets, RETAIL, KOSARAK, and ACCIDENTS, they perform worse than without delta encoding enabled. On these datasets, the values are already low so that delta encoding has only a small effect. Moreover, it introduces even large values whenever the difference of the last value of a transaction with the length field of the next transaction is encoded. A transaction’s last value is almost always larger than the length value of a succeeding transaction. This leads to a negative difference that is encoded as a large 32-bit value. For this reason, also 4-GAMMA performs worse than without delta encoding on these datasets because for each such large value, 4 consecutive values are encoded using 32 bits. GAMMA is also affected by this issue but it has to encode only the large value alone. Hence, it is superior over 4-GAMMA on the small datasets but again inferior on the large datasets. Nevertheless, all static encodings show a better compression performance than BINARY. Except on WIKILINKS, always the best of them requires 2x up to 3.5x less space to encode a full dataset. The space saving is even higher if the ASCII representation is considered.

We made similar observations for the third configuration in which shared-base delta encoding is applied directly on the binary file. As illustrated in Figure 4.6c, the static encodings show a similar behavior regarding their compression ratios on the small and large datasets. They are only higher than for “regular” delta encoding. On the small datasets, the difference is larger because of the length value recoding. It has a large impact on shared-base delta encoding.

---

7This “overflow” value is during decompression correctly recoded into a negative distance.
4. Compression

coding because \( k \) values—in our case 4 values—get large when a transaction’s last element is used as shared base. This happens more often for the small datasets, which consist of mainly short transactions. For example, NULL and VARINT achieve only a compression ratio of about 60\% on \textit{retail} and \textit{kosarak}. This is 10\% to 15\% higher than for regular delta encoding. The difference is smaller for the large datasets. Here, it is only about 3\% to 7\%. As we will see later, the throughput of shared-base delta encoding is much higher than for the regular one. Hence, one needs to find a tradeoff between throughput and compression ratio.

Figure 4.6d illustrates the results for the last configuration where only the transactions are delta-encoded. This configuration has the highest costs for applying delta encoding but leads to the smallest integer values compared to the other configurations. Hence, it leads to the best compression ratios. Interestingly, all static encodings perform similar. Only on \textit{accidents}, GAMMA requires less than half of the space that VARINT or NULL requires for encoding the file. Compared to BINARY, it even achieves a file size reduction of 8.5x.

To summarize, static encodings are well suited for compressing datasets used for itemset mining. These datasets contain—after delta encoding is applied—many small integer values, which can be well compressed. As a result, all static encodings significantly outperform BINARY. Furthermore, our vectorized encodings show a competitive compression performance with their sequential counterparts and other encodings; NULL achieves similar compression ratios like VARINT on all tested datasets and configurations. Also 4-GAMMA is competitive with GAMMA; it even achieves on some datasets and configurations much better compression ratios than GAMMA. As we will show in the next section, the vectorized implementations of NULL and 4-GAMMA have a superior compression and decompression throughput, which makes them the encodings of choice for compressing datasets in itemset mining.

4.3.2. Performance results

This section contains the results of our performance experiments. We measured the single-threaded as well as the multi-threaded performance of all algorithms that were discussed in this chapter. We used the same hardware and software setup that is described in Section 3.3.1 on page 65. To ensure fair comparison, all implementations are hand-tuned: Both the sequential and the vectorized versions make extensive use of the available instruction set for efficient processing. Before processing, the complete datasets were loaded into main memory. We also allocated sufficient memory before the operations to avoid expensive \texttt{malloc} calls during processing. Furthermore, we used the \texttt{linux mlock} function for ensuring that the allocated arrays are main memory resident, i.e., none of them is swapped out to disk at any time. For the multi-threaded experiments, we used OpenMP to employ TLP. The dataset is partitioned into multiple partitions so that the threads can independently process these partitions.

All performance experiments were conducted on synthetic datasets; this allows us to adjust the compression ratio to evaluate its influence on the performance of the algorithms. The datasets themselves consist of non-negative integer values that follow a Zipf distribution with Zipf parameter \( z \). A parameter of \( z = 0 \) corresponds to a uniform distribution (no compression possible), while parameter values of \( z > 1 \) represent distributions that are heavily skewed towards small integers. Since the values are all larger than zero, GAMMA can process them directly; there is no need for a value mapping to remove zeros. The integer values within the datasets are further accumulated so that they form a monotonically increasing integer sequence. This is a prerequisite for delta encoding, which is employed within all experiments.
4.3. Experiments

For each algorithm, we generated two datasets of 1GB: a dataset with low compression potential and a dataset with high compression potential.

In all experiments, we measured the wall-clock time using the \texttt{gettimeofday} and used it to calculate the throughput of the algorithms, i.e., how much values are processed in a certain operation per second. Note that the encoding/decoding bandwidth can be calculated from the throughput; a throughput of 1B values/second for 32-bit values corresponds to a bandwidth of 4GB/s. We evaluated the performance of three different operations: (1) \texttt{encode\_copy}, (2) \texttt{decode\_copy}, and (3) \texttt{decode\_only}. The first and the second function involve loading uncompressed/compressed values from main memory and storing of compressed/uncompressed values back into main memory, respectively. The third function reads compressed values from main memory and decompresses and sums them up, i.e., it does not put the uncompressed data back into main memory.

As a baseline for comparison, we measured the throughput of all three functions when working on uncompressed integers. The \texttt{encode\_copy} and \texttt{decode\_copy} function then reduce to a simple \texttt{memcpy} operation. We implemented a \texttt{memcpy} operation, which achieves a throughput of 1.8B and 3.15B values per second when one or twelve threads are used, respectively.\(^8\) Similarly, the \texttt{decode\_only} function reduces to an array sum. It achieves a throughput from 4.0B (one thread) up to 10.85B (twelve threads) values per second. These numbers apply for 32-bit integer values; the throughput is doubled when 16-bit integer values are used. As can been seen, the \texttt{memcpy} as well as the array sum operation do not scale well if the number of threads is increased. The reason for that is the limited memory bandwidth. Our test system allows to read from and write to main memory about 43GB/s and 16GB/s, respectively. Hence, if only few instructions are issued per memory request, then the system’s memory bandwidth—i.e., maximum number of memory requests—limits the throughput when all threads are used.

\textbf{Basic static encodings}

In the first set of the performance experiments, we compare the existing scalar static compression algorithms. More precisely, we compare the throughput of \texttt{null32}, \texttt{varint32}, and \texttt{gamma32}. We further include the “pure” delta encoding (\texttt{delta32}) that does no compression of the encoded values but allows to measure the overhead of delta encoding. We set \(z_{\text{low}} = 0.8\) and \(z_{\text{high}} = 1.25\) to obtain a dataset with low and high compression potential, respectively.

In the first experiment, we examine the throughput for the single-threaded execution; the results are shown in Figure 4.7a. Each group of bars corresponds to one function, and each individual bar corresponds to an algorithm. For all algorithms except \texttt{varint}, there is no difference whether the compression potential of the dataset is low or high. They all require the same amount of instructions regardless of whether small values or large values are encoded and a single thread does not fully utilize the available memory bandwidth. However, \texttt{varint} benefits from smaller values because it requires less instructions when such values are encoded or decoded. Therefore, the throughput for all three operations is almost doubled if the high instead of the low potential dataset is used. In general, \texttt{delta32} is always the fastest of the four algorithms; it achieves for all three operations a throughput of about 1.35B values/second. This amounts to 75\% of the \texttt{memcpy} throughput and 33\% of the \texttt{array sum}’s throughput. \texttt{null32} is the second fastest of the scalar algorithms; it achieves a throughput of roughly 0.5B values/second for all three operations. The algorithms \texttt{varint32} and \texttt{gamma32} are between

\(^8\)Our own implementation was a few percent faster than the corresponding Linux library function.
4. Compression

Figure 4.7.: Throughput of the scalar static encodings

We observe that GAMMA32 compresses values faster than VARINT32 while for decompressing values it is the other way around. Both algorithms have high costs because of the many shift/mask instructions and exceed in none of the tasks 0.26B values/second. However, the main costs of VARINT32—the expensive segmenting of the integer value into 7-bit parts—could be avoided if the algorithm is changed. We implemented an alternative version of VARINT32 that avoids this segmentation by encoding the number of 7-bit parts into the lowest significant bits of the first encoded byte so that the 7-bit parts can be stored as a continuous chunk. It encodes values roughly 2.7x and 4x faster than VARINT32 and achieves independently of $z$ for both other operations a throughput of 0.26B values/second. Nevertheless, it is still slower than NULL32.

In the second experiment, we evaluated the throughput for a multi-threaded execution of the four scalar compression algorithms. We used the same setup as in the previous experiment but varied the number of threads that are used for the processing of the three tasks from one to twelve. Figure 4.7b–4.7d illustrate the results for the dataset with high compression potential. All compression algorithms—except for DELTA32—scale well if the number of threads is increased. They achieve linear speedups until the number of threads matches the number of physical cores. For 7 up to 12 threads, the throughput still increases but at a slower pace because the threads compete for the execution units of the cores. NULL32 achieves at best a throughput of 3.15B values/second for encoding and roughly 4B values/second for decoding (with and without writing back the values). Again, GAMMA32 and VARINT32 are much slower than NULL32. DELTA32 has only few instructions in the main loop and is mainly limited by the memory bandwidth of the system. For the encode and decode function, already four threads fully saturate the memory bandwidth. The highest throughput is achieved with
4.3. Experiments

(a) Influence of manual loop unrolling (high compression potential dataset)

(b) Different versions of delta encoding on 4-NULL32

Figure 4.8.: Throughput of sequential and vectorized null suppression

5 or 6 threads, while it even decreases for 7 and more threads. For this reason, NULL32 is superior over DELTA32 for more than 7 (decoding) and 11 threads (encoding) because it requires less of the system’s memory bandwidth. Nevertheless, DELTA32 always achieves the highest throughput (up to 8.9B values/second) for the third task where data is only read.

Null suppression

Within the second set of experiments, we compare different versions of null suppression. As before, we set $z_{\text{low}} = 0.8$ and $z_{\text{high}} = 1.25$ to obtain datasets with low and high compression potential, respectively. For regular delta encoded values, the corresponding compression ratios of null suppression are 95% and 38%.

In the first experiment, we evaluate the impact of data dependencies on the different versions of null suppression. To remove data dependent latencies, we process the integers being compressed or decompressed in batches (cf. Section 4.1.5) and manually unroll the core loop of the compression algorithms under test. For that purpose, the compression masks of $b$ blocks of 4 integers are combined and stored consecutively up front. We evaluate the impact of this batching/unrolling in the first experiment with disabled delta encoding; the respective results for the high compression potential dataset are shown in Figure 4.8a. In general, we observe only a small effect on the performance of NULL32. If the main loop is unrolled two times ($b = 2$), encoding is only 1.1x faster than if it were not unrolled ($b = 1$). The same holds true for decoding; it achieves at best a 1.15x higher throughput when it is unrolled. For $b = 4$, decoding is up to 1.2x faster than for $b = 1$ but the encoding throughput even decreases slightly. In general, NULL suffers more from the many instructions that were required for encoding and decoding than from the data dependencies. K-NULL32 benefits more of unrolling. It encodes values up to 1.35x faster when the loop is unrolled four times. For K-NULL32’s decoding, unrolling is even more effective. The decode_copy and decode_only function is for $b = 4$ about 2.5x and 2.8x faster, respectively. Furthermore, we observed a similar behavior for the 16-bit version (not shown). In the following experiments, we set $b = 4$ since both, NULL as well as K-NULL, benefit from unrolling.9

9Note that all these effects only occur for “pure” encoding and decoding. If these operations are combined with some other calculations, then the impact of the data dependencies may soften and hence K-NULL achieves these speedups already for $b = 1$. 

91
4. Compression

In the second experiment, we evaluate the influence of the different versions of delta encoding that can be combined with K-NULL32. We compare plain null suppression (NONE), shared-base delta encoding (SHARED), and regular delta encoding (REGULAR). Again, we use a dataset with low and high compression potential. As can be seen in Figure 4.8b, the encoding throughput is influenced by neither the compression potential of the dataset nor the delta encoding version being applied. On both datasets, it remains almost the same because vectorized encoding of null suppression requires many instructions whereas both, SHARED and REGULAR, can be vectorized and consist of only few instructions. For this reason, they increase the costs of encoding by only a small fraction. The throughput of decoding, however, is influenced by the dataset’s compression potential and the delta encoding version. It increases when the compression potential of the dataset is higher. The reason for that are the fewer memory requests that need to be issued while processing the same amount of values. The throughput further increases, if delta encoding is disabled. With a single thread NONE achieves roughly 2.55B values/second for the decode_only function on the high compression potential dataset. Of both delta encoding versions, the overhead of SHARED is relatively low; NONE is only between 1.05x and 1.10x faster than it. In contrast, the overhead—caused by the sequential processing—of REGULAR is much higher so that NONE is up to 1.3x faster.10

As mentioned in Section 4.3.1, SHARED has lower compression ratios compared to REGULAR (cf. the discussion on page 87). For this reason, there is a tradeoff between compression ratio and throughput. Whenever the compression ratio is less important and higher throughput is preferred, SHARED is the algorithm of choice.

In the next two experiments, we compare the throughput of NULL and K-NULL for a varying number of threads and different integer precisions. In both experiments, we employed SHARED for delta encoding within K-NULL and used the high compression potential dataset. Figure 4.9a shows the results for single-threaded execution of NULL16, NULL32, K-NULL16, and K-NULL32. The throughput of NULL16 and NULL32 is almost equal because both version require the same amount of instructions to encode or decode a value. In contrast, K-NULL16 can process eight values with same amount of instructions that K-NULL32 requires for four values. Therefore, K-NULL16 is much faster than K-NULL32. Independently of the integer precision, K-NULL achieves always a higher throughput compared to NULL. More specifically, K-NULL16 has a 4.6x and 5x higher encoding and decoding throughput compared to its sequential counterpart, respectively. For the decode_only function, K-NULL16 is even 6.5x faster than NULL16 and achieves a throughput of roughly 3.8B values/second. The achieved speedups of K-NULL32 are lower compared to this. Of the three operations, encoding is the most expensive one for K-NULL32 because there is no parallel population count instruction. Nevertheless, it is still 2x up to 3.85x faster than NULL32.

The results for the multi-threaded execution with up to twelve threads are shown in Figure 4.9b–4.9d. As can be seen, NULL16 performs similarly to NULL32; is is slightly faster than the latter when the compressed or uncompressed values need to be written back (cf. Figure 4.9b and 4.9c) because it requires less memory bandwidth. Both, K-NULL16 and K-NULL32, do not scale as well as their scalar counterparts. Similarly to DELTA32, they are both limited by the system’s memory bandwidth: K-NULL16 achieves for the compression task its peak throughput of roughly 7B values/second already with 4 threads whereas K-NULL32 saturates the memory bandwidth using 6 threads and achieves at most 4.5B values/second for

10We observed larger differences on other platforms where SHARED is up to 1.5x faster than REGULAR. Furthermore, SHARED is expected to show a better performance as the width of the SIMD registers increases.
4.3. Experiments

this task. The same behavior can be observed for decoding (see Figure 4.9c) where both implementations almost achieve the peak throughput with 4 threads. Notice that the encoding and decoding throughput of K-NUL32 is about 1B values/second higher than for the memcpy operation because less bandwidth is required for loading or storing of compressed values. If data is only read from main memory as for the decode_only function, then K-NUL16 and K-NUL32 scale almost linear; K-NUL16 encodes up to 24.6B values/second using twelve threads and is thus 6.4x faster than NUL16. For the same configuration, K-NUL32 achieves a throughput of 18.4B values/second, which amounts to a speedup of 4.5x compared to NUL32 and is almost twice of the throughput being achieved when working with uncompressed 32-bit integer values (10.85B). The throughputs are even higher when working on datasets with very high compression potential (z = 1.75).

Gamma encoding

In the third set of the performance experiments of this chapter, we compare the throughput of gamma and 4-gamma. We only use a high compression potential dataset, which is generated by setting \( z_{\text{high}} = 1.75 \).

The results for the single-threaded experiments are shown in Figure 4.10a. As already mentioned earlier, bitwise processing has a high impact on bandwidth. None of the implementations exceeds 1B values/second for any of the tasks. Nevertheless, there is a large performance difference between gamma and 4-gamma. If 16-bit values are decoded, K-gamma is 6.4x (7.5x without writing back) faster than gamma. For encoding of 16-bit values, however, K-gamma is only 3x faster because it contains a large sequential fraction in which the shared
4. Compression

prefix is obtained. The throughputs are lower when 32-bit values are processed. Again, k-
gamma’s speedup for encoding is lower (3.1x) than for decoding (3.75x); k-gamma32 encodes
and decodes about 0.55B and 0.45B values/second, respectively.

We observe similar performance improvements for the multi-threaded execution of both
vectorized encodings. Figure 4.10b, 4.10c, and 4.10d illustrate the throughput of the three
operations for a varying number of threads. In general, all gamma encodings are rather
compute than memory intensive so that they all scale well if the number of threads is increased.
As before, the throughput increases at a slower pace as soon as more than 6 threads are
used because the threads compete for the execution units. Interestingly, vectorized decoding
profits more of Hyperthreading than vectorized encoding and even scalar decoding. While
k-gamma32’s encoding throughput is higher than its decoding throughput when a single
thread is used, it is vice versa for twelve threads. To summarize, k-gamma16 encodes 5.35B
values/second and decodes 8.2B values/seconds when twelve threads are used. It is then up
to 8.25x faster than gamma16. k-gamma32 also achieves high throughputs—3.6B and 4.15B
values/second for encoding and decoding, respectively—when twelve threads are used. It is
then up to 4.15x faster than gamma32 and also faster than the memcpy function. Hence, the
overhead for compression becomes negligible when the vectorized encodings are used.

4.4. Summary and Discussion

In this chapter, we discussed compression algorithms that could be used to compress the
intermediate state of frequent-itemset mining algorithms and the datasets being mined. First,
we classified integer compression algorithms that were often used in the area of databases
4.4. Summary and Discussion

and information retrieval. Based on that, we found that static encodings are well suited for compressing frequent-itemset mining datasets because they are much faster than entropy encodings and they do not require knowledge about the data characteristics (e.g., maximum value, exact value distribution) before they can be applied. Null suppression and elias gamma encoding are two popular representatives of static encodings, which are often combined with delta encoding. For all three encodings, we discussed efficient sequential as well as parallel implementations. In the latter implementations, ILP is utilized by avoiding control and data dependencies within the main loop; TLP is employed by dividing the dataset being compressed into independent partitions. Efficiently exploiting DLP, however, is more challenging. For this reason, we proposed two different data layouts, which form the foundation of our vectorized implementations $k$-wise null suppression, $k$-gamma encoding, and the multiple versions of vectorized delta encoding.

We evaluated the compression performance of the discussed static encodings on various realistic datasets. We found that they are—combined with delta encoding—well suited for compressing large datasets. The tested encodings achieve file size reductions of up to 8.5x compared to an efficient binary file representation and up to 6.4x compared to the standard ASCII file representation. In average, the compressed files are 2.5x smaller than in the former and 3x smaller than in the latter representation. Hence, a 1TB dataset in the ASCII representation is in average compressed to only about 330GB. This allows to hold large datasets in main memory so that they need to be loaded only once even if they are repeatedly mined.

Besides evaluating the compression performance, we also measured the throughput of sequential and parallel implementations of the discussed static encodings. We found that the vectorized encodings achieve much higher throughputs than their sequential counterparts, i.e., their throughput is in some cases up to 6.5x higher. $k$-wise null suppression is the fastest among all tested algorithms; it decompresses up to 24.6B values/second on our test system that has six cores. For this reason, it is well suited for compressing datasets that are often decompressed when they are iteratively mined. Even if $k$-gamma encoding achieves a smaller throughput than $k$-wise null suppression, it is still useful for compressing datasets on which null suppression has a lower compression performance. Furthermore, $k$-gamma encoding could serve as the basis of more elaborate bitwise encodings because gamma encoding—of which $k$-gamma inherits from—resembles to other bitwise encodings like colomb codes [Golomb, 1966] or rice codes [Rice, 1979]. Most of our parallel optimizations on gamma encoding thus could easily be applied on these encodings so that also existing mining algorithms that employ them could benefit from our optimizations; e.g., VIPER [Shenoy et al., 2000] uses a combination of run-length encoding with colomb codes (see Section 2.2.2 on page 19).
Part III.

Frequent-itemset mining algorithms
5. cApriori

In this chapter\textsuperscript{1}, we propose cAPRIORI (short for compact APRIORI) as a memory efficient and highly scalable version of the candidate-generation based algorithm APRIORI. Both algorithms differ basically only in how the filtered transactions are internally represented and support counting is performed. cAPRIORI employs a transaction representation that is optimized for datasets with many small transactions. It has a small memory footprint even for large of such APRIORI-suited datasets and allows a fast conversion of them. The latter is typical one big strength of cAPRIORI compared to other mining algorithms that are based on complex data structures (e.g., FP-GROWTH) and therefore usually require much more time for converting a dataset. Besides the optimized data layout, cAPRIORI is founded on efficient parallel subset counting and simple but effective database pruning techniques. Parallel counting is—depending on the number of candidates—performed using either replicated or partitioned data structures. For the latter, we propose a modified producer–consumer processing scheme.

In Section 5.1, we introduce cAPRIORI’s internal transaction representation; it holds the filtered transactions that are obtained after the second scan of a dataset. We further discuss cAPRIORI’s multi-threaded dataset conversion and underlying memory management. Both are crucial for a highly scalable parallel mining algorithm because the conversion amounts to a large fraction of the overall runtime and standard memory allocators are not optimized for situations where many threads repeatedly allocate small objects.

We discuss efficient parallel support counting in Section 5.2. We start the discussion with parallel variants for direct counting, which is used for obtaining the frequent items as well as the frequent-2 itemsets. This includes counting using atomic increments as well as replicated or partitioned data structures. Thereafter, we review state-of-the-art algorithms for counting subsets that comprise more than 2 items; we discuss the algorithms’ advantages and possible drawbacks. Based on that, we propose optimizations that increase ILP and reduce the memory footprint of the subset counting algorithms. As for parallel direct counting, we discuss different variants for parallel subset counting.

We propose and discuss simple candidate and database pruning techniques in Section 5.3. They are used to reduce the number of candidates and trim or prune transactions during support counting; less candidates or transaction may significantly reduce the effort of subset counting. Thereby, our techniques are effective but incur only low overhead.

Finally, we present the results of our experimental evaluation of cAPRIORI in Section 5.4. We explain the experimental setup and give an overview about the used synthetic and realistic datasets; both the setup and datasets are also used for evaluating the algorithms in Chapter 6 and 7. The actual experiments are split into two parts. In the single-threaded experiments, we measure the impact of our optimizations, database pruning, and compare cAPRIORI with highly-optimized existing APRIORI implementations. We found that cAPRIORI is often about 10x faster than its competitors. In the multi-threaded experiments, we examine the scalability

\textsuperscript{1}The material in this chapter has been developed jointly with Tim Kiefer, Thomas Kissinger, and Wolfgang Lehner. The chapter forms an extended version of prior published work \cite{Schlegel2013b}.
of \textsc{CAPRIORI}'s core components as well as its overall scalability on large datasets. We observed an almost linear scalability on our large test system that has 32 cores.

5. \textsc{cApriori}

In this section, we present \textsc{cApriori}'s internal transaction representation, which is optimized for \textsc{Apriori}-suited datasets. We discuss its advantages and how transactions can be converted into it in parallel. We further propose a tailor-made memory management that has a large impact on \textsc{cApriori}'s performance. It is also employed by our two mining algorithms \textsc{CEclat} and \textsc{CFP-growth}, which will be proposed in Chapter 6 and Chapter 7, respectively.

In what follows, we assume that the dataset being mined is already fully available in main memory (e.g., in an in-memory database or loaded previously from disks or network). Otherwise, loading the dataset might be the main bottleneck of the overall processing, so most of our optimizations would be meaningless. The compression algorithms from Chapter 4 can be exploited to reduce the transfer time or to load datasets that are larger than the available main memory. Furthermore, we assume that the data must not be changed because of iterative mining; recall that a user might invoke mining repeatedly until certain metrics are fulfilled. Hence, the mining algorithms' intermediate state that usually comprises filtered transactions in a certain layout and candidates must fit in the remaining available memory.

5.1. Data representation

In this section, we present \textsc{cApriori}'s internal transaction representation, which is optimized for \textsc{Apriori}-suited datasets. We discuss its advantages and how transactions can be converted into it in parallel. We further propose a tailor-made memory management that has a large impact on \textsc{cApriori}'s performance. It is also employed by our two mining algorithms \textsc{CEclat} and \textsc{CFP-growth}, which will be proposed in Chapter 6 and Chapter 7, respectively.

In what follows, we assume that the dataset being mined is already fully available in main memory (e.g., in an in-memory database or loaded previously from disks or network). Otherwise, loading the dataset might be the main bottleneck of the overall processing, so most of our optimizations would be meaningless. The compression algorithms from Chapter 4 can be exploited to reduce the transfer time or to load datasets that are larger than the available main memory. Furthermore, we assume that the data must not be changed because of iterative mining; recall that a user might invoke mining repeatedly until certain metrics are fulfilled. Hence, the mining algorithms' intermediate state that usually comprises filtered transactions in a certain layout and candidates must fit in the remaining available memory.

5.1.1. Page-wise layout

All efficient frequent-itemset mining algorithms transform the transactions of the dataset being mined into an internal representation on which the mining takes place. The chosen representation has a large impact on the mining algorithm's performance and memory footprint. Recall that \textsc{Apriori}-based algorithms store the filtered transactions in the horizontal layout, i.e., the transaction database is represented as a set of transactions of which each consists of only frequent items. As discussed in Section 2.2.1, there exist various representations like list-based or array-based layouts and even more elaborate data structures like tries. For \textsc{cApriori}, we employ a \textit{page-based transaction representation}, where the filtered transactions are stored clustered in pages based on their length, i.e., all filtered transactions that share a page have the same length. In the following, we discuss the layout in more detail.

To better understand the benefits of the page-wise layout, we analyze the filtered transactions of various realistic datasets and varying $\xi$ values in a small experiment. As for all \textsc{Apriori}-based algorithms, \textsc{cApriori} works usually only well on datasets that have a small average transaction cardinality, i.e., in a range up to a few tens of items per transaction. For datasets with larger transactions, subset-checking is unreasonably expensive so that, for example, \textsc{FP-growth} is better suited for mining them. Hence, for this experiment we used only datasets that have a small average transaction cardinality.

Figure 5.1 illustrates the distribution of the length of filtered transactions for the realistic datasets \textit{retail} and \textit{kosarak} and various choices of $\xi$. The former and latter dataset have an average transaction cardinality of 10.3 and 8.1 items, respectively. We made the following observations:

- For large $\xi$ values, e.g., $\xi = 10\%$, almost all filtered transactions have less than 5 items.
- Even as $\xi$ decreases to 1%, about 90% of them still have less than 5 items.
Only for low $\xi$ values, there occurs a noticeable amount of longer transactions, i.e., roughly 5% of the transaction have more than 18 items. Nevertheless, 90% of the transactions have less than 16 and 12 items on retail and kosarak, respectively. As $\xi$ decreases further on these datasets, the sweet spot of CAPIORI is left and other mining algorithms like FP-GROWTH perform better.

A large number of transactions contains only a single frequent item. These transactions must not be stored but can be used to reduce the number of candidates for the following iterations. We propose such an optimization in Section 5.2.3.

We obtain similar results for the BMS-POS and wikilinks dataset.

Generally, many small transactions make the page-based transaction representation efficient because the length of the transactions within a page must not be stored multiple times; each page stores the length of its containing transactions only once. Since a length field requires as much memory as a single item, the space reduction is 33% for length-2 transactions and 25% for length-3 transactions. The overall savings are similar because filtered transactions of such lengths occur most frequently in datasets suitable for APRIORI (cf. Figure 5.1). Nevertheless, it is not useful to maintain pages for each transaction length. Especially for long transactions, there is almost no space reduction and the pages for certain lengths (e.g., for more than 20 frequent items) usually contain only few transactions. For this reason, we only partition the transactions up to a length $p_{\text{max}} = 16$. All filtered transactions that contain more than $p_{\text{max}}$ items are stored in pages with explicit length information assigned to each transaction. For mining, we maintain $p_{\text{max}}$ lists where each list connects pages that contain transactions of the same length.

The conversion from a dataset to the page-wise layout is simple and cheap; $p_{\text{max}}$ pages are maintained in which filtered transactions are inserted depending on their length. Whenever a page is full, a new page is created and the full page is enqueued into the respective global page list. Parallel conversion works similar, except that each thread maintains $p_{\text{max}}$ pages and parses independent parts of the dataset. To minimize the synchronization costs of the enqueue operation, we set the page size to 1MB. Notice that the filtered transactions have a different order in the page-wise layout, which, however, does not influence the mining result.

The reduced memory footprint and cheap dataset conversion are not the only advantages of the page-wise layout. Scanning the transactions is cheaper than for list-based and array-based layouts because there is no need to decode the length information for each transaction.
5. cApriori

separately. Hence, data dependencies disappear which greatly improves ILP. Besides faster scanning, the page-wise layout is amenable for candidate and database pruning techniques; we discuss such techniques in Section 5.3. Lastly, the page-wise layout eases load distribution and balancing for multi-threaded support counting. The pages provide a proper granularity for work distribution and the load of a page can be predicted based on the length of the transactions it contains. Pages with large transactions induce more load because longer transactions have more subsets that need to be counted. For this reason, these pages are processed first to avoid load imbalance at the end of an iteration.

The filtered transactions are represented in the pages with only as many bytes as required to encode the number of frequent items. Such an optimization was previously applied by Orlando et al. [2003a] for their array-based transaction representation and requires a recoding from the actual item id to an internal id that lies in the range from 0 to \( n - 1 \) where \( n \) denotes the number of frequent items. Hence for 256 or less items, each transaction item is 8-bit encoded, whereas otherwise the transaction items are 16-bit encoded. More than 65,536 frequent items are uncommon for frequent-itemset mining but in such a case, the items would be encoded using larger data types.

Compression could be employed to further reduce the size of the filtered transactions within the pages. The page-wise layout, however, is already very efficient so that gain of compression is expected to be low. For example, if null suppression is employed, then a compression ratio of at best 9/16 can be achieved.\(^2\) For this reason, we do not compress cApriori’s internal transaction representation. Nevertheless, compression might be more beneficial for rare itemset mining where the number of frequent items could be larger than 65,536. In this case, the filtered transactions could be compressed page-wise. Thereby, there is no need to intermix the code for mining with the compression code, i.e., a page can be temporarily fully decompressed and then the regular algorithm can start processing (e.g., subset counting).

5.1.2. Memory management

Memory management is an essential part of efficient frequent-itemset mining algorithms because all of them have complex allocation pattern and repeatedly allocate and deallocate memory. During mining, for example, APRIORI grows and shrinks candidates, ECLAT creates and deletes intermediate tid-lists, and FP-GROWTH builds and deletes conditional trees. Thereby, the sizes of the allocations vary strongly. The regular memory allocation functions like `new` and `delete` within C++ or `malloc` and `free` within C, however, are intended for applications with simple allocation pattern and thus cause problems. First of all, the functions all maintain complex data structures (e.g., free-lists), which are updated whenever memory is allocated or deallocated. This leads to serious performance degradations when a large number of allocations is performed, for example, for small chunks of memory like nodes in a tree. Memory fragmentation—as the second problem—increases the memory footprint of the mining algorithm. It is caused by the varying sizes of the allocations and worsens when multiple threads are used. Lastly, calls to the allocation functions are internally serialized—using futexes or spinlocks to avoid race conditions—which hinders scalability.

Our tailor-made memory management avoids the problems of the standard allocators. We use lightweight memory pools, which provide memory that grows and shrinks like a stack. The pools provide a simple interface with basically only two functions. The function `grow`\(^2\) Assuming the items are encoded using 16 bit and null suppression uses a single bit to distinguish between an 8-bit and 16-bit value. If the items are 8-bit encoded, then null suppression is even useless.
returns a chunk of memory from the end of the stack and increases the stack by the size of the allocated chunk. Contrary, the shrink function reduces the stack to a certain size. The memory allocation and deallocation of each of the mining algorithms’ core data structures is thus be realized with this interface. For that purpose, each thread within all of our algorithms has often multiple own memory pools assigned from which it allocates memory without any synchronization. In CAPRIORI, for example, each thread grows an own page stack by the size of a page whenever a new page must be created. Besides faster allocation, the memory pools allow an easier control on which processor the memory should reside, i.e., an NUMA-aware memory allocation.

Each memory pool is maintained using only a few variables. The base pointer holds the address of a continuous memory chunk (i.e., the start of the stack) whereas the stack pointer refers to the end of the stack. Each pool has a large chunk of virtual memory assigned, which is provided by a memory-mapped file. These files are mapped on consecutive addresses in the process’ address space with a large distance between each pair of pools (e.g., 16GB). The pools are increased or decreased using the thread-safe linux function mremap. To avoid repeated calls of it, we internally increase or decrease the stack chunk-wise—usually using at least 128MB chunks. Because virtual memory is not mapped to physical memory until it is touched (i.e., read or written), this does not increase the physical memory usage.3

5.2. Support counting

In this section, we discuss CAPRIORI’s support counting phase. Recall that APRIORI-based algorithms generate before each iteration a set of candidates for which the support is obtained by counting the subsets of the filtered transactions. The candidate generation itself is typically inexpensive compared to the actual counting on large datasets; we observed—using the gprof profiler—that it often requires less than 1% of the time within an iteration. Hence, there is no need to parallelize it until the number of threads is very large. The remaining 99% of the time are required for subset counting, so it is worthwhile to optimize and parallelize it.

We start the discussion with parallel direct counting, which is also used in CECLAT (Chapter 6) and CFP-growth (Chapter 7). We then discuss existing sequential trie-based subset counting algorithms and provide optimizations for them. Based on that, we present parallel trie-based subset counting algorithms.

5.2.1. Parallel direct counting

As discussed in Chapter 2, direct counting is used for obtaining the frequent items within all mining algorithms and for obtaining the frequent 2-itemsets within APRIORI-based algorithms. In both cases, counting is performed by increasing values in a large lookup table. In what follows, we discuss three different approaches for parallel direct counting. The selection of the optimal approach is influenced by the size of the lookup table.

If the frequent items should be obtained, the size of the lookup table depends on the number of distinct items that are within the dataset being mined, i.e., for each distinct item a single 4-byte counter is maintained. For market-basket analysis datasets (e.g., retail and BMS-POS), the number of distinct items is typically below 100,000. For web-mining datasets (e.g.,

---

3Only when the stack shrinks then some physical memory might not be returned to the operating system. In our algorithms, however, we rarely shrink the stacks so that this behavior is acceptable.
webdocs or wikilinks), however, the number of distinct items easily reaches several millions. Hence, the lookup table used for counting could require less than a megabyte but also up to hundreds of megabytes. Similarly, the number of frequent items from which the frequent 2-itemsets are obtained is typically in the range of a few hundred up to a few ten thousands. If each ordered pair of frequent items is mapped to a 4-byte counter, then the respective lookup table varies from several kilobytes up to a few gigabytes. Thus, the lookup table for direct counting fits in some cases in the processor’s caches.

![Parallel direct-count approaches](image)

**Figure 5.2:** Parallel direct-count approaches

While the direct counting technique is straightforward to implement for sequential processing, it is more involved for parallel processing. In general, there are three parallel direct counting approaches that differ in the way how they deal with parallel increments. Using a single shared table for all threads is prohibitively expensive because synchronization (e.g., locks or atomic increments) is required to avoid inconsistent results caused by race conditions. Figure 5.2a illustrates the shared table approach.

The replicated table approach—shown for two threads in Figure 5.2b—maintains a local table for each thread in which the thread can increment the items’ count values independently from other threads. Synchronization is only required after counting is finished to merge the local tables into a single table that contains the final result. Besides its simplicity, the replicated table approach has a much higher memory consumption than its sequential counterpart. Hence, it cannot be employed for very large tables when a large number of threads is used for counting. Furthermore, even for small tables with only a few megabytes, it has a limited scalability because the local tables may not fit into the caches anymore while a single global table does fit into the caches (at least in the large last-level cache).

The partitioned table approach uses only a single table for all threads, however, it is partitioned so that each thread increments the count values for those items that lie within a certain range. In Figure 5.2c, for example, the first thread counts all values from 1 to 3 while the second thread counts values from 4 to 6. For realistic scenarios, the ranges are usually much larger and the threads have multiple ranges assigned for a better load balancing. Furthermore, for 32-bit count values, the size of the ranges must be a multiple of 16 because otherwise false sharing might occur. The actual parallel counting of the items can be done in two ways: (1) all threads scan the complete dataset thereby counting only their assigned values and (2) each thread has an assigned part of the dataset and distributes scanned values to the responsible threads. The former is only useful, if scanning of the data is not expensive; it should thus not be applied within the first two scans because they suffer from high parsing costs. Also it has a limited scalability since the threads do much redundant work, i.e., each thread checks...
5.2. Support counting

for each value whether it is in its range or not. The latter, however, requires communication between the threads. For that purpose, we employ a modified producer–consumer scheme, which we explain below in more detail. Indeed, partitioned counting induces communication overhead, it requires only as much space for the count table as the shared table approach or sequential processing. Compared to replicated processing, the caches are thus much better exploited, which pays off for large tables.

There are also hybrid approaches between these base approaches possible. For example, a replicated table could be assigned to each processor whereas threads that run on the same processor share a partitioned table. As soon as the overhead for partitioning is paid, however, it seems more efficient to use only a single lookup table for all threads of a system because then each processor has to hold only parts of the full table, which improves cache efficiency. Only when communication between the processors is expensive (in terms of bandwidth) or too many partitions are required, then hybrid approaches might be beneficial.

Multiple-producer/multiple-consumer processing model

In the following, we describe our multiple-producer/multiple-consumer processing model, which is derived from the basic producer–consumer model (cf. Section 2.5.3 on page 45). It is employed for (1) partitioning the subsets being counted and (2) communicating them to the threads responsible for increasing their respective count values in the partitioned count table. The basic idea is that each thread is always in one of two states. As producer, it partitions the input data based on a certain partitioning function whereas as consumer, it processes data in its partition. For parallel direct counting, the transactions from the input dataset \((k = 1)\) or the page-wise layout \((k = 2)\) form the input data and processing means to increase the count values in the count table.

![Figure 5.3: Multiple-producer/multiple-consumer processing with four partitions and two threads](image)

Synchronized work queues are used to communicate partitioned data between the producers and the consumers. Each thread has a single work queue assigned that contains the partitioned data the thread is responsible for, i.e., data that lies in the thread’s assigned partitions. To avoid excessive communication overhead that would occur if a producer enqueues single data elements, the producers collect data in local buffers and enqueue chunks of data. For that purpose, the local buffers have pages assigned that float in cycles during processing. They are (1) filled by a producer, (2) put into a work queue, (3) processed by a consumer, (4) put
into an *empty-pages queue*, and finally (5) obtained by a producer from the latter queue to start the cycle again. Figure 5.3 illustrates the components of the processing model (except for the empty-pages queue) when two threads are employed that divide the input data into four partitions. Notice that the work queues are filled by multiple threads but each of them is read by only a single thread.

For parallel direct counting, each producer calculates the positions of the subsets’ count values and communicates these positions to the consumers. For \( k = 1 \), the item itself is used as position while for \( k = 2 \), it is calculated from the two items as discussed in Section 2.2.1 on page 16. A consumer thus only increases the count values using the calculated positions that are in the pages of its work queue. The full algorithm works as follows:

**Producer** A thread that is in producer mode continuously reads chunks of transactions; as long as a single chunk contains further transactions, it repeats the following steps:

1. Read a transaction from the input chunk.
2. For each size-\( k \) subset \( s \) of the transaction repeat the following steps:
   a) Obtain the partition id \( p_{id} \) and the count-table position \( c \) for \( s \).
   b) If the page of the local buffer assigned for \( p_{id} \) is full, then enqueue the page into the work queue assigned for \( p_{id} \) and get a new page from the empty-pages queue and assign it to the local buffer.
   c) Insert \( c \) into the page of the local buffer assigned for \( p_{id} \).

If the thread’s work queue is *not* empty, it switches to *consumer mode*. Otherwise, it reads the next chunk from the input data and the cycle starts again.

**Consumer** A thread in consumer mode is responsible for increasing the count values for all subsets that are in its partition. It repeats the following steps:

1. If the thread’s work queue is empty, switch back to *producer mode*.
2. Dequeue the next page \( p \) from the assigned work queue.
3. For each count-table position \( c \) in \( p \), increase the \( c \)-th value in the global count table.
4. Enqueue \( p \) into the empty-pages queue.

The algorithm finishes, as soon as there are no further transaction chunks and all pages in all work queues are processed. Clearly, non-empty pages in the local buffers must also be enqueued into the work queues before counting is finished. The questions that remain are (1) which partition function should be employed, (2) how to balance the load and distribute the partitions, and (3) how many partitions should be used.

Partitioning the items in the first iteration is performed based on the items themselves. The count-table position of an item \( i \) is inserted into the partition \( x \) using \( x = (i/256) \mod p \) where \( p \) denotes the number of partitions. The division is implemented using a shift instruction and is required to avoid false sharing that occurs when two consecutive count values are in different partitions but are in the same cache line. The item pairs are similarly partitioned in the second iteration: The count-table position of a pair of items \( i \) and \( j \) is inserted into the partition \( x = i \mod p \). By using only the first item for obtaining the partition id of an item pair, we reduce the calculation effort during counting since we only need to calculate the
partition id once for a set of subsets generated from a transaction, e.g., the subsets ac, ad, and af for a transaction acdf all fall in the same partition.

The partitions themselves are distributed in a round-robin manner. This may lead to load imbalance, which, however, is compensated by our processing model. All threads can always become producers for which load is always available. Only a single consumer should not be overloaded since their work queues would otherwise overflow—leading to contention for all threads inserting in such a queue. For this reason, it is always the thread’s highest priority to process pages of its work queue because this work cannot be distributed to other threads (at least as long as the partitions are not redistributed). It thus switches also to be a consumer when it is blocked by a full queue. Fortunately, the tasks of the consumer are cheaper than the tasks of the producer; this limits the possibility of queue contention.

Finally, the number of partitions, work queue sizes, number of floating pages, and size of a page influence the efficiency and overhead of the processing model. If, for example, the pages are too large and too many partitions are used, then the size of the local buffers explodes. We empirically found that 256 partitions, a page size of 32KB, a task queue with at most 256 entries and 1024 · t floating pages where \(t\) denotes the number of threads works best on our four-socket test system. These variables may be different on other systems.

Summing up, the processing model employed for direct counting is well suited for multi-processor systems. All threads can hold the parts of the global count table they are responsible for in the processor’s memory on which they are running. Only the pages in the work queues need to be exchanged between the processors for which the interconnects provide sufficient bandwidth. The processing model thus should scale very well to a large number of processors and threads. Nevertheless, the model’s major drawback is the effort required for partitioning the data. Many operations need to be performed per item whereas basically only a single memory request is required per item for sequential direct counting. This might limit its applicability for small count tables. For large count tables, however, the partitioning weakens this drawback because it increases spatial and temporal locality, which increases cache line utilization. This is also suggested by our experiments in Section 5.4.3.

5.2.2. Lookup-based subset counting

For subset counting in the third and later iterations, there exists a large number of algorithms. The most efficient among them employ lookup-based data structures. In what follows, we discuss state-of-the-art algorithms and discuss optimizations for them.

Counting using prefix arrays

As discussed in Chapter 2, Lucchese et al. [2004a] propose to employ direct counting also for the third iteration. Size-3 subsets are mapped to positions in a large lookup table, which basically provides a count value for each ordered combination of three frequent items. Although only a single memory reference is required to count a subset, this approach has unreasonable high memory requirements; the table comprises about 0.6GB already for 1000 frequent items. Besides, large lookup tables also exhibit a low cache hit ratio, which outweighs the benefits of the fewer memory references during counting. Hence, direct counting in the third iteration is only useful for a small number of frequent items where counting is typically not expensive.

Perego et al. [2001] use a similar approach with lower memory requirements. They use a prefix array, which holds a pointer for each ordered pair of frequent items. These pointers refer
5. cApriori

to values in a second large array, which contains the count values for the size-3 candidates. The memory referred between two pointers can be seen as a leaf node containing count values for all candidates that share the same first two items. Subset counting is basically performed by finding a subset’s leaf node and increasing its associated count value.

Figure 5.4: Prefix array with three leaf nodes

Figure 5.4 illustrates the data structure used by Perego et al. [2001] for 10 candidates and 7 frequent items. The prefix array comprises 15 pointers of which only three refer to leaf nodes. A single memory reference is sufficient to find a leaf node in which—if it exists—another memory request is required for increasing the subset’s associated count value. Notice that although we plot three separated leaf nodes, they are stored in a continuous array.

As can be divined from Figure 5.4, the subset counting approaches that rely on prefix arrays have usually large memory requirements because of the large array; it often consists of many pointers set to NULL. Furthermore, there were many useless subset checks performed. For example, the subsets \{3, 4, 5\}, \{3, 4, 6\}, and \{3, 5, 6\} are all checked even when there exists no candidate that starts with the item 3. Nevertheless, the checks might be performed quite fast so that it is difficult to predict the performance of this approach. For this reason, we implemented it (in what follows denoted as direct subset counting) and evaluate its performance in our experimental section.

Recursive subset counting using tries

Trie-based counting [Borgelt, 2003, Bodon, 2004] is another state-of-the-art subset counting method. Its basic idea is to organize all candidates within a prefix-hash tree [Coffman and Eve, 1970]. In the simplest variant of such a trie, each node consists of \(n\) pointers where \(n\) denotes the number of frequent items; the trie’s depth is given by the length of the candidates. Only the leaf nodes do not contain pointers; they hold integer values representing the count values of the candidates. Each candidate (with items ordered ascending) forms a path within the trie. At the root node, the first item identifies which pointer of the root node should be followed; the second item identifies which pointer of the next node should be followed; and so forth. The candidate’s last item identifies its associated count value at the leaf node. The transactions’ subsets (again ordered ascending) are counted in a similar fashion. Their items identify a path within the trie; if the path is available, then the subset’s last item is used to increase a count value within the leaf node.

To minimize the effort for subset counting, Borgelt [2003] proposed a recursive counting procedure, which ensures that each node is loaded only once when the subsets of a single transaction are counted. Figure 5.5 contains a code snippet that illustrates how the recursive subset counting function \texttt{count()} works when it is applied on the aforementioned trie. It takes four parameters: a trie node \texttt{node}, a pointer to a part of a transaction \texttt{ta}, the length
5.2. Support counting

```cpp
void count(Node * node, ItemType * ta, unsigned ta_length, unsigned depth) {
    // check whether the lowest level is reached
    if(depth == mDepth) {
        // count the occurrences within a leaf node
        for(int i = 0; i < ta_length; i++) {
            ((unsigned*)node)[str[i]]++;
        }
    } else {
        // traverse down the trie
        unsigned remain = ta_length - (mDepth - depth);
        for(int i = 0; i < remain; i++) {
            Node * succ = node->s_node[str[i]];
            if(succ != NULL) {
                count(succ, str + 1 + i, ta_length - 1 - i, depth + 1);
            }
        }
    }
}
```

Figure 5.5.: Code snippet for recursive subset counting.

of the part `ta_length`, and the current depth `depth`. The internal member variable `mDepth` denotes the length of the candidates. The function is called using the root node, the full transaction whose subsets should be counted, and `depth=1`. Basically, in each recursion call either all remaining items of a transaction are counted within a leaf node (`depth == mDepth`) or the function is further called for all non-empty suffixes of the passed transaction substring. The trie path used for a single recursion call is determined by the first item of a suffix. Whenever the path is not available, then the subsets of the respective suffix are not within the candidates.

Borgelt [2003] and Bodon [2004] employ the trie for indexing the candidates and—at the same time—for storing so far found frequent itemsets. Each node has therefore besides the `n` pointers also `n` count values, which denote the support of an itemset represented by the path to a count value. In each iteration, the trie is extended by a new level, which holds the candidates of this iteration. These new leaf nodes do not contain pointers until the next level is added in the next iteration. Borgelt [2003] further stores at each node two variables that denote the first item and last item within a node; they are used to avoid unused pointers within the nodes.

![Figure 5.6: Prefix-hash tree for indexing](image)
5. cApriori

Figure 5.6 illustrates an example trie used for counting subsets in the third iteration; the trie has a height of 3 and represents 10 candidates, which are also used in Figure 5.4. All nodes—except the leaf nodes in the third level—hold pointers and count values and represent so far obtained frequent itemsets. The frequent itemset \( \{d \rightarrow 4, e \rightarrow 5\} \), for example, is represented by the rightmost node and has a support of 321. Furthermore, the intervals above each node denote the items that are within a node, e.g., the root node contains only the items \( a \rightarrow d \) (1–4). The two rightmost nodes in the second level are not required for the actual subset counting in the third iteration. Such nodes are denoted as *dead-end paths* because they do not contain pointers to leaf nodes that represent the candidates of the current iteration. To avoid visiting them during counting, Borgelt [2004] marks them each using a single bit and Bodon [2004] copies these nodes in a separate list.

We implemented the tries of Bodon [2004] and Borgelt [2004] and evaluate their performance in our experiment section. We denote the former, which does not store intervals at the nodes, as *simple subset counting* and the latter as *advanced subset counting*. We employ in both counting methods Borgelt’s recursive subset counting procedure, which is much more efficient than counting methods that generate subsets beforehand and traverse the trie for each subset.

**Trie-improvements**

All three subset counting algorithms are already highly optimized and allow only minor improvements. In the following, we provide optimizations that have indeed a small impact on the sequential performance but may significantly reduce the algorithms’ memory footprint. A small footprint is highly beneficial for parallel processing because the threads often share the same caches. Hence, any memory reduction lowers the pressure on them and increases cache line utilization.

Our first optimization aims on removing unused data elements within the nodes that are in the tries used by simple and advanced subset counting. The basic idea is to remove the *count* fields and unused pointers to dead-end paths within the inner nodes since these are not required for counting in an iteration. For this reason, we divide the single trie into an *itemset trie* for storing so far found frequent itemsets and a *count trie*, which holds only the candidates of a single iteration and has *count* fields only in the leaf nodes. The former is thus solely used for candidate generation and is increased by a level whenever new frequent itemsets are obtained whereas the latter is solely used for support counting. The separation of the tries greatly reduces the number of unused data elements that are in the caches during counting. Only the trie has to be always rebuild before counting, which however is negligible compared to the costs for support counting when large datasets are mined.

The count trie after applying the optimizations.
5.2. Support counting

Figure 5.7 depicts the count trie that corresponds to Borgelt’s trie shown in Figure 5.6. The illustrated trie has only three instead of five inner nodes. The inner nodes are further smaller because they do not contain count fields and pointers to dead-end nodes. Thus, less memory is required, which improves the cache line utilization.

Besides removing unused fields within the trie used for counting, the size of the fields itself can be reduced. The nodes of a count trie consist of almost only pointers to child nodes. On a 64-bit machine (e.g., both of our two test systems), a single pointer requires 64 bit or 8 bytes and can address $2^{64}$ bytes of memory. The count trie, however, is typically much smaller so that 32-bit pointers or sometimes even 16-bit pointers are sufficient to address its nodes. For this reason, we provide the small-pointer optimization, where the size of the pointers is adjusted to the size of the trie. Each pointer then holds only an offset to a base address, which refers to the continuous memory area provided by the trie’s memory pool. Whenever a pointer needs to be dereferenced, it is computed by adding the offset to the base address.

Another optimization resembles to Borgelt’s interval optimization but can be applied to direct and simple subset counting without adding fields to their data structures. The basic idea is to reduce the size of the leaf nodes by avoiding count fields in them that are newer accessed. For example, the first three count values in the leaf node that holds candidates with the prefix $\{a \rightarrow 1, c \rightarrow 3\}$ are newer accessed because none of the subsets $\{a, c, a\}$, $\{a, c, b\}$, and $\{a, c, c\}$ is counted. Hence, the last value of each prefix leading to a leaf node denotes how many count values can be omitted within it; i.e., the leaf node array is shrunk by these values. To avoid runtime overhead during counting, we adjust all pointer to leaf nodes. When the first valid element of a leaf node is given by $n$, we subtract $n$ from the referring pointer. Hence, the correct addresses are automatically calculated during counting. We denote this optimization as overlapping count-array since the pointers overlap to other leaf nodes.

Finally, we provide an optimization that improves ILP in subset counting by avoiding the many function calls that are required when the recursive count() function is employed. We achieve this by transforming this function to multiple loop-based functions; each of these functions is tailor-made for a certain trie depth whereby it is sufficient to provide functions for tries with three up to six levels. The remaining iterations often amount to only a small fraction of the overall runtime on APRIORI-suited datasets, so the recursive count() function is sufficient in these iterations. Besides reducing the amount of function calls, the transformation enables the compiler to apply further optimizations. The optimization, however, may work better for small tries that fit almost within the caches. If the trie is very large, the latency of the memory requests dominates counting so that the avoidance of function calls has only a small impact.

Figure 5.8 illustrates the function count_depth3(), which is the loop-based variant of the recursive count() function (cf. Figure 5.5) for a count trie with three levels. As can be seen, there is a single loop for each of the three levels. The first loop iterates over the first $n - 2$ items of the transaction, where $n$ denotes its length; it traverses down to the second loop for all items that have a child node within the root node. This procedure is repeated in the second loop within the root’s child nodes that are indexed by $ta[i1]$. The third loop is used to count all subsets for which a child node was found within the root’s child node at $ta[i2]$.

---

4Rugina and Rinard [2001] employ a similar techniques to improve the performance of recursive divide and conquer programs.
void count_depth3( Node * root, ItemType * ta, unsigned ta_length )
for( int i1 = 0; i1 < ta_length - 2; i1++ ) {
    // start traversing the root node, i.e., the first level
    Node * node1 = root->s_node[ta[i1]];
    if( node1 != NULL ) {
        for( int i2 = i1 + 1; i2 < ta_length - 1; i2++ ) {
            // traverse the nodes of the second level
            Node * node2 = node1->s_node[ta[i2]];
            if( node2 != NULL ) {
                // increase the counters within the leaf nodes
                for( int i3 = i2 + 1; i3 < ta_length; i3++ ) {
                    ((unsigned*)node2)[ta[i3]]++;
                }
            }
        }
    }
}

Figure 5.8.: Code snippet for loop-based subset counting.

5.2.3. Parallel trie-based subset counting

For counting subsets using tries in parallel, we have basically the same options as for parallel
direct counting. The shared count-trie approach requires the least changes to the algorithms.
All threads traverse the same trie and increase the count values within the leaf nodes using
atomic increments or small critical regions. Clearly, this approach has again high synchroniza-
tion costs, which strongly limit its applicability. The replicated count-trie approach is
similar to the full replication employed by Jin et al. [2005]. The count trie is split into a
shared upper part and a replicated lower part. The shared part comprises all the inner nodes,
which are only read by the threads, while the lower part comprises the leaf nodes, which are
updated during counting. As mentioned before, the counting without synchronization comes
at the price of a multiple times larger memory consumption compared to the single-threaded
execution. For this reason, replicated counting is only useful when the number of leaf nodes
is rather low, i.e., for moderate or high $\xi$ values or in the later iterations. The partitioned
count-trie has the same memory footprint as the trie used for sequential subset counting be-
cause only a single trie is used. The count values in this trie, however, are partitioned. Each
thread has an assigned number of partitions in which it increases the count values without
synchronization. To avoid that all threads scan all pages, we use the previously described
multiple-producer/multiple-consumer scheme to exchange the positions of count values that
need to be increased by another thread. Recall that the processing scheme incurs overhead
but increases temporal and spatial locality.

Replicated counting in CAPRIORI is implemented as follows: a single memory pool is used
to provide memory for the shared upper part of the count trie. After the candidates are
inserted, all inner nodes are stored within the pool’s respective memory area and pointers
to inner nodes contain always the offset to the starting address of this area. Pointers to leaf
nodes, however, contain an offset to the address 0x0. Since the leaf nodes are basically only
a large array of integer values, this address can simply be exchanged to obtain a valid copy
of the leaf nodes. Hence, each thread adds the leaf offsets to the address of its own memory, which is obtained from the thread’s memory pool. Clearly, all count fields within this area have to be set to zero before counting.

Partitioned trie-based counting in cAPRIORI is implemented similar to partitioned direct counting. Only the partition criteria is different: All leaf nodes—and thus count values—are in the same partition \( x \) with \( x = i \mod p \), where \( i \) denotes the second last item of a candidate and \( p \) denotes the number of partitions. For example, the candidates \( acd, ace, \) and \( bcd \) are in the same partition whereas \( ade \) is in another partition.

5.3. Candidate and Database Pruning

Besides optimizing the underlying data structures and algorithms, subset counting can further be sped up by candidate and database pruning. As discussed in Chapter 2, candidate pruning reduces the number of candidates by exploiting the frequencies of the so far obtained frequent items or itemsets whereas database pruning trims or prunes transactions using the so far obtained candidates or frequent itemsets. Both methods may significantly reduce the effort for subset counting but incur itself overhead. This overhead clearly should not outweigh the reduced effort. For this reason, we propose and discuss in the following pruning techniques that are effective and produce only small overhead.

5.3.1. Candidate pruning

For APRIORI-suited datasets, we observe that often a large number of borderline candidates, which may only marginally satisfy a given \( \xi \) value, can be pruned when the information about the length of the filtered transactions is utilized during counting. The basic idea is as follows: If an item \( \alpha \) occurs in \( n \) transactions and it is frequent with \( n \geq \hat{\xi} \), then it cannot not be part of a frequent itemset when it occurs in \( t \) filtered transactions that contain only this item where \( \hat{\xi} > n - t \). For example, if \( \hat{\xi} = 100 \) and \( \alpha \) occurs in 120 transactions of which 30 contain only \( \alpha \) and thus no other frequent item, then \( \alpha \) is indeed frequent but can only occur in 90 transactions combined with other frequent items. Hence, none of these combinations can be frequent so that all candidates that contain \( \alpha \) can be removed.

Pruning borderline candidates using false-positive candidate items or itemsets works for candidates with an arbitrary number of items. In what follows, we focus only on pruning candidate 2-itemsets using false-positive candidate items because a candidate reduction is often most beneficial for direct counting, i.e., obtaining the frequent 2-itemsets.

The first step of borderline candidate pruning is to obtain the false-positive candidate items. This can be done by using a second frequency counter for each frequent item. Each of these counters holds the number of filtered transactions that contain solely the respective item and is maintained during the dataset conversion. Note that the counters cannot be maintained during the first scan because at this point the frequent items are not known and thus the transactions are not yet filtered. After the conversion, the false-positive candidate items can be obtained by subtracting the second counters from the first.

To get an early insight about the effectivity of this candidate pruning technique, we perform a small experiment where we use the four—already previously used—APRIORI-suited datasets and count the number of false-positive candidate items for \( \xi \) values that lead to few (around 50) and many frequent items (around 10,000).
5. cApriori

Table 5.1.: False positive frequent 2-candidate items on realistic datasets

<table>
<thead>
<tr>
<th></th>
<th>retail</th>
<th>kosarak</th>
<th>BMS-POS</th>
<th>wikilinks</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\xi$ (in %)</td>
<td>1.25</td>
<td>0.006</td>
<td>1.07</td>
<td>2.62</td>
</tr>
<tr>
<td># frequent items</td>
<td>50</td>
<td>10340</td>
<td>50</td>
<td>10102</td>
</tr>
<tr>
<td># false positives</td>
<td>11</td>
<td>41</td>
<td>2</td>
<td>176</td>
</tr>
<tr>
<td>candidate reduction (in %)</td>
<td><strong>39.2</strong></td>
<td>0.7</td>
<td>7.84</td>
<td><strong>19.0</strong></td>
</tr>
</tbody>
</table>

Table 5.1 illustrates the number of frequent items and false positives for the datasets retail, kosarak, BMS-POS, and wikilinks. We further plot the reduction of candidate 2-itemsets that is given by $100 \cdot (1 - \frac{(n-t)^2}{n^2})$ where $n$ and $t$ denote the number of frequent items and false positives, respectively. As can be seen, there is a high candidate reduction as long as the number of frequent items is small. Almost 40% of the frequent 2-candidates can be pruned on retail. For many frequent items, however, the possible candidate reduction is rather low. At best 3.6% of the candidates can be pruned on wikilinks. The main reason for the decreasing pruning effectiveness is that the number of transactions that contain only a single frequent item decreases with a decreasing $\xi$ value.

Unfortunately, the effectiveness of this candidate pruning technique is not as high as expected. Especially for a large number of frequent items, the candidate reduction is too small to speed up counting; the overhead for pruning may even outweigh the benefits. Contrary, candidate pruning works well for few items but usually it is not required in this case because the number of candidates is then already low. For this reason, we do not integrate this pruning technique in cAPRIORI. Nevertheless, it still might be useful in other scenarios, e.g., for other datasets with even smaller transactions, application areas with rather few frequent items, or in computing environments (e.g., GPUs) where main memory is more valuable.

5.3.2. Database pruning and trimming

As for all APRIORI-based algorithms, cAPRIORI scans in each iteration the converted database completely. Each filtered transaction that is removed using pruning thus reduces the scan time in the later iterations. Functions calls (i.e., to count()) are avoided and—more importantly—subset counting for such transactions is not required.

A simple transaction pruning technique is to remove transactions that are too small during an iteration. For example, the transactions with only two items are not required for obtaining the support of candidates with three and more items. cAPRIORI’s page-wise layout eases pruning of such transactions; all lists with pages that contain transactions with insufficient length are simply skipped during support counting. Since none of the skipped pages (and thus skipped transactions) is physically accessed, there is no overhead for parsing the transaction’s length, branching, or copying transactions. Such overhead might occur when pruning is applied to other data layouts. Notice that the pages with explicit length information could not simply be skipped, however, the number of iterations is typically smaller than $p_{\text{max}}$ so that these transactions will not be skipped anyway.

Although the memory of the skipped pages could also be released, it is not necessary because cAPRIORI’s peak memory usage is often reached after the frequent 2-itemsets are obtained and thus before pruning takes place. If they still should be released, the memory manager would have to be changed because mmap does not allow to release parts within a memory

114
mapped file. Assigning pages of the same size into own memory mapped files, however, would solve this issue.

Trimming transactions, i.e., removing items that are not required in later iterations from the transactions, is often even more beneficial than database pruning. Already a small decrease of the average transaction length has a large impact on subset counting, i.e., 120 size-3 subsets have to be counted for a transaction with 10 items whereas only 56 size-3 subsets have to be counted for a transaction with 8 items. However, reducing a transaction’s length from 5 to 4 items might not be useful so that some trimming techniques with perfect results but large overhead are rather counterproductive.

We observed that trimming is already very effective if it is based on the generated candidates of an iteration. After the candidates itemsets are generated, we exploit that not all frequent items occur in them in each iteration. On retail with $\xi = 0.106\%$, for example, 1998 items are frequent whereas only 166 of them occur in candidate itemsets of the forth iteration. Thereby, the largest of these 166 candidate items has the id 684. Recall that the ids are assigned to the frequent items based on their frequency; less frequent items get a larger id. For this reason, the largest id used in candidate itemsets—in what follows denoted as borderline id—is often much smaller than the largest id of the frequent items since the less frequent items have a higher probability to not occur in any candidate. Length trimming removes all items with an id larger than the borderline id from the transactions being counted because these items are not part in any of the iteration’s candidates and thus are never be counted, i.e., it removes all items larger than 684 from the transactions in our previous example. Dead-item trimming further removes all non-candidate items from the transactions, i.e., frequent items that occur in none of the candidate itemsets. Hence, the 519 of the non-candidate items that have an id smaller than the borderline id are also removed in our example.$^5$

We illustrate the effectivity of both trimming techniques using a small experiment. We run CAPRIORI on the four APRIORI-suited datasets using varying $\xi$ values. During support counting, we measure the average length of the transactions being counted. This includes also transactions that could be pruned after trimming. Most of the transactions are further counted multiple times, e.g., a transaction with more that three items would be counted three times when CAPRIORI requires three iterations to complete.

![Figure 5.9a: Effectivity of the trimming techniques](image-a)

![Figure 5.9b: Effectivity of the trimming techniques](image-b)

Figure 5.9a illustrates the average length of the transactions being counted on retail for the two trimming techniques and without trimming. For high $\xi$ values (i.e., $\xi \approx 5\%$),

$^5$The id of the most frequent item is 0.
trimming has no effect because the transactions are fairly short. As the $\xi$ value decreases, however, trimming significantly reduces the average transaction length, e.g., it decreases from 12.6 to 5.3 when dead-item is applied and $\xi = 0.02\%$. While length trimming has already a large impact on retail, it is not as efficient on kosarak (Figure 5.9b). We observed similar results on the BMS-POS and wikilinks dataset (not shown); trimming is always very effective on both datasets for low $\xi$ values whereas dead-item trimming reduces the average transaction length by at least 1 item more than length trimming.

We integrated both trimming techniques in cApriori. Length trimming is called before each count() call. It is implemented using a simple loop that runs backwards over a transaction and decreases the transaction’s length until an item occurs whose id is smaller than or equal to the borderline id. If the transaction is too small after trimming, it is skipped. We do not, however, materialize the new length because this involves copying the transaction and writing its new length back to memory; this might even more expensive than running the loop in each iteration again. Dead-item trimming is applied on a complete page before subset counting for the page’s transactions starts. It is performed using a small lookup table that stores for each frequent item whether it can be pruned or not. After the non-candidate items of a single transactions are removed, the remaining items are collapsed and also materialized. Collapsing is required so that the count() interface can be used without any changes. The materialization of the trimmed transactions differs for pages with fixed-length transactions and pages with variable-length transactions. In the former, the trimmed transactions remain in the page and are backfilled with MAX_INT values. During subset counting, we avoid counting these values using length trimming. We do not move trimmed transactions to other pages because this would complicate the page management and load balancing when multiple threads are used. Contrary, the pages containing variable-length transactions are rewritten. The trimmed transactions are stored one after another without fragmentation. Hence, the number of used bytes in a page decreases with each iteration. Transactions that can be pruned after trimming are removed from both page types.

In our experiments, we observed that both trimming techniques incur only small overhead. Even if only a few items can be pruned, the overall performance improves—or at least stays constant—with trimming. Hence, there is no need to activate or deactivate it using a threshold. As for transaction pruning, trimming could also reduce CApriori’s memory usage but there is not need for it. If it is still required, then pages that contain transactions of the same length could be merged and empty pages could be released.

5.4. Experiments

In this section, we present the experimental results for the algorithms proposed in this chapter. We evaluate CApriori’s single-threaded as well as its multi-threaded performance. For the former experiments, we compare CApriori with other algorithms that are of the same class, i.e., only with available, highly optimized Apriori-based algorithms, because—as discussed earlier—each algorithm class is superior over all other classes for certain datasets or $\xi$ values. We postpone a comparison with other frequent-itemset mining algorithms to Appendix C starting at page 195. The experiments in this chapter suggest that CApriori is up to an order of magnitude faster than the tested existing Apriori implementations. It further scales well on our large test system.
5.4. Experiments

5.4.1. Setup

In the experiments in this and the following chapters, we use a four-socket NUMA multiprocessor system that consists of four Intel E7-4830 processors, each equipped with eight cores and a 24MB last-level cache. Each processor has 32GB of main memory assigned so that 128GB are available for the complete system. The cores runs at a frequency of 2.13GHz and support dynamic frequency scaling. In favor of more consistent results, we turned this feature off and run the cores always with 2.13GHz. The E7-4830 processor supports all SSE instruction sets up to SSE4.2 and Hyperthreading. Hence, up to 64 threads can run in parallel.

We use a 64-bit linux (Ubuntu server 10.04) as operation system. As before, all algorithms were written in C++ and compiled using Intel Parallel Composer 2011; we used -fast as only optimization flag. OpenMP pragmas were used to enable thread-level parallelism. We implemented—similar as Orlando et al. [2003a]—most of our components using templates and select an appropriate variant at runtime depending on the number of frequent items. If, for example, the number of frequent items is below 257, then all filtered transactions can be represented using 8-bit arrays and the 8-bit versions of dataset conversion and support counting functions are selected. We measure in all experiments the wall clock time by using linux gettimeofday (within the code in our algorithms) or linux time (at process level for existing algorithms). We capture the memory consumption by adding the maximum resident set size of a process and the size of the system’s used swap space. In our algorithms, the former and latter are obtained using <sys/resource.h> and <sys/sysinfo.h>, respectively. For the competing algorithms under test, we use a small tool that periodically captures both values until the process finishes. In favour for more consistent timings, we increase the priority of the algorithms’ processes using linux nice.

In all experiments, we do not return the actual result itemsets and their respective count values; we only report the number of found frequent itemsets for each length. This does not influence the effort for mining because these numbers are only known after all frequent itemsets are found. It reduces, however, the number of system calls of competing algorithms that would occur when writing these sets into a file. Hence, it allows a more fair comparison of the algorithms. Furthermore, all datasets used for our experiments have the same ASCII format as it was used in the FIMI competition [Goethals and Zaki, 2004]. Item identifier are separated using a whitespace symbol while transactions form a single line separated using a carriage return symbol. Also the algorithms have no prior knowledge (e.g., number distinct items, longest transaction, etc.) about the datasets. As previously mentioned, we consider a scenario where all data is already loaded in main memory but itself is read-only. Hence, the file access is quite fast but the main memory cannot be used exclusively for the mining algorithm. For example, if the dataset is 100GB, then only 28GB are available for the algorithm’s internal data structures on our particular system. To provide a fair starting situation for all competing algorithms, we store the datasets within a RAM disk. We thus do not need to change their file interfaces—i.e., whether they use mmap, read, or gets—for employing them to load the datasets with in-memory speed. Clearly, some of the interfaces are more efficient than the others. For low $\xi$ values, however, reading a cached dataset often amounts only for a small fraction of overall runtime. In our algorithms, we always load the data using linux mmap because it eases access to a file and is among the fastest file interfaces.

We created the RAM disk using linux ramfs, which is a temporary block-based filesystem. It uses only as much main memory as the size of the files that are stored within it—in our case the single dataset. Basically, the performance for accessing a file is equal as if the file is
5. cApriori

cached within linux. One issue of ramfs, however, is that it does not support an NUMA-aware memory allocation. Linux allocates randomly available physical pages from the memory of all four processors while a file is copied into the RAM disk. Often, the complete memory of a single processor is occupied, which influences the performance of our parallel algorithms since then, they cannot apply an NUMA-aware memory allocation strategy anymore. For this reason, we implemented a memory blocker tool that allows us to occupy memory before creating the RAM disk. For example, before loading a 100GB dataset into the RAM disk, we block 7GB in each of the four local memories. Linux then allocates equally across all local memories the remaining 100GB for the RAM disk. Thereby, linux mlock is used to prevent swapping blocked memory to disk. After the dataset is loaded, the blocked memory is freed and we run the frequent-itemset mining algorithm under test.

Before a mining algorithm can read the transactions of a dataset, the dataset must be mapped into the process address space, i.e., the ramdisk’s physical pages that contain the dataset are mapped into the virtual address space of the process. For large datasets with several hundred gigabytes, this mapping procedure can take several tens of seconds because linux maintains the page tables sequentially. It requires, for example, about 9 seconds to map a 100GB dataset. The mapping forms a large sequential fraction and thus may limit—depending on the chosen $\xi$ value—the maximum speedup of the mining algorithm. Furthermore, the page table requires a noticeable amount of memory, e.g., already 200MB are required for a 100GB dataset. The main reason for the large mapping time and large page table is that linux maintains the page mapping at a 4KB granularity. For each 4KB page, there is an entry in the page table. Huge pages, i.e., pages with up to 2MB, can reduce the number of such entries but they cannot be used for the ramfs ramdisk. In what follows, we always state explicitly whether the mapping time is included or not. In each experiment, we measure it by passing the flag MAP_POPULATE to mmap to populate the page tables during mmap’s invocation.

Datasets

We use several realistic and synthetic datasets for evaluating the different frequent-itemset mining algorithms in this and the following chapters. Unfortunately, the realistic datasets, which are discussed in Chapter 4 on page 85, are all very small. The datasets retail, kosarak, and accidents, for example, fit together in the cumulated L3-cache (4 × 28MB) of our test system. Furthermore, even the largest dataset—the webdocs dataset—comprises only 1.4GB of memory. Hence, mining the freely available realistic datasets is not challenging. For this reason, we use these datasets mainly for the non-performance experiments.

To provide larger datasets for our performance experiments, we generated synthetic datasets using the IBM Quest Dataset Generator [Agrawal and Srikant, 1997]. This generator allows to build datasets of arbitrary size: thereby the dataset’s characteristics (e.g., distinct items, average cardinality) can be controlled to obtain more realistic datasets. We used this feature to build large datasets that follow the same characteristics of the realistic datasets while containing more transactions. For example, the quest-retail dataset has the same average cardinality and number of distinct items as the retail dataset but it has about 25000x more transactions, i.e., instead of 4MB it comprises 100GB. Despite the same characteristics,

---

6The blocked memory plus the dataset should be smaller than the available memory because the system itself requires main memory for the operating system. Otherwise, thrashing might occur—leading to an unusable system state.

7Obtained via the proc filesystem using cat /proc/meminfo | grep PageTables.
the synthetic datasets behave differently compared to the real-world datasets because the generator is based on a simple model. For this reason, we build a tool that allows us to combine a realistic with a synthetic dataset. It exchanges the \( n \) most-frequent items of each transaction of the synthetic dataset by the \( n \) most-frequent items of a randomly chosen transaction from the respective realistic dataset. Thereby, \( n \) is set to three quarters of the average transaction cardinality but at most to 25. For example, we exchanged the 7 most-frequent items of each quest-retail’s transactions because this dataset has an average transaction cardinality of 10. More details about characteristics of the real-world and synthetic datasets can be found in the appendix (see Section A.1).

5.4.2. Single-threaded

We start by evaluating the single-threaded performance of cApriori. The purpose of the experiments is to show (1) the benefits of our optimizations and transaction pruning techniques and (2) that our algorithms are competitive with existing Apriori implementations. The latter is a precondition for multi-threaded cApriori because running an algorithm with multiple threads is not reasonable when it is single-threaded several orders of magnitude slower than an other single-threaded algorithm that solves the same task.

As mentioned earlier, Apriori-based algorithms—and thus also cApriori—usually perform best on datasets that consist of mainly short transactions. We furthermore observe that FP-Growth often outperforms these algorithms when the obtained frequent itemsets become longer and the overall number of frequent itemsets gets large, i.e., for very small \( \xi \) values. Hence, we designed the following experiments so that we stay in the sweet spot of Apriori-based algorithms. We use rather larger \( \xi \) values, which leads to result sets that contain mainly short frequent itemsets, and use datasets that consist of many small transactions. The synthetic datasets quest-retail-short, quest-kosarak-short, quest-BMS-POS-short, and quest-wikilinks-short are well suited for this task. They are based on the similarly-named realistic datasets, each comprise 4GB, and have an average transaction length of less than 20 items.

Subset counting

In the first set of experiments, we evaluate the different subset counting algorithms and the impact of the discussed optimizations. We compare the simple, advanced, and direct subset counting algorithm. For the former two algorithms, we implemented a recursive and a loop-based variant whereas direct is realized only loop-based. We furthermore implemented for all three algorithms the small-pointer and overlapping count-array optimization.

We start by evaluating the impact of the trie optimizations. For this purpose, we run cApriori for each of the three subset counting algorithms on the aforementioned four datasets and apply the optimization techniques one by one. We start measuring the runtime when no optimization is applied and then incrementally apply both optimizations and finally exchange the recursive with the loop-based variant (for simple and advanced only). We plot only the time for subset counting—i.e., for the third and later iterations—for two different \( \xi \) value setups. In the high \( \xi \) values setup, subset counting amounts on each dataset to about 25\% of the overall runtime; this fraction increases to 75\% in the low \( \xi \) values setup. Hence, the optimizations have a higher impact in the second setup.
5. cApriori

![Normalized performance graph for cApriori with different optimizations on various datasets]

(a) High ξ values

(b) Low ξ values

Figure 5.10.: Normalized execution time of SIMPLE with different enabled optimizations on various datasets

Figure 5.10a illustrates the normalized runtime reduction on SIMPLE when the high ξ values are used. As can be seen, the small-pointer and overlapping count-array optimization have only a small impact on the subset counting performance. For quest-wikilinks-small and ξ = 0.27%, they even increase the runtime because the trie is very small, i.e., it is in each iteration smaller than 140KB and fits in the processor’s L2-cache. Hence, the additional address calculation overhead of the small-pointer optimization is not amortized by fewer cache misses. Contrary, loop-based SIMPLE is always faster than recursive SIMPLE. It is most beneficial on quest-kosarak-small where it achieves a speedup of 1.4x when ξ = 0.02%.

The small-pointer and overlapping count-array optimization are slightly more effective when low ξ values are used (see Figure 5.10b) because the trie is then much larger than before. Hence, the memory reduction of about 10% to 40% (for both optimizations) leads to less cache misses. The speedup of the loop-based SIMPLE variant, however, decreases to at best 1.12x. As the counting data structures grow, more time is spend per memory access, which almost outweigh the fewer function calls. Nevertheless, the speedup remains constant as ξ is further decreased.

We obtain similar results also for DIRECT and ADVANCED. In general, the overall runtime reduction is rather low for the optimizations. Single-threaded CAPRIOI is in average about 10% faster when all three optimizations are enabled. As soon as multiple threads are used, however, the optimizations are much more beneficial. Often many threads share the same caches; e.g., up to 16 threads share a common L3-cache on our four-socket test system. Any memory reduction thus reduces the pressure on this cache. Also fewer cache lines have to be transferred between the processors when the shared trie is smaller.

We next compare the runtime of three subset counting algorithms directly with each other. We enable all discussed optimizations for each of them and again use the four datasets. Besides the algorithms’ full runtime, we also plot the time required for (1) obtaining the frequent items and the dataset conversion and (2) obtaining the frequent 2-itemsets in the second iteration.

Figure 5.11a illustrates the results for the quest-retail-small dataset. As can be seen, frequency counting and the dataset conversion amount for a large fraction of the overall runtime for large ξ values (i.e., ξ > 0.05) but play only a minor role for small ξ values. The time required for obtaining the frequent 2-itemsets grows as ξ decreases and has a large impact on the overall runtime for ξ < 0.05. The execution time of the three subset counting algorithms varies greatly. ADVANCED shows always the best performance and is for certain ξ...
5.4. Experiments

Figure 5.11.: Execution times of the different subset-counting algorithms on various datasets

values almost twice as fast as DIRECT, which suffers from many unsuccessful lookups and—as we will show—a much higher memory consumption. SIMPLE’s runtime is always between DIRECT and ADVANCED; it requires less memory than the former but more than the latter subset counting algorithm. Clearly, a lower memory footprint leads to a better cache usage, which significantly improves the algorithms runtime. ADVANCED further benefits from the annotated intervals at its nodes; they reduce the number of iterations within a count call.

We obtain basically the same results for quest-kosarak-small as shown in Figure 5.11b. Again, obtaining the frequent 2-itemsets has a large impact on the performance and ADVANCED performs always best. On the datasets quest-BMSPOS-small (Figure 5.11c) and quest-wikilinks-small (Figure 5.11d), subset counting always amounts to a large fraction of the overall runtime. In which iteration the most time is spend, however, differ between the datasets. On quest-BMSPOS-small most time is spend during the third iteration because there is a very large number of candidate 3-itemsets; quest-wikilinks-small has long transactions so that the later iterations are very costly. Again, ADVANCED is superior on both datasets in terms of memory consumption and runtime. SIMPLE and DIRECT perform similar; they the have almost the same memory footprint, which is dominated by a large count array.

Since subset counting is heavily influenced by the memory usage of the data structures, we measured it for the trie and the respective count array in each iteration. On quest-retail-small, quest-kosarak-small, and quest-BMS-POS-small, the data structures require at most memory during the third iteration, i.e., obtaining the frequent 3-itemsets. For quest-wikilinks-small the peak is reached between the fifth and eighth iteration (depending on the chosen \( \xi \) value) because of the dataset’s long transactions.
Table 5.2. Size of the largest count-trie during mining (in MB)

<table>
<thead>
<tr>
<th>dataset</th>
<th>quest-retail-small</th>
<th>quest-kosarak-small</th>
<th>quest-BMS-Pos-small</th>
<th>quest-wikilinks-small</th>
</tr>
</thead>
<tbody>
<tr>
<td>ξ</td>
<td>0.05%</td>
<td>0.02%</td>
<td>0.01%</td>
<td>0.55%</td>
</tr>
<tr>
<td>SIMPLE</td>
<td>2.9 780</td>
<td>5.5 1691</td>
<td>2.4 85</td>
<td>0.1 6.8</td>
</tr>
<tr>
<td>ADVANCED</td>
<td>0.1 292</td>
<td>0.1 928</td>
<td>0.3 46</td>
<td>0.1 0.6</td>
</tr>
<tr>
<td>DIRECT</td>
<td>15.1 1577</td>
<td>45.8 5631</td>
<td>11.3 90</td>
<td>0.1 3.8</td>
</tr>
</tbody>
</table>

Impact of database pruning

In the next set of experiments, we evaluate the impact of our two trimming techniques, length trimming (LENGTH) and dead-item trimming (DEAD-ITEM), on the runtime of the three subset counting algorithms. For that purpose, we use the four APRIORI-suited synthetic datasets and measure on them CAPRIORI’s overall runtime for various ξ values. We further use the respective realistic datasets because we observe a significantly different pruning effectiveness between the synthetic and realistic datasets. We choose the ξ values so that they are (1) rather small because pruning and trimming is then typical much more effective but (2) still in the sweet spot of APRIORI-based algorithms. The chosen ξ values differ even between a realistic dataset and its synthetic counterpart because they have different sweet spots. Since pruning only affects the subset counting procedure, the time required for the other phases (i.e., obtaining the frequent items, dataset conversion, and obtaining the frequent 2-itemsets) is constant. We mark this time using a dashed line in the following figures.

Figure 5.12a depicts APRIORI’s overall runtime for the three subset counting algorithms on kosarak with ξ = 0.2%. Here, subset counting amounts to the largest fraction of the overall runtime. As for the previous experiments, ADVANCED performs better than SIMPLE and DIRECT when trimming is not enabled. The latter algorithms, however, benefit more than ADVANCED from trimming. If length trimming is enabled, their overall runtime reduces to about 80%; it further reduces to 60% when dead-item trimming is enabled. Contrary, ADVANCED does not benefit from length trimming; it even slightly slows down because of the additional trimming overhead, which does not pay off since ADVANCED already internally trims the transactions using the intervals at the nodes. Nevertheless, if dead-item trimming is enabled, it is 15% faster than without trimming. Despite the lower improvements, ADVANCED is still the fastest of the three subset-counting algorithms.

We obtain similar results for the BMS-POS dataset—see Figure 5.12c. Again, most of the overall runtime is spend for subset counting. On this dataset, however, trimming is less ben-
5.4. Experiments

Subset counting algorithm
Overall runtime (in seconds)

0
1
2
3
4
5
6

simple advanced direct
without
length
dead-item

(a) kosarak
(\(\xi = 0.2\%\))

(b) BMS-POS
(\(\xi = 0.16\%\))

(c) wikilinks
(\(\xi = 0.52\%\))

(d) quest-kosarak-small
(\(\xi = 0.0047\%\))

(e) quest-BMS-POS-small
(\(\xi = 0.019\%\))

(f) quest-wikilinks-small
(\(\xi = 0.38\%\))

Figure 5.12.: Runtime improvements for trimming on various datasets

Efficient because BMS-POS has fewer distinct items and smaller transactions than kosarak; i.e., the former and latter have 6.5 and 8.1 items per transaction, respectively (cf. Appendix A.1 on page 189). Hence, less items can be pruned before subset counting. The overall runtime reduces to 74\% for simple and 77\% for direct when dead-item trimming is employed. Advanced does not benefit from both trimming techniques.

The results for wikilinks and \(\xi = 0.52\%\) are depicted in Figure 5.12c. On this dataset, the phases before subset counting consume roughly two-third of the overall runtime when trimming is not enabled. As for the previous datasets, length trimming has only a positive effect on simple and direct whereas dead-item trimming is beneficial for all three subset-counting algorithms—again it is most beneficial for simple and direct. The effect of trimming further increases as the \(\xi\) value is decreased. For lower \(\xi\) values, however, we leave the sweet spot of APriori-based algorithms because the average transaction length of wikilinks is higher than for the other datasets.

Trimming is not as beneficial on the synthetic datasets as it is on the realistic datasets. On quest-kosarak-small with \(\xi = 0.0047\%\) (Figure 5.12d), most of the time is spent for counting in the second iteration, i.e., for obtaining the frequent 2-itemsets. In the later iterations, only a few frequent items have an id smaller than the borderline id while many of them have a larger id. For this reason, only length trimming considerably reduces the runtime on simple and direct because—as before—advanced already benefits from implicit trimming. Dead-item trimming is more beneficial in the later iterations, which only have minor impact on the overall performance. Therefore, it reduces the runtime by at most 1\%.

On quest-BMS-POS-small (Figure 5.12e), trimming has only a minor impact because this dataset has small transactions as its realistic counterpart. The impact of trimming is similar
on quest-wikilinks-small (Figure 5.12f). Although it works well on this dataset, subset counting only amounts to a small fraction of the overall runtime. As for wikilinks, it is not useful to decrease the $\xi$ value since then the transactions get too long for CAPRIOI.

We observe no improvement for trimming on retail (not shown). This dataset has too few transactions that even the candidate generation between the iterations and the trie generation have a serious impact on the overall performance. The results for the respective synthetic dataset, quest-retail-small, are similar to them of quest-kosarak-small; i.e., only length trimming is considerably beneficial for SIMPLE and DIRECT.

To summarize, our transaction trimming and pruning techniques reduce almost always the time required for subset counting; the impact on the overall runtime, however, differs strongly between the used datasets and $\xi$ values. Database pruning is typically more effective (1) for datasets with long transactions and (2) for low $\xi$ values that lead to high subset counting effort in the later iterations. Unfortunately, such datasets or very low $\xi$ values are often not the sweet spot of APRIORI-based algorithms. The impact of database pruning is thus rather limited, raising the question whether more elaborate pruning or trimming techniques are even useful. Of the three subset counting algorithms, ADVANCED benefits least of transaction trimming; it does already some implicit pruning in the count() function using the annotated intervals. For this reason, the performance of SIMPLE and DIRECT gets closer to the performance of ADVANCED; SIMPLE is sometimes on par with ADVANCED whereas DIRECT is always the slowest algorithm. Nevertheless, ADVANCED is almost always the most efficient subset counting algorithm. Therefore, we make it the standard subset counting algorithm in CAPRIORI.

**Comparison with existing Apriori implementations**

In the next set of experiments, we compare CAPRIORI with publicly available APRIORI implementations. We use the same datasets as used in the previous experiments. In CAPRIORI, we enable all optimizations, use ADVANCED as subset counting algorithm, and employ dead-item pruning. As competitors, we use two highly optimized sequential APRIORI implementations from Bodon [2003] and Borgelt [2004].

Bodon provides several versions of its APRIORI implementation. In our experiments, we use Version 2.1.3 obtained from the FIMI repository [FIMI, 2004] because the newer versions—even though they have an up to 25% smaller memory footprint—are up to 1.85x slower than this version.\(^8\) The implementation itself (in what follows referred to as APRIORI-BODON) relies on C++ standard data structures and uses the standard allocators for memory allocation. It parses the full dataset before subset counting starts and stores all transactions in a tree in which duplicate transactions are stored only once. The tree is realized using a std::map and the filtered transactions are represented using 32-bit integer values that are stored in arrays of the type std::vector. The employed parser is rather inefficient because it uses the getc file interface to read the dataset. Borgelt’s APRIORI implementation (APRIORI-BORGELT) is implemented in C and uses the standard allocators malloc and realloc for memory allocation. After starting, it converts the complete dataset into an internal representation to avoid the overhead for a repeated parsing\(^9\) of the dataset. Thereby, the transactions are not filtered because the frequent items are first obtained after the conversion. Before subset

\(^8\)All of Bodon’s APRIORI versions can be obtained from http://www.cs.bme.hu/~bodon/en/apriori/.

\(^9\)Apriori-Borgelt’s parser can handle many different file formats, user defined item and transaction separators and is therefore much more expensive than a specific parser that works only a single file format.
counting starts, `APRIORI-BORGELT` sorts and filters the transactions and builds a trie-based representation of them. Subset counting is then solely performed on this trie, which greatly reduces the time for counting. We further tried Goethal’s `APRIORI` implementation\(^{10}\), which also relies on C++ data structures. It parses the dataset in each iteration completely using `getc` and is therefore multiple times slower than the other implementations. For this reason, we do not provide results for this implementation.

Figure 5.13a illustrates the execution time of the algorithms under test on `quest-retail-small`. As can be seen, `CAPRIORI` is almost always the fastest of the three algorithms for the chosen \( \xi \) values. For \( \xi = 0.042\% \), it requires only 68s to finish whereas the better of both other algorithms requires 687s; this amounts to a speedup of 10x. Compared to the other algorithms, `CAPRIORI`’s strengths are its phases before the actual mining, i.e., the tasks before the expensive subset counting. Obtaining the frequent items and the dataset conversion takes only 30s (cf. Figure 5.11a). Contrary, `APRIORI-BORGELT` requires 460s before it starts mining the frequent itemsets. This time includes, however, all the steps to build the trie-based transaction representation, which greatly speeds up subset counting and leads to an almost constant total execution time until very small \( \xi \) values are used. The execution time even decreases on this dataset for \( \xi < 0.024\% \)—caused by caching effects—until it increases again to 1100s for \( \xi = 0.001\% \) (not shown). `APRIORI-BORGELT`’s efficient subset counting is also the reason that it is faster than `CAPRIORI` for \( \xi < 0.003\% \). `APRIORI-BODON` is always slower than `CAPRIORI` but performs better than `APRIORI-BORGELT` for \( \xi > 0.04\% \). It has high costs\(^{10}\)\end{footnote}

\(^{10}\)The implementation was obtained from [http://adrem.ua.ac.be/~goethals/software/](http://adrem.ua.ac.be/~goethals/software/).
5. cApriori

![Graph](image)

(a) quest-retail-small  
(b) quest-kosarak-small

Figure 5.14.: Memory footprint of cAPRIORI and the existing APRIORI implementations during subset counting because it does not apply any pruning or trimming technique and has a rather inefficient recursive subset counting procedure. Furthermore, the initial parsing and conversion of the dataset is also much slower than for cAPRIORI because it uses the standard C++ allocators.

We observe similar results on the datasets quest-kosarak-small (Figure 5.13b), quest-EMS-POS-small (Figure 5.13c), and quest-wikilinks-small (Figure 5.13d). On all three datasets, cAPRIORI usually performs best and achieves speedups of up to 16x within the measured $\xi$ value range. Only for very low $\xi$ values, the competing algorithms show a better performance. APRIORI-BORGELT is faster than cAPRIORI on quest-kosarak-small as $\xi$ gets smaller than 0.00035%. On quest-wikilinks-small, APRIORI-BODON performs best for $\xi < 0.15\%$ whereas it is usually the slowest algorithm on all other datasets.

Besides the execution time of the algorithms, we evaluate their memory usage during mining. Figure 5.14a illustrates the peak memory consumption on quest-retail-small of the three algorithms under test. As can be seen, cAPRIORI has the lowest memory footprint; it requires for the chosen $\xi$ values always less than 2.1GB of main memory. Most of the space is occupied by the filtered transactions. For $\xi = 0.01\%$, for example, about 1.6GB are required for the filtered transactions whereas only 0.5GB are required for the data structures during mining. APRIORI-BODON has much higher memory requirements because of its transaction trie; the quest-retail-small dataset has many small transactions with a rather small number of duplicates so that this trie is not beneficial. APRIORI-BODON has only for $\xi > 0.042\%$ a smaller memory footprint than APRIORI-BORGELT and approaches for small $\xi$ values 15GB of main memory. APRIORI-BORGELT has also high memory requirements because of its initial dataset conversion. It thus requires on quest-retail-small always 5.7GB of main memory even if the result set does not contain any frequent item. Additionally to the internal representation of the dataset, APRIORI-BORGELT’s transaction trie grows as the $\xi$ value decreases so that the algorithm’s overall memory usage approaches 11GB.

We obtain similar results on quest-kosarak-small—see Figure 5.14b. Again, cAPRIORI has the smallest memory footprint but requires on this dataset up to 4GB of memory when $\xi \approx 0.005$; 1.8GB are then occupied by the filtered transactions and 2.2GB are used during mining. More specifically, the count array for obtaining the frequent 2-itemset from 32,176 frequent items requires roughly 2GB. Hence, it is responsible for the peak memory consumption. APRIORI-BODON and APRIORI-BORGELT have for the same reasons a higher memory consumption than on quest-retail-small.
5.4. Experiments

We observe—independently of the $\xi$ value—an almost constant memory usage for all three algorithms on quest-BMS-POS-small (not shown). CAPRIORI again has the lowest memory footprint and requires only 2GB of main memory of which—except for 47MB—all is used for storing the filtered transactions. Contrary, APRIORI-BODON and APRIORI-BORGELT have a high peak memory usage of 14GB and 12GB, respectively. Both algorithms suffer from the many small transactions, which introduce large overhead when they are stored in their data layouts. CAPRIORI’s page-wise layout, however, is well suited for storing such transactions.

All three algorithms have an almost constant or slowly growing memory usage on the quest-wikilinks-small dataset for the used $\xi$ values (not shown). On this dataset, however, APRIORI-BODON has always the best memory footprint with only about 190MB. The dataset seems to have many duplicate transactions, which are stored only once in APRIORI-BODON’s transaction tree. CAPRIORI requires up to 400MB of which most is used to hold the converted database. APRIORI-BORGELT’s footprint is 4.7GB since it initially converts the full dataset.

To summarize, CAPRIORI is competitive with existing highly efficient APRIORI-based implementations. Moreover, it has a superior runtime and memory footprint for reasonable $\xi$ values and thus forms a meaningful basis for a multi-threaded mining algorithm.

5.4.3. Multi-threaded

In the following, we provide the results of the evaluation of multi-threaded CAPRIORI. We start the discussion with the evaluation of the three parallel direct count approaches, which are also used in the multi-threaded versions of CECLAT and CFPGROWTH (see Chapter 6 and 7). We then evaluate the performance of parallel direct counting for obtaining the frequent 2-itemsets in CAPRIORI. Finally, we illustrate CAPRIORI’s overall runtime and memory footprint when it runs using multiple threads. We do not compare our implementation to other parallel algorithms because there are no multi-threaded APRIORI versions publicly available.

In all of our mining algorithms, we need to parse the input dataset twice. Although, sequential parsing is straightforward, parallel parsing is slightly more involved. In our scenario, all datasets are stored in the previous mentioned ASCII format where each line denotes a transaction. For parallel parsing, the dataset is partitioned into multiple equal-sized chunks (e.g., chunks of 1MB). Single item identifiers as well as transactions may thus span over two chunks. Hence, the item identifier or transactions might be parsed partially if all chunks are processed independently. For this reason, we always start parsing after the first separator symbol within a chunk and continue parsing until the first separator symbol of the subsequent chunk occurs. For parsing the item-identifier within the first scan, a thread starts within its assigned chunk and stops within the subsequent chunk at a symbol that separates two item identifier, i.e., carriage return or blank symbol. Parsing transactions within the second scan works similarly, however, the separator symbol is a carriage return.

**Parallel direct counting**

In the first set of the multi-threaded experiments, we evaluate parallel direct counting used for obtaining the frequent items of a dataset. For that purpose, we measure the performance of the shared (SHARED), replicated (REPLICATED) and partitioned (PARTITIONED) table approach on different datasets and a varying number of threads. Recall that the performance of parallel counting depends on the number of distinct items a dataset contains. For this reason, we use
three datasets that are representative for many datasets: \texttt{quest-retail}, \texttt{quest-netflix} and \texttt{quest-webdocs} have 17 thousand, 480 thousand, and 5.2 million distinct items, respectively.

All three parallel counter rely on our memory pools to allocate memory for the count tables. The underlying \texttt{mmap} function returns zero-initialized memory so that further initialization is not necessary. Hence, large arrays can be allocated that occupy only as many physical space as is touched during processing. The physical space required by \texttt{shared} and \texttt{partitioned} is as large as the count table with \( n \) entries where \( n \) denotes the largest distinct item being counted. For \texttt{replicated}, multiple multiple memory pools are used. As mentioned before, all parallel counter process the dataset in batches; the threads iteratively gather and process a single batch until all of them are processed. Afterwards, the frequent items are obtained from the count table (or count tables for \texttt{replicated}) and subsequently they are sorted with respect to their frequency. Based on these frequencies, lookup tables that map the dataset’s identifier to internal identifier and vice versa are build. Except for the parallel counting, all other steps are processed by only a single thread. This was found to be sufficient because they amount to only a small fraction of the overall runtime.

Figure 5.15a illustrates the runtime of \texttt{replicated} and \texttt{partitioned} on the \texttt{quest-retail} dataset without the time required for dataset mapping. For this dataset, each table requires only 68KB, so it fits into the dedicated L2-caches of the processor’s cores. Therefore, \texttt{replicated} scales very well. A single thread processes the complete dataset in 272s while 32 threads require only 9.6s for the same task. This constitutes a speedup of 28.3x. Hyperthreading has only a small impact on the performance; with 64 threads the runtime slightly decreases to 7.2s. Counting is thus faster than mapping the dataset, which takes 9s. Independently of the number of threads, \texttt{partitioned} has a much higher runtime than \texttt{replicated} because it suffers from the overhead of the producer–consumer scheme, i.e., for distributing the count values to the respective threads. \texttt{partitioned} requires 361.5s for the complete dataset using only a single thread and 20.4s (17.3s) for 32 (64) threads. \texttt{shared} is not competitive at all (not shown). The overhead for the atomic increments increases the single-threaded counting time to 1285s. With more threads, the counting time increases even further.

On the \texttt{quest-netflix} dataset, \texttt{partitioned} scales better than \texttt{replicated}—see Figure 5.15b. On this dataset, a single count table is about 1.8MB and still fits for \texttt{partitioned} into the L2-caches because it is stored partitioned within the caches of multiple cores. For \texttt{replicated}, however, each core has to hold the full 1.8MB so that the L2-cache is not sufficient anymore. For more than 16 threads, the count tables of all threads even do not fit
into the 32MB L3-cache. For this reason, \textsc{replicated} scales only well up to 8 threads. In this range, it is faster than \textsc{partitioned}. For 16 and more threads, the reverse is true. For example, \textsc{partitioned} is 2.5x faster than \textsc{replicated} for 64 threads. Again, \textsc{shared} has unreasonable high execution times that even increase with more threads.

The results are similar on \textsc{quest-webdocs} (not shown). Interestingly, \textsc{replicated} performs better than on \textsc{quest-netflix} despite the larger tables. This can be explained with the better locality of the items within the dataset. Some items occur very often whereas other occur rarely. Nevertheless, \textsc{partitioned} is always about 1s faster than \textsc{replicated} for 32 and 64 threads. It requires 17.8s for mining the full dataset with 64 threads.

To summarize, \textsc{replicated} usually performs best on datasets that have less than a few 100 thousand distinct items whereas \textsc{partitioned} performs best in all other cases. As expected, \textsc{atomic} is never competitive because of the expensive atomic instructions. Hence, the selection of the best counter solely depends on the number of distinct items of the dataset being mined. Without any prior available statistics, a small sample of the dataset could be used to get an estimate about the number of distinct items. Also, one may switch from one counter implementation to the other during runtime depending on the so far obtained distinct items. In the algorithms of this thesis, we did not implement such an automatic counter selection. We always select the best counter by hand—using a flag. We furthermore always select \textsc{replicated} in experiments where only up to 8 threads are used.

**Obtaining the frequent 2-itemsets**

In the next set of experiments, we evaluate parallel direct counting used to obtain the support of the frequent 2-itemsets within \textsc{apriori}'s second iteration. We implemented only the replicated (\textsc{replicated}) and partitioned (\textsc{partitioned}) count table approach since the shared table approach was already not competitive for obtaining the frequent items where the count tables are less frequently accessed than for counting the frequent 2-itemsets. We further use the same datasets as used in the single-threaded experiments to (1) keep the time for the experiments small and (2) avoid thrashing, which might occur for \textsc{replicated} when the count tables approach several hundred megabytes. We expect, however, the same results for the datasets' large counterparts.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure5.16.png}
\caption{Parallel direct 2-counting on \textsc{quest-retail-small}}
\end{figure}

Figure 5.16a illustrates the runtime for obtaining the frequent 2-itemsets on \textsc{quest-retail-small} with $\xi$ in the range from 0.10\% to 0.02\% and 32 threads. We further plot the runtime of
REPLICATED for 64 threads and time required for obtaining the frequent items and the dataset conversion. As can be seen, REPLICATED is only faster than PARTITIONED for $\xi > 0.05\%$. For such $\xi$ values, the number of frequent items is below 1300, which limits the number of size-2 candidates to 844,350; a single count table thus requires less than 3MB so that the size of the replicated count tables remains acceptable. In this $\xi$ range, however, counting the size-2 candidates accounts for only a small fraction of the overall runtime. As $\xi$ further decreases, more frequent items are obtained and, with that, the count tables holding the size-2 candidates grow considerably. PARTITIONED is then much more efficient than REPLICATED because more candidates remain in the caches during counting. Interestingly, both counting approaches do not benefit of Hyperthreading for any of the measuring points; the overhead for the additional replicated tables or the queues is not offset by the SMT threads.

Figure 5.16b depicts the memory usage of the two counting approaches (excluding the size of the converted dataset). PARTITIONED has higher memory requirements than REPLICATED for high $\xi$ values because it has to maintain the pages (i.e., that are within the queues and free page queue) required by the producer–consumer processing scheme. This memory, however, remains constant as $\xi$ decreases. Only the single count table grows by about 350MB from $\xi = 0.09\%$ to $\xi = 0.02\%$. Contrary, the memory footprint of REPLICATED grows from 74MB to 9.46GB in the same range. If 64 threads are used, then even twice as much memory is required. Notice that both approaches require roughly the same amount of memory for $\xi = 0.043\%$, but PARTITIONED is at this point already much more efficient than REPLICATED.

To evaluate the scalability of both count table approaches, we varied the number of threads for certain $\xi$ values from 1 to 64. Figure 5.17a illustrates the time required for obtaining the frequent 2-itemsets on quest-retail-small with $\xi = 0.06\%$. Thereby, we include the time for the earlier phases—illustrated using a dashed line—which are still run using 32 threads. As can be observed, REPLICATED performs best; it is more than four times as fast as PARTITIONED when only a single thread is used because the count table is quite small so that the partitioning overhead does not pay off, i.e., a single table contains count values for “only” 818,560 candidates on this dataset when $\xi = 0.06\%$. Nevertheless, PARTITIONED scales well up to 32 threads whereas REPLICATED scales only up to 16 threads. This effect is increased when $\xi$ decreases to 0.03\% (Figure 5.17b). The count table then contains 51,974,110 candidates generated from 10,196 frequent items. REPLICATED scales only up to 8 threads; if more threads are employed, the runtime even increases until it is almost as high as for a
single thread. Again, PARTITIONED scales well up to 32 threads and is already for 8 threads faster than REPLICATED.

We obtain similar results for the datasets quest-kosarak-small and quest-wikilinks-small. If the size-2 candidates for more than 2220 (\(\xi \approx 0.02\%\)) and 2245 frequent items (\(\xi \approx 0.03\%\)) on the former and latter dataset should be counted, then PARTITIONED is more efficient than REPLICATED. Notice, however, that the subsequent mining on quest-wikilinks-small for such low \(\xi\) values would be very expensive for \(cApriori\). As mentioned earlier, other mining algorithms are more efficient on this dataset for such low \(\xi\) values. We further found that REPLICATED performs always best on quest-BMSPOS-small since this dataset contains less than 1800 distinct items.

Summing up, REPLICATED is more efficient for high \(\xi\) values where the count tables are rather small and candidate counting typically is not expensive. As soon as the count tables grow considerably, the pressure on the shared caches increases, which limits REPLICATED’s scalability. PARTITIONED is then more efficient; it scales well with an increasing number of threads and is thus well-suited for multiprocessor systems. Nevertheless, it has higher initial memory requirements for the producer–consumer scheme so that one should switch from one counter to another depending on the number of frequent items. In \(cApriori\), we set the threshold for switching to 2200 frequent items. This threshold may be different on other systems that have larger caches or allow more threads run in parallel.

**Overall runtime**

We obtain similar results for the overall mining procedure. The runtime for a varying number of threads on quest-retail-small is illustrated in Figure 5.18a. REPLICATED scales only well up to 4 threads whereas PARTITIONED scales well up to 32 threads. For this reason, PARTITIONED is multiple times faster than REPLICATED when a large number of threads is used. The overall memory usage is illustrated in Figure 5.18b. As can be seen, PARTITIONED’s memory footprint grows strongly as the number of threads is increased. For 64 threads, it requires more than 25GB while the converted dataset comprises only about 1GB. PARTITIONED’s memory footprint increases too—caused by the floating pages—but remains acceptable even for 64 threads. If necessary, it could further be decreased if the floating page size is reduced at the cost of a slightly worse runtime.
The overall runtime of cAPRIORI on quest-kosarak-small and quest-BMS-POS-small is similar (not shown). REPLICAED performs worse than PARTITIONED as soon as the count trie exceeds about 15–25MB. Moreover, REPLICAED is unusable when the count trie is larger than 300MB. Hence, only PARTITIONED can be employed if a large number of threads is used and the count trie reaches a certain size; REPLICAED is thus only useful for small count tries.

Finally, we run parallel cAPRIORI on the large datasets quest-retail, quest-kosarak, quest-BMS-POS, and quest-wikilinks. We integrate REPLICAED and PARTITIONED trie-based subset counting into cAPRIORI and switch from the former to the latter, when the count trie exceeds 20MB. Figure 5.19a illustrates the execution time on quest-retail with $\xi = 0.09\%$ when the number of threads is varied from 1 to 64. As can be seen, cAPRIORI scales very well as the number of threads is increased. The runtime decreases from 1150s when only a single thread is used to 59.7s when 32 threads are used. This constitutes to a speedup of 19.3x. We do not achieve linear speedups on this dataset because the $\xi$ values must be set rather high on the large datasets. The converted dataset on quest-retail with $\xi = 0.09\%$ amounts to 17GB—still less than 20% of the dataset’s original size—so that the remaining space that can be used for mining is fairly low on our test system (cf. Figure 5.19b). Because of the high $\xi$ values, converting the dataset is almost as expensive as the subsequent mining. Converting the dataset, however, does not scale as well as mining because it is limited by the time used for mapping physical pages to virtual pages while increasing the page pools. This issue would disappear as soon as Linux performs the page mapping in parallel.

We obtain similar results for the other large datasets. Except on quest-wikilinks, the converted datasets occupy much of the remaining 28GB on our large test system. Rather high $\xi$ values must therefore be used, so converting the dataset has a large impact on the overall mining time. Only the conversion, however, limits the speedup. All other components scale well so that cAPRIORI scales almost linear on quest-wikilinks where the time for mining is much higher than for the preceding steps.

5.5. Summary and Discussion

In this chapter, we proposed cAPRIORI as a memory efficient and highly scalable version of APRIORI. cAPRIORI uses a novel internal representation for the transaction database in which the filter transactions are organized in pages. Each of these pages holds only transactions of the same length, which reduces the memory footprint, enables cost-free transaction pruning,
and allows a fast conversion of the dataset being mined. Support counting in CAPRIORI is performed in the first two iterations and the remaining iterations using direct counting and highly optimized tries, respectively. The respective data structures are either replicated or partitioned to enable parallel support counting. For partitioned subset counting, we employ the multiple-producer/multiple-consumer processing scheme, which is a novel processing model for multiprocessor system comprising many cores. CAPRIORI is further founded on a fast parser, a lightweight memory management, and simple but effective database pruning techniques. All of these points are crucial for CAPRIORI’s superior performance, which was observed in our experiments: CAPRIORI it often about 10x faster than existing highly optimized APRIORI implementations and has usually a much lower memory footprint than them. This makes it the algorithm of choice for processing large datasets with small transactions and almost arbitrary $\xi$ values. CAPRIORI further scales almost linear on our four-socket test system.

Although CAPRIORI is typically much faster than the competing APRIORI implementations, there is still room for improvements. CAPRIORI’s fast dataset conversion into the page-based transaction representation is performed with almost only sequential access in main memory. It thus quite fast and amounts often to only a small fraction of the overall runtime but comes at the price of a more expensive subset counting when rather small $\xi$ values are used or the dataset’s average transaction cardinality is high. In such cases, a trie-based representation [Borgelt, 2003, Bodon, 2004] or even sorting the filtered transactions internally [Chester et al., 2009] lead to a much faster subset counting and thus better overall performance. The trie creation or sorting the transactions, however, has to be performed in parallel because otherwise this would introduce a large sequential fraction within a parallel mining algorithm. Building a trie using multiple threads is the main topic in the second part of Chapter 7 (cf. Section 7.2 starting at page 163). These techniques can be reused to obtain a parallel trie-based APRIORI implementation. Sorting variable strings, which is basically comparable to sorting transactions, can be done by inserting them into a trie and traversing it afterwards in a depth-first order to obtain the sorted strings; this approach [Sinha et al., 2007] is currently considered as the best way for sorting strings. The filtered transactions thus could be sorted accordingly by inserting them into an FP-tree and traversing it in depth-first order. Again, the parallel trie building algorithm of Chapter 7 is well suited for this task.
6. cEclat

In this chapter\(^1\), we present cECLAT (short for compact ECLAT), a memory-efficient and highly-scalable version of ECLAT. cECLAT differs from the latter in two core aspects: It uses a more sophisticated internal transaction representation and employs several optimizations within support counting. More specifically, cECLAT represents transactions using tid-limaps, which are a combination between tid-lists and tid-bitmaps. They combine the advantages of both data structures and often require significantly less memory than them. Hence, cECLAT’s memory footprint is usually small so that even for large datasets out-of-core mining is effectively avoided. Parallel support counting is performed using our efficient intersection algorithms from Chapter 3 and equivalence class partitioning.

In Section 6.1, we introduce the tid-limap as the core data structure of cECLAT. We start by explaining the logical representation of a tid-limap, i.e., how transactions are represented in this data structure. Based on that, we discuss a respective physical representation, which is tailor-made for our speculative intersection algorithms from Chapter 3 and is further optimized for a low memory usage. Thereafter, we explain the sequential as well as the parallel conversion of the transaction database into tid-limaps.

We discuss cECLAT’s parallel support counting in Section 6.2. This includes details about the applied search strategy, the integration of the intersection algorithms, various optimizations that speed up the intersections, and the memory management of the threads.

We provide the results of our experimental evaluation of cECLAT in Section 6.3. Basically, we use the same setup as in Chapter 5 but employ ECLAT-suited datasets in our experiments. As in the previous chapter, we perform single-threaded and multi-threaded experiments. The purpose of the former is to illustrate the impact of our optimizations (data layout and intersection algorithms) as well as compare cECLAT’s performance with existing highly optimized ECLAT implementations. We observe that cECLAT is often an order of magnitude faster than them. Within the multi-threaded experiments, we investigate cECLAT’s scalability for an increasing number of threads. We achieve high speedups of up to 25x on our four-socket system comprising 32 cores.

6.1. Data representation

In this section, we propose the tid-limap as the core data structure of cECLAT. It forms a combination between the tid-list and the tid-bitmap and is used to represent the filtered transactions. We start our discussion with the logical representation of tid-limaps and thereafter give details about its physical representation, which is tailor-made for our intersection algorithms proposed in Chapter 3. In the remainder, we discuss the multi-threaded conversion from a given dataset into tid-limaps.

---

\(^1\)The material in this chapter has been developed jointly with Tomas Karnagel, Tim Kiefer, and Wolfgang Lehner. The chapter forms an extended version of prior published work [Schlegel et al., 2013a].
6. cEclat

6.1.1. Logical representation

The logical and physical data representation of the transactions in the vertical layout has a large impact on cECLAT’s performance. Since the multi-threaded intersection process is limited by the memory bandwidth (cf. Section 3.3.2), one major goal is to minimize the memory required for representing the tid-sets, which on the side delays expensive out-of-core processing. Recall that after the dataset is parsed, a set of tids needs to be stored for each of the frequent items. As discussed in Chapter 2, the tids for an item can be represented as either a tid-list or a tid-bitmap. The former is considered as more space-efficient for sparse datasets while the latter is superior on dense datasets. Most ECLAT implementations [Burdick et al., 2001, Borgelt, 2003] choose—based on a certain threshold—one of both representations, which is then used for the complete mining process. In the following, we propose a hybrid representation that is memory efficient and tailor-made for our intersection algorithms.

A tid-limap consists of a tid-list and a tid-bitmap and stores a frequent item’s tids partly in one and partly in the other data structure. The tid-bitmap section of a limap encodes certain tids within a fixed-length bitmap. Each bit corresponds to a single transaction; it is set if the corresponding transaction contains the tid-limap’s respective frequent item; otherwise, the bit is unset. The tid-list section contains the tids of all remaining transactions in which the frequent item occurs and that are not encoded within the bitmap section. For multiple tid-limaps, i.e., for each frequent item of a dataset a single tid-limap is created, the length of the tid-lists may be different while the bitmaps of all limaps have all the same length. Figure 6.1 illustrates tid-limaps for \( n \) frequent items \( i_1 \) to \( i_n \). The tid-bitmap section of each tid-limap consists of 6 bits; all bits that are in the same column belong to the same transaction whereas all bits in the same row belong to the same frequent item. The second and forth bit in the tid-bitmask of item \( i_1 \) indicate that this item occurs within the transactions \( t_2 \) and \( t_4 \). The respective tid-list has three elements and indicates that \( i_1 \) also occurs within the transactions \( t_7, t_8, \) and \( t_9 \).

![Figure 6.1: Tid-limaps for \( n \) frequent items](image)

The tid-limap representation is only superior—in terms of memory consumption—to the pure tid-bitmap or tid-list representation, if the optimal of both inner data structures is chosen for each of the transactions being converted. Roughly speaking, sparse transactions, which contain only few frequent items, should be encoded in the tid-list section whereas dense transactions, which contain many frequent items, should be encoded in the tid-bitmap section of a set of tid-limaps. Hence, we need to find the optimal segmentation of a dataset’s transactions into these two transaction types.
As for tid-lists or tid-bitmaps in ECLAT, tid-limaps are used in CECLAT to represent the base data (i.e., the converted and filtered transactions) and intermediate results. In the following, the former tid-limaps are denoted as *base-limaps* and the latter are denoted as *intermediate-limaps*. The optimal transaction segmentation for the base-limaps can be determined after the first scan of a dataset and thus before it is converted. In what follows, we denote with \( n \) the number of frequent items that are revealed during the first scan while \( m \) denotes the items of a transaction after infrequent items are pruned away. These two parameters are sufficient to chose the optimal representation for a transaction being converted. For that, consider what happens if a transaction is inserted into one of the two sections:

- If the transaction is inserted into the tid-bitmap section, then a bit must be added to each bitmap of all of the \( n \) tid-limaps. In Figure 6.1, for example, a seventh column needs to be added in the tid-bitmap section. Hence, a transaction would be represented using \( \lceil \frac{n}{8} \rceil \) bytes.

- If the transaction is inserted in the tid-list section, then a list item must be added to \( m \) tid-limaps since \( m \) frequent items occur within the transaction. As we will explain later, we use our hierarchical list layout from Section 3.2.2 (see page 62) to encode the lists so that only about 2 bytes were required for storing a single list item. Hence, the transaction would be represented within the tid-list section using roughly \( 2m \) bytes.

The optimal representation for each transaction can now be calculated as follows: each transaction that satisfies \( 2m < \frac{n}{8} \) is considered as a `sparse` transaction while all other transactions are considered as `dense` transactions; i.e., the former are stored within the tid-bitmap section whereas the latter are stored within the tid-list section. This segmentation is optimal for the base-limaps but may not be optimal for the intermediate-limaps.

Intermediate-limaps represent the tids of frequent itemsets that comprise at least two items. They store the tids in which these itemsets occur and are created from base-limaps (for frequent 2-itemsets) or other intermediate-limaps (for frequent itemsets comprising more than 2 items) during mining. They typically contain less tids than the limaps they are created from. This affects the efficiency of the tid-bitmap section because it does not shrink as the number of tids decreases whereas the tid-list section is likely to shrink. For example, consider the intersection of the first two items in Figure 6.1. The tid-list of the \{i_1, i_2\}-itemset would be empty but the tid-bitmap still requires six bits.\(^2\) Hence, tid-lists are often more efficient in the deeper recursions; they then require less space than tid-bitmaps and the intersection costs shrink because less list items must be intersected. For low \( \xi \) values, the tid-lists even get much more efficient than tid-bitmaps. This effect is validated in our experimental evaluation.

As a result, we add an additional parameter to the previously introduced classification rule. A transaction is now `sparse` (and otherwise `dense`) if it fulfills \( m < \frac{n}{b_{sparse}} \) where \( b_{sparse} \) is denoted as transaction classifier parameter and is used to adjust the size of the bitmap section. For \( b_{sparse} = 16 \), the new rule is equal to the original classification rule. In the extreme cases, \( b_{sparse} = \infty \) and \( b_{sparse} = 0 \), the limaps behave like regular tid-bitmaps and tid-lists, respectively. Thus for \( \xi \) values that lead to deep recursion levels, \( b_{sparse} \) should be smaller than 16 to increase the tid-list section of the limaps. Unfortunately, an optimal value for \( b_{sparse} \) cannot be determined before mining. We discuss an appropriate setting that is empirically obtained within our experimental evaluation (cf. Section 6.3.2).

\(^2\)Clearly, this does only matter if the itemset has a larger support than \( \xi \) and thus is required for subsequent intersections.
6. cEclat

6.1.2. Physical representation

Besides the transaction assignment, also the specific implementation of the tid-limap influences cEclat’s execution time and memory footprint. The tid-lists within the limaps are represented using the hierarchical layout proposed in Section 3.2.2 on page 62, i.e., a list is split in smaller sub-lists that each have at most 65,536 list items. This allows us to exploit the speculative list intersection algorithms that do not require a pre- and post-processing step for the intersection of two limaps. Furthermore, each list item is encoded using only about 2 bytes, which already includes the bytes required for encoding the length of a list. The tid-bitmaps are stored using an array of 64-bit integer values (64-bit aligned) to allow the fast intersection and support counting with only and and popcount instructions.

The actual arrangement of the inner data structures differs between base-limaps and intermediate limaps. The base-limaps are stored chunk-wise together; a single chunk contains a range of at most 65,536 tids for each frequent item. Thereby, the tid-bitmaps and tid-lists are stored separately in different chunks. The size of a tid-bitmap chunk is thus fixed and given by n · 65536/8 bytes where n denotes the number of frequent items. Contrary, the size of a tid-list chunk depends on the sparseness of the transactions. If the dataset contains many small transactions, then the tid-lists are sparse and a tid-list chunk is much smaller than a tid-bitmap chunk. For dense datasets, however, their size could be similar. As we will explain later, the chunk-wise memory layout eases the conversion of the dataset into the base-limaps.

The intermediate-limaps are stored separately from each other; the tid-bitmap and tid-list of each of them are stored—in chunks of 65,536 tids—consecutively in memory. This separation of the limaps eases the generation of new intermediate-limaps during support counting.

Compression techniques could be used to further reduce the memory footprint of the limaps. As for cApriori, however, the expected gain of compression is low. The bitmaps are already a very efficient representation of the dense transactions. Mainly the sparse transactions could benefit from compression but since there are only 2 bytes per tid required, the compression rate is at best 50% with the discussed byte-wise static encodings (cf. Chapter 4). Using a small experiment with various realistic datasets (e.g., netflix, webdocs), we observed an average compression rate of about 60%, i.e., roughly 9.5 bit were required per tid. The bitwise static encodings provide similar compression rates. If compression should be applied anyway, it could be used to compress both, base-limaps and intermediate-limaps. In both cases, the tid-lists are compressed once and may be decompressed multiple times. Hence, a high decompression speed is key to a good performance.

6.1.3. Conversion into the vertical layout

After obtaining the frequent items within the first scan, the transactions of the dataset must be converted into the vertical layout, i.e., the transactions are converted into tid-limaps. The sequential conversion is straight-forward; for each transaction the decision is made whether it is a dense or sparse transaction; then its tid is appended in the list section of the corresponding tid-limaps or the bitmap section of all tid-limaps. Depending on the dataset and chosen \( \xi \) value, this conversion may account for a large fraction of the algorithm’s runtime. Therefore, it cannot be done sequentially within the multi-threaded version of cECLAT, since otherwise it would introduce a large sequential fraction—thus limiting the maximum speedup. For this reason, we propose a parallel conversion that partitions the transactions and processes them independently. The basic idea is inspired by Zaki et al. [1997b]; they propose a parallel
conversion for PAReClAT (see Chapter 2 on page 25). Unlike their solution, however, we do not require any knowledge about the tids that are within the partitions of the dataset (i.e., the tid of the first and last transaction of each partition).

The main difficulty of the multi-threaded conversion is that tids cannot be assigned in an ascending order to the transactions as it is done in the sequential conversion. This, however, is even not necessary because the tid that is assigned to a transaction can be arbitrarily chosen; there is no need that the \( n \)-th transaction must be represented as value \( n \) within the tid-lists or at the \( n \)-th position of the tid-bitmaps. Only two conditions must always hold: (1) the chosen tid of a single transaction must be the same in all tid-limaps that contain the transaction and (2) it may only be assigned to one transaction. Otherwise, the intersection result becomes ambiguous if two limaps are intersected and at least two transactions are represented using the same tid within them. Using the chunk-wise memory layout, both conditions can be easily met because a transaction is represented in only a single chunk. Such a chunk represents a range of transactions for all limaps. A sparse transaction gets an own value within a tid-list chunk whereas a dense transaction gets an own position within a tid-bitmap chunk. For this reason, threads can assign unique tids\(^3\) to transactions without interfering with other threads as long as they create the chunks independently from each other.

During the conversion, each thread maintains a local tid-bitmap chunk with 65,536 bits for each frequent item and additional space for holding the items of 65,536 filtered transactions. Dense transactions are directly inserted into the local tid-bitmap chunk. If the dense transaction consists of \( n \) items that fulfill \( \xi \), then \( n \) particular bits must be set to “1” in the column that corresponds to the transaction. The space for a tid-bitmap chunk is always fully allocated for 65,536 transactions even if they may not be completely used. The additional space is required to hold temporarily the transactions that should be stored in a tid-list chunk. This is necessary because the transactions cannot be inserted on-the-fly into such a chunk as long as the lengths of the tid-lists are unknown; these lengths are required since the tid-lists are stored one after another continuously in memory. They, however, are first available after all transactions that should be inserted into a chunk are read and filtered; only then the transactions can be converted into tid-lists.

The multi-threaded conversion itself works in two recurring phases. It takes transactions and the frequencies of the distinct items as input and returns a list of tid-list chunks and a list of tid-bitmap chunks.

**Process transactions:** Each thread loads a chunk of transactions, prunes infrequent items within them, and partitions the transactions. Sparse transactions are collected in a thread-local buffer whereas dense transactions are directly inserted into a thread-local tid-bitmap chunk.

**Append chunks:** Whenever a tid-bitmap chunk is full, it is appended to the global tid-bitmap chunk list. Similar, whenever 65,536 sparse transactions are collected, they are converted into a tid-list chunk that is afterwards appended to the global tid-list chunk list. Synchronization between the threads is only required for this append operation. After appending is finished, memory is gathered for a new tid-bitmap or tid-list chunk and the thread continues processing transactions.

---

\(^3\)Each tid is unique because it can be identified using the chunk itself and the value (in a tid-list chunk) or the position (in a tid-bitmap chunk) within it.
6. cEclat

As soon as all transactions are processed, the threads append their partially filled chunks (i.e., the local tid-list and tid-bitmap chunk of each thread) to the global chunk lists. The partially filled chunks could also be merged to obtain full chunks and thus reducing their number. We found, however, that the fraction of partially filled chunks is negligible for large datasets; we thus do not merge them. After the dataset is converted, cECLAT starts support counting.

6.2. Support counting

As for ECLAT, cECLAT employs equivalence class clustering (cf. Section 2.2.2 on page 17), i.e., the equivalence classes are processed one after another. In multi-threaded cECLAT, each thread has an own equivalence class based on the frequent items assigned. Whenever a thread finishes processing its assigned class, it gathers one of the remaining classes. To avoid a load imbalance while processing the last classes, we process the classes in the order of their mining effort; heavy classes are processed first whereas the light classes remain for the end. For that purpose, we sort the frequent items with respect of their frequency. Since the classes for more frequent items have typically a larger weight, we start mining them first.

The coarse-grained assignment of the equivalence classes to threads has two issues. First of all, it works only if enough frequent items are found. For example, if there are 32 threads but only 20 frequent items, then 12 threads are not busy at any time during support counting. Secondly, cECLAT’s memory footprint grows as the number of threads is increased because for \(n\) threads, the intermediate limaps of \(n\) different classes must reside in main memory at the same time. Both issues, however, are negligible for reasonable \(\xi\) values and dense datasets. If \(\xi\) is set too high so that only few frequent items occur, then mining is usually not expensive at all and thus it is already for a small number of threads sufficiently fast. Also, the memory usage per equivalence class is small for dense datasets because the base-limaps are short for such datasets—leading to short intermediate-limaps.

To reduce the size of the intermediate results and reduce effort during mining, we employ the diffset optimization, which is described in Chapter 2 in Section 2.2.2. More specifically, we represent the frequent items and frequent 2-itemsets using regular sets and larger frequent itemsets using diffsets. Hence, the limaps are intersected within the first recursion level and for all other recursion levels, the difference operation is used. Thereby, there is no difference whether a limap represents regular tid-sets or diffsets. We observed in almost all cases significant performance improvements with this optimization enabled.

Counting the support of a single candidate itemset, i.e., the intersection or difference operation on two tid-limaps, is performed chunk-wise using the algorithms proposed in Chapter 3. Always the two tid-bitmaps or two tid-lists that belong to the same chunk are intersected or the difference is formed of. The tid-bitmaps are processed first. Thereafter, the tid-lists are processed using the speculative intersection or difference algorithm. If base-limaps are used as input, the tid-bitmaps and tid-lists need to be gathered from all tid-bitmap and tid-list chunks of the converted base data. Otherwise if intermediate-limaps are used as input, the chunks of each tid-bitmap or tid-list are stored continuously in memory. The result of the intersection or difference operation is always an intermediate-limap.

Support counting of a candidate itemset can be further speed up by stopping the intersection or difference operation on two limaps as soon as the given \(\xi\) value cannot be fulfilled. For example, if \(\hat{\xi} = 150\) and the two limaps being intersected have 160 and 190 tids, then the intersection can be stopped after 10 mismatches (i.e., there are 10 values in the first limap that...
6.3. Experiments

do not occur in the second limap). We adapt this optimization, which is denoted as short-circuited intersection within the original ECLAT algorithm (cf. Section 2.2.2 on page 19), also for support counting when the diffset optimization is employed. As soon as the difference between the number of tids of the two limaps is larger than the difference of the first limap’s support and $\xi$, then the difference operation can be stopped because the support of the newly generated intermediate-limap would be smaller than the $\xi$ value. Checking these constraints in each loop iteration of the employed intersection or difference algorithm, however, introduces a large overhead so that the early abort is not beneficial anymore. For this reason, we check the constraints only after a chunk of the limap is fully processed. Whenever the constraints are violated, the remaining chunks are skipped. In our experiments, we obtained a performance improvement between 10% and 20% with this optimization enabled.

All newly created intermediate-limaps are stored within a continuous memory region that is provided by a stack memory pool of our tailor-made memory management. The memory region, i.e., the stack, grows with each newly created intermediate-limap that has a support larger than $\xi$. Intermediate-limaps that are temporarily created during an intersection or difference operation and do not satisfy $\xi$ are overwritten as the next intermediate-limap is created. For each recursive call of mine(), we maintain the start address of the used memory within this call to adjust the size of the stack later. It is shrunk to the stored address whenever the recursion level is decreased because a call of mine() returns. Hence, there is no memory fragmentation during support counting. Within the multi-threaded version of cECLAT, each thread maintains an own stack memory pool for storing its intermediate-limaps.

6.3. Experiments

In this section, we provide the results for our experimental evaluation of cECLAT. As in Chapter 5, we first provide results for the single-threaded experiments and thereafter discuss the outcome of the multi-threaded experiments. Again, we compare cECLAT only with ECLAT-based algorithms and defer a comparison with other algorithms to Appendix C starting on page 195. Our experiments suggest that cECLAT has a superior memory footprint and is often much faster than the existing highly efficient ECLAT implementations. Depending on the chosen $\xi$ value, cECLAT is up to 8x faster and has an up to 16x smaller memory footprint. Furthermore, multi-threaded cECLAT scales well up to 32 threads on our four-socket system comprising 32 cores.

6.3.1. Setup

We use the same hardware and software setup as used in the previous chapter (see Section 5.4.1 on page 117). cECLAT and all of its optimizations are implemented in C++ and compiled using Intel C++ Composer XE (-fast is used as optimization flag). We reused certain components of cAPRIORI; this includes the parallel parser, the memory management, and the frequent item counter. We further integrated the set intersection algorithms of Chapter 3 and implemented the set difference counterparts. For each intersection (or difference) algorithm, we build an own cECLAT executable where the underlying algorithm is directly integrated; i.e., there is no runtime overhead for selecting the intersection function. We similarly enable or disable the different optimizations. Multi-threaded cECLAT is realized using OpenMP.

All experiments were conducted on our four-socket system, which has 128GB main memory, supports all required SIMD instructions, and allows up to 64 threads (with Hyperthreading
6. cEclat

The datasets are main memory resident—located in a RAM disk. They were generated using the IBM Quest Data Generator [Agrawal and Srikant, 1997] and enhanced using small realistic datasets (see Section 5.4.1 on page 118 for more details). We measured the execution times of our algorithms using `gettimeofday`; `linux time` was used to measure the runtime of the competing algorithms. As before, all algorithms do not return the actual frequent itemsets, which has no impact on the performance of the algorithms; only a histogram containing the frequencies of the frequent $k$-itemsets for each $k$ is returned.

6.3.2. Single-threaded

In the following set of experiments, we investigate the single-threaded performance of cECLAT. We evaluate the influence of the transaction classifier parameter as well as the performance of the different underlying intersection algorithms that were proposed in Chapter 3. We further compare cECLAT with available sequential Eclat implementations. If not mentioned otherwise, cECLAT always employs the diffset optimization as well as the short-circuit intersection technique. We do not employ other optimizations like perfect extension pruning or the range intersection technique (cf. Section 2.2.2 on page 19) because they are often only beneficial for $\xi$ values that lead to unreasonable large result sets.

We basically use the same setup as it is used for evaluating the single-threaded performance of cAPRIORI (cf. Section 5.4.2); all used synthetic datasets are 4GB and the linux process of the algorithm under test is always bound to the first processor. We mainly use two dense datasets (quest-webdocs-small and quest-netflix-small) and a sparse dataset (quest-retail-small) but tried also other datasets. The mix of dense and sparse datasets allows to highlight the benefits and drawbacks of the possible internal transaction representations since transactions of dense datasets are usually represented using tid-bitmaps whereas transactions of sparse datasets are represented as tid-lists.

Impact of transaction representation and intersection algorithm

We run several experiments to evaluate the impact of the limap parameter $b_{\text{sparse}}$ that is used to classify transactions into dense or sparse transactions. Recall that for a larger $b_{\text{sparse}}$ more transactions are represented within the tid-bitmap section and thus less of them are represented within the tid-list section of the limaps. Furthermore, the limaps should be the representation with the lowest memory footprint for the converted database when $b_{\text{sparse}} = 16$ while the memory footprint during mining is typically lower for $b_{\text{sparse}} < 16$.

We observed that $b_{\text{sparse}}$ has almost no impact on sparse datasets like quest-retail-small. On such datasets, only few transactions are represented within the limaps’ tid-bitmap section, even when $b_{\text{sparse}} = 16$. On dense datasets like quest-webdocs-small, however, the choice of $b_{\text{sparse}}$ has a strong impact on the limaps’ memory footprint and intersection execution time. Figure 6.2a illustrates cECLAT’s execution time on quest-webdocs-small for a varying $b_{\text{sparse}}$ and various choices of $\xi$ values. We varied $b_{\text{sparse}}$ from 0 (all transactions are represented as tid-lists) to 16 (most of the transactions are represented as tid-bitmaps). As one can see, the execution time strongly depends on the chosen $\xi$ value. For $\xi \geq 10\%$ on this particular dataset, it is more beneficial to use larger values of $b_{\text{sparse}}$ because there are mainly small frequent itemsets. The stable tid-bitmap size has then only a minor impact. As the $\xi$ value decreases, more larger frequent itemsets occur. With that, the tid-bitmaps get more and more ineffective because of their stable size. In contrast, the diffsets (i.e., tid-lists) get
smaller with each recursion step, leading to a much faster intersection. Already for $\xi = 8\%$, cECLAT performs best when $b_{\text{sparse}} = 3$. Notice that the speculative intersection algorithm is used for intersecting the tid-list sections of two limaps; the threshold might thus be higher for other intersection algorithms. For even lower $\xi$ values ($\xi < 8\%$), the lowest execution time is obtained for $b_{\text{sparse}} \leq 2$.

The memory footprint of cECLAT does not correlate with its execution time when $b_{\text{sparse}}$ is varied (assuming in-core processing). Figure 6.2b illustrates cECLAT’s peak memory footprint when mining quest-webdocs-small. Independently of the chosen $\xi$ value, the least memory is required for $b_{\text{sparse}} \geq 10$. Nevertheless, contrary to the memory required for purely representing the transactions, which is—as expected—always minimal for $b_{\text{sparse}} = 16$, the minimal memory footprint during mining depends on $\xi$; for example, the least memory is required for $b_{\text{sparse}} \approx 11$ when $\xi = 8$. Note that the memory consumption always increases for $b_{\text{sparse}} > 16$ (not shown). We observed similar results also on other datasets like quest-netflix-small and quest-netflix$^T$-small. Since cECLAT’s execution time is usually better for smaller $b_{\text{sparse}}$ whereas its memory footprint is better for larger $b_{\text{sparse}}$, one need to find a compromise between both. As the memory consumption increases first for $b_{\text{sparse}} \approx 7$, we found this value as a good tradeoff between performance and memory consumption. Hence, we use this value in the following experiments.

We next compare the performance of the different intersection algorithms proposed in Chapter 3. The cECLAT implementations list-branch, list-branchless, and list-spec use only tid-lists (i.e., tid-limaps with $b_{\text{sparse}} = 0$) and differ in the intersection algorithm they employ. Bitmap uses bitmap intersection and represents the transactions using only tid-bitmaps (i.e., tid-limaps with $b_{\text{sparse}} = \infty$) whereas limap uses tid-limaps as underlying data structure (with $b_{\text{sparse}} = 7$) and employs bitmap as well as the speculative list intersection.

Figure 6.3a shows the runtime for the dense dataset quest-webdocs-small. As one can see, list-spec is always the most efficient of the three tid-list intersection algorithms. It achieves speedups from 2.2x up to 3.0x compared to list-branchless and 3.3x up to 4.4x compared to list-branch. The speedup is higher for smaller $\xi$ values because the average selectivity of the performed intersections decreases as $\xi$ gets smaller. Recall that the correlation between selectivity and speedup is discussed in Chapter 3 (see page 66). Bitmap and limap are more efficient than list-spec for large $\xi$. For example, limap is 1.6x faster than list-spec for

\footnote{As $\xi$ decreases, the number of tid-lists increases—because more items are considered as frequent—but the lists differ more in length and the items they contain.}
6. cEclat

![Graphs showing runtime and memory footprint for different data sets and support levels.](image)

Figure 6.3.: Running cEclat using different data layouts and intersection algorithms

$\xi = 12\%$. For small $\xi$, however, LIST-SPEC is much faster than LIMAP and BITMAP, e.g., it is twice as fast as LIMAP for $\xi = 7\%$. The speedup even increases for smaller $\xi$ values. The main reason of their inferior performance is the stable size of the tid-bitmaps during the mining recursions. Although this affects BITMAP more than LIMAP, they both get eventually slower than LIST-BRANCHLESS and even LIST-BRANCH for very small $\xi$ values. We obtain a similar overall behavior also on quest-netflix-small.

The implementations memory usage on quest-webdocs-small is depicted in Figure 6.3b. All three list-based cEclat implementations require the same amount of main memory since they employ the same memory layout and differ only in the underlying intersection algorithm. As can be observed, their memory footprint is much higher compared to BITMAP or LIMAP. For $\xi = 12\%$ and using tid-lists, cEclat has a peak memory usage of 503MB, of which 471MB are required for the base-lists and 32MB are required for the recursion. BITMAP and LIMAP require only about 285MB; this amounts to a 1.75x smaller memory footprint. As $\xi$ decreases to $6\%$, cEclat requires 641MB when it uses tid-lists (584MB for base-lists and 57MB for intermediate-lists) whereas it requires only 396MB when it uses tid-limaps (356MB for base-limaps and 40MB for intermediate-limaps). BITMAP has always a larger memory footprint than LIMAP but the difference between them is small, e.g., 35MB for $\xi = 6\%$. Interestingly, the intermediate results require only a fraction of the space required by the base data. Within all runs, about 10% of the memory usage is caused by them during mining. We obtain similar results on quest-netflix-small, although, LIMAP’s memory footprint is in average only 1.2x lower than the memory footprint of cEclat using tid-lists.

In the next set of experiments, we compare the implementations performance on the sparse dataset quest-retail-small. In general, APRIORI-based as well as FP-GROWTH-based al-
algorithms are better suited for mining sparse datasets, e.g., LIMAP (or LIST-SPEC) requires for \( \xi = 0.4\% \) about 100s to mine \texttt{quest-retail-small} whereas our \textsc{cApriori} implementation from the previous chapter requires only 31s for the same task (a single thread is used in both cases). Nevertheless, we still include this experiment to illustrate the adaptivity of the limap. As can be seen in Figure 6.3c, the best performance is achieved by LIMAP and LIST-SPEC. Both implementations achieve a similar performance because almost all transactions are within the limaps’ tid-list section; only a few transactions are represented as tid-bitmaps so that LIMAP is slightly faster than LIST-SPEC. Again BITMAP performs well for larger \( \xi \) values but its performance degrades dramatically as \( \xi \) decreases, i.e., it is slower than all other implementations for \( \xi < 0.3\% \). LIST-BRANCH and LIST-BRANCHLESS show a similar performance because the average intersection selectivity is high on this dataset, i.e., many lists contain similar items. For this reason, there is a smaller performance difference between the list intersection algorithms than for the datasets \texttt{quest-webdocs-small} or \texttt{quest-netflix-small}. LIST-BRANCH is even faster than LIST-BRANCHLESS whereas both are outperformed by LIST-SPEC and LIMAP; the latter two are both from 1.7x (for large \( \xi \)) up to 2.1x (for small \( \xi \)) faster than LIST-BRANCH.

Finally, Figure 6.3d illustrates the memory usage of the implementations on \texttt{quest-retail-small}. LIMAP and all \textsc{cEclat} implementations that use solely tid-lists have the same memory footprint because—as mentioned before—almost all transactions are represented within the tid-list section of the limaps on this particular dataset. BITMAP has the largest space requirements; they increase rapidly for low \( \xi \) values because of the many transactions, which have aside only a small cardinality. Hence, the bitmaps are large but consist of almost only zeros.

To summarize, the speculative intersection is very beneficial within \textsc{cEclat}. On dense datasets, which are well suited for \textsc{cEclat}, the speculative intersection decreases the runtime by up to 3x (or 4.4x if the regular list intersection is replaced). The experiments further indicate that BITMAP—although it is often considered to be superior over list intersection algorithms—is only for large \( \xi \) values competitive with LIST-SPEC. For small \( \xi \) values, BITMAP shows the worst performance of all tested intersection algorithms.\footnote{The worse bitmap intersection performance makes it questionable whether Eclat implementations for GPUs, which completely rely on tid-bitmap intersections, can compete for low \( \xi \) values with implementations that rely on tid-list intersections and are executed on CPUs.} In combination with tid-lists, tid-bitmaps can reduce the memory footprint of \textsc{cEclat}. Since, however, LIMAP is often slower than LIST-SPEC, a tradeoff between performance and memory usage has to be found.

**Comparison with Competitors**

In the next set of experiments, we compare \textsc{cEclat-list} and \textsc{cEclat-limap} with existing \textsc{Eclat} implementations. Both of our implementations rely on the speculative intersection and differ only in the data layout they employ. \textsc{cEclat-list} uses solely tid-lists (i.e., limaps with \( b_{\text{sparse}} = 0 \)) whereas \textsc{cEclat-limap} uses tid-limaps (with \( b_{\text{sparse}} = 7 \)) as underlying data structure.

Goethals provides an \textsc{Eclat} implementation\footnote{The implementation was obtained from \url{http://adrem.ua.ac.be/~goethals/software/}.}, which also includes the diffset optimization. This implementation—in the following referred as \textsc{Eclat-Goethals}—is similar to \textsc{cEclat-list} but relies on C++ standard data types; it uses the \texttt{std::vector} data structure for representing the tid-lists as well as the \texttt{set_intersection} and \texttt{set_difference} function from the C++ standard template library. Both functions rely on the simple merge-based
6. **cEclat**

![Graphs showing runtime of different implementations on various datasets.](image)

**Figure 6.4.a:** Runtime of certain cECLAT and ECLAT implementations on different datasets

Intersection that has two branches in the core loop (cf. Figure 3.2 on page 54). **Eclat-Goethals** further allocates memory using `new`, i.e., it does not provide an own memory manager, and it employs perfect extension pruning (cf. Section 2.2.2 on page 21), which is usually beneficial when mining with low \( \xi \) values. The second Eclat implementation used for comparison is obtained from Borgelt [2003]. This implementation includes several optimizations (e.g., diffsets, bitmap and range intersection), which were employed either automatically based on the density of the datasets being mined or using command-line arguments. For our experiments, we set these arguments to obtain regular **Eclat** and an **Eclat** version that employs the range intersection optimization. We denote the former as **Eclat-Borgelt** and the latter as **rEclat-Borgelt**. Besides the so far described implementations, we also tried the Eclat implementations from Schmidt-Thieme [2004] and Zaki and Gouda [2003]. The former was not running on our system while the latter requires a different input file format. Hence, we do not use them in our evaluation.

Figure 6.4a depicts the performance of the five implementations on **quest-webdocs-small**. As can be seen, both c**ECLAT** implementations perform much better than the existing implementations for \( \xi \geq 8\% \). For example, they both require about 120s until completion when \( \xi = 9\% \). The best competing Eclat implementation, **rEclat-Borgelt**, requires 309s for the same task, which amounts to a 2.5x higher runtime. For lower \( \xi \) values, the performance of **cEclat-limap** deteriorates seriously because of the expensive bitmap intersections at the depth recursion levels. **cEclat-list** performs then better than **cEclat-limap** but it is slower than **rEclat-Borgelt** for \( \xi < 6\% \). For \( \xi = 6\% \), however, the result set contains already 22.7 million itemsets so that it is questionable whether such low \( \xi \) are meaningful at all. The better performance of **rEclat-Borgelt** results mainly from the range intersection
6.3. Experiments

Figure 6.5.: Peak memory usage of the ECLAT-based implementations under test

optimization because both of Borgelt’s implementations spend almost the same time (about 240s) for parsing and converting the dataset. Nevertheless, the range intersection optimization works better on the synthetic datasets than on realistic datasets. Figure 6.4c illustrates the implementations’ performance on webdocs that is the realistic counterpart of the quest-webdocs-small dataset. Here, cECLAT-LIST is always superior over rECLAT-BORGELT.

Figure 6.4b and Figure 6.4d illustrate the performance of the five implementations for the synthetic quest-netflix-small and realistic netflix dataset, respectively. On the former dataset, both cECLAT implementations are much faster than the competing algorithms for high \( \xi \) values (> 9.5%). Again, rECLAT-BORGELT has the best performance for low \( \xi \) values; it is faster than cECLAT-LIMAP for \( \xi < 8.6\% \) (> 45.2 million frequent itemsets) and faster than cECLAT-LIST for \( \xi < 7.75\% \) (> 350 million frequent itemsets). On the netflix dataset, both cECLAT implementations are always faster than the other implementations.

We observe similar results also on several other datasets. ECLAT-BORGELT and ECLAT-GOETHALS always perform worse than both cECLAT implementations; rECLAT-BORGELT is often only competitive for low \( \xi \) values where the result set is unreasonable large. The worse performance of ECLAT-GOETHALS results from the missing memory management, i.e., the repeated expensive new and delete calls. Furthermore, the used C++ list intersection and difference algorithm of the standard template library has the same inferior performance as the branch intersection. In fact, it is even slower. ECLAT-BORGELT suffers from malloc and free calls that are performed within each call of the recursive mining function recurse; it uses also a slow intersection algorithm with two branches in the main loop. Furthermore, all three competing implementations spend much time for parsing and converting the dataset; this is a huge drawback when mining large datasets with moderate \( \xi \) values.

We next evaluated the peak memory usage of the five ECLAT-based algorithms under test. Figure 6.5a depicts the results for the datasets quest-webdocs-small, webdocs, quest-netflix-small, and netflix for \( \xi \) values that lead to moderate mining effort. As can be seen, all three competing ECLAT implementations have a much higher peak memory usage than both cECLAT implementations. cECLAT-LIMAP is up to 18.5x more space efficient than them; even cECLAT-LIST has an up to 15.8x lower memory usage. The high memory usage of the competing implementations results from two main issues. Firstly, they all convert the complete dataset into an internal format (i.e., tid-lists for ECLAT-GOETHALS or a bag of transactions for Borgelt’s implementations) to avoid the costs for parsing the dataset twice. Hence, the dataset is only scanned once but the internal format contains infrequent items,
6. cEclat

which are removed after the first scan took place. CECLAT avoids storing these items by scanning—and thus parsing—the dataset twice. Secondly, the internal representation of the tid-lists of all three competing implementations is not as efficient as the one used in cEclat; they use 32-bit integer values to represent items within the tid-lists whereas cEclat uses tid-lists with roughly 16 bit per value or limaps with less than 16 bit per value.

The dataset itself has a huge impact on the implementations’ memory usage. For similar \( \xi \) values, quest-webdocs-small and quest-netflix-quest lead to a different size of the internal representation during mining although both datasets have the same size (i.e., 4GB). This can be explained with the characteristics of the datasets; the former dataset has shorter transactions and more distinct items than the latter one. Interestingly, the memory footprint of the competing algorithms is larger than the webdocs dataset itself when it is mined. The datasets’ characteristics influence further the memory required for the intermediate results. Figure 6.5b illustrates the results for \( \xi \) values that lead to a high mining effort. On quest-netflix-small and netflix there is almost no difference as if higher \( \xi \) values are used (cf. Figure 6.5a). For quest-webdocs-small and webdocs, however, the implementations have a higher memory footprint as \( \xi \) is decreased. Thereby, the relative increase is larger for cEclat because the competing implementations only partially reuse memory that was previously used for the infrequent items and was released after the conversion. Thus in absolute numbers, the competing algorithms require more additional memory as \( \xi \) decreases than cEclat.

The high memory usage of the competing implementations renders out-of-core experiments useless. Even for large \( \xi \) values with possibly no frequent items at all, these implementations would already go out-of-core during the conversion of the dataset. Although the conversion issue could be resolved when two scans are performed as for cEclat, it would require major changes on the implementations.\(^7\) For this reason, we omit out-of-core experiments for the Eclat and cEclat implementations.

To summarize, our single-threaded cEclat implementations are competitive with available Eclat implementations. For \( \xi \) values that lead to reasonable large result sets, they are much faster than the existing implementations and have a superior memory footprint. Hence, it is reasonable to run cEclat using multiple threads.

6.3.3. Multi-threaded

Finally, we evaluate the performance of multi-threaded cEclat. We use the same setup as used for evaluating multi-threaded CAPRIORI, except that we mine the dense datasets quest-webdocs and quest-netflix—each one comprising 100GB. As before, we vary the number of threads from 1 to 64 and measure the overall execution time and memory consumption.

Figure 6.6a depicts the execution time of cEclat-limap and cEclat-list when mining quest-webdocs with \( \xi = 0.2\% \). As can be seen, both cEclat implementations scale well up to 32 threads—we achieve speedups up to 15.5x. For example, cEclat-limap finishes mining after 41.5s when 32 threads are used instead of 622s when only a single thread is used. There is further a small improvement when 64 threads (with Hyperthreading enabled) are used, i.e., the speedup increases to 17x. We do not achieve higher speedups, e.g., linear speedups, on quest-webdocs because of (1) the initial costs for mapping the dataset, (2) load imbalance.

\(^{7}\)The additional costs for parsing the dataset the second time are low compared to the conversion costs itself, i.e., the random access in main memory required for adding a tid to various tid-lists is more expensive than parsing a single line (or transaction) in the dataset. Clearly, this holds true only when the dataset is cached in main memory.
6.3. Experiments

![Graphs showing runtime and memory footprint for cEclat with varying numbers of threads on quest-webdocs and quest-netflix datasets.](image)

**Figure 6.6.:** Results for multi-threaded cECLAT and a varying number of threads during mining, and (3) the limited memory bandwidth of our test system. As for CAPRIO, already about 9s are required for mapping the 100GB dataset into the process address space. Without this mapping time, the maximum speedup increases to 22x when 64 threads are used. The load imbalance occurs because quest-webdocs resembles more to a sparse dataset than to a dense dataset. Although, it has the same cardinality as webdocs or quest-webdocs-small, it has much more transactions than these smaller datasets. For this reason, very long tid-lists occur for ξ = 0.2% whereas the result set contains only 59 frequent items. Hence, cEclat’s load partitioning, which distributes equivalence classes to the threads based on the frequent items, does not work well anymore. Besides this issue, the dataset’s sparseness further leads to an extensive memory usage as the number of threads is increased (see Figure 6.6b). Most of the space is required for holding the large intermediate tid-lists or tid-limaps that are generated by the threads simultaneously. We therefore conclude that quest-webdocs is—because of its sparseness—unsuitable for cEclat. As we will see in the next Chapter, CFP-GROWTH requires less time and space when mining it. Finally, cEclat’s overall speedup is also limited by the test system’s low memory bandwidth the threads compete of. Both, the bitmap and speculative list intersection, require a high bandwidth that easily exceeds the system’s bandwidth when all hardware threads are used.\(^8\) Hence, the threads slow each other down during mining.

\(^8\)The memory configuration of our test system is not optimal. Each of our processors is equipped with only one memory cardridge filled with four memory modules whereas the optimal configuration would be to equip each processor with two memory cardridges that are each filled with eight memory modules. For this reason, the memory bandwidth is reduced to roughly one forth of the specified peak memory bandwidth.
6. cEclat

We achieve slightly better speedups on the quest-netflix dataset—the execution times for $\xi = 11\%$ are illustrated in Figure 6.6c. This dataset is more suitable for cEclat because its transactions have much larger cardinality than the transactions of quest-webdocs (cf. Appendix A.1 on page 189). This leads to more frequent items and a larger recursion depth even for high $\xi$ values. For $\xi = 11\%$, there are 348 frequent items so that load imbalance does not occur as on quest-webdocs. Only the test system’s memory bandwidth and the mapping effort limit a linear speedup. We achieve speedups of up to 25x when the initial mapping time is not included. cEclat-list is faster than cEclat-limap for 32 and more threads because—although it requires more main memory for the intermediate results—it requires less memory bandwidth during the many intersections. The implementations’ overall memory usage on quest-netflix is illustrated in Figure 6.6d. As on quest-webdocs, more space is required as the number of threads is increased, but it is still below 7GB for cEclat-list and 3GB for cEclat-limap even if 64 threads are used. Hence, the additional space usage is negligible compared to the size of the dataset.

To summarize, cEclat scales well as the number of threads is increased. Only the system’s limited memory bandwidth as well as the time spend for mapping the dataset into the process address space hinders linear scalability on dense datasets like quest-netflix. One remaining issue, however, is the increased memory footprint of cEclat when the number of threads is increased. This might render problematic if very many threads are used. Furthermore, the partitioning of the load based on the frequent items might be too coarse-grained in such cases.

6.4. Summary and Discussion

In this chapter, we introduced cEclat, which forms an improved version of Eclat. cEclat uses tid-limaps to represent the filtered transactions that are obtained within the second scan of the dataset being mined. A single tid-limap consists of a tid-list and tid-bitmap and requires in average only 2 bytes of memory per tid; it is intersected using our parallel list and bitmap intersection algorithms of Chapter 3. Various optimizations are further employed to speed up parallel support counting, which is realized by partitioning the equivalence classes. We evaluated the performance of cEclat on several realistic and synthetic datasets. We found that the tid-limap has always the smallest memory footprint but performs only better for larger $\xi$ values than the pure tid-list approach. For small $\xi$ values, the tid-list representation combined with the speculative list intersection algorithm performs best. This holds also true for the comparison with existing, highly efficient implementations of Eclat. Single-threaded cEclat achieves—except for very large result sets on some datasets—high speedups over all of these implementations; it has further always the smallest memory footprint on all tested datasets and $\xi$ values. Moreover, cEclat is highly scalable, i.e., scales well up to 32 threads on our large test system, which has 32 cores.

A remaining issue is load balancing in cEclat. Although cEclat scales well on our test systems, the employed equivalence class partitioning scheme, i.e., partitioning based on the frequent items, will become an issue as the number of threads is largely increased. The number of frequent items will then be too small to keep all threads busy. A solution for this problem would be to build equivalence classes based on the combination of two and more frequent items. This, however, requires that the threads jointly build the tid-sets for the respective equivalence classes since initially (i.e., after the conversion) tid-sets were only created for the
6.4. Summary and Discussion

frequent items. Hence, synchronization is required for building these common tid-sets whereas they can be then mined without synchronization.

Another issue of CECLAT is the adjustment of the transaction classifier parameter $b_{\text{sparse}}$. On sparse datasets with even $b_{\text{sparse}} = 16$, almost all transactions are represented as tid-lists, which is in this case the best representation regarding high mining performance and low memory footprint—although our experiments indicate that FP-GROWTH or APRIORI often show a better performance on sparse datasets than ECLAT. On dense datasets, the tid-bitmaps are the representation of choice when the memory footprint is the main concern. In fact, CECLAT’s memory footprint could be as twice as high if only tid-lists are used instead of tid-limaps comprising tid-bitmaps. For low execution times, however, it is beneficial to use a larger amount of tid-bitmaps ($b_{\text{sparse}} \approx 16$) only when the result itemsets are mainly short; otherwise it is better to have larger tid-list sections within the limaps ($b_{\text{sparse}} < 16$) because the intermediate results for the deeper recursions are then lower, which leads to faster intersections. Hence, we need to find a tradeoff between memory usage and performance for a given dataset and $\xi$ value. If main memory is rare, we should always use $b_{\text{sparse}} = 16$ to avoid expensive thrashing. In the remaining cases, statistics could be gathered within the first scan of the dataset or even a sample could be mined to estimate the length distribution of the returned itemsets. An optimal value for $b_{\text{sparse}}$ could be selected based on this estimate.

CECLAT could also be improved with further optimizations. In our experiments, we found the perfect extension pruning and the range intersection optimization—both employed by existing ECLAT implementations—indeed useless or even counterproductive when mining datasets with high $\xi$ values but beneficial for low $\xi$ values. The former optimization could be added to CECLAT without greater changes. The latter optimization, however, requires that the filtered transactions are sorted lexicographically. As mentioned in Chapter 5, sorting might be responsible for a large fraction of the mining algorithm’s runtime and thus must be performed in parallel. The parallel tree building algorithm proposed in the next chapter could be used for that purpose: An FP-tree could be build in parallel from the filtered transactions and is afterwards traversed in depth-first order to obtain the sorted transactions.
7. CFP-growth

In this chapter\(^1\), we propose parallel CFP-growth, which constitutes a memory efficient and highly scalable multi-threaded version of FP-growth. CFP-growth works solely on a compressed FP-tree that requires significantly less memory than a regular FP-tree. This allows us to avoid out-of-core processing in many cases where it is needed with FP-growth, and reduces the cost of out-of-core processing in the remaining cases due to favorable access patterns. CFP-growth’s data structures introduce only a small overhead; it has thus a similar runtime as FP-growth when the dataset is small, but is significantly faster when the data is large. The parallel version of CFP-growth relies on efficient tree partitioning and load balancing techniques. The main idea is to incorporate knowledge from the tree being built into the tree partitioning such that the load can be evenly distributed among a large number of cores. This is required to achieve good scalability.

In Section 7.1, we explain the sequential version of the CFP-growth. It basically differs from FP-growth only in the employed data structures. Instead of one “core” data structure for complete mining process, CFP-growth uses for each phase a tailor-made data structure. The CFP-tree is optimized for the build phases; it is based on structural differences to the FP-tree and a highly tuned physical representation by means of a ternary tree. The CFP-array is used within each mine phase and comprises an array-based physical representation of the FP-tree using intelligent node ordering, indexing, and compression.

We propose scalable parallel versions of CFP-growth and FP-growth in Section 7.2. For both algorithms, the tree building step constitutes the main reason for bad scalability because the load is not equally distributed among the threads. We thus discuss the requirements of efficient tree partitioning schemes, which form the foundation for an effective load distribution and balancing. We start by discussing the issues of the state-of-the-art partitioning scheme and present tree-specific partitioning schemes, which incorporate characteristics of the tree being built. The tree-specific partitionings allow a much better load distribution and balancing. The tree itself is built using the multiple-producer/multiple-consumer processing model, which was also used in cApriori in Chapter 5. In CFP-growth, however, producer threads distribute transactions of the input dataset to consumer threads that insert them into their exclusively assigned subtrees.

Finally, we report the results of our experimental evaluation in Section 7.3. We conducted an extensive evaluation of CFP-growth and FP-growth with various datasets. For CFP-growth, we observed an order of magnitude decrease in memory consumption on all the datasets we experimented with, whether real world or synthetic. When applied to very large datasets, the reduced memory consumption leads to multiple order of magnitude performance improvements when compared to plain FP-growth. For evaluating the parallel versions of the algorithms, we compare the efficiency of the partitioning schemes and examine the algorithms scalability on very large datasets. If the range partitioning scheme is employed, both algorithms scale perfectly on our multiprocessor system that has 32 cores.

\(^1\)The material in this chapter has been developed jointly with Rainer Gemulla and Wolfgang Lehner. The chapter forms an extended version of prior published work [Schlegel et al., 2011a].
7. CFP-growth

7.1. CFP-growth

In this section, we propose the data structures that are used in CFP-growth. We first discuss physical representations of the FP-tree and analyze its compression potential when it is represented by a ternary tree. We then discuss structural changes that increase the compression potential further and—based on that—propose the ternary CFP-tree. Finally, we discuss the CFP-array and its efficient conversion from a ternary CFP-tree.

7.1.1. Reviewing the FP-tree

An important part of all prefix tree-based algorithms is the design and implementation of the physical representation of the tree. During the build phase, one wants to quickly find (or create) the prefix corresponding to the current transaction. In what follows, we refer to the children of a node in the FP-tree as the direct suffixes of the node. Thus, to insert the transaction \{2, 4\} into the FP-tree shown in Figure 2.5 on page 21, we have to find the node of item 2 among the direct suffixes of the root, and then item 4 among the direct suffixes of the selected node. The counts of both nodes are increased by one. How the search among the direct suffixes is performed depends on the underlying physical representation.

Nodes in all physical representations share the fields id, count, nodelink, and parent; the former two fields store the item ids and their respective count values while the latter two fields are used to navigate between all nodes that have the same id and for backward traversals to the root node. The main difference of the physical representations lies in the layout and number of fields they use to connect the direct suffixes. Clearly, their insertion speed and memory consumption depends on that. Variable-sized nodes that hold \(n\) pointer for \(n\) direct suffixes suffer from memory fragmentation and overhead for copying the nodes whenever a new direct suffix is added. Hence, fixed-sized nodes are preferable. For example, a binary tree representation requires only two additional fields per node: sibling and suffix. The former connects all direct suffixes in a linked list while the latter is used for traversing “downwards” to the first of a node’s suffixes. Indeed, the nodes all have equal size but searching within the linked list is usually expensive because it leads to many cache-misses for a large tree. To solve this problem, a sibling list itself could be represented as a binary tree which leads to a representation that is similar to a ternary tree [Sleator and Tarjan, 1985].
A node in a ternary FP-tree has two fields, \texttt{left} and \texttt{right}, for searching within a node’s suffixes that form a binary tree and a \texttt{suffix} field that connects each node with the binary tree of its direct suffixes. As the other representations, a node has also the \texttt{id}, \texttt{count}, \texttt{parent} and \texttt{nodelink} fields. Figure 7.1 shows a ternary search tree for the FP-tree of Figure 2.5. For readability, parent pointers and nodelinks have been omitted. Note that left and right pointers connect nodes within a level of the FP-tree; suffix pointers move down one level.

The insertion performance of a ternary FP-tree for large datasets is often more than twice as high as for other physical representations we tested. Since each node has the same size, the ternary FP-tree also facilitates simple and efficient memory management without fragmentation. Its major drawback, however, is the high memory consumption: the five pointers alone require 40 bytes per node on a 64-bit systems. On trees with billions of nodes, this high memory requirement forces out-of-core computation and thus severely slows down computation. It is the impetus of this section to drive down this memory requirement while maintaining fast execution times. Although we focus in the following on the ternary FP-tree only, most of our techniques can be applied to other physical designs.

\textbf{Compression Potential}

To get some intuition about the size of an uncompressed FP-tree and its compression potential, we exemplarily analyze a ternary-tree FP-tree built for the \texttt{webdocs} dataset of the FIMI repository [Lucchese et al., 2004b]. With a $\xi$ value of 10\%, the FP-tree is built from 1,661,662 transactions with an average length of 46.78 distinct items. The tree comprises 50,407,635 nodes, each consisting of seven 4-byte fields (we used 32-bit pointers), and requires 1.4GB of memory. Thus even a relatively small dataset and a high value of $\xi$ value may lead to substantial memory consumption.

<table>
<thead>
<tr>
<th>Leading 0s $\rightarrow$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{item}</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>98</td>
<td>0</td>
</tr>
<tr>
<td>\texttt{count}</td>
<td>0</td>
<td>&lt;1</td>
<td>&lt;1</td>
<td>&gt;99</td>
<td>0</td>
</tr>
<tr>
<td>\texttt{nodelink}</td>
<td>67</td>
<td>33</td>
<td>&lt;1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>\texttt{parent}</td>
<td>66</td>
<td>34</td>
<td>&lt;1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>\texttt{suffix}</td>
<td>65</td>
<td>32</td>
<td>&lt;1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>\texttt{left}</td>
<td>&lt;1</td>
<td>&lt;1</td>
<td>0</td>
<td>0</td>
<td>99</td>
</tr>
<tr>
<td>\texttt{right}</td>
<td>&lt;1</td>
<td>&lt;1</td>
<td>0</td>
<td>0</td>
<td>99</td>
</tr>
</tbody>
</table>

\textbf{Table 7.1.}: Fields and number of leading zero bytes in an FP-tree for the Webdocs data

The in-memory representation of our example FP-tree is summarized in Table 7.1. As can be seen, most of the space is spent for storing zero bytes. We make the following observations:

- Roughly 53\% of the total memory consists of zero bytes. In most cases, four out of the seven fields in each node start with 3 or 4 leading zero bytes; the remaining three fields have none or only one leading zero byte.

- The \texttt{item} field, which stores identifiers rather than actual items, ranges from 1 to 262 so that two of the field’s four bytes are always zero; only 2\% of the nodes actually require 2 bytes to store the item.
7. CFP-growth

- Almost all nodes require only one byte for the count field (but the maximum value is 1,429,525).

- Only one third of the forward-traversal pointers left, right, and suffix are non-zero. This is because every ternary tree with n nodes has $2n + 1$ zero pointers and $n - 1$ non-zero pointers.\(^2\) In contrast, the backwards-traversal pointer parent and the sideways-traversal pointer nodelink are rarely zero.

- About two-thirds of all non-zero pointers require 4 bytes of memory; most of the remaining pointers require 3 bytes. This is true even though we used a virtual address space so that pointer values start at zero and are tight.

We observed similar patterns in our experiments with other datasets. As argued below, the CFP-tree improves the compression potential further and is guaranteed to be effective under mild assumptions on the data.

Lightweight Compression

As we have seen, the FP-tree has a huge compression potential. The challenge, of course, is to select compression techniques that balance compression ratio and runtime overhead accordingly. We found that entropy-based compression as well as all bit-level compression techniques have too high runtime overhead. This overhead is magnified because the FP-tree is traversed many times. For this reason, we employ static encodings that allow reasonable compression rates with only small overhead. We use variable byte encoding and three different versions of null suppression that are all described in Chapter 4 starting on page 75. We use the null suppression version from Reghbati [1981], which uses a single bit to decide whether a full word is available or not. The other two versions compress at byte-level granularity and differ only in the maximum number of encoded leading zero bytes, i.e., whether 0–3 or 0–4 leading zero bytes can be encoded.

Every bit or byte saved when storing each node of the FP-tree has significant impact on the overall memory consumption on large trees with billions of nodes. Since the various static encodings differ in compression ratio and runtime cost, we employ different compression techniques for different parts of the CFP-tree and the CFP-array. In fact, each of the four selected static encodings will play an important role.

7.1.2. The CFP-tree

As mentioned previously, the CFP-tree is a variant of the FP-tree tailored to the build phase of FP-growth. By design, most of the fields stored in a CFP-tree node have small values so that static encodings are very effective. We exploit this compression potential in an optimized ternary representation, the ternary CFP-tree. After being built, the CFP-tree is converted to an CFP-array, which are tailored to the mine phase. The CFP-array draws from intelligent node ordering, indexing, and variable byte suppression.

\(^2\)Proof: A node added to a ternary tree transforms a zero pointer into a non-zero pointer, but also adds three new zero pointers. The smallest ternary search tree with 1 node has three zero pointers.
Logical representation

This section introduces the CFP-tree; its physical representation is discussed in the next section. The main goal of the CFP-tree is to make values as small as possible so that the effectiveness of static encoding is maximized. The structure of the CFP-tree is identical to the structure of the corresponding FP-tree, but the information stored in its nodes is different. We illustrate our discussion using the CFP-tree shown in Figure 7.2; it corresponds to the FP-tree of Figure 2.5 on page 21.

In what follows, we only describe differences between the FP-tree and CFP-tree. The first such difference concerns the link structure. An analysis of FP-growth shows that the build and mine phases require disjoint sets of links. In fact, to build an FP-tree, only child links are necessary because traversal during insertion is from top (root) to bottom (inner node or leaf). Since the CFP-tree is designed for only the build phase, we can safely omit parent links or nodelinks. As shown in Figure 7.2, this leads to a significant reduction of the number of links. Note that omitting these links is not beneficial for the FP-tree itself because they are needed during the mine phase. In contrast, the CFP-array handles parent and nodelinks in a different fashion; they are reconstructed (but not stored) during the transition from the CFP-tree to the CFP-array.

The remaining differences concern the data stored in each node. Since an item may occur many times in the FP-tree, both the FP-tree and the CFP-tree make use of item identifiers instead of working with the actual items. A supplementary dictionary is used to store the mapping between identifiers to actual items. We assign identifiers in increasing order of item frequency: The k-th most frequent item gets identifier k. With this choice of item identifiers, frequent items require less bits to store; e.g., each of the 256 most frequent identifiers can be stored in just one byte. Under the assumption that item frequencies are skewed (which makes frequent itemset mining useful), the space savings are significant. Moreover, the value of the identifier stored in each node is expected to be close to the identifiers stored in its children; cf. Figure 2.5. This property is exploited in the CFP-tree, which replaces the item field of the FP-tree by a Δitem field. The Δitem field stores only the difference of the item identifier from the respective parent node. To obtain the actual identifier, simply accumulate the values of all Δitem fields on the path from the root node to the node of interest. For example, node (4 | 13490) of the FP-tree of Figure 2.5 has item identifier 4. The corresponding node in...
7. CFP-growth

<table>
<thead>
<tr>
<th>Leading 0s →</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Δitem</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>100%</td>
<td>0%</td>
</tr>
<tr>
<td>pcount</td>
<td>0%</td>
<td>0%</td>
<td>&lt;1%</td>
<td>3%</td>
<td>97%</td>
</tr>
</tbody>
</table>

Table 7.2: Fields and number of leading zero bytes in the CFP-tree for the Webdocs data

the CFP-tree of Figure 7.2 is labeled (2 | 13490); its item identifier can be reconstructed by adding the Δitem fields of nodes (1 | 131732), (1 | 41648), and (2 | 13490).

Perhaps surprisingly, delta encoding is not beneficial for count values. Although the counts along the path from the root to a leaf form a non-increasing integer sequence, the count of a node is often much smaller than, and very rarely equal to, the count of its parent; Figure 2.5 illustrates this property. In fact, the delta to the parent is often greater than the actual count value. Moreover, storage of delta counts would lead to decreased efficiency when updating the tree: The delta counts of a large number of nodes would have to be modified to incorporate an insertion or update.

Instead of using deltas, the CFP-tree employs non-cumulative count values. It replaces the count field of the FP-tree by a pcount field (for partial count). Recall that the insertion of a prefix into the FP-tree increases all counts along the path corresponding to the inserted prefix by 1 (because all these paths are contained in the transaction). Instead, insertion into the CFP-tree increases only the pcount field of the final node in the path. One can show that the value of the count field of a node in an FP-tree is equal to the sum of the pcount fields of the corresponding CFP-tree node and all its children. Thus, for every corresponding pairs of nodes, we have count ≥ pcount. For example, consider node (1 | 41648) in Figure 7.2. To obtain the count of the node, take its pcount value add the pcount values of nodes (1 | 27250), (1 | 12520), and (2 | 13490). This gives 94,908, which equals the count of the corresponding node in Figure 2.5. Note that the sum of all non-cumulative count values of a CFP-tree is equal to the number of transactions that generated the tree. Often, the CFP-tree has many more nodes than transactions so that the average value of the non-cumulative count is less than 1; this makes lightweight compression extremely effective.

Table 7.2 shows the distribution of leading zeros for the fields in the CFP-tree built on the webdocs dataset. When compared to Table 7.1, one observes that the pcount field is almost always zero, while count is never zero. This means that there is a huge number of nodes that do not correspond to a full transaction in the database (these nodes correspond to subsets of transactions). We also observe that delta coding of item identifiers shows only a minor improvement on this particular dataset. This is because the number of frequent items in this dataset is very low (just 262); significant improvements can be observed for smaller support values or datasets with more distinct items.

The Ternary CFP-tree

The ternary CFP-tree is a highly-compressed physical representation of the CFP-tree. It exploits the fact that the CFP-tree contains many small values. Each node in the ternary tree consists of compressed versions of the data fields from the CFP-tree (Δitem, pcount) and, in most cases, the relevant pointer fields of the ternary tree (left, right, suffix). We use different compression techniques for the various fields so as to minimize the overall
memory consumption. Moreover, we reduce the number of pointers by embedding small leaf nodes into their parents and representing “chains” in the tree as arrays.

Since the data fields $\Delta_{\text{item}}$ and $\text{pcount}$ usually have very small values, we compress them using leading zero byte suppression. Recall that leading zero byte suppression makes use of a compression mask that indicates how many zero bytes have been omitted. The length of this mask is either 3-bit (0–4 bytes omitted) or 2-bit (0–3 bytes omitted). In all our experiments, the $\text{pcount}$ value was equal to zero for the majority of the nodes in the CFP-tree; cf. Table 7.2. In contrast, the $\Delta_{\text{item}}$ field is arguably never 0. Thus, a 3-bit compression mask pays off for $\text{pcount}$, while a 2-bit compression mask is sufficient for $\Delta_{\text{item}}$. Note that apart from increased compressibility, usage of non-cumulative counts also reduces the (de)compression overhead because only a single $\text{pcount}$ field has to be updated when inserting a prefix that already exists in the tree.

We reduced the size of each of the three pointer fields $\text{left}$, $\text{right}$, and $\text{suffix}$ from 64 bits to 40 bits, which is sufficient to address 1TB of main memory. Furthermore, all three pointers are compressed using null suppression, i.e., we use a presence bit to indicate whether or not a pointer is null. This amounts to 3 presence bits in total. The pointer fields of a ternary CFP-tree with $n$ nodes compress to a fraction of $\frac{n-1}{3n}$ of the size of the respective fields in the ternary FP-tree (since there are $n-1$ non-zero pointers). For large $n$, the size of the pointers in the ternary CFP-tree averages to 43 bits, while the size of the pointers in the ternary FP-tree averages to 120 bits (also assuming 40-bit pointers).

To summarize, a standard node of the ternary CFP-tree is stored as follows. The first byte holds the compression mask for all fields ($2+3+3=8$ bits). The subsequent bytes hold $\Delta_{\text{item}}$ and $\text{pcount}$ field without leading zero bytes, followed by 0–15 bytes containing the non-zero pointers. Figure 7.3 gives an example of ternary CFP-tree compression as discussed thus far. The first two bits of the compression mask are set to 11 and indicate that $\Delta_{\text{item}}=3$ has three leading zero bytes. The next three bits of the compression mask are set to 100 because $\text{pcount}=0$. The remaining bits are set to 010 and indicate zero values for the $\text{left}$ and $\text{right}$ pointers and a non-zero value for the $\text{suffix}$ pointer. The compressed node requires 7 bytes in total.

In addition to standard nodes, the CFP-tree comprises embedded leaf nodes and chain nodes. Embedded leaf nodes are based on the observation that the smallest possible standard node requires just 3 bytes: one byte for each the compression mask, the $\Delta_{\text{item}}$ field, and the $\text{pcount}$ field, and zero bytes for the pointers (i.e., the node is a leaf). Since this is less memory than required for storing a pointer, we embed such small leaf nodes within the respective pointer...
7. CFP-growth

field of their parents. The first byte of the pointer field is used to distinguish between a pointer (≠255) and an embedded leaf node (=255); our memory manager never uses memory addresses that start with 255. All leaves with $0 \leq \Delta \text{item} < 256$ and $p\text{count} < 16,777,216$ are stored in the parent. Thus an embedded leaf consists of a marker byte, one byte for the value of $\Delta \text{item}$, and three bytes for the value of $p\text{count}$. We found that in particular for datasets with small average transaction length (i.e., datasets with a CFP-tree of low height), node embedding achieves significant memory reduction.

Interestingly, experiments on several datasets with low $\xi$ values suggest that often more than 90% of the nodes in a CFP-tree follow exactly the node layout of the example of Figure 7.3: a $\Delta \text{item}$ value lower than 256, a zero $p\text{count}$ value, zero left and right pointers, and a suffix pointer. Note that the suffix pointer consumes most of the space in the node. This gives rise to a further optimization technique in the form of chain nodes, which were originally proposed for Patricia tries [Knuth, 1973]. The key idea is as follows: Whenever there is a chain of multiple nodes of the pattern described above, we store just the item identifiers of the nodes in the chain. To indicate the presence of such a chain node, we set all three bits of the $p\text{count}$ compression mask to one. This reduces the number of possible states of $p\text{count}$ in a standard node, but forms no problem in practice. The remaining 5 bits of the compression mask store the number of nodes in the chain, followed by that many bytes each containing one $\Delta \text{item}$ value. The chain node is terminated by a suffix pointer that connects it to the child of the last node in the chain. In total, a chain node requires $m + 6$ bytes for a chain of length $m$. For example, if six FP-tree nodes are merged into a single chain node, then the space consumption of each node averages to only 2 bytes.

Memory Management

We extended the memory manager (cf. Section 5.1.2 on page 102) that is used in cApriori and cEclat to support fast memory allocation and deallocation for variable-sized nodes, i.e., the extended memory manager is aware of (1) our compression scheme leads to different node sizes and that (2) node sizes may change as new transactions are processed (e.g., count values may grow, references to new nodes may be added). We exploit the fact that the memory footprint of a compressed node is within a fixed range, i.e., from 7 up to 24 bytes (three 40-bit pointers; 9 byte for $\Delta \text{item}$, $p\text{count}$, and compression mask); smaller nodes are stored as embedded nodes.

Figure 7.4 illustrates our memory manager. The memory is divided into two continuous parts: used memory (consisting of nodes and free memory chunks) and unused—but pre-allocated—memory. A next-free pointer separates these two parts. In used memory, we connect all free memory chunks of the same size with a LIFO queue. The elements of the

---

3The remaining seven states allow $p\text{count}$ values taking 0–6 bytes, which is sufficient for all datasets in which the most frequent item occurs less than $2^{14}$ times.
queue are stored directly in the memory chunks being managed. This is possible because
the queue actually contains the locations of these free memory chunks, and only 5 bytes are
needed to store such a location. Whenever a node grows/shrinks from $b_1$ bytes to $b_2$ bytes,
we dequeue a $b_2$-byte chunk from the $b_2$-byte queue. If no such chunk exists, we create a new
$b_2$-byte chunk at the next-free pointer, which we subsequently increase by $b_2$ bytes. We
then store the updated node in the so-obtained chunk, and enqueue the node’s old $b_1$-byte
chunk to the $b_1$-byte queue (thereby freeing its memory).

The extended memory manager avoids fragmentation because almost all released memory
chunks from growing nodes are eventually reused. In experiments, we found that often more
than 99% of the chunks are reused. Although the extended memory manager is very efficient,
it still imposes overhead whenever a node shrinks or grows. For this reason, we made use of
compression techniques that avoid fluctuations of the node sizes to the extent possible.

7.1.3. The CFP-array

The CFP-array is a data structure optimized for the mine phase of CFP-growth. Recall
that the mine phase of FP-growth performs both sideways and backward traversal and
requires quick access to the value of item and count. To provide the navigational and data
information, the CFP-array consists of an array of compressed triples and a small index. One
of the key observations behind this layout is that the CFP-array is static, i.e., built once and
never changed. We can thus safely ignore update costs. In what follows, we describe how
to build the CFP-array from a standard FP-tree. After that, we will discuss the conversion
process from a ternary CFP-tree that is slightly more involved.

Consider a representation of the FP-tree as an array of quadruples of form (item, count,
parent, nodelink). An FP-tree with $n$ nodes corresponds to an array with exactly $n$ elements.
Observe that the order of the FP-tree nodes in the array does not matter, but that nevertheless
each node has a certain position in the array. The CFP-array exploits this fact by picking a
node-to-position mapping that allows us to omit the nodelink field.

We assign positions to nodes such that for any pair of nodes $u$ and $v$ with $u.item < v.item$,
the position of $u$ is smaller than the position of $v$. This property ensures that the item
field is non-decreasing and that all nodes corresponding to a particular item are clustered
together; they form consecutive subarrays within the CFP-array. Since the purpose of the
nodelink field is to connect nodes that correspond to the same item, and since we gave all
those items consecutive positions and thus made them easy to find, the nodelink field is
redundant and can be omitted. Instead, we maintain a small item index (also in the form
of an array) that maps each distinct item to its first occurrence in the CFP-array; we call
this position the starting position. Sideways traversal is performed by processing the nodes
from the starting position onwards. The traversal ends when a node with a different item is
encountered. Observe that the item index also contains all the information that is required to
determine the item of a node at some position $i$: It is the item that with the largest starting
position less than or equal to $i$. Thus, the item field could potentially be omitted as well, but
we chose not to do so for performance reasons.$^4$

Just as done in the CFP-tree, we increase the compression potential of static encodings by
making the fields of the CFP-array as small as possible. As before, we replace the item field
by the difference $\Delta$item to the item identifier of the parent. To obtain the actual identifier

$^4$If we omit the item field, we have to reconstruct it for each backward traversal via binary search in the item
index.
of a node during a backward traversal, subtract all $\Delta_{\text{item}}$ values on the path from the leaf node to the node of interest (excluding its $\Delta_{\text{item}}$). The identifier of the leaf node itself is obtained via the item index. To encode the parent field succinctly, we do not store the parents global position in the CFP-array. Define the local position of a node as its position within its subarray, i.e., the subarray that contains all the nodes with the same item. We replace the parent field by a $\Delta_{\text{pos}}$ field, which contains the delta between the local position of the child and the local position of its parent. The conversion of the CFP-tree into the CFP-array is performed such that this delta encoding is very effective. The final field of each triple is the count value. In contrast to the CFP-tree, we do not store partial counts because we do not have direct access to the descendants of a node, and thus cannot efficiently reconstruct counts from partial counts.

An example FP-tree and the corresponding CFP-array are shown in Figure 7.5. There are three subarrays, one for each of the three distinct items. Node (3 | 3692) is stored in the subarray for item 3 at local position 2 (we use 0-based local positions). Its parent is (2 | 21194), which is stored at position 1 in its subarray. Thus, we have $\Delta_{\text{item}} = 3 - 2 = 1$ and $\Delta_{\text{pos}} = 2 - 1 = 1$. Note that node (2 | 21194) does not have a parent; this is indicated by setting $\Delta_{\text{item}} = \text{item} = 2$.

![Figure 7.5: Conversion of an FP-tree to a CFP-array](image)

We compress each of the three fields $(\Delta_{\text{item}}, \Delta_{\text{pos}}, \text{count})$ using variable byte encoding in the order specified. We prefer this technique because it requires no separate compression mask, and because both $\Delta_{\text{item}}$ and $\text{count}$ require almost always just one byte (cf. Tables 7.1 and 7.2). The field order allows us to perform backwards traversal without decoding the count field; we thus avoid the expensive length lookups associated with variable byte encodings. Note that since each array element is stored using a variable number of bytes, we use local positions that refer to the byte offset of a node within its subarray in the physical representation. This allows us to directly access a node given its local position.

**Conversion**

Conversion of a ternary CFP-tree to its respective CFP-array is required in between every build phase and its subsequent mine phase. We perform this transformation in two passes over the ternary CFP-tree. The first pass is required to determine the size of the CFP-array, while the second pass is used for the actual transformation. The transformation is
not performed in-place so that the main memory has to be large enough to hold both data structures simultaneously. Fortunately, this does not increase the peak memory consumption of CFP-growth because (1) a CFP-tree can be partitioned into smaller subtrees, which can be converted one after another, and (2) a subtree can be discarded immediately after its conversion, i.e., only subtree being converted needs to be held additionally to its CFP-array. The additional memory from the conversion is reused for building the conditional trees during the mine phase. Nevertheless, the additional memory should be much smaller than the memory required by the entire CFP-array because the subtrees require often only a small fraction of its size. For this reason, the partitioning of the CFP-tree plays an important role; the partitions should have a similar size because the largest partition (i.e., subtree) determines the size of the additional memory. Obtaining such a partitioning is discussed in the following section. Besides the small additional memory consumption, our experiments indicate that the conversion consumes only a small fraction of the overall execution time of CFP-growth.

The conversion process of a single subtree is illustrated in Figure 7.5. The size and starting position of each subarray in the CFP-array can be determined via a recursive depth-first traversal of the CFP-tree. For each node, we compute the space consumption of the variable byte encodings of the count, Δpos, and Δitem fields and accumulate it individually for each item. The transformation itself also performs a recursive depth-first traversal, and writes each node directly into its final position within the CFP-array. This position is easily determined from the information gathered in the first pass. The Δpos fields are obtained via a stack that contains the path taken from the root to the current node during the traversal.

7.2. Parallel CFP-growth

In this section, we discuss a parallel version of the CFP-growth algorithm. It is optimized for multiprocessor systems providing a large number of threads. For that purpose, we start discussing the problems of current parallel versions of FP-growth that arise when they are executed on such systems. We then present our solutions, which are applicable both to CFP-growth as well as FP-growth. Similarly to the previous section, we first discuss the build phase and thereafter the mine phase of parallel CFP-growth.

7.2.1. Lock-free tree building

To build the initial CFP-tree, the dataset needs to be scanned twice: the first scan reveals the frequent items while the second scan is used to build the tree. As before, we consider that all input data is already loaded in main memory (see the discussion in Section 5.1 on page 100). Otherwise, it might not be reasonable to optimize or parallelize the tree building step, in case loading the data is the main bottleneck.

Obtaining the frequent items in parallel is done as for the parallel versions of cApriori and cEclat. The dataset is divided into equal-sized parts that can be processed independently by multiple threads. Building the CFP-tree in parallel is more involved. As discussed in Chapter 2, all threads must build a single tree together to avoid the unaffordable high memory consumption of the multiple tree approach where each thread builds an own tree. Building the single tree in parallel, however, requires either synchronization between the threads or a partitioning of the tree. As discussed earlier, locks and other synchronization methods cannot be used because of their high costs so that the tree has to be partitioned. Hence, we divide the initial CFP-tree into multiple smaller subtrees that can be maintained in parallel without
synchronization, i.e., each thread has a single or multiple subtrees exclusively assigned in which it can insert new nodes and increase the nodes’ count values without locking or other synchronization primitives.

Similar as for partitioned counting (cf. Section 5.2.1), each thread must obtain and process all the transactions that belong to its assigned subtrees. For obtaining these transactions, each thread can either (1) read the complete dataset or (2) read parts of the dataset and communicate transactions that belong to other threads to them. We do not use the first variant because the redundant work performed by each thread limits scalability. Instead, we use the second variant and employ—as for partitioned counting in \textsc{cApriori}—the multiple-producer/multiple-consumer model to communicate transactions between the threads. We postpone details about how the processing model is employed in \textsc{CFP-growth} to Section 7.2.3 (staring on page 170) and will first explain how a good partitioning of a CFP-tree can be obtained.

A \textit{partitioning} of a CFP-tree describes how the tree is split into independent, non-overlapping subtrees or partitions. Its properties have a large impact on the scalability of \textsc{CFP-growth}. Recall that the key for a good performance and scalability is keep all threads busy during processing. Thus, a partitioning must allow to equally distribute the load among the available threads, however, it must also be adapted to the specifics of the CFP-tree and the employed processing model. For \textsc{CFP-growth}, a partitioning must fulfill the following properties:

**Partition size** Each partition has a \textit{weight} that denotes the fraction of the time for building the respective subtree and the time for building the full tree. For example, a partition weight of 1/2 means that half of the time for building the full tree is spend for building the partition’s respective subtree; it thus limits the maximum speedup to \textit{two} because the thread that processes the partition requires half of the time of the sequential algorithm (cf. Amdahl’s Law [Amdahl, 1967]). Hence, the partition with the highest weight determines the maximum speedup. For this reason, none of the partition weights shall be much greater than 1/t when the tree is build with t threads.

**Number partitions** The number of partitions affects the effectiveness and efficiency of load distribution and the processing model. If there are only few partitions, the load might not be equally distributed. For example, if the number of partitions is equal to the number of threads, then the load can only equally distributed when all partitions have all the same weight. However, if the number of partitions is too high, the overhead for the processing model considerably increases (cf. Section 7.2.3). We found empirically that eight times the number of threads is a good tradeoff for the number of partitions, e.g., 256 partitions for 32 threads.

**Stable partitions** All partitions should be stable during tree building, i.e., none of the respective subtrees should be split, because the nodes of a subtree are stored within the same continuous memory area. Thus, if a subtree is split into two subtrees, several of their nodes share the same cache lines. This leads to false sharing and—in case a multiprocessor system is used—expensive communication between different processors. Copying the subtrees to separate memory areas would solve the problem but this could be quite expensive for large subtrees. Hence, the partitioning should be obtained before the tree is build and then not be changed.
Summing up, a partitioning should have $8 \cdot t$ partitions for $t$ threads whereby none of the partitions shall have a weight that is larger than $1/t$. Furthermore, the partitioning should be stable because of the high costs of a split of a subtree. In what follows, we describe how to obtain a partitioning that fulfills these requirements.

### 7.2.2. Tree partitioning schemes

A tree partitioning scheme is used for obtaining a partitioning for a CFP-tree. As discussed in Chapter 2, existing parallel versions of FP-GROWTH [Chen et al., 2006, Li and Liu, 2007] only employ the hot-item partitioning scheme. We will now explain it in more detail and discuss why it does not fulfill the aforementioned desired properties. In the following, we discuss how the transactions are assigned to the partitions. All transactions that are in the same partition form eventually an independent subtree of the final CFP-tree. We assume throughout that the items of the transaction being partitioned are ordered with respect to their frequency in the transaction database, i.e., the transaction’s first item occurs more frequent in the database than its second item.

A hot-item partitioning divides the transactions into partitions based on the availability of the first $n$ global frequent items (also denoted as hot items) within them. From a higher-level view, assigning a transaction to a partition could be seen as traversing a perfect binary decision tree of height $n$. Within each level the decision is made based on the presence of a certain item, i.e., the availability of the $i$-th most frequent item is used for the decision at the $i$-th level within the tree. Notice that the decision tree must not be materialized for this assignment.

Figure 7.6 illustrates the decision tree for the hot-item partitioning with $n = 3$ hot items; each transaction is assigned to one of eight partitions using three decisions. A transaction containing the items $\{2, 3, 7, \ldots\}$ would be put into the partition P3 because the transaction does not contain the item 1 but contains the items 2 and 3.

Besides of its simplicity, this partitioning scheme is unsuitable for tree building using a large number of threads. As indicated by our experiments on several realistic datasets (cf. Section 7.3.4), it often fails to provide partitionings where the weight of the largest partition is below $1/20$ and the number of partitions is low (e.g., 256 partitions for 64 threads). The main reason for this behavior is that not all of the most frequent items co-occur with all other most frequent items frequently together. Depending on the dataset, they emerge only with certain other frequent items. In some realistic datasets, moreover, none of the most frequent items co-occurs frequently together with another of the most frequent items. Hence, increasing the number of partitions does not solve the problem. Even for the large number of $2^{24}$ partitions, our experiments indicate that there often still exists a partition with a weight of $1/10$. For some datasets, the weight of the largest partition is even as high as $1/3$, which
7. CFP-growth

limits the maximum speedup to 3. As we will discuss later, increasing the number of partitions beyond $2^{12}$ would render the partitioning unusable for our processing model; the overhead for assigning the transactions to their partitions would be unacceptable high whereas most of the partitions would be empty.

Tree-specific partitioning schemes

A partitioning usually has more well-balanced partitions that have a similar size if it is adapted to the tree being built. For obtaining such a tree-specific partitioning, we require (1) knowledge about the final CFP-tree like the weights of certain subtrees as well as (2) a partitioning scheme that is capable of integrating this knowledge into a partitioning. Clearly, the exact weights of a CFP-tree’s subtrees are only available after it is build. Fortunately, approximate weights are sufficient for a good tree-specific partitioning and they can be derived using a small sample of the transaction database. We will now discuss two different partitioning schemes that are capable of integrating knowledge about the final tree into their partitionings and differ only in the way how they assign a transaction to a partition.

The tree-specific hot-item partitioning scheme uses—as the partition scheme it derives from—binary decisions based on the presence of the most frequent items to assign transactions to partitions. The decision tree, however, has variable length paths, i.e., it is not a balanced (perfect) tree anymore. The path length is adjusted so that the partitions that are at the end of the paths have a similar weight. For example, if almost all transactions include the most frequent item 1, then all paths to leaf nodes including the “1-is-present” edge are usually longer than the paths including the “1-is-not-present” edge. Notice that the decision tree must be materialized for this partitioning scheme.

![Tree-specific hot-item partitioning](image)

Figure 7.7.: Tree-specific hot-item partitioning

Figure 7.7 illustrates a tree-specific hot-item partitioning. The respective decision tree has also six nodes as the tree shown in Figure 7.6 but this time its paths have a different length. In this specific example, the decision tree’s paths on the right side are in average longer contributing to the fact that there exist more transactions that include the most-frequent item 1 than transactions not including it. Half of the partitions are contributed to transactions that include the two most-frequent items, i.e., the partitions P4, P5, P6, and P7.

The tree-specific hot-item partitioning leads often to more equally-sized partitions compared to the “pure” hot-item partitioning. Nevertheless, it does not provide a good partitioning for
some datasets and still has many almost empty partitions. Especially, for datasets where the most frequent items do not co-occur frequently, the decision tree degenerates to a single path; it is located on the decision tree’s left side and comprises only “not-present” decisions. Besides the unsuitable partitioning for such datasets, the transaction–partition assignment is getting expensive because roughly $n$ decisions must be made while searching the respective partition for a transaction that has the $n$-th frequent item as its first item.

The range partitioning scheme avoids the problems of both hot-item partitioning schemes. Instead of checking the presence or absence of the hot items, it is based on checking whether the first few items of a transaction lie within certain ranges. For example, a range partitioning could put all transactions whose first item is smaller than 10, in the range from 10 to 20, and greater than 20 into three different partitions. For a larger number of partitions, the range checking is applied recursively on further items of the transaction; whenever the $n$-th transaction item falls into a range that covers only a single item, the $(n + 1)$-th item may be checked. This allows a very fine-grained partitioning of the transactions. For a better understanding of the range partitioning scheme, we will now describe a data structure that is suitable for obtaining efficiently the partition for a given transaction.

The range partition tree is a prefix tree that is used for the transaction–partition assignment. For each transaction, it is traversed from its root node to a certain leaf node that constitutes a partition the transaction belongs to. A node within the tree has multiple entries; each of them is either a single-item entry or a range entry. A single-item entry at level $l$ is followed by all transactions whose $l$-th item is equal to the entry’s item. It refers either to a partition (i.e., a leaf node) or to a further inner node. In the latter case, the $(l + 1)$-th transaction item is checked at the next node. A range entry always refers to a leaf node; if it is located on level $l$, it is followed by all transactions whose $l$-th item is within the entry’s item range. The last entry of a node is always a range entry; it is followed by all transactions that did not match to a previous entry of the node. For the traversal, each entry holds an item value and a pointer to the next node. A flag is used to mark leaf nodes. To distinguish between a single-item entry and a range entry, the difference to the item value of the subsequent entry is used. If the difference is 1, then it is a single-item entry while otherwise it is a range entry.

An example of a range partition tree is depicted in Figure 7.8. As in the partitioning examples before, transactions are put into one of eight partitions. Within the first node—the
root node with two single-item entries and two range entries—the mapping is based on the first item \( t[0] \) of the transaction. For \( t[0] = 1 \), the decision is made based on the transaction’s second item \( t[1] \) and—if \( t[1] = 2 \)—also its third item \( t[2] \). Otherwise, if \( t[0] = 2, t[0], 3 \leq t[0] \leq 5 \), or \( t[0] \geq 6 \), the transaction is assigned to partition P5, P6, or P7, respectively.

The actual search procedure for obtaining a transaction’s assigned partition is quite fast and works as follows: Starting at the root node, the tree is traversed—using a transaction’s items—until a leaf node is found. Within a node at level \( l \), the transaction’s \( l \)-th item is compared against the item values of the entries. Always the entry is followed that is either (1) the node’s last entry or (2) followed by the entry with the smallest item value that is larger than the transaction’s \( l \)-th item. For finding this entry within a node, we sequentially scan all of its entries. This can be done very fast because all of a node’s entries are stored continuously in memory. Furthermore, a range partition tree usually fits well into the L1-cache—even for many partitions. As soon as a leaf node is reached, the respective partition number is returned.

The range partitioning scheme is a super-set of the hot-item partitioning as well as the arithmetic partitioning scheme [Buehrer et al., 2006], i.e., all possible partitionings created by these schemes could be reproduced by the range partitioning scheme. For this reason, it has a much larger search space for possible partitionings but allows more elaborate ones that lead to more equally-sized partitions. For example, both hot-node partitioning schemes fail to provide equally-sized partitions for datasets where the most frequent items do not co-occur frequently together. A possible range partitioning for such datasets, however, would be to partition the transactions solely using the transaction’s first item, i.e., define \( p \) ranges for this item where \( p \) denotes the number of partitions.\(^5\)

### 7.2.3. Multiple-producer/multiple-consumer tree-building

Building the initial CFP-tree could be done either with or without pre-processing of the dataset being mined. For example, Li and Liu [2007] employ approximate sorting of the transactions before building the tree; this improves the temporal locality while building it but comes at the cost of an additional run over the transaction database—after counting the frequent items and before building the tree. Moreover, it would increase the memory footprint of CFP-GROWTH dramatically because the dataset has to be replicated while approximately sorting the transactions within it. Although the sorted copy of the dataset could be stored in a more efficient representation than the original dataset, the main memory might not be sufficient when pre-processing is applied on large datasets. For this reason, we omit any such step and build the CFP-tree while scanning the dataset the second time.

#### Creating the partitioning

Before actually building the CFP-tree, we first have to generate a tree-specific partitioning and assign its partitions to the available threads. As mentioned before, we require (approximate) weights of certain subtrees of the CFP-tree being build to create such a partitioning. To support that, we introduce a tailor-made data structure: The WFP-tree—for weighted frequent-pattern tree—is identical to an FP-tree but has no node- or parent-links and further

\(^5\)Depending on the defined ranges, such a partitioning could also be created with the arithmetic partitioning scheme [Buehrer et al., 2006] that partitions the transactions solely using their first item. Nevertheless, it cannot provide partitionings where the ranges of items other than the transaction’s first item are checked.
7.2. Parallel CFP-growth

stores in its nodes the weight of the following subtree instead of the regular count value. This weight value of a node is basically the opposite of the partial count, i.e., it is the sum of the node’s count value and the weight values of all of the node’s direct children. Figure 7.9a illustrates an FP-tree that is build using a small number of transactions; the corresponding WFP-tree is illustrated in Figure 7.9b. The weight of the node (1, 202) in the WFP-tree, for example, is obtained by adding the count value of its corresponding node (1, 140) of the FP-tree to the weight values of its direct children nodes (2, 57) and (3, 5) of the WFP-tree.

A WFP-tree is build from a small transaction sample from the dataset that is used to build the CFP-tree. Whenever a sampled transaction is inserted into a WFP-tree, the weight value of each node that corresponds to the prefix of the transaction is individually increased by the length of the remaining transaction. For example, if a transaction \{1, 2, 3\} is inserted into the WFP-tree depicted in Figure 7.9b, then the weight values of the nodes (1, 202), (2, 57), and (3, 5) are increased by 3, 2, and 1, respectively. Since the dataset is located in main memory, the sample can be extracted from the complete file. This reduces the chance of obtaining a biased sample. We found, however, that looking at the first few thousand transactions already provides us a good sample that is often sufficient for obtaining a good partitioning. Building the WFP-tree itself introduces some costs but pays off with a much faster building of the full CFP-tree. As our experiments suggest, this overhead is almost negligible for large datasets, mainly because the WFP-tree is small; it easily fits into the processor’s caches. Also, the sample could be drawn in parallel and combined before the generation of the partitioning.

The partitioning, i.e., a range partitioning, is created by traversing the WFP-tree. Sets of its nodes represent partitions that are iteratively split until the desired number of partitions is obtained. We employ a simple greedy-based algorithm that splits the largest partitions in each iteration; it starts with a single partition that comprises the full tree. Furthermore, a heap is maintained for quickly finding the largest partition. For each split, the following two steps are executed:

1. If the largest partition comprises only one node, then its child nodes are split in two partitions that have a similar weight (as equal as possible). Otherwise, its current nodes are split into two partitions with similar weight.

2. The two new partitions are inserted into the heap; it is then reorganized so that the partition with the largest weight is the top of the heap.

We illustrate the algorithm on our example WFP-tree that is shown in Figure 7.9b. Table 7.3 depicts the node sets available after each iteration. The algorithm starts with a single
large partition that comprises all children of the root node. The initial partition is split into a partition comprising the node \( (1, 202) \) and a second partition comprising the nodes \( (2, 48) \) and \( (3, 15) \). After the second iteration, the partition \{ \( (1, 202) \) \} is split into two partitions by dividing the child nodes of \( (1, 202) \) into two sets. Finally, \{ \( (2, 58), (3, 15) \) \} is split because it has the largest weight.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Partition set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>{ ( (1, 202) ), ( (2, 58) ), ( (3, 15) ) }</td>
</tr>
<tr>
<td>1.</td>
<td>{ ( (1, 202) ), { ( (2, 58) ), ( (3, 15) ) }</td>
</tr>
<tr>
<td>2.</td>
<td>{ ( (2, 58) ), ( (3, 15) ) }, { ( (2, 57) ) }, { ( (3, 5) ) }</td>
</tr>
<tr>
<td>3.</td>
<td>{ ( (2, 57) ) }, { ( (2, 48) ) }, { ( (3, 15) ) }, { ( (3, 5) ) }</td>
</tr>
</tbody>
</table>

Table 7.3.: Created partitions using the greedy-based partitioning algorithm

In each of the algorithm’s iterations, the found partitions are annotated at the nodes of the WFP-tree; the range partition tree is eventually obtained by traversing this annotated tree. Afterwards, the partitions are (1) ordered based on their weight and (2) assigned in a round-robin manner to the threads. Clearly, the load is not always completely equally distributed because often splitting the largest partition in half does not lead to optimal partitioning. Nevertheless, it efficiently reduces the size of the largest partition and thus avoids the large sequential fraction that might occur for a partition with a large weight. More elaborate partitioning algorithms might be necessary for a very large number of threads but in all of our experiments on our 32-core test system the greedy algorithm was found to be sufficient.

**Build phase**

After the partitioning is created, the actual tree is build in parallel. For that purpose, the transactions need to be assigned to their respective partitions. As mentioned before, we partition the transactions at runtime using the multiple-producer/multiple-consumer model proposed in Section 5.2.1 on page 105. The model is slightly adapted for its use in CFP-growth: Multiple producer partition the transactions and distribute them to consumers that insert them into their assigned partitions (i.e., subtrees). Communication is performed using work queues that contain pages filled with transactions. Each consumer has an assigned work queue containing the transactions it should process. We collect multiple transactions within buffers before inserting them into the respective queues to avoid synchronization costs of the enqueue operation. Hence, each producer maintains for each partition such a buffer. This efficiently reduces the synchronization overhead that would be unacceptable high if each transaction would be inserted into one of the work queues individually. The complete tree-building process is then executed in two phases:

1. The tree is build using a multiple-producer/multiple-consumer processing model. Each of the participating threads is always in one of two states and switches between them based on certain conditions. The both states are as follows:
   a) A producer continuously reads pages from the input file, parses them, and partitions the transactions according the partitioning scheme. Partitioned transactions are temporary buffered in pages; whenever a page is full, it is put into the work queue of the thread that is assigned to process these pages.
7.2. Parallel CFP-growth

After each parsed input page, the producer switches to consumer state if there is at least one page in its work queue.

b) A consumer continuously reads a page from its assigned work queue and inserts the transactions within it into the respective subtree. Whenever a consumer’s work queue is empty, it switches back to producer state.

2. The tree must be converted into a tree optimized for the mine phase. This step is started as soon as all threads are finished with “producing”, i.e., a thread can already convert its assigned subtrees while other threads still process pages of their work queues.

As in partitioned counting in \textsc{Apriori}, switching threads from producer to consumer state and vice versa leads to a fine-grained load balancing. Threads with less “insertion” work tend to be more often a producer while other threads are often a consumer and insert transactions in their assigned subtrees. Thereby, processing pages from a threads work’s queue has always the highest priority because it can only be done by this specific thread, i.e., to keep the number of pages within the queues small and avoid a potential sequential fraction at the end of the build phase. If there is still an imbalance at the end of the tree-building process, it is usually compensated while converting the subtrees into mine-optimized subtrees. After all threads finished “producing”, they process the remaining pages that are within their work queues and then start converting their subtrees. Earlier finished threads can help converting the subtrees of threads that took longer processing the pages of their work queues. As found in our experiments, both techniques are very effective so that the load can be statically balanced. Dynamic load balancing techniques (e.g., redistributing assigned partitions during processing) do not need to be applied.

As before, one major concern of our processing model is the space required for buffering the transactions. Each producer requires at least a single page for each partition; this amounts to 16,384 pages for 64 threads and 256 partitions. For 32KB pages—empirically found to be a good tradeoff between memory consumption and synchronization effort—we require 0.5GB for all these pages. This value is further increased because additional “free” pages are necessary to ensure asynchronous processing of producers and consumers. Nevertheless, even if the number of threads and partitions is doubled, this is still acceptable because the memory can be reused during the mine phase. For a large number of partitions like 65,536 partitions and 64 threads, however, already 132GB are required. Hence, such large number of partitions must be avoided, which renders problematic for the hot-item partitioning that usually improves with more partitions. If the number of threads is largely increased, i.e., on future systems, more elaborate processing models need to be developed; see Section 7.4 for a discussion.

Minimization of communication between producers and consumers is further key to good performance since it could be a potential bottleneck—especially on multiprocessor systems. For this reason, we remove redundant items from the page-wise exchanged transactions. Each page holds besides the number of transactions it contains, also the number of the partition (i.e., the partition id) in which the transactions should be inserted. Recall that each thread has a single work queue but multiple partitions assigned. The partition id can be used to remove all items of a transaction that are already covered by the partitioning. For example, consider the range partitioning illustrated in Figure 7.8; all transactions that are within the partitions P1, P2, and P3 include the items 1 and 2—they pass the two respective single-item entries—and thus there is no need to communicate these items for each transaction that is with one of these partitions.
7. CFP-growth

In general, each single-item entry within the range partition tree represents a covered node that is not stored in any subtree. Hence, the range partition tree needs to provide count values for these nodes since a transaction that consists only of covered items is not inserted in any of the subtrees and thus would not be counted. Furthermore, a parent and nodelink pointer are also required for the subsequent mining. All of these fields were not compressed within the range partition tree because it is usually small and often traversed. It is further replicated for each thread to avoid synchronization while increasing the count values of the covered nodes in parallel.

7.2.4. Mining Phase

As for sequential CFP-growth, the mining starts directly after the tree is build and converted. During the parallel tree mining step, smaller conditional CFP-trees are recursively built from the initial CFP-tree using multiple threads. As for parallel FP-growth, the mining step is already well partitioned: All conditional CFP-trees that share the same first item within their associated pattern can be processed independently from the other conditional trees. Figure 7.10 illustrates conditional trees that are generated from a CFP-tree. In this particular example, two threads are used for mining. The first thread \( t_1 \) mines all conditional trees that start with item \( d \) while the second thread \( t_2 \) mines the conditional trees starting with \( c \).

![Figure 7.10.: Conditional trees for a CFP-tree (processed by two threads)](image)

Partitioning the mining task based on the global frequent items reduces thread synchronization to a minimum; it is only required for writing into the result set and dynamically assigning the frequent items to the threads. Both is fairly inexpensive compared to the costs of the repeated tree traversals that are necessary for mining the initial or a conditional tree. Nevertheless, two major issues might occur for this approach when the number of threads is largely increased: (1) more conditional trees must be held at the same time in main memory and (2) load imbalances might occur because of the coarse-grained work distribution.

During the mining step, additional memory is required for holding the conditional trees. For parallel mining, each thread has an own memory manager that provides a large continuous memory area used as a stack. A thread’s stack holds the conditional trees during the recursion steps. Whenever a new conditional tree is created, it is put on top of the stack. It is first removed from the stack, after it is itself and all of its potential successors are fully mined. For example, in Figure 7.10, the three conditional trees with the respective pattern \{d\}, \{d, c\}, and \{d, c, b\} must reside in main memory at the same time to mine the pattern \{d, c, b\}. After it is mined, the respective conditional tree can be removed from the stack while the \{d, c\}-conditional tree can first be removed after the \{d, c, a\}-pattern is mined. With only
7.3. Experiments

one thread, the peak memory required for mining is given by a single of these recursion paths. With \( n \) threads, conditional trees of \( n \) recursion paths may reside at the same time in memory. In our example in Figure 7.10, this amounts to six conditional trees while using two threads; for sequential processing at most three conditional trees must be hold in main memory at any time. Hence, the memory consumption increases when more threads are used. Nevertheless, for our tested datasets and reasonable \( \xi \) values, we did not experience a significant grow of the memory required by the algorithm. This might change for a larger number of threads. In such cases, a shared projection approach might be helpful in which multiple threads mine certain—possibly large—conditional trees together; for example, two threads in Figure 7.10 could build the \( \{d\}\) -conditional tree together and then mine independently smaller conditional trees based on it. This would reduce the memory requirements at the cost of probably reasonable increased synchronization costs.

Besides the possibly higher memory consumption, the partitioning based on the global frequent items might lead to load imbalance, especially, when the number of available threads is close to the number of frequent items. On our test systems and the used datasets and \( \xi \) values, such a situation did not occur. We experienced only very small load imbalances. However, if the number of threads is increased further, then the load imbalance might increase too. Also here, the shared projection approach could improve the load distribution. Furthermore, a task-based programming model would allow a better partitioning and thus improves load balancing. For example, after building the \( \{d\}\) -conditional tree, all of its direct successors (\( \{d,c\} \), \( \{d,b\} \), and \( \{d,a\} \)) could be defined as tasks and be processed by different threads. It must only be ensured that the \( \{d\}\) -conditional tree is not removed from the stack until all of its successors completely mined.

7.3. Experiments

We run experiments with different versions of FP-growth, which mainly differ in the core data structure and partitioning scheme they employ. In what follows, we restrict our attention to the comparison of these versions only; a detailed comparison with other frequent-itemset mining algorithms is available in Appendix C starting at page 195. We compare both sequential and parallel implementations of “plain” FP-growth and CFP-growth. Our experiments suggest that CFP-growth has a significantly smaller memory footprint than FP-growth. This allows us to process much larger datasets in main memory and thus avoiding the expensive out-of-core processing. Furthermore, the parallel implementations scale almost linear when they employ the range partitioning scheme.

7.3.1. Setup

We basically use the same hardware and software setup as used in both previous chapters (cf. Section 5.4.1 on page 117). We use the four-socket system, which has 32 cores and 128GB main memory. We implemented CFP-growth—including the ternary CFP-tree and the CFP-array—as well as a highly efficient version of FP-growth. As before, the algorithms were implemented in C++ and compiled using Intel C++ Composer XE. We reused components from the algorithms of the previous chapter, e.g., the parallel parser, parallel direct counting, and the basic memory manager. Each subtree of an FP-tree or a CFP-tree has such a memory manager assigned, which chunk-wise gathers memory (i.e., in 16MB chunks) using mremap while the respective tree grows. Thus, they avoid expensive malloc calls when new
nodes are created. Recall that the memory manager is extended for CFP-growth to (1) reduce the pointer size to 40 bits, (2) provide unpadded chunks of memory, and (3) “recycle” the memory of nodes that grow during processing.

Although, we consider a main-memory only scenario, in some experiments we use the swap space to illustrate the costs of out-of-core processing. The swap space is 128GB on an HP server disk (3G) with 1TB and 7200rpm. HP QuickSpecs [2012] specifies the disk’s average seek time as smaller than 8.5ms and maximum transfer rate as 3Gb/s.

For CFP-Growth, we restrict the maximum length of a single chain node to 15; longer chains are broken into multiple chain nodes. Although longer chains may have lower memory requirements, we found that both runtime overhead as well as memory fragmentation increase. We perform construction of new chain nodes only when a new leaf is inserted; we do not merge existing chain nodes; they may even be split when subsequent transactions are inserted.

In both of our implementations of FP-growth and CFP-growth, the initial FP-tree and CFP-tree is partitioned, respectively, so that each thread has always eight assigned partitions. Thus, for the single-threaded execution the initial tree has 8 partitions while for 32 threads it has 256 partitions. For 64 threads with enabled Hyperthreading, we also use 256 partitions. Besides load balancing for parallel processing, partitioning serves two purposes: (1) it reduces the time for building the tree since it increases the temporal locality and (2) it limits the additional memory requirements for converting a CFP-tree into a CFP-array. The latter is only beneficial for CFP-growth.

We use the realistic and synthetic datasets that are described in Section 4.3.1 (on page 85) and Section 5.4.1 (on page 118), respectively. The realistic datasets are—because of their small size—only used within the non-performance experiments. For the performance experiments, we use large synthetic datasets that are generated using the IBM Quest Dataset Generator [Agrawal and Srikant, 1997] and afterwards enhanced using realistic datasets. As before, we consider in all following experiments that the complete dataset is already available in main memory (see the description in Section 5.4.1 on page 117). We also do not return the found itemsets during the performance evaluation: we only report the number of found frequent itemsets for each length. This does not affect the mining effort, i.e., the number of created conditional trees stays the same whether the itemsets are returned or not. Similar to the cApriori and cEclat experiments, we measure the algorithm’s memory consumption by obtaining the process’ resident set size and the system’s swap space using <sys/resource.h> and <sys/sysinfo.h>, respectively. Linux gettimeofday is used to obtain the runtime; linux nice is used to increase the process’ priority to get more consistent results.

### 7.3.2. Compression Ratio

The purpose of the first set of experiments is to evaluate the compression ratio of our data structures on a large variety of real world datasets. We measure the average node size in bytes for both the ternary CFP-tree and the CFP-array. The average is taken over the number of nodes in the corresponding FP-tree; in the ternary CFP-tree, the number of nodes may be less because multiple FP-tree nodes may be merged into a single chain node.

To obtain the node size of state-of-the-art implementations of FP-growth, we investigated the source code of various FP-growth implementations. Similar to other implementations, in the FP-growth implementation of Grahne and Zhu [2003] each node requires 40 bytes of memory, i.e., the FP-tree is based on a binary tree and thus each node has besides the id and count field four pointers: child, sibling, parent, and nodelink. During compilation,
the compiler performs alignment optimization, which increases the node size to 48 bytes; this behavior can be avoided with appropriate modifications of the source code. Assuming such a modification, 40 bytes per node would be the baseline for “regular” FP-GROWTH implementations. In our own FP-GROWTH implementation, we build the FP-tree in two phases as for CFP-GROWTH. The FP-tree is based on a ternary tree and requires in the build phase three pointers (child, left, right) and in the mine phase only two pointers (parent, nodelink). We convert the “build” tree into a “mine” tree in-place so that the nodes of both trees require always three pointers; the child and left pointer is used as parent and nodelink pointer after the conversion, respectively, while the third pointer is not used in the “mine” tree. For this reason, our optimized FP-GROWTH implementation lowers the baseline to 32 bytes.

Figure 7.11: Average node sizes for different datasets

Figure 7.11a shows the average node size of a ternary CFP-tree built from the retail, kosarak, accidents, netflix, netflix, and webdocs datasets with various choices of $\xi$ values; the $\xi$ value is adjusted for each dataset so that the tree is build from the 50, 250, and 1250 most frequent items of it. A table with the respective $\xi$ values can be found in Appendix A.2 on page 189. We omit results for the synthetic datasets, which have similar characteristics. The average node size heavily depends on the number of frequent items that are used to build the CFP-tree: for 50 frequent items, it varies from 6.78 bytes (retail) to 2.67 bytes (netflix). The datasets with a larger average node size (e.g., retail, accidents, and wikilinks) benefit less from node chaining. Their respective CFP-trees branch out more compared to the CFP-trees build from the other datasets. When the number of frequent items increases, i.e., for smaller $\xi$ values, the impact of the branching lowers because the tree paths are getting longer. This leads to longer chains so that the average node size further decreases. It is below 5 bytes for all datasets and 250 frequent items which corresponds to a 8x reduction of memory consumption compared to regular FP-growth. A node then occupies in average less space than a single pointer. The lowest observed average node size of 1.48 bytes is obtained for a tree that is build using the netflix dataset and 1250 frequent items. This corresponds to a 25x size reduction compared to the 40-byte node of a regular FP-tree. In general, the smaller average node size for a large number of frequent items is beneficial because more frequent items usually lead to more nodes within the trees. For these specific datasets, the trees had 10x to 45x more nodes when they contained 1250 instead of 50 frequent items.
7. CFP-growth

Results for the CFP-array are given in Figure 7.11b. Here, the average node size is dominated by the ∆pos field. This field requires the most space in the datasets that lead to large trees with many nodes. It is usually lower if there are less nodes within the array and/or the number of frequent items is high. For example, the netflix dataset has much less nodes for the same number of frequent items within its tree than netflix'. For all tested datasets, the average node size is below 4.5 bytes; it is close to 3 bytes in the best cases. These node sizes correspond to an 9x–13x reduction of memory consumption. Depending on the number of frequent items, the CFP-array dominates the memory consumption of CFP-GROWTH.

7.3.3. Sequential CFP-growth

We next investigate the performance of FP-GROWTH and CFP-GROWTH implementations when only one thread is used. The purpose of these experiments is to (1) illustrate the superior memory footprint of CFP-GROWTH and (2) show that our implementations are competitive with other highly optimized FP-GROWTH implementations. As mentioned before, only if our sequential implementations are competitive, then it is meaningful to parallelize them.

As for the experiments in the previous chapters, we reduce the time for experimentation by running the algorithms on datasets that are smaller than the datasets used for evaluating the parallel algorithms. For that purpose, we use only a subset of the transactions that are within the large synthetic datasets; the smaller datasets each comprise 4GB. We further limit the available main memory of our system using the memory-blocker tool, which is described in Chapter 5 on page 117, to highlight the algorithms’ behavior under memory pressure. The algorithms under test can use only up to 1GB of main memory that is available in the dedicated memory of the first processor. Linux numactl is used to bind the algorithm’s process to this processor.

Build Phase and Conversion

We start with investigating the build phase of CFP-GROWTH, including the conversion of the ternary CFP-tree to the CFP-array. For this set of experiments, we use the quest-netflix'-small dataset; we also repeated the experiments with other synthetic datasets and observed similar results. We compare our FP-GROWTH and CFP-GROWTH implementation against the best-performing FP-GROWTH implementation—proposed by Grahne and Zhu [2003]—of the FIMI repository. Our FP-GROWTH implementation is based on a ternary tree and is therefore in the following denoted as FP-GROWTH-TT. Grahne and Zhu [2003] use a binary tree as physical representation in their implementation; we thus refer it as FP-GROWTH-BT.

Figure 7.12a shows the time for the initial build phase of the mining algorithms under test. For CFP-GROWTH, this includes the time for building the ternary CFP-tree and converting it into the CFP-array. For FP-GROWTH-TT, the time comprises building the ternary FP-tree and its in-place conversion while for FP-GROWTH-BT it is just the time to build the binary FP-tree (without conversion). We vary the ξ value during the experiments, but show the corresponding number of FP-tree nodes on the x-axis. We also plot CFP-GROWTH’s time required for scanning the data. Recall that FP-GROWTH-based algorithms perform two scans: The first scan counts the frequency of each item, while the second scan builds the prefix tree. The time after the second scan is consumed by the conversion for both of our implementations. We found that FP-GROWTH-TT and CFP-GROWTH perform similarly for small prefix trees with up to 20 million nodes. FP-GROWTH-BT, however, is even for such small number of nodes
7.3. Experiments

![Graph](image)

(a) Tree build time  
(b) Tree memory requirements

Figure 7.12.: Single-threaded tree building on quest-netflix$^T$-small

about 5x to 6x slower than the other algorithms because its parser is not as optimized\(^6\) as our parser and—more importantly—it is much more expensive to build a binary tree instead of a ternary tree. This holds true also on many other large datasets. The extra runtime for converting a ternary FP-tree or CFP-tree is relatively small. For example, the conversion of a ternary CFP-tree with 20M nodes in a CFP-array requires only 1.7s; counting the frequent items and building the tree takes 28.5s.

As the tree size increases, the build time of the binary and ternary FP-tree explodes. This is because the tree size quickly exceeds the available main memory (1GB) so that thrashing occurs; see Figure 7.12b. In fact, for a tree size of about 30M nodes, the binary FP-tree of FP-GROWTH-BT required 1.1GB of main memory and took roughly 2h to build. FP-GROWTH-TT behaves similar, although it can build slightly larger trees in a reasonable time because of its smaller nodes. In contrast, CFP-GROWTH can process much larger trees without thrashing. The CFP-array, which is always larger than the CFP-tree for the used dataset, hits the main memory limit for a tree size of roughly 300M nodes. As we go beyond this size, the runtime of CFP-GROWTH starts to increase more quickly but remains acceptable. For example, a prefix tree of 450M nodes corresponds to a 0.6GB CFP-tree and a 1.8GB CFP-array. Still CFP-GROWTH requires only about 800s to build and convert the tree. To see why, observe that the CFP-tree fits entirely into main memory so that thrashing only occurs during conversion. Memory access during conversion is almost sequential so that the amount of thrashing is expected to be quite low. As we go to even larger tree sizes, thrashing increases heavily. For example, to build and convert a tree of 600M nodes, CFP-GROWTH requires 4,000s.

**Overall Performance**

We move on to experiments that cover the overall performance of FP-GROWTH and CFP-GROWTH. Again, we use FP-GROWTH-TT and FP-GROWTH-BT for this evaluation. In contrast to the latter implementation, both of our implementations do not perform any particular optimization of the mining step—except for the single-path optimization (cf. Section 2.2.3 on page 22). Hence, FP-GROWTH-BT might be more efficient on small datasets or when mining with unreasonable high $\xi$ values. We focus, however, on mining of very large datasets where tree-building is often as expensive or even more expensive than mining.

\(^6\)The file is read using `getc`, which is more expensive than reading a file that is loaded using Linux `mmap`. 

177
7. CFP-growth

The total execution time and memory consumption for various $\xi$ values are shown in Figures 7.13a and 7.13b, respectively. As before, the $x$-axis shows the tree sizes of the initial FP-tree corresponding to the choice of $\xi$ values. As can be seen, CFP-GROWTH and FP-GROWTH-TT behave similar up to 20 million nodes. For larger trees, FP-GROWTH-TT exceeds the available main memory and its overall execution time increases significantly. For example, at 32M nodes ($\xi = 6.28\%$), it has a peak memory consumption of 1.8GB and requires 14,877s to complete. For the same $\xi$ value, CFP-GROWTH has a peak memory consumption of 0.29GB and requires 48.2s to complete. This corresponds to a speedup of 310x. The speedup further increases for larger trees. FP-GROWTH-BT is always inferior when compared with the other two mining algorithms. Up to 20 million nodes, it is about 6x slower because of its expensive tree building, which significantly influences the overall performance. With more nodes, FP-GROWTH-BT’s performance also degrades seriously because of its large memory consumption.

CFP-GROWTH scales well up to an initial tree size of 270M nodes. At this point, the peak memory consumption hits the main memory size of 1GB. Compared to FP-GROWTH, which reaches this size at 25M nodes, CFP-GROWTH can perform pure in-core processing for a 10.8x larger tree. Note that the non-linear increase of runtime and memory consumption of CFP-GROWTH results from the fact that, as the $\xi$ value decreases, more and more itemsets become frequent. To find these itemsets, more work has to be done and more conditional CFP-trees have to be created and stored in main memory.

We obtain similar results also on other large datasets. Our algorithms are always superior over other tested implementations of FP-GROWTH when mining large datasets. Hence, it is reasonable to parallelize them.

7.3.4. Parallel CFP-growth

We run several experiments to evaluate the performance as well as the scalability of parallel versions of FP-GROWTH and CFP-GROWTH. As before, we used OpenMP for obtaining the parallel versions. The producer-consumer processing model is realized—similar as for parallel cApriori—using a non-blocking synchronized queue that uses a small critical section while enqueuing or dequeuing pages. Parts of the dataset are distributed using a parallel for construct. The conversion of the subtrees is performed using the task construct. Each thread creates a task for each of its eight subtrees; these “conversion” tasks are eventually executed
7.3. Experiments

by threads that are already finished with their work. The subsequent tree-mining step is parallelized using parallel for where each thread recursively mines all conditional subtrees for a certain item. We always use the guided scheduling policy to dynamically assign load to threads that are not busy.

If the range partitioning scheme is used, we need to build a small tree using a transaction sample to obtain the range partitioning that is required for building the actual tree. We did not spend much effort in drawing this sample. In our current implementation, we simply use the first few thousand transactions of the dataset; the sample tree is usually about 10MB and thus below 24MB, which constitutes the size of an L3-cache of the employed processors. Hence, the sample tree could be build quite fast within the caches.

Tree partitioning schemes

Before evaluating the parallel tree building itself, we measure the efficiency of the used partitioning schemes. For that purpose, we compare the hot-item (HOT) and the range partitioning scheme (RANGE) on various datasets and a varying number of partitions. A good measure for the partitioning efficiency is the weight of the largest partition; it determines directly the maximum speedup of the tree-building step. To obtain the weight of a partition, we add up the length of all transactions—excluding covered items—that would be inserted into it; we do not build the subtree and count the number of its nodes. This is sufficient because the partition’s cumulated transactions length corresponds very well with the number of nodes of the respective subtree.

As similarly done before, we individually set the ξ value for each dataset so that the number of frequent items in the result set is fixed. We observe that the number of frequent items (or the ξ value) has only a minor impact on the weight of the largest partition. For a smaller number of frequent items, the largest partition is usually slightly smaller than for more frequent items. On the retail dataset, for example, the largest of 256 partitions contains for 50 frequent items about 12% of the load while for 10,000 frequent items it grows to about 14%. For this reason, we set the ξ value for each dataset so that 250 frequent items are in the result set. The number of partitions was set to 256 as it is sufficient for balancing the load of 32 threads. We also provide results for HOT with 65,536 and $2^{24}$ partitions because for a larger number of partitions its efficiency is usually increased. For the latter number of partitions, however, the overhead for the partitioning gets unacceptable high; already the pointer of a single producer to the $2^{24}$ pages require 128MB. For this reason, we include the respective results mainly to provide an upper limit for the efficiency of HOT.

Figure 7.14 illustrates the size of the largest partition for various realistic datasets. We also include results for two synthetic datasets, quest-webdocs and quest-netflix$^T$, which are both used in the subsequent experiments. The dotted line plotted at 3.125% marks the maximum partition size that is required to distribute the load equally among 32 threads. The hot-item partitioning (HOT) often fails to provide such an equal load distribution. For 256 partitions, its largest partition contains on accidents, netflix, and webdocs more than one-third of the overall load. Hence on these datasets, the theoretical maximum speedup is at best 3x. Also for $2^{16}$ partitions, the largest partition contains—except for kosarak and webdocs—about 10% of the load. Only for the very large number of $2^{24}$ partitions, the largest partition of HOT shrinks for half of the datasets below 3.125%. However, on retail, netflix$^T$ and wikilinks even this large number of partitions is not sufficient. On the latter dataset, the largest partition limits the speedup to 6x. As expected, a large number of the
7. CFP-growth

2^{24} partitions are empty; about 97% to 99% do not contain a single transaction. In contrast, less than 5% of the partitions are empty if 256 partitions are used. RANGE provides a more equal load distribution. The largest of 256 partitions contains for all datasets less that 1.5% of the load. For most datasets, it is even around 0.8%. If the number of partitions of RANGE is increased, the size of the largest partition also decreases further. For example, it is between 0.1% and 0.45% on all datasets, if 1024 partitions are used (not shown). Hence, RANGE also supports an effective load balancing even if the number of threads is substantially increased.

Both synthetic datasets behave similar as their realistic counterparts—although HOT’s largest of 256 partitions on quest-netflix $^T$ is noticeably smaller than for netflix $^T$. It contains only 11.3% of the load on the former dataset but 17.6% on the latter dataset. Interestingly, before our modification of the synthetic datasets (see Section 5.4.1 on page 118), HOT’s largest partition within both synthetic datasets contained about 60% up to 99% of the load while RANGE’s largest partition was below 3%. Hence, HOT does not work on traditional quest datasets while RANGE works well on them.

Build phase

In the next set of experiments, we evaluate the tree building performance of FP-GROWTH and CFP-GROWTH for both partitioning schemes, HOT and RANGE. As before, we measure the wall clock time. We do not include the time for mapping the dataset into main memory and counting the frequent items because the results are similar as for multi-threaded cApriori (cf. Section 5.4.3 on page 127). The measured time includes the time for building the partitioning and the actual tree, as well as the tree conversion. Creating the partitioning induces almost no overhead for HOT and requires only few milliseconds—typical below 50ms—for RANGE. This includes the time for drawing the sample and creating the range partition tree. The number of partitions is set to 8 times the number of threads but at most to 256, i.e., we use 256 partitions for 32 as well as 64 threads. The threads’ work queues have 128 entries and the page size is set to 32KB. This showed to be a good trade-off between locking effort and the space overhead—which is at most 0.75GB for 64 threads—of the partitioning process.

As datasets, we use quest-webdocs and quest-netflix $^T$ that both comprise 102GB and are well-suited for FP-GROWTH. Again, adjust the $\xi$ value so that the result contains a fixed number of frequent items. For quest-webdocs, we use 256 frequent items ($\xi = 8.2\%$) which results in an FP-tree that has about 47 million nodes. We use only 128 frequent items ($\xi = 6.5\%$) for quest-netflix $^T$; the respective tree comprises 487 million nodes and occupies
14.5GB of memory when FP-GROWTH is applied. As shown in Figure 7.14, the largest of 
HOT’s 256 partitions for quest-webdocs contains about 41% of the load while for quest-
netflix it contains only 11%. Hence, HOT’s maximum speedup for building the tree should 
be 2.4x and 9x for the former and latter dataset, respectively.

Figure 7.15a illustrates the tree build time in seconds for a varying number of threads on 
quest-netflix. As can be seen, FP-GROWTH and CFP-GROWTH scale only well up to eight 
threads when HOT is applied. As the number of threads is further increased, the build time 
indeed decreases—the largest partition gets smaller because of the more partitions—but it 
goes never below 37s for FP-GROWTH and 50s for CFP-GROWTH, which constitutes a speedup 
of 14.5x and 12.7x, respectively. It is above the expected maximum speedup of 9x because the 
producer–consumer processing model also does some load-balancing, i.e., threads with only 
small partitions are more often producer while the thread with the largest partition is usually 
always a consumer. This works better for FP-GROWTH than for CFP-GROWTH because 
inserting in a CFP-tree is more costly so that the proportionally less expensive producing has 
a lower impact as a counterpart of consuming. Interestingly, the maximum speedup is achieved 
with 32 threads. Building the tree using 64 threads—with Hyperthreading enabled—is even 
slower because the thread that processes the largest partition seems to be slowed down by 
the second thread that is executed on the same core; the additional threads, however, cannot 
speedup processing because of the unequal load distribution. Contrary to HOT, both mining 
algorithms scale linear up to 32 threads (w/o Hyperthreading) when they employ RANGE. 
The speedup increases further to 44.3x for FP-GROWTH and 46.7x for CFP-GROWTH when 
64 threads (with Hyperthreading) are used.

Figure 7.15b shows the tree build time for quest-webdocs and a varying number of threads. 
Here, HOT scales only well up to two threads; already for four threads, the speedup is limited to 
3.3x and 3.2x for FP-GROWTH and CFP-GROWTH, respectively. Again, as with more threads 
the number of partitions grows to 256, the largest partition slightly decreases. Nevertheless, 
HOT’s speedup never exceeds 6.3x (FP-GROWTH) and 5.4x (FP-GROWTH). RANGE scales 
well up to 32 threads for both mining algorithms. For FP-GROWTH, the runtime decreases 
from 614s to 20.3s for 32 threads, which constitutes a speedup of 30.1x. If Hyperthreading 
is enabled, the runtime even decreases further by 5s for both mining algorithms. We obtain 
similar numbers for CFP-GROWTH combined with RANGE: Its runtime decreases from 632s 
(single-threaded) to 14.1s (64 threads with Hyperthreading), which constitutes a speedup of
7. CFP-growth

Figure 7.16: Parallel tree building on quest-netflix

44.7x. Hence, RANGE permits good scalability for the tree-building step of both, FP-GROWTH and CFP-GROWTH.

In the following experiment, we evaluate the time required for building the tree and its memory consumption for a varying number of nodes on quest-netflix. The number of threads was set to 64 (using Hyperthreading). We include the time for the first scan, i.e., the time for mapping the dataset into the process' address space as well as the time for counting. After the dataset is loaded, only 24.5GB are available for holding the FP-tree or CFP-tree on our particular system. As mentioned in Chapter 5, already 200MB of this memory are required for storing the page table of the process (obtained via linux/proc/meminfo). This could be lowered if linux hugetables would be used, however, ramfs supports only 4K-pages.

Figure 7.16a illustrates the time for building the tree. Both mining algorithms require—independently of the employed partitioning scheme and the number of nodes—the same time for mapping the file into main memory and counting the frequent items. This time is constant at approximately 24s and is illustrated using a dotted line. As one can see, the time for both mining algorithms grows very slowly when they employ RANGE. The small growth is mainly caused by the shape of the tree; it gets broader instead of deeper as the number of nodes is increased. If HOT is employed, however, the building time grows significantly as the number of nodes is increased because the largest partition also grows with more nodes. Hence, the load imbalance gets worse which leads to more unused threads. As soon as 740 million nodes are reached, the FP-tree requires more than the available main memory (see Figure 7.16b). From this point on, tree building gets prohibitively expensive for FP-GROWTH. The CFP-tree has much lower memory requirements; they are even smaller when RANGE is applied because equally-sized partitions reduce the additional memory requirements during tree conversion. This amounts to about 1.5GB for a CFP-tree comprising 2 billion nodes.

Building the tree in parallel behaves similar also if other datasets (e.g., quest-retail and quest-webdocs) are used. Furthermore, the subsequent mining has often only a small impact on the memory consumption for reasonable ξ values. Only for quest-webdocs, the time for mining easily explodes if the tree gets large because it has many distinct items combined with long transactions.
Overall performance

In our final experiments, we compare the overall performance of parallel FP-growth and CFP-growth. We use both partitioning schemes and measure the time until the results are available; we do not include the time that is required for Linux to end the process after leaving the main() function. The $\xi$ values are set to the same values as used for evaluating the time for building the tree—$\xi = 6.5\%$ for quest-netflix$^T$ and $\xi = 8.2\%$ for quest-webdocs.

Figure 7.17a shows the overall runtime for a varying number of threads on quest-netflix$^T$. As before, the mining algorithms only scale well up to eight threads when hot is applied while they scale much better when using range. The maximum speedup achieved for 32 threads is 17.1x and 25.6x when hot and range is applied, respectively. The speedup is mainly limited by the time for mapping the dataset into the process’ address space. If this time is omitted, the mining algorithms scale perfectly when range is applied. If Hyperthreading is enabled, they are even up to 46.7x faster than their sequential versions. For hot, however, the mining algorithms’ speedup is limited by 19.9x—independently of the number of employed threads.

We obtain similar results when mining quest-webdocs. As can be seen in Figure 7.17b, the mining algorithms scale much better when they employ range instead of hot. For the latter, the maximum speedup is limited to 8x. When using range, the mining algorithms are up to 25x faster than their respective sequential versions. Only the time for mapping the dataset as well as a small load imbalance during mining that only occurred on quest-webdocs prevent perfect scalability for both parallel mining algorithms when they employ range.

The algorithms behave similarly on other datasets like quest-retail, quest-kosarak and quest-wikilinks. Also here, range is always superior over hot which leads to a much faster tree building. This results in a better overall performance because—for large datasets—tree building usually amounts to a large fraction of the overall runtime. For this reason, range is an important building block of scalable FP-growth versions.

7.4. Summary and Discussion

In this chapter, we introduced CFP-growth that is a highly memory efficient version of FP-growth. CFP-growth relies on two highly-tuned data structures: the CFP-tree and the CFP-array. Both data structures exploit a combination of structural changes to the FP-tree
and lightweight compression techniques. The resulting reduction of memory consumption—a factor of 7x–25x in our experiments—allows in-core processing for significantly larger datasets. Hence, expensive out-of-core processing could be avoided for such datasets. In the second part of this chapter, we proposed scalable parallel versions of FP-GROWTH and CFP-GROWTH. We improved the load balancing of the tree-build step, which is the main scalability issue in previous parallel FP-GROWTH versions, by introducing the range partitioning scheme. A partitioning from this scheme is always customized to the tree being build; sampling is used for this “customization” process. The tree itself is build using the multiple-producer/multiple-consumer processing model. Experiments are carried out using large datasets that have the same properties as real-world datasets. We achieve linear scalability with high speedups up to 46.7x on a system with 32 cores (Hyperthreading is enabled). Our experiments further suggest good scalability also for an increasing number of threads. Hence, parallel CFP-GROWTH is both, memory-efficient and highly scalable, on a wide range of datasets.

In our experiments, CFP-GROWTH’s is usually as fast as “plain” FP-GROWTH when there is no memory pressure. Only on certain datasets that comprise mainly short transactions, we found CFP-GROWTH to be up to 2x slower than FP-GROWTH. This gap might be widen because there are several optimizations for improving the cache utilization of FP-GROWTH that cannot be applied to CFP-GROWTH. For this reason, CFP-GROWTH is not always the algorithm of choice; FP-GROWTH is preferable as long it can operate in-memory, e.g., for small datasets or high $\xi$ values. Otherwise, CFP-GROWTH is superior over FP-GROWTH. One thus has to consider the expected memory requirements—which are only fully available after the algorithm is finished—when choosing the best-suited of both algorithms for a given dataset and $\xi$ value. A good estimate for the memory requirements might be obtained from a sample tree build from a small part of the transaction database. As an alternative, one could also switch to CFP-GROWTH during processing as soon as FP-GROWTH goes out-of-core.

The efficiency of our range partitioning scheme strongly depends on the WFP-tree that is build from a sample of the transaction database. In our experiments, we use the first few thousand transactions of a dataset to obtain a reasonable sample. For a larger number of threads, however, more elaborate partitionings might be necessary so that the sample should be drawn from the full dataset. This could also be done in parallel to improve the sample quality without increasing the time for building it. Furthermore, the multiple-producer/multiple-consumer model might not scale up to a very large number of threads (e.g., more than thousand threads). Its main drawback is that each thread needs to maintain a page per partition. Therefore, the memory required grows quadratic as the number of threads and thus the number of partitions is increased. A solution for this problem might be to assign transactions to partitions using multiple stages, i.e., start with a coarse-grained assignment and refine it with each additional stage similar to a trie-based sort algorithm.
Part IV.

Summary
8. Conclusion

“The Age of Serial Computing is over...”
– Charles E. Leiserson

There are many highly-optimized serial algorithms for frequent-itemset mining, which constitutes still an active research area because of its many application areas. The serial algorithms, however, cannot cope with the ever-increasing datasets that occur in many of the application areas because the serial performance of processors almost stopped increasing in recent years. Instead, processors as well as complete systems are getting more and more parallel so that the huge datasets can only be mined in a reasonable time when the available parallelism is effectively used. Thereby, not only thread-level parallelism should be employed but also data-level and instruction-level parallelism since both offer great potential for speeding up algorithms. Nevertheless, all the computing power is meaningless when the running threads of a parallel algorithm have to wait most of their time for data being accessed. It is thus crucial to keep all of the algorithms required data in main memory; this is challenging because of the large datasets and the mining algorithm’s excessive memory consumption.

In this thesis, we propose algorithms for frequent-itemset mining that (1) exploit the available parallelism of modern processors and (2) are memory efficient at the same time. For that purpose, we investigated the performance hot spots of three well-established algorithms and optimized them by employing appropriate parallelism. For APRIORI and ECLAT, we considerably reduced the time that is consumed for obtaining the support of certain candidate itemsets. More specifically, we speed up APRIORI’s repeatedly executed frequency counting using a scalable counting algorithm that is based on a variant of the producer–consumer model. The algorithm is carefully optimized and effectively employs available parallelism. To cope with ECLAT’s particular hot spot—the many list and bitmap intersections—we propose a list intersection algorithm that relies on speculative comparisons and therefore can exploit available, powerful parallel-comparison instructions. With that, the intersection algorithm is much faster than other state-of-the-art list intersection algorithms. Additionally, we discuss and revise an efficient parallel bitmap intersection algorithm. Finally, we propose a tree partitioning scheme for parallel FP-GROWTH. It allows scalable multi-threaded tree building, which is one of the major performance hot spots of this algorithm.

Besides the parallel improvements, we provide further memory-efficient versions of APRIORI, ECLAT, and FP-GROWTH. CAPRIORI uses a compact representation for the filtered transactions that are stored in the horizontal layout. Furthermore, the memory required for storing the candidates remains almost constant as the number of threads is increased. Similarly, CECLAT uses a compact layout for the transactions stored in the vertical layout; each transaction is either represented within bitmaps or integer lists—always considering the better of both representations. For CFP-GROWTH, we even apply explicitly compression techniques; the nodes of the respective CFP-tree are always compressed so that the tree requires only a small fraction of the memory required of a regular FP-tree. We furthermore propose two parallel compression algorithms that can be used in scenarios, where large datasets must be kept in main memory or transferred very frequently.

We carried out many experiments on very large datasets that comprise up to 100GB. On such datasets, our algorithms were always superior—in terms of runtime as well as memory...
8. Conclusion

consumption—over existing highly optimized algorithms. The better runtime results mainly from very efficient data structures, which are amenable for exploiting parallelism. The multi-threaded versions of our algorithms also scale very well on our test system that has 32 cores. The superior memory consumption comes from the compact data structures and applied compression techniques. Thus, we can conclude that we achieved our goal of providing scalable as well as memory efficient algorithms for frequent-itemset mining.

Further work directions

The mining algorithms proposed in this thesis scale very well on current available systems and—very likely—also on near future systems. Nevertheless, modern hardware continuously evolves so that with new hardware there will occur always new research opportunities. While the core ideas of our parallel algorithms might be valid for a long time, some of our proposed techniques need to be refined with increasing parallelism like more cores, wider vector registers, and more specialized instructions; for example, using a transaction sample to obtain a reasonable partitioning is valid even when the number of threads is largely increased. Contrary, the employed multiple-producer/multiple-consumer processing model will most likely not scale up to thousand threads, which might be available on systems in several years.

Besides this continuously refinement of the parallel algorithms, itemset mining algorithms may also benefit from other emerging technologies:

Algorithms for heterogeneous hardware There is a trend towards the development of processors with heterogeneous cores. A prominent example is the IBM Cell processor, which has a single general-purpose core and eight cores that are tailor-made for vectorized processing. Furthermore, AMD and Intel provide processor architectures that have CPU and GPU cores on the same chip; these cores are connected using a fast interconnect and can be exploited in algorithms using OpenCL. Interesting research questions could now be: (1) which algorithms may benefit of certain cores, (2) which core type is best suited for executing which parts of an algorithm, and (3) how to balance the load between the different cores.

Building tailor-made hardware Field programmable gate arrays (FPGAs) or other customizable processors (e.g., Tensilica’s Xtensa series) allow to build custom hardware that is tailor-made to a specific problem. Such custom circuits or processors are usually superior to a regular CPU when they process certain performance hot spots of an algorithm with few cycles while the regular CPU would require many or expensive instructions for processing them. Hence, it would be an interesting research topic to speed up certain hot spots of itemset mining algorithms with customized hardware. For example, eclat’s time required for the often repeated intersection of lists or bitmaps may be significantly reduced when they are performed using intersection circuits or instructions that are implemented in hardware. This implementation must thereby not necessarily reduce the latency of single intersection but could improve the intersection throughput.

There exist also many other topics that could be listed here; for example, modern storage technologies like flash and phase-change memory as well as many-core accelerator cards may also help for mining very large datasets. For all such optimizations and new algorithms, this thesis may be the starting point.
A. Datasets and Code tables

In this chapter, we provide (1) details to the datasets used in this thesis, (2) experimental parameters, and (3) code tables for integer compression techniques of Chapter 4.

A.1. Datasets

In the experiments of this thesis, we used several realistic and synthetic datasets. Most of the realistic datasets were taken from the FIMI repository [FIMI, 2004]. The datasets retail [Brijs et al., 1999] and BMS-POS [Zheng et al., 2001] contain anonymous customer transactions and belong to the retail business application area; retail comprises data from a Belgian supermarket store while the data in BMS-POS is taken from a large electronics retailer. The kosarak dataset [FIMI, 2004] contains click-stream data of a Hungarian online news portal. The accidents dataset [Geurts et al., 2003] is built from a database that contains statistics about traffic accidents in Belgium. Finally, webdocs [Lucchese et al., 2004b] is created from a collection of web documents; it thus belongs to the web-mining application area.

We further build three datasets from publicly available realistic datasets that were not originally intended for itemset mining. The netflix dataset is built from a dataset that contains movie ratings and was used in the Netflix Prize competition.\(^1\) The dataset thus can be assigned to the recommender systems application area. The netflix\(^T\) dataset is the transposed version of the netflix dataset, i.e., instead of storing which movie was seen by which users, netflix\(^T\) holds the association which user has seen which movies. The wikipedia dataset is built from the links of the wikipedia web pages of 2009.

For our performance experiments, we generated synthetic datasets using the IBM Quest Dataset Generator [Agrawal and Srikant, 1997]. We built small (4GB) and large datasets (100GB). Each of them resembles to a realistic dataset in terms of the characteristics average cardinality and distinct items, i.e., quest-retail has almost the same average cardinality and number of distinct items as retail, but about 25,000x more transactions than it.

All datasets were stored in plain text files, which follow the standard FIMI format. Each line consists of a list of items that constitute a single transaction. The average storage space per occurrence of each item is below 8 bytes. Table A.1 summarizes the characteristics of the realistic and synthetic datasets. All synthetic datasets start with the prefix “quest-”.

A.2. Chosen Support for FP-growth

In Section 7.3.2 on page 175, we evaluated the compression performance of the CFP-tree and CFP-array. For that purpose, we mined all realistic datasets—except BMS-POS—with certain \(\xi\) values. Table A.2 provides for each of these datasets the chosen \(\hat{\xi}\) values that lead to 50, 250, and 1250 frequent items in the result set when mining them.

\(^1\)See http://www.netflixprize.com/ for more information.
## A. Datasets and Code tables

### Table A.1.: Summary of datasets

<table>
<thead>
<tr>
<th>Name</th>
<th># TA</th>
<th>Avg. card.</th>
<th>Distinct items</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>retail</td>
<td>88,162</td>
<td>10.3</td>
<td>16,470</td>
<td>4MB</td>
</tr>
<tr>
<td>BMS-POS</td>
<td>515,597</td>
<td>6.5</td>
<td>1,657</td>
<td>11MB</td>
</tr>
<tr>
<td>kosarak</td>
<td>990,002</td>
<td>8.1</td>
<td>41,270</td>
<td>31MB</td>
</tr>
<tr>
<td>accidents</td>
<td>340,183</td>
<td>33.8</td>
<td>468</td>
<td>34MB</td>
</tr>
<tr>
<td>netflix(^T)</td>
<td>480,190</td>
<td>209.2</td>
<td>17,770</td>
<td>519MB</td>
</tr>
<tr>
<td>netflix</td>
<td>17,771</td>
<td>5,654.1</td>
<td>480,189</td>
<td>727MB</td>
</tr>
<tr>
<td>wikilinks</td>
<td>5,706,071</td>
<td>22.8</td>
<td>3,773,865</td>
<td>962MB</td>
</tr>
<tr>
<td>webdocs</td>
<td>1,692,082</td>
<td>177.2</td>
<td>5,267,656</td>
<td>1.4GB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th># TA</th>
<th>50 items</th>
<th>250 items</th>
<th>1250 items</th>
</tr>
</thead>
<tbody>
<tr>
<td>quest-retail-small</td>
<td>86M</td>
<td>10</td>
<td>17k</td>
<td>4GB</td>
</tr>
<tr>
<td>quest-retail</td>
<td>2,220M</td>
<td>10</td>
<td>17k</td>
<td>100GB</td>
</tr>
<tr>
<td>quest-kosarak-small</td>
<td>115M</td>
<td>8</td>
<td>41k</td>
<td>4GB</td>
</tr>
<tr>
<td>quest-kosarak</td>
<td>2,900M</td>
<td>8</td>
<td>41k</td>
<td>100GB</td>
</tr>
<tr>
<td>quest-BMS-POS-small</td>
<td>128M</td>
<td>7</td>
<td>2k</td>
<td>4GB</td>
</tr>
<tr>
<td>quest-BMS-POS</td>
<td>3,300M</td>
<td>7</td>
<td>2k</td>
<td>100GB</td>
</tr>
<tr>
<td>quest-wikilinks-small</td>
<td>72.5M</td>
<td>23</td>
<td>3,773k</td>
<td>4GB</td>
</tr>
<tr>
<td>quest-wikilinks</td>
<td>1,950M</td>
<td>23</td>
<td>3,773k</td>
<td>100GB</td>
</tr>
<tr>
<td>quest-netflix-small</td>
<td>0.17M</td>
<td>5654</td>
<td>480k</td>
<td>4GB</td>
</tr>
<tr>
<td>quest-netflix</td>
<td>4.4M</td>
<td>5654</td>
<td>480k</td>
<td>100GB</td>
</tr>
<tr>
<td>quest-netflix(^T)-small</td>
<td>3.78M</td>
<td>210</td>
<td>18k</td>
<td>4GB</td>
</tr>
<tr>
<td>quest-netflix(^T)</td>
<td>100M</td>
<td>210</td>
<td>18k</td>
<td>100GB</td>
</tr>
<tr>
<td>quest-webdocs-small</td>
<td>4.3M</td>
<td>177</td>
<td>5,200k</td>
<td>4GB</td>
</tr>
<tr>
<td>quest-webdocs</td>
<td>110M</td>
<td>177</td>
<td>5,200k</td>
<td>100GB</td>
</tr>
</tbody>
</table>

### Table A.2.: Chosen \(\hat{\xi}\) values in the compression potential experiment in Section 7.3.2

<table>
<thead>
<tr>
<th>Name</th>
<th># TA</th>
<th>50 items</th>
<th>250 items</th>
<th>1250 items</th>
</tr>
</thead>
<tbody>
<tr>
<td>retail</td>
<td>88,162</td>
<td>1,075</td>
<td>419</td>
<td>141</td>
</tr>
<tr>
<td>kosarak</td>
<td>990,002</td>
<td>10,490</td>
<td>3,550</td>
<td>1,002</td>
</tr>
<tr>
<td>accidents</td>
<td>340,183</td>
<td>165,501</td>
<td>203</td>
<td>N/A</td>
</tr>
<tr>
<td>netflix(^{-1})</td>
<td>480,190</td>
<td>139,900</td>
<td>81,280</td>
<td>20,280</td>
</tr>
<tr>
<td>netflix</td>
<td>17,771</td>
<td>4,753</td>
<td>2,953</td>
<td>1,986</td>
</tr>
<tr>
<td>wikilinks</td>
<td>5,706,071</td>
<td>45,000</td>
<td>15,640</td>
<td>5,426</td>
</tr>
<tr>
<td>webdocs</td>
<td>1,692,082</td>
<td>365,500</td>
<td>174,500</td>
<td>46,330</td>
</tr>
</tbody>
</table>
A.3. Code tables

In this section, we give example codewords for $k$-gamma encoding and $k$-gamma$_0$ encoding for $k=2$. Table A.3 shows input values $x_1$ and $x_2$, the corresponding codewords $v_1$ and $v_2$ and the shared prefix $p_1$ (including the separator bit). Note the difference in encoding input blocks that consist of only zero-or-one values.

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$p_1$</th>
<th>$v_1$</th>
<th>$v_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>01</td>
<td>10</td>
<td>00</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>01</td>
<td>10</td>
<td>01</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>01</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>01</td>
<td>11</td>
<td>00</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>01</td>
<td>11</td>
<td>01</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>01</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>01</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>001</td>
<td>100</td>
<td>000</td>
</tr>
</tbody>
</table>

Table A.4.: Example codewords for $k$-gamma$_0$

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$p_1$</th>
<th>$v_1$</th>
<th>$v_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>01</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>001</td>
<td>10</td>
<td>00</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>001</td>
<td>10</td>
<td>01</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>001</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>001</td>
<td>11</td>
<td>00</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>001</td>
<td>11</td>
<td>01</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>001</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>001</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0001</td>
<td>100</td>
<td>000</td>
</tr>
</tbody>
</table>
B. Detailed algorithm descriptions

As discussed in Chapter 3, the parallel merge intersection algorithm is the basis for an efficient parallel merge difference algorithm. Recall that the difference set $C$ of two sets $A$ and $B$ contains all values that are in $A$ but not in $B$. Thus the difference algorithm differs from the intersection algorithm only in the branch condition that is used for copying a value of $A$ into the result set $C$; i.e., a value of $A$ in the sequential difference algorithm is copied into $C$ whenever the condition $(A[i_a] < B[i_b])$ evaluates to true. In the parallel difference algorithm, we speculatively compare multiple values of $A$ with multiple values of $B$.

```c
int difference(short *A, short *B, int l_a, int l_b, short* C) {
    // initialize the variables
    int count = 0;
    short i_a = 0, i_b = 0, write_back_mask = 0xFF;
    // repeat the loop body until the end of a list is reached
    while(i_a < l_a && i_b < l_b) {
        // 1. Load the vectors
        __m128i v_a = _mm_load_si128((__m128i*)&A[i_a]);
        __m128i v_b = _mm_load_si128((__m128i*)&B[i_b]);

        // 2. Full comparison of the two vector variables
        __m128i res_v = _mm_cmpestrm(v_b, 8, v_a, 8,
            _SIDD_UWORD_OPS|_SIDD_CMP_EQUAL_ANY|_SIDD_BIT_MASK);
        int r = _mm_extract_epi32(res_v, 0);
        unsigned short a7 = _mm_extract_epi32(v_a, 7);
        unsigned short b7 = _mm_extract_epi32(v_b, 7);
        i_a += (a7 <= b7) * 8;
        i_b += (a7 >= b7) * 8;
        write_back_mask -= r; // remove the common values in A

        // 3. write back all values that were not found in B
        if(a7 <= b7) {
            __m128i p = _mm_shuffle_epi8(v_a, sh_mask[write_back_mask]);
            _mm_storeu_si128((__m128i*)&C[count], p);
            count += _mm_popcnt_u32(write_back_mask);
            write_back_mask = 0xFF;
        }
    }
    // copy remaining values of A
    count += copy_remaining_values(A, i_a, v_a, write_back_mask, C);
    // return the number of values that are in the difference set
    return count;
}
```

**Figure B.1.** Code snippet for the parallel merge difference algorithm
B. Detailed algorithm descriptions

A code snippet for the 16-bit parallel merge difference algorithm is illustrated in Figure B.1. For the ease of explanation, we assume that the length of each vector is a multiple of eight. A version for arbitrary set lengths requires some special treatment in the final loop iterations. After initializing certain variables, the loop body is repeated until the end of an input set is reached. As for the parallel merge intersection, 8 values of each sorted input set are loaded into vector variables using two SIMD load instructions within each iteration. Thereafter, the full comparison is performed using the `PCMPESTRM` instruction and the index variables are increased using predication. The difference to the parallel merge intersection algorithm is now that the result bitmap mask is ORed with the variable `write_back_mask`, which marks all values of \( v_a \) that are so far found in \( B \). The reason is that the same values in \( v_a \) might be compared multiple times with other values in \( v_b \). Whenever a value of \( v_a \) is found in \( v_b \), it has to be marked so that it is not copied into the result set. Copying takes place whenever the index \( i_a \) is increased, i.e., the condition \((a7 <= b7)\) evaluates to true. It is performed similar to the parallel merge intersection algorithm but here the variable `write_back_mask` is used to (1) select the appropriate permutation mask and (2) increase the number of common values stored in `count`. After the values in \( v_a \) that do not occur in \( B \) are shuffled to the front of the vector variable \( p \), they are written back into \( C \) using a single SIMD store instruction and the next iteration starts. Once the loop body is left, the possibly remaining values of \( A \) are copied into \( C \), since these values cannot occur in \( B \) when the loop body is left with the condition \(( i_b < l_b )\) evaluated to false. Notice that some values in \( v_a \) may already be marked as found in \( B \) and thus they must not be copied into \( C \).
In this chapter, we compare the single-threaded performance of our three algorithms \textsc{cApriori}, \textsc{CEclat}, and \textsc{CFP-growth}. The purpose of these experiments is to give an intuition about which algorithm performs well on which dataset. Clearly, this evaluation is not exhaustive because we only use a small set of datasets and do not examine the many factors (e.g., memory pressure, dataset size) that influence the algorithms’ performance.

We use the same setup that is described in Chapter 5 (cf. Section 5.4.1 on page 117). We use our four-socket multiprocessiong system, the process’ thread is pinned to the first socket, the datasets are main memory resident, and we measure the algorithms’ full runtime using \texttt{gettimeofday}. We further enable the same optimizations that were enabled within the experiments for evaluating the algorithms’ competitiveness with existing implementations. We set $b_{\text{sparse}} = 7$ within \textsc{CEclat} and employ the range partitioning with 4 partitions within \textsc{CFP-growth}. Furthermore, there is no memory pressure on the test system.

\textsc{cApriori} performs very well on the datasets \texttt{quest-retail-small} (Figure C.1a), \texttt{quest-kosarak-small} (Figure C.1b), \texttt{quest-BMS-POS-small} (Figure C.1c). These datasets have mainly short transactions and are thus well-suited for this algorithm. For some ranges of $\xi$ values, \textsc{cApriori} is an order of magnitude faster than the better of the other two algorithms. Only for small $\xi$ values, \textsc{CFP-growth} performs best because then the size of the filtered transactions increases so that \textsc{cApriori}’s subset counting gets expensive. \textsc{CEclat} is never competitive with the other two algorithms on these datasets; it suffers from the many small transactions that lead to long tid-lists and make the many intersections very expensive.

Contrary, \textsc{CEclat} and \textsc{CFP-growth} typically perform best on the datasets that have longer transactions. On \texttt{quest-wikilinks-small} (Figure C.1d), \textsc{cApriori} indeed is slightly faster than the other algorithms for large $\xi$ values, however, \textsc{CFP-growth} is the algorithm of choice for low $\xi$ values. \textsc{CEclat} performs best on \texttt{quest-netflix-small} (Figure C.1e) and \texttt{quest-netflix-small} (Figure C.1f) for high $\xi$ values. On the latter dataset, it is up to 3x faster than the better of the other two algorithms. Again, however, \textsc{CFP-growth} has the lowest execution time for small $\xi$ values since then the expensive tree building pays off.

Although, \texttt{quest-netflix$^7$-small} has rather long transactions, \textsc{cApriori} is much faster than the other two algorithms on this dataset (see Figure C.1g). The reason is that only a small number of candidates is created. These candidates thus easily fit into the caches so that \textsc{cApriori}’s subset counting is—despite of the long filtered transactions—quite fast. Contrary, building the CFP-tree in \textsc{CFP-growth} does not pay off and \textsc{CEclat} suffers from many long tid-lists in the first few iterations.

Summing up, none of the algorithms is always superior over the other algorithms. \textsc{cApriori} performs best as long as the transactions are short. \textsc{CEclat} is only on one dataset the clear winner, which however is attributed to the fact that we have only a small number of datasets. \textsc{CFP-growth} outperforms both other algorithms on almost all datasets as the $\xi$ value gets sufficiently small because summarizing the transactions using the CFP-tree then pays off. This competitive advantage might disappear when similar transaction summarizing techniques are integrated in \textsc{cApriori} and \textsc{CEclat} (cf. the discussion in Section 5.5 and 6.4).
C. Further Experimental Results

Figure C.1.: Execution time of \texttt{cAPRIORI}, \texttt{CECLAT}, and \texttt{CFP-GROWTH} on various datasets
References


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


TESLA M-Class Whitepaper. TESLA M-Class GPU Computing Modules. NVIDIA Corp., 2011.
REFERENCES


S. White. High-performance power-efficient x86-64 server and desktop processors using the core codenamed “bulldozer”. In *Hot Chips 23*, 2011.


REFERENCES


List of Figures

1.1. Outline of this thesis .................................................. 6
2.1. Association rule mining process ........................................ 7
2.2. Dataset characteristics of different application areas .................. 11
2.3. Breath-first search space traversal ...................................... 14
2.4. Itemsets in the equivalence classes \([a], [b], [c], \text{and} [d]\) .................. 18
2.5. An FP-tree ............................................................. 21
2.6. Branch in main loop ..................................................... 38
2.7. Branchless version ....................................................... 38
2.8. Multiprocessor system with 4 processors ................................. 43
2.9. Employed parallel algorithm models ..................................... 45
2.10. Parallelization of a loop using OpenMP ................................. 46

3.1. Scalar merge intersection of two sorted sets with 22 comparisons; the resulting
intersection set consists of 7 elements (gray shaded). ...................... 53
3.2. Code snippet for a list intersection implementation \textit{with branches} .......... 54
3.3. Code snippet for a \textit{branchless} list intersection implementation ......... 55
3.4. Intersection of bitmaps consisting of 40 bits (set bits are gray shaded). .... 56
3.5. Code snippet for a bitmap intersection implementation ................ 57
3.6. The full comparison of two vectors using the \texttt{PCMPESTRM} instruction. ..... 59
3.7. Code snippet for list intersection using speculative comparisons and SIMD
instructions. ......................................................................... 60
3.8. Parallel merge of two sorted sets with only three full comparisons and three
end-pair comparisons. The latter comparisons on the particular last value of
each vector (bold) are used to decide which 8 values are compared next. ...... 61
3.9. The 32-bit representation of 8 values (in hexadecimal representation) and their
respective hierarchical representation. ........................................ 63
3.10. Varying the selectivity for intersections in small domains ............... 66
3.11. Varying the selectivity and domain coverage for intersections in large domains
(32-bit values only) ................................................................ 67
3.12. Multi-threaded intersection results (32-bit only) ............................. 68

4.1. SIMD data layouts \((k = 4)\) .................................................. 79
4.2. Null suppression using the \texttt{PSHUFB} instruction \((k = 4)\) .................... 80
4.3. Code snippet for vector null suppression (decompression only) .......... 81
4.4. Memory representation of Elias gamma and \(k\)-gamma encoding of four example
codewords .......................................................................... 82
4.5. Code snippet for shared-base delta encoding (decode only). ............... 84
4.6. Compression ratios for various realistic datasets (lower is better) .......... 87
4.7. Throughput of the scalar static encodings ................................... 90
LIST OF FIGURES

4.8. Throughput of sequential and vectorized null suppression .......................... 91
4.9. Throughput of NULL and K-NULL on the high potential dataset .................. 93
4.10. Throughput of GAMMA and k-GAMMA encoding on the high potential dataset . 94

5.1. Transaction length distribution for various datasets ................................ 101
5.2. Parallel direct-count approaches .............................................................. 104
5.3. Multiple-producer/multiple-consumer processing with four partitions and two threads ................................................................. 105
5.4. Prefix array with three leaf nodes ............................................................. 108
5.5. Code snippet for recursive subset counting .............................................. 109
5.6. Prefix-hash tree for indexing ................................................................. 109
5.7. Count trie after applying the optimizations ............................................... 110
5.8. Code snippet for loop-based subset counting ............................................ 112
5.9. Effectivity of the trimming techniques ..................................................... 115
5.10. Normalized execution time of SIMPLE with different enabled optimizations on various datasets ................................................................. 120
5.11. Execution times of the different subset-counting algorithms on various datasets 121
5.12. Runtime improvements for trimming on various datasets .......................... 123
5.13. Runtime of CAPRIORI and the existing APRIORI implementations on various datasets ................................................................. 125
5.14. Memory footprint of CAPRIORI and the existing APRIORI implementations ... 126
5.15. Parallel direct counting for obtaining the frequent items on various datasets 128
5.16. Parallel direct 2-counting on quest-retail-small ....................................... 129
5.17. Parallel direct 2-counting on quest-retail-small for a varying number of threads ................................................................. 130
5.18. Overall performance on quest-retail-small and $\xi = 0.03\%$ ...................... 131
5.19. Overall performance on quest-retail and $\xi = 0.09\%$ .............................. 132

6.1. Tid-limaps for $n$ frequent items ........................................................... 136
6.2. Varying the limap transaction classifier $b$ within cECLAT .......................... 143
6.3. Running cECLAT using different data layouts and intersection algorithms .... 144
6.4. Runtime of certain cECLAT and ECLAT implementations on different datasets 146
6.5. Peak memory usage of the ECLAT-based implementations under test ........... 147
6.6. Results for multi-threaded cECLAT and a varying number of threads ........... 149

7.1. A ternary FP-tree .......................................................... 154
7.2. A CFP-tree .......................................................... 157
7.3. A CFP-tree node and its in-memory representation .................................. 159
7.4. Illustration of our memory manager ...................................................... 160
7.5. Conversion of an FP-tree to a CFP-array .............................................. 162
7.6. Hot-item partitioning with 8 partitions ............................................... 165
7.7. Tree-specific hot-item partitioning ...................................................... 166
7.8. A range partition tree with eight partitions ............................................ 167
7.9. FP-tree and WFP-tree build from a small transaction sample ..................... 169
7.10. Conditional trees for a CFP-tree (processed by two threads) ..................... 172
7.11. Average node sizes for different datasets ............................................ 175
LIST OF FIGURES

7.13. Overall single-threaded mining performance on $\text{quest-netflix}^T$-small ........ 178
7.14. Size of the largest partition for various datasets (lower is better) .............. 180
7.15. Tree build time for a varying number of threads (lower is better) .............. 181
7.16. Parallel tree building on $\text{quest-netflix}^T$ ........................................ 182
7.17. Full mining time for a varying number of threads .................................... 183

B.1. Code snippet for the parallel merge difference algorithm ............................... 193

C.1. Execution time of $\text{cApriori}$, $\text{cEclat}$, and $\text{CFP-growth}$ on various datasets 196
List of Tables

3.1. Setup overview for the set-intersection experiments . . . . . . . . . . . . . . . . 65

4.1. Average encoding/decoding bandwidths and compression ratios for popular
    linux compression tools on realistic mining datasets . . . . . . . . . . . . . . . . 75

5.1. False positive frequent 2-candidate items on realistic datasets . . . . . . . . . . 114
5.2. Size of the largest count-trie during mining (in MB) . . . . . . . . . . . . . . . 122

7.1. Fields and number of leading zero bytes in an FP-tree for the Webdocs data . . 155
7.2. Fields and number of leading zero bytes in the CFP-tree for the Webdocs data 158
7.3. Created partitions using the greedy-based partitioning algorithm . . . . . . . . 170

A.1. Summary of datasets . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 190
A.2. Chosen \( \xi \) values in the compression potential experiment in Section 7.3.2 . . 190
A.3. Example codewords for \( k \)-gamma . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 191
A.4. Example codewords for \( k \)-gamma\(_0\) . . . . . . . . . . . . . . . . . . . . . . . . . . . . 191
Selbständigkeitserklärung


Benjamin Schlegel
Dresden, 19. März 2013