NTNU – Trondheim
Norwegian University of
Science and Technology

Mining Frequent Intra- and Inter-Transaction Itemsets on Multi-Core Processors

Sebastian Zalewski

Master of Science in Computer Science
Submission date: June 2015
Supervisor: Kjetil Nørvåg, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem description

With the trend of increasingly parallel processor architectures, we aim to investigate how these can be utilized in frequent itemset mining. The task is 1) to study and improve existing techniques for frequent intra-transaction itemset mining on multi-core processors, and 2) to develop new techniques for inter-transaction mining on such processors.

Assignment given: 15.01.15
Supervisor: Kjetil Nørvåg

Abstract

The main focus of this report is on frequent intra- and inter-transaction itemset mining, specifically regarding parallel algorithms on shared memory multi-core processor systems. Multiple state-of-the-art algorithms are presented for both type of frequent itemset mining problems. Three novel parallel algorithms are presented and implemented, these include a parallel intra and two parallel inter algorithms. In addition to this, three state-of-the-art intra algorithms are implemented as well.

A thorough experimental procedure is conducted in order to analyse the implemented methods in-depth. This procedure includes the creation of multiple synthetic datasets with various characteristics. All intra and inter algorithms are tested on these datasets, in order to see how the behaviour of the algorithms changes with different dataset attributes. Multiple real datasets, which include already existing and created datasets, are used during the experiments as well. All these tests include a comparison between the algorithms, with different algorithm parameters.

The results from the experiments show that the utilization of multi-core processors on frequent itemset mining methods is good. In the best cases, on the tested system, the speedup was almost 35 times better than single threaded execution. However, the results also show that whenever a large quantity of frequent itemsets are generated, the system bus becomes the bottleneck. All in all the viability of parallelization of frequent intra- and inter-transaction itemset mining algorithms is advantageous.

Sammendrag

Hovedfokuset i denne rapporten ligger på generering av frekvente intra- og inter-transaksjons elementsett, spesifikt angående parallelle algoritmer på systemer med delt minne og flerkjernesprosessorer. Flere relevante algoritmer er presentert for begge problemområder. Tre nye parallelle algoritmer er presentert og implementert, disse inkluderer en parallell intra metode og to parallelle inter algoritmer. I tillegg til dette er tre relevante intra algoritmer implementert også.

En grundig eksperimentell fremgangsmåte er brukt for å analysere de implementerte metodene i dybden. Denne fremgangsmåten inkluderer etablering av flere syntetiske datasett med forskjellige karakteristikker. Alle intra og inter algoritmer er testet på disse datasettene, for å se hvordan oppførselen av algoritmene endres med forskjellige datasett attributter. Flere ikke-syntetiske datasett, som inkluderer allerede eksisterende og egen-produserte datasett, er brukt under forsøkene i tillegg. Alle disse testene inkluderer en sammenligning mellom algoritmene med ulike parametre.

Resultatene fra forsøkene viser at anvendbarheten av flerkjernesprosessorer på algoritmer for frekvente elementsett er bra. I de beste tilfellene, på systemet testene ble uført, var hastighetsøkningen nesten 35 ganger så bra som kjøring på en tråd. Derimot viser resultatene at når en stor mengde med frekvente elementsett blir generert, blir systembussen flaskehals. Alt i alt er anvendbarheten av parallellisering av algoritmer for frekvente intra- og inter-transaksjons elementsett fordelaktig.

Contents

Chapter 1

Introduction

This chapter consists of an introduction to the problem statement that this thesis covers in Section 1.1. The contributions of this report are presented in Section 1.2, as well as the thesis outline in Section 1.3.

1.1 Background

The goal of data mining procedures is to identify non-intuitive patterns in large quantities of data. If the data can be interpreted as a list of sets, with unique items in each set, a common mining procedure called *frequent itemset mining* (FIM) can be used to process the data. The algorithm is able to identify frequently appearing subsets of items within the dataset being mined. There are multiple applications for this type of task, for example in retail stores, e-business, web log mining, machine learning, and so forth. A couple of examples of the applications of FIM can be finding the following rules:

- If a certain laptop is bought, a specific computer mouse is often bought with it.
- Often when the price of stock A decreases, the price of stock B and C increases the same day.
- Whenever there is a traffic jam at street A and B, there also is a traffic jam at street C and D.

The relationship between the items above, i.e. whenever set A appears set B co-occurs, is called an association rule. In order to create association rules for a given dataset, we must first find the frequent itemsets. This is often the most resource heavy procedure, and hence the most interesting during algorithm construction.

Frequent itemset mining, as explained above, is one-dimensional and called frequent *intra*-transaction itemset mining. When we expand this problem statement to cover multidimensional transactions, it is called frequent *inter*-transaction itemset mining. The applica-

bility of such algorithms expands. Not only is it possible to find frequent itemsets in one transaction at a time, but it is also possible to find frequent itemsets spanning over multiple transactions. Practical applications of this include stock market analysis [6], text document analysis [22], e-business, web log mining, and so on. Some examples of inter-transaction association rules can be the following:

- If a certain laptop is bought, it can be shown that it is likely that the customer will buy a laptop battery one year later.
- The price of stock A increases, three days later the price of stock B decreases and the price of stock C increases.
- If there is a traffic jam at street A and B, then 10 minutes later there will be a traffic jam at street C and 20 minutes later at street D.

Whenever implementing such algorithms, and focusing on optimization, it is important to keep computer hardware in mind. The increase in CPU frequency has decreased during the last years, due to physical limitations, and the focus of CPU manufactures has shifted towards creating multi-core processors. Hence, in order to utilize modern processors to their fullest potential it is a necessity to program parallel algorithms. This introduces many challenges because of data synchronization and its costs. Often multiple threads are trying to access the same data, which results in data that needs to be locked. This again causes other threads to wait for data access, which results in sequential execution and impacts scalability negatively. However, by keeping the synchronization cost in mind it is often possible to create efficient parallel algorithms.

With a large number of cores it often leads to greater amount of data that needs to be transferred in the memory hierarchy. This is because multiple threads are asking for access to memory, either to write or read data. Since the cost of transferring data between the different memory hierarchies is expensive, it is important to create data-structures which minimize the transfer time. By grouping important data together fewer cache misses can be expected because of spatial locality. If the size of the data structures is minimized, more data can be stored at a given memory hierarchy. This leads to smaller transfer time. Often compression of data can be viable if the data transfer time becomes the bottleneck. Even though this can cause more CPU instructions to be executed, the overall performance gain can potentially be significant.

1.2 Contributions

This thesis focuses on parallel intra- and inter-transaction itemset mining algorithms on shared memory systems with multi-core processors. Multiple different in-memory algorithms are implemented with scalability in mind. Thorough experiments are performed with different parameters and datasets, in order to see how the different variables are affecting the implemented methods. The following research questions are addressed:

RQ1 How can we utilize multi-core processors efficiently in order to minimize mining time for frequent intra- and inter-transaction itemset mining algorithms?

RQ2 What impact do different dataset characteristics have on the performance for the focused algorithms?

RQ3 With the increasing amount of cores available on modern computers, how does the data transfer time between primary memory and CPU affect the scalability?

RQ4 How viable is it to apply compression techniques in order get a better cache utilization? Are the additional CPU cycles worth it?

In combination with these research questions, the following are the main contributions of this thesis:

- Multiple existing parallel state-of-the-art frequent intra-transaction itemset mining methods are implemented. These include CFP-Growth [27], ShaFEM [35], and a parallel Eclat algorithm that is based on two existing Eclat algorithms [37][28].
- A new parallel frequent intra-transaction itemset mining algorithm is introduced and implemented. This is a hybrid method between two already existing state-of-the-art algorithms, which is able to preserve their strengths.
- Two novel parallel inter-transaction itemset mining algorithms are described and implemented. These are believed to be the first parallel algorithms for this type of task.
- A new dataset is created consisting of stock data (see Section 7.2.2). Multiple synthetic datasets are created with variations in transaction count, average transaction length, and item count (see Section 7.2.3).
- A thorough analysis is given between all these implemented algorithms with the created, and some already existing, datasets. This analysis consists of a comparison between the algorithms, and an analysis of the behaviour of each algorithm for variations in different parameters.

1.3 Thesis outline

The content of this thesis is organized in the following manner. Firstly an introduction is given to hardware, and parallel programming frameworks in Chapter 2. An introduction to frequent intra- and inter-transaction itemset mining is given in Chapter 3. This chapter also includes the most fundamental frequent itemset mining methods. Some state-of-the-art frequent intra-transaction itemset mining algorithms, as well as the novel hybrid method, are presented in Chapter 4. This is again followed by some of the state of the art methods for frequent inter-transaction itemset mining methods given in Chapter 5. Here the two novel inter algorithms, i.e. EP-Index-BTFITI and FGP-Index-BTFITI, are presented as well. The implementation details of the implemented methods used during experiments,

as well as the new algorithms created, are presented in Chapter 6. Experiments and results are documented and shown in Chapter 7. Finally, we conclude this thesis in Chapter 8. Answers to all the research questions and a description of further work is given in this chapter as well.

Chapter 2

Overview of hardware and parallel programming frameworks

This chapter gives a short introduction to different storage devices, and its importance during algorithm construction in Section 2.1. An introduction to parallel systems, and their advantages, is given in Section 2.2. Lastly some tools for parallel programming are presented in Section 2.3.

2.1 Data storage

In this section a short description is given of the most commonly used memory devices. It is important to understand the memory hierarchy when implementing algorithms with performance in mind. One of the reasons for this is that data transfer time often becomes the bottleneck when processing large amount of data. By implementing data structures that are able to minimize data usage, the algorithm can potentially gain a lot of performance. This can result in minimal access to secondary storage, or minimal accesses to primary memory, because the data can be stored in a higher level in the memory hierarchy.

2.1.1 Secondary storage

Secondary storage devices are non-volatile units, which are able to store data even if the systems power supply is turned off. The most common types of secondary storage, in modern computers, are hard disk drives (HDD) and solid-state drives (SSD). A hard disk drive usually consists of multiple rotating magnetic disks, that are together capable of storing up to hundreds even thousands of gigabyte of data. The rotation speed on different HDDs

typically ranges from 5400 to 15000 rpm. Multiple factors are determining the amount of time it takes to acquire data from a hard disk drive. These factors include seek time, rotational delay and block transfer time [8]. A typical media transfer rate for a 7200 rpm HDD is around 130 MB/s [29], however in practice transfer time is lower due to seek time and rotational delay.

A solid state drive is a much faster, and more expensive, data storage unit than an HDD. Data is stored in NAND Flash memory, which allows for faster data transfer throughput than in a magnetic disk. Typically the read speed can be around 500 MB/s, while the write speed is often lower and can be around 250 MB/s. Average access time is typically 0.1 ms. Solid state drives are usually more expensive than HDDs, and hence often have less space available. However, SSD are getting larger and it is possible to get SSDs with hundreds of gigabytes available storage space. [21]

2.1.2 Primary memory

Primary memory in most systems consists of dynamic random-access memory (DRAM). This is a volatile data storage unit that requires power in order to keep data in memory. It allows the system fast random-access to different memory locations. Typical memory access time ranges from 10-100 nanoseconds, and data transfer rate can be up to multiple gigabytes per second [15]. It is slower than cache, but much faster than secondary storage devices.

The operating system uses virtual memory, in order to be able to allocate more memory than is available on RAM. This is done by combining system memory with secondary storage. Whenever there is memory available in RAM, memory is allocated from this unit. However if the primary memory runs out of space, data from the system memory needs to be swapped to secondary storage. If we take into comparison the speed of DRAM and SSD or HDD, this will result in a major decrease in performance. [23]

2.1.3 Cache

Because it takes time to transfer data from primary memory over the system bus, CPUs have cache on their chip. Modern CPUs typically have multiple levels of cache, where each level has different amount of available memory and a different speed. For example, in the system used during experiments, there are three levels of cache (see figure 2.1). These are L1, L2, and L3. The size of L3 cache is around 35 MB, and is shared between all cores on the chip and also is the slowest. L2 cache is smaller, consisting of 256 KB, but each core has its own L2 cache. The same applies for L1. However, there are two types of L1 caches, one instruction cache and a data cache. Both of these caches are able to store 32KB of data and are the fastest cache on the chip. The access time for on-chip cache can be around 100 picoseconds [15], which is between 100-1000 times faster than DRAM.

Data is transferred in blocks of memory, i.e. cache blocks, which often consists of 8 to 16 times the amount of space it takes to store data in a single memory location. By trans-

ferring data in blocks the system accomplishes *spatial* locality, i.e. data access to a location often leads to another data access close to this location. Therefore it is important to group together the most important data when implementing high performance algorithms. Whenever data is not found in cache, it is called a *cache miss*. If, on the other hand, the information is found it is called a *cache hit*. When a cache miss occurs, the system needs to look for that information either in a higher cache level or in primary memory. [23]

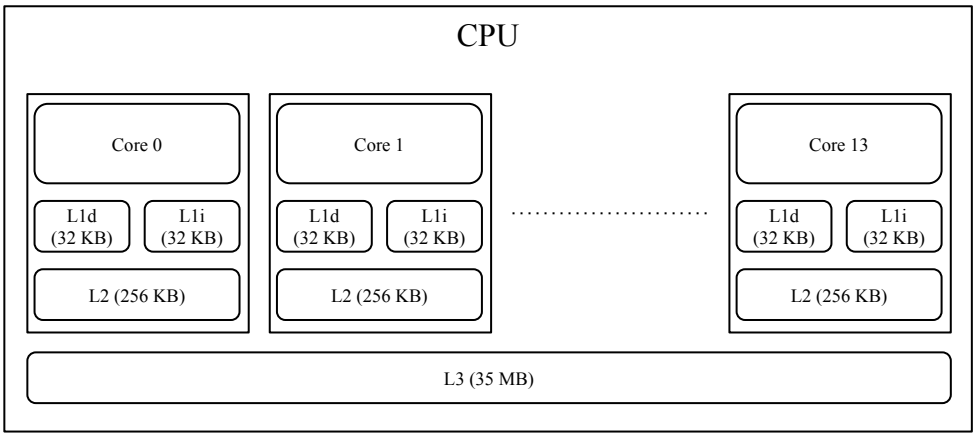


Figure 2.1: Diagram of the CPU that was used during experiments.

2.2 Parallel systems and its properties

This section gives a short introduction to parallel systems and its properties. Firstly a short introduction is given for different data processing architectures, followed by an introduction to different memory systems. Lastly the potential speedup that is possible to obtain within such systems is discussed.

2.2.1 Introduction

Multiple different computer systems exist with different types of properties. Flynn's taxonomy [10] provides a classification for different data processing architectures. These are listen in Table 2.1.

	Single Data Stream	**Multiple Data Stream**
Single Instruction	SISD	SIMD
Multiple Instruction	MISD	MIMD

Table 2.1: Flynn's taxonomy

Previously regular computers used to be SISD systems, i.e. machines with single core processors. This was sufficient because single cores in microprocessors had an average increase of performance by 50% per year between 1986 and 2002. Afterwards the increase in performance decreased down to 20% [23]. Hence, microprocessor manufacturers had to switch their focus from single-core processors to multi-core. These types of processors can be seen as SIMD systems. This means that they execute one instruction at a time, but are able to process multiple data streams simultaneously.

2.2.2 Memory systems

A distributes system can be seen as a MIMD system. This means that it can consist of multiple computers executing different instructions on multiple data. All these computers have their own memory, and hence the memory system can be described as a distributed-memory system. A single computer that consists of a CPU, or multiple CPUs, can be described as using a shared-memory system. This means that all the cores in the system share the same memory. An illustration of these concepts is shown in Figure 2.2.

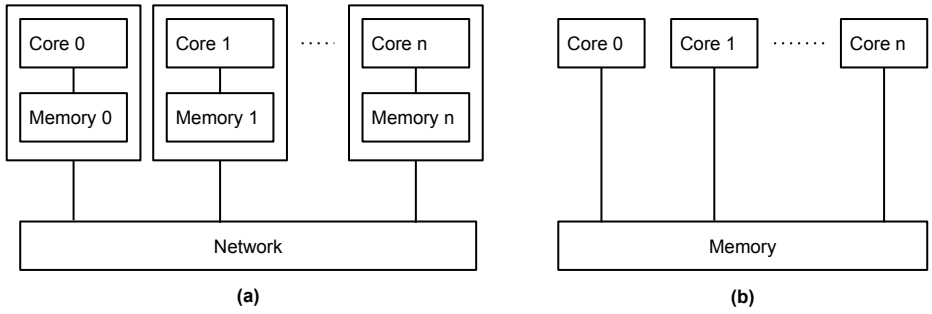


Figure 2.2: (a) A distributed memory system. (b) A shared-memory system.

In a shared-memory system with multiple processors the memory can be accessed in different ways. If all processors can access memory through a shared interconnect, the system can be classified as a *uniform memory access* (UMA) system. An example of a UMA system is shown in Figure 2.3. There are also architectures where each processor has quick access to their own memory, but in order to access the memory of another CPU they need to get that data directly through that CPU. These type of systems are called *nonuniform memory access* (NUMA) systems, an example is shown in Figure 2.4.

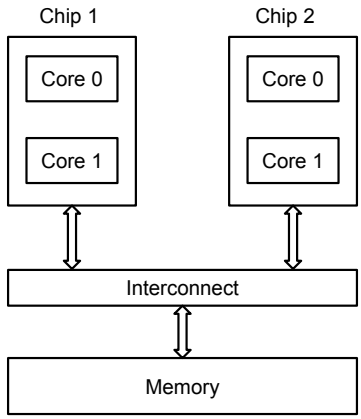


Figure 2.3: UMA system.

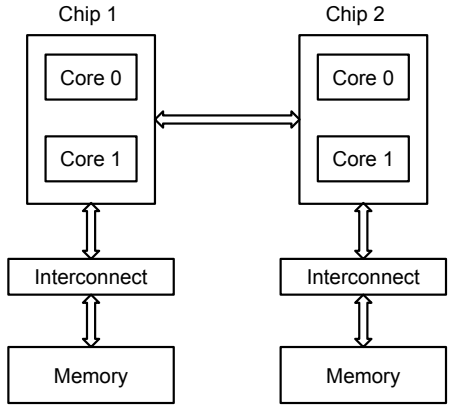


Figure 2.4: NUMA system.

2.2.3 Scalability

In [2] Gene Amdahl made an observation about task parallelization that later lead to Amdahl's law. This law say that the speedup of a parallel execution of a procedure, compared to sequential, can potentially equal the amount of parallel processing units available. The only factor limiting the speedup is the sequential part of a procedure, i.e. the portion of the algorithm that is not possible to parallelize. Equation 2.1 illustrates Amdahl's law, where f indicates the fraction of the code that can be parallelized and m denotes the number of

parallel execution units.

$$Speedup = \frac{1}{1 - f + \frac{f}{m}} \tag{2.1}$$

However, there are some rare cases where the speedup of an application can exceed the limit given by Amdahl's law. Multi-core processors often have small caches for each core that are not shared. By using more cores during execution, more cache memory becomes available and thus can result in better performance. This can potentially breach the barrier given by Amdahl's law.

Amdahl's law might seem restrictive, because let's say that we have an algorithm where 20% of it is sequential. Then the best theoretical speedup, with 10 cores, is supposed to be only $\frac{1}{1-0.8+\frac{0.8}{10}} \approx 3.57$. If we scale the system up to 100 cores, then the speedup is $\frac{1}{1-0.8+\frac{0.8}{100}} \approx 4.8$, which is a small increase. However, Gustafson reevaluated Amdahl's law in [12] and come to the conclusion that with more parallel execution units available the problem size also needs to scale. The concept of scaled speedup was introduced:

$$Scaled\_speedup = m + (1 - m) \cdot s \tag{2.2}$$

In the equation m can be seen as the amount of cores available, while s indicates the fraction representing the sequential part of the algorithm. The idea is that the workload is not fixed, and hence with increasing amount of cores it is possible to get a good speedup with a larger amount of data to process.

2.3 Parallel programming frameworks

In order to utilize a multi-core processor efficiently it is needed to create separate instances of either threads or processes that can run asynchronously. A process consists of the following elements [23]:

- Executable machine code
- A memory block
- Resource information
- Security information
- State information

A thread on the other hand is more light weight and doesn't contain as much information. Thus, it is faster to create a thread than a process. A process can create and manage multiple threads, this is illustrated in Figure 2.5.

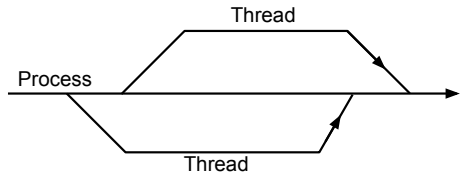


Figure 2.5: A process with two threads.

In order to create parallel algorithms it is important to have an efficient way to tell the operating system to create either threads or processes. Efficient communication between these created instances is also important. This is often accomplished through parallel programming frameworks. Some well known parallel frameworks including MPI, OpenMP and Pthreads are presented in this section.

2.3.1 MPI

The parallel programming framework MPI stands for Message Passing Interface. This framework contains methods that make it possible for the application to create multiple processes. Methods that allows the program to communicate efficiently between the created processes is also a fundamental part of the framework. MPI can be used in heterogeneous environments, i.e. systems which contain multiple different processes. This framework can be used with programming languages such as C, C++, and Fortran. [33]

2.3.2 OpenMP

OpenMP is a framework that is designed for shared-memory systems, and is well suited for incremental parallelization of existing software. It is a high level framework that provides good scalability, and can be used with C, C++ and Fortran. In order to parallelize a section of a code, compiler directives can be used. The framework also contains a runtime library with a wide variety of functions, e.g. methods for synchronization by using locks. [7]

2.3.3 POSIX Threads

POSIX Threads, also known as Pthreads, allows the programmer to create multi-threaded programs. Pthreads are mainly aimed at POSIX system, where POSIX stands for Portable Operating System Interface. However, there are variants of this framework for Windows as well. Compared to OpenMP, the programmer has better thread control and hence this can be seen as a lower level framework. Rather than define sections of code that are parallel, it is possible to instantiate a thread and let it call a function. [4]

Chapter 3

Introduction to frequent itemset mining

An introduction to intra- and inter-transaction association rules, and its most basic concepts, are presented in Chapter 3.1. The most well known frequent itemset mining methods are also covered. This includes Apriori in 3.2, Eclat in 3.3, and FP-Growth in 3.4.

3.1 Introduction

Transaction based databases can have a great amount of non intuitive information contained within them. Tools are needed in order to extract valuable information from these datasets. This section contains information of some well known methods for extracting this type of data, i.e. *association rule learning* and *frequent itemset mining.*

3.1.1 Association rule learning

When analysing transaction databases of retail stores, some interesting patterns can often be found in the data. For example, in some scenarios it can be shown that customers who buy a set of products X also buy another set of products Y. The relationship between these sets can be written as $X \rightarrow Y$, i.e. when X occurs in the transaction Y often co-occurs. These two sets however must be disjoint; otherwise the result would not be as interesting. This relationship between these sets is called an association rule. In order to measure the strength of an association rule we need to define two measurements, i.e. *support* and *confidence*. These are defined as follows:

$$support(X \rightarrow Y) = \frac{\sigma(X \cup Y)}{N} \tag{3.1}$$

$$confidence(X \rightarrow Y) = \frac{\sigma(X \cup Y)}{\sigma(X)} \tag{3.2}$$

The function $\sigma(Z)$ returns the amount of items within set Z. N represents the number of transactions in the database. Support measures the frequency of an itemset in the transaction. This value tells if the itemset simply occurred by chance, or if this is a set that is likely to occur. Confidence denotes the strength of an association rule, i.e. the probability that an itemset occurs given another itemset. However, in order to find association rules we need to first find frequent itemsets. This problem is defined in the next section. [30]

3.1.2 Frequent itemset mining

If we have a set of symbols called items represented by J, an itemset is defined as a subset $I \subseteq J$. Let's say we have a sequence of itemsets $\tau = \langle t_1, ..., t_n \rangle$, this collection of data is often called a transaction database. A cover of itemset $I \subseteq J$ is defined as:

$$cover_J(I) := \{t \in \tau | I \subseteq t\} \tag{3.3}$$

The cover consists of transactions containing the itemset I. Support value for a given transaction is defined as:

$$sup_J(I) := |cover_J(I)| \tag{3.4}$$

This value is the number of transactions that contain the itemset I. The goal of frequent itemset mining is to find frequently occurring itemsets, where the frequency is a user defined threshold called *minimum support*. Minimum support is often abbreviated as *minsup*. The set containing all frequent itemsets is defined as:

$$F_{J,minsup} := \{I \subseteq J | sup_J(I) \geq minsup\} \tag{3.5}$$

Support value is often differentiated between relative support and absolute support. Where the relative support is defined as $\frac{|cover_\tau(I)|}{|\tau|}$, and hence is a value in the range $[0, 1]$. A support value of 1 indicates that the itemset is contained within all transactions, whereas a support of 0 says that the itemset is nonexistent. Absolute measurement is simply $|cover_\tau(I)|$, i.e. the amount of transactions that contain the given itemset. [3]

3.1.3 Dataset characteristics

A dataset for a frequent itemset mining algorithm consists of a list of transactions. Each transaction is a set of unique items. The main characteristics of a dataset are the number of transactions, average transaction length and the number of unique items. It has been

shown that different algorithms perform differently based on dataset characteristics. This information can be used in order to create adaptive methods for frequent itemset mining [9]. Density is an important key factor when it comes to performance for different FIM methods. Dataset density is defined as average transaction length divided by the number of unique items in the dataset [39]. When this value is hight the dataset is said to be dense, otherwise it can be characterised as a sparse dataset.

3.1.4 Inter-transaction association rules

This subsection is based on the description of inter-transaction itemset mining given in [32]. Association rule learning, as described in Section 3.1.1, is single-dimensional. The found rules are among items that are in the same transaction, these rules are called intra-transaction association rules. Inter-transaction association rules, on the other hand, describe the association relationship between items in different transactions. Thus, these association rules are multidimensional. An example of the applicability of such rules can be taken from the stock market. For example, let's say that the prices of stock A and B goes up the same day and in three days stock C will drop. A classical association rule would not be able to predict this, since it would only take into account the behaviour of stocks during the same day. However, since inter-transaction association rules are multi-dimensional they also have the ability to predict this kind of behaviour.

A transaction database for inter-transaction itemset mining is similar to a regular itemset mining transaction database. The only difference is that the items have an attribute that describes a property of the item. This can be, for example, time or location. More formally the transaction database can be defined to contain transactions in the form of (d, E), where $E \subseteq \Sigma = \{e_1, e_2, ..., e_u\}$ and $d \in Dom(D)$. Σ is a set of items, and $Dom(D)$ denotes the domain of D. $Dom(D)$ contains the properties of the items, e.g. time of occurrence (day 1, day 2, ... day N).

In order to reduce processing time, and since it is not always interesting to find association rules spanning over a certain amount of transactions, the mining parameter *maxspan* is defined denoted as w. This parameter defines that rules which span over w transactions should not be taken into consideration. A *sliding window* of size w, is used during mining. The sliding window is defined as a block of w intervals along domain D, where it always starts from an interval that contains a transaction. An example of this is shown in Figure 3.1, where the sliding windows are denoted as W.

Each sliding window forms a *megatransaction* M, where M is defined as a transaction that contains all items in the given sliding window. More formally a megatransaction is defined as follows: $M = \{e_i(j) | e_i \in W[j], 1 \leq i \leq u, 0 \leq j \leq w-1\}$, where u is the number of items in the transaction database and w is the maxspan. For example the sliding window $W2$ in Figure 3.1 forms the megatransaction $\{b(0), c(0), a(1)\}$. Since the items in a megatransaction contain item properties as well, they are called *extended-items*. The set of all possible extended items is denoted as Σ'. With this in mind, *inter-transaction itemset* and *intra-transaction itemset* can be defined formally. An *intra-transaction itemset* is defined as $A \subseteq \Sigma$, where Σ was defined as all unique items in the transaction database.

An *inter-transaction itemset* is defined as $B \subseteq \Sigma^{'}$, i.e. a set of extended items, such that $\exists e_i(0) \in B, 1 \leq i \leq u$.

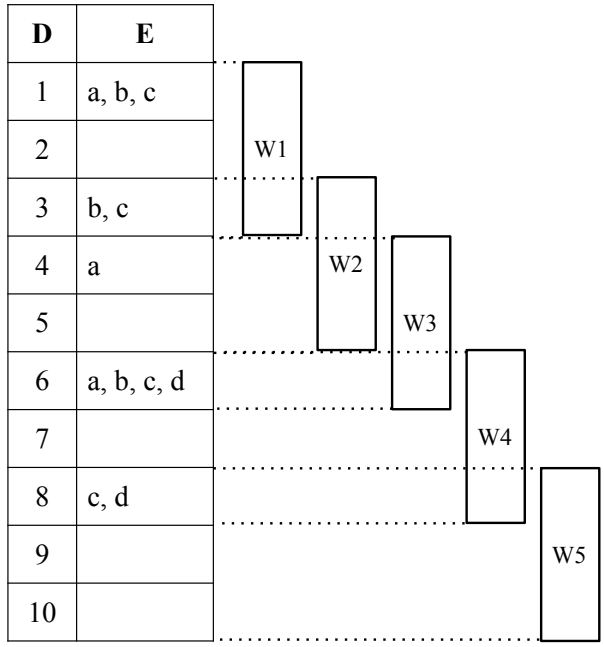


Figure 3.1: Inter-transaction database with five sliding windows, i.e. $W1, W2, W3, W4$, and $W5$. $W4[0]$ contains the items $\{a, b, c, d\}$ and $W4[2]$ contains $\{c, d\}$.

Inter-transaction association rule learning is quite similar to classical association rule learning. However, as stated earlier, the main difference is multidimensionality. An inter-transaction association rule is given in the form $X \Rightarrow Y$, which implies that if X occurs it is likely that Y also will co-occur. The criteria for inter-transaction association rules, as given in [32], are given below:

1. $X \subseteq \Sigma^{'}, Y \subseteq \Sigma^{'}$
2. $\exists e_i(0) \in X, 1 \leq i \leq u$
3. $\exists e_i(j) \in Y, 1 \leq i \leq u, j \neq 0$
4. $X \cap Y = \emptyset$

Whereas in regular association rule learning $X, Y \subseteq \Sigma$, and j is always 0 in statement 3 above. Confidence and support is used in order to determine what a good association rule is. These measurements are defined below.

$$support = \frac{|T_{xy}|}{N} \tag{3.6}$$

$$confidence = \frac{|T_{xy}|}{|T_x|} \tag{3.7}$$

N is the number of transactions in the database, T_{xy} is the set of megatransactions that contain the extended-items from X and Y, and T_x contains the set of megatransactions that contain items from X. As in classical association rule learning the goal is to find the rules that satisfy the minimum confidence and minimum support thresholds. The process of finding the association rules consists of two stages. Firstly, all frequent inter-transaction itemsets need to be found. Then the association rules can be created from the found frequent itemsets. The main bottleneck is the first stage, and hence is often seen as the most important stage to optimize during algorithm construction.

3.2 Apriori

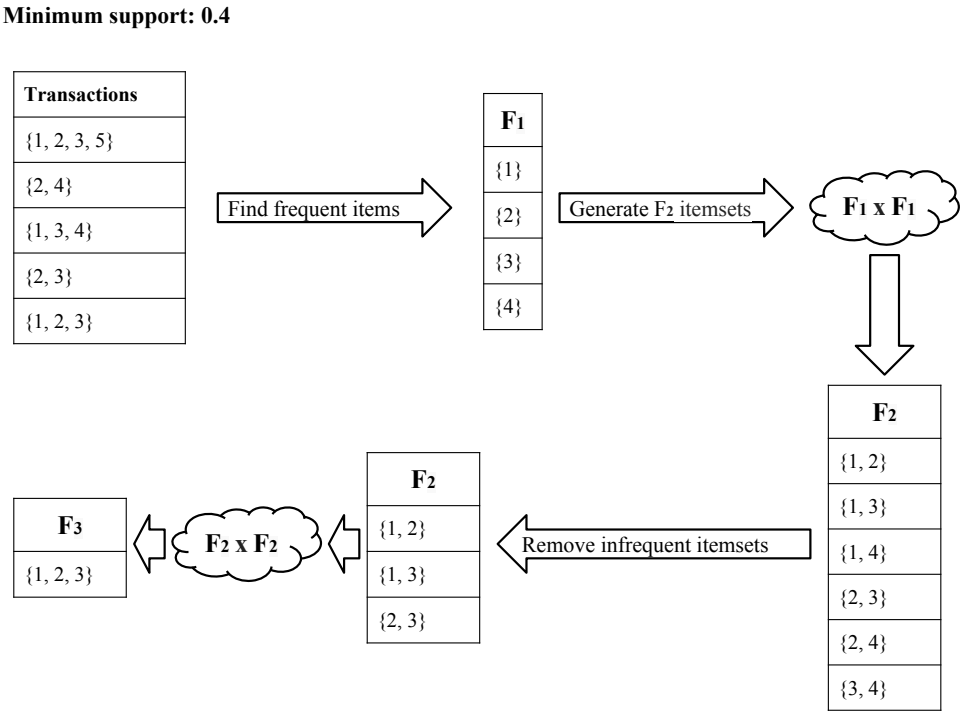


Figure 3.2: Example of the Apriori algorithm with minimum support set to 0.4.

One of the best known algorithms for frequent itemset mining is the Apriori algorithm [1]. It works by iteratively creating frequent itemsets of increasing cardinality. Firstly the

algorithm starts by doing an initial scan where the frequent items are found, F_1. In order to generate candidates C_k, where k is the iteration, the frequent itemsets F_{k-1} are used. The itemsets in F_{k-1} are merged with the itemsets in the same set where the first $k-2$ items are equal, and the last item is not equal. This will create itemsets of size k. To keep the search space small, a pruning step needs to be performed. For each transaction all itemsets of size k are created. These itemsets are then scanned against the generated candidates C_k. If a match is found, the support value of the candidate is increased. The last step is to remove all the infrequent itemsets, and then increase the iteration value k by one. All these steps are repeated until the set of candidates, C_k, is empty.

Figure 3.2 shows an example of the apriori algorithm. Minimum support is set to 0.4, i.e. in order for an itemset to be frequent it must be contained within at least two transactions. The final frequent itemsets found from the example are: {1}, {2}, {3}, {4}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}.

3.3 Eclat

In this section the well known Eclat [38] algorithm is presented. This is an algorithm that is well suited for dense datasets, and often has good performance when processing data with a high minimum support. Similarly to Apriori, Eclat uses candidate generation in order to find frequent itemsets. However, while Apriori uses a horizontal data representation Eclat uses a vertical data layout. What this means is that, in the internal data layout, each row consists of an item and a TID-list which denotes in which transaction the item is within. In Apriori this is the other way around, each row represents a transaction and also contains information of which items are within that given transaction. Another difference between Eclat and Apriori, is the way the candidates are being generated. In Eclat a depth-first traversal is done, in order traverse the candidate space. Apriori uses a breadth-first type of traversal. The way depth-first traversal is done in Eclat, is by employing something called equivalence class mining. An equivalence class is a group of itemsets that share the first *k* items. For example, the itemsets {1, 3, 4, 5} and {1, 3, 4, 7} are in the same equivalence class, while the itemset {1, 2, 4, 6} is not in the same equivalence class. Under candidate generation equivalence classes are created recursively, where for each recursive call the size of the equivalence class is increased by one. This is done in a depth-first traversal fashion, so that all possible frequent candidates are generated. If there are not generated any new frequent itemsets during a call, the function does not call itself recursively.

In order to check if a new generated itemset is frequent, set operations are used. For example, let's say we have the itemsets {1, 2, 3} and {1, 2, 4}. If we combine these sets together we get the itemset {1, 2, 3, 4}. To see if this is frequent we need to intersect the TID-lists of the two itemsets that generated the new one. The TID-list contains all transaction identifiers for a given item, i.e. all transactions that the item is a part of. In this example we define the TID-list of {1, 2, 3} to be τ({1, 2, 3}) = {t1, t3, t5, t6} and for {1, 2, 4} to be τ({1, 2, 4}) = {t3, t6, t7}. The intersection of the TID-lists will be as follows: {t1, t3, t5, t6} $\cap$ {t3, t6, t7} = {t3, t6}. This means that τ({1, 2, 3, 4}) = {t3, t6}. The support count

for an itemset is the number of TIDs in the TID-list. For the itemset {1, 2, 3, 4} the support count is |{t3, t6}| = 2.

A detailed description of how the algorithm works is given in the pseudocode in Figure 3.3[1]. Here we see that the algorithm starts off by finding the frequent items, and then transforming the database to a vertical layout. Then the recursive function *Bottum-up* is called, which consists of candidate generation and testing the frequency of the candidates generated.

Algorithm 1: Eclat algorithm

Data: Transaction database.
Result: Frequent itemsets.

```
1  Find frequent items F_1;
2  Transform database to vertical layout;
3  Function Bottom-up(F_k)
4      for each itemset α_i ∈ F_k with i = 1, 2, ..., |F_k| do
5          F_{k+1} = ∅;
6          for each α_j ∈ F_k with j = i + 1, ..., |F_k| do
               /* τ(α) is the tid-list of α, and β = α_i ∪ α_j   */
7              τ(β) = τ(α_i) ∩ τ(α_j);
8              if |τ(β)| ≥ ξ then
9                  Add β to the frequent itemsets F_{k+1};
10             end
11         end
12         if F_{k+1} is not empty then
13             Bottom-up(F_{k+1});
14         end
15     end
```

Figure 3.3: Eclat algorithm

3.4 FP-Growth

The FP-Growth [13] algorithm is often considered good for sparse datasets. That is datasets with a high amount of unique items compared to average transaction length. Candidate generation is not needed because of its trie data structure, which often also results in less memory usage. This data structure is explained in Section 3.4.1. The FP-tree is actively used in the mining procedure, which is presented in Section 3.4.2.

[1] The code is based on the pseudocode given in [28]

3.4.1 FP-Tree

The first step in the FP-Growth algorithm is to find all the frequent items from the transaction database. These items are then sorted in support descending order. Then the FP-tree can be created. Initially the FP-tree only contains the root element, i.e. *null*. For each transaction in the dataset all infrequent items are removed, and the remaining items are sorted in support descending order. These items are added to the tree, in the given order, where each item creates a node with a support count value. These nodes are connected to each other, and create a path in the tree. Whenever an equal path of items is added, the support count value gets incremented by one in each node. This is done in order to indicate that another transaction with these items exists. If a path that is not present in the tree is inserted, then new nodes need to be created in the tree in order to represent this new path. The support count values for each node in this path are set to one. If a partially equal path, i.e. a path where the first items are equal to the existing path, is added to the tree. Then the new path is created from the point where the path diverges from the existing path. The first nodes containing the equal items gets their support value incremented by one, whereas the last newly created nodes have support count set to one.

A table that contains all frequent items is also created; this table is called a *header table*. Each item, in the table, contains a link to the first node containing the given item. All nodes, that contain the same item, are connected to each other through a linked list. This means that it is possible to find all nodes containing a specific item through the header table.

Transactions
{1, 2, 3, 5}
{2, 4}
{1, 3, 4}
{2, 3}
{1, 2, 3}

Table 3.1: Transactions

Let's say that we have the transactions given in Table 3.1. We set the minimum support to 0.4, and then remove all infrequent items and sort the items in support descending order. The result is shown in Table 3.2.

Item	Support count
2	4
3	4
1	3
4	2
5	1

Transactions
{2, 3, 1}
{2, 4}
{3, 1, 4}
{2, 3}
{2, 3, 1}

Table 3.2: To the left: support count in support descending order. To the right: transactions with frequent items in support descending order.

The corresponding FP-tree for the given transactions, in Table 3.2, will look as depicted in Figure 3.4. In this figure we see the header table with all items contained in the FP-tree, with a link to the first node containing the given item. Each node contains an item value, a support count value, a link to its parent or child node, and a link to a node containing an equal item.

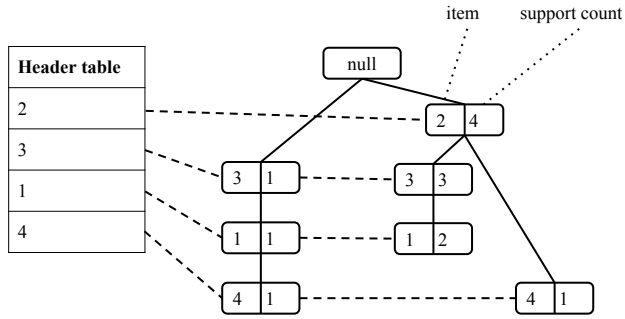


Figure 3.4: The FP-tree corresponding to the transactions given in Table 3.2. The value on the left in the nodes represents the item, value on the right represents the support count.

3.4.2 Mining

After the FP-tree is build, the mining procedure can be started. First all paths ending with a specific frequently occurring item must be found. These paths are called *prefix paths*. Prefix paths can be easily found by following the linked lists in the header tables. For each node in the linked list a prefix path can be found by following the parents links to the root node. When all the prefix paths are found for the given item, they are merged together into a tree. This tree is called a *conditional pattern tree*. From this tree all infrequent items are removed, and the remaining items can be combined with the last element in the tree in order to form a frequent itemset with a cardinality of two. After this another item is chosen in the *conditional pattern tree*, and all of its prefix trees are found. The same procedure is followed in order to find frequent itemsets of a size of three. This can be done recursively until all frequent itemsets are found.

Chapter 4

Frequent intra-transaction itemset mining on multi-core processors

In this chapter some of the parallel state-of-the art methods, as well as the novel hybrid algorithm, for frequent intra-transaction itemset mining are presented. Firstly the memory efficient CFP-Growth is presented in Section 4.1. ShaFEM with its dynamic mining method is presented in Section 4.3. In Section 4.2 a short description is given for two different parallel Eclat algorithms. Lastly the novel hybrid algorithm, that is a combination of ShaFEM and CFP-Growth, is presented in Section 4.4.

4.1 CFP-Growth

In order to achieve a high level of performance for frequent itemset mining, it is important to keep as much data in memory as possible. However, memory can become a constraint for large datasets. This is why compression of data in memory can be a benefit for some algorithms. CFP-growth [27] is a compact FP-growth algorithm with a high focus on memory efficiency. The following subsections are going to introduce this method, including the sequential and parallel versions of the algorithm.

4.1.1 Introduction

CFP-growth consists of three main phases, i.e. CFP-tree construction, CFP-tree to CFP-array conversion, and a final mining phase. Firstly the tree needs to be constructed from the dataset. This tree focuses on memory efficiency, and stores only the most essential information. After the tree is constructed from the dataset, it needs to be converted to a

CFP-array. This is done with both of the data structures in memory during the conversion. When this phase is finished the CFP-tree is discarded and deleted from memory. Lastly the CFP-array is used during the mining phase. This data structure is optimized for this procedure.

4.1.2 CFP-Tree

In contrast to a FP-tree a CFP-tree is a ternary tree, i.e. each node can have up to three children. Since upwards pointers are not needed, during the construction phase, a node contains only references that are pointing to nodes beneath it. Each node can have up to three pointers; these are called left, suffix, and right. Neighbouring nodes are placed in left or right pointer to a specific node, depending on the item value. If the item value is greater than the current node the reference is placed in right pointer, otherwise in left pointer. Direct children are placed in the suffix node. Let's say that the following itemsets, in the following order, are added to a ternary tree: {1, 3, 4, 5}, {1, 2}, and {1, 4, 6}. The tree will then have the form as given in Figure 4.1.

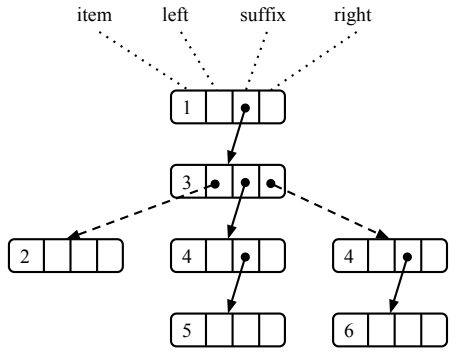


Figure 4.1: A ternary tree created from the following itemsets (in the given order): {1, 3, 4, 5}, {1, 2}, and {1, 4, 6}.

With the ternary tree it is possible to perform binary search in order to find a specific child node, instead of performing linear search. To reduce memory usage, the item and count value of each node is stored with only as many bytes as needed. For example all values smaller than 2^8 are stored with only one byte, else if the value is smaller than 2^{16} but larger or equal to 2^8 the values are stored with two bytes, and so on. Here we see that it is beneficial to have as small values for item and count as possible. It is possible to decrease the values further by only storing delta item and partial count instead. Delta item is the difference of the item value of the current node and its parent. Hence, if we insert the following transaction to the tree: {1, 3, 4, 7}. The delta item values will be the following: {1, 2, 1, 3}. Partial count is a bit more complicated on the other hand. This value is only incremented in the last node an item is added from a transaction. For example if we look at the previous transaction, i.e. {1, 3, 4, 7}, only the node containing item 7 gets its partial

count incremented. Thus, the total count of a node is the sum of the partial count of all its children and itself. This includes the children's children as well. A visual example of this is given in Figure 4.2. Here we see the equivalent of Figure 4.1 but with delta item (denoted as $\Delta item$) and partial count (denoted as *pcount*).

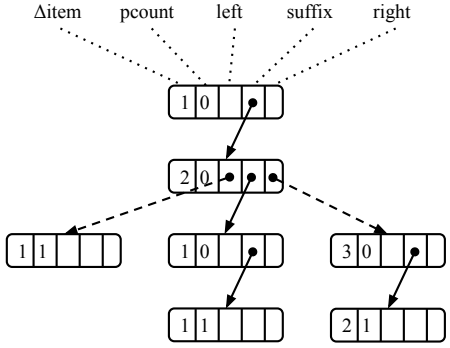


Figure 4.2: The equivalent of Figure 4.1 but with *pcount* and $\Delta item$.

In order to calculate the count value for the first node, i.e. the node that contains item 1, we need to sum the *pcount* value of all its children. Therefore the count value is $0 + 1 + 0 + 0 + 0 + 1 + 1 = 3$.

The nodes are stored as bit arrays in order to minimize memory usage. A bit array for a given node consists of multiple parts, these include a bitmask, delta item, partial count, left pointer, suffix pointer, and right pointer. The bitmask indicates how many bytes to use in order to store the $\Delta item$ and *pcount* value. It also specifies which pointers are *null* pointers, which are not stored in the bit array. The bitmask consists of eight bits, i.e. a byte. The first two bits tell how many preceding zero bytes the 32-bit $\Delta item$ variable consists of. Since this value is always greater than or equal to one only two bits are needed. After all, the highest number of preceding zeros for this value is three, in which case the value is smaller than 256. The next three bits in the bitmask are specifying how many preceding zero bytes the 32-bit *pcount* value consists of. This value can be zero, and hence three bits are needed in order to tell if up to four zero bytes are contained in *pcount*. Lastly, the three bits in the bitmask indicate if left, suffix, or right pointer is *null* or not respectively. In a 64-bit program the pointer size is eight byte, however rarely does a system need to use pointers of that size. With a five byte pointer it is possible to manage a memory of size $2^{5 \cdot 8}$ byte $= 1099511627776$ byte $= 1\text{TiB} \approx 1\text{TB}$. We can define and hardcode a pointer to be five bytes in CFP-growth.

An example of a node representation is shown in Table 4.1. From the bitmask we see that $\Delta item$ contains three preceding zero bytes. This is because the first two bits are 11 which are representing three in binary. The next three bits are 001 and indicate that *pcount* has one leading zero byte. We can see that only suffix pointer contains a value, this is because the last three bits of the bitmask contain the value 010. The first bit indicates that *left* is null, next bit is 1 and hence *suffix* is not *null*, since the last bit is 0 *right* is *null*.

	Value	Bitset representation
bitmask		11001010
Δ*item*	42	00101010
pcount	78231	00000001 00110001 10010111
left	*null*	
suffix	32525146	00000000 00000001 11110000 01001011 01011010
right	*null*	

Table 4.1: CFP-node example.

In some cases child nodes can be represented with less than five bytes (pointer size). Thus, it can be an advantage to store these in the pointer place holder of a node. These are called *embedded leaf nodes*. A node can be embedded in the pointer place holder if $\Delta item < 256$ and $pcount < 16{,}777{,}216$. In this case only four bytes are needed, one in order to store $\Delta item$ and three to store *pcount*. The first byte, in the pointer place holder, is set to 255 in order to mark this as an embedded leaf node.

To further compress the CFP-tree something called *chain nodes* are used, these type of nodes were initially proposed for Patricia tries [25]. The idea is to store multiple nodes in one single node. Nodes can be stored in chain nodes if the $\Delta item < 256$, $pcount = 0$, and all pointers except the suffix pointer are *null*. The compression mask for *pcount* is set to 111, in order to specify that this is a chain node. The rest of the bits, i.e. the first two and last three bits, are used in order to specify how many nodes the chain node contains. Let's say we have five elements with the required characteristics, the $\Delta item$ values are as follows: 1, 4, 2, 2, and 1. Then the chain node can be represented in the following way:

00111101 00000001 00000100 00000010 00000010 00000001 <suffix pointer>

The last five bytes of a chain node always represent a suffix pointer.

4.1.3 CFP-Array

The CFP-array is an optimized data structure for the mining phase in CFP-growth. Two depth first traversals of the CFP-tree are needed in order to construct the CFP-array. The first traversal is needed in order to determine the size of the array, in the second pass the values in the array are set. Node elements in the CFP-array are ordered with an increasing item value. Each set of elements with equal items are grouped together in the array, these groupings are called sub-arrays. A separate array is created that contains a reference to the first element in each sub-array; this array can be seen as a header table in a regular FP-tree. An element in the array consists of $\Delta item$, Δpos, and *count*. The $\Delta item$ value is the same as in the CFP-tree. However the *pcount* value is not used, instead *count* is used. This value is the same value as in a regular FP-tree. The Δpos item contains information that tells where its parent is located in the array. In order to understand what this value represents, *local position* needs to be defined. This is the position in the sub-array it is contained in, where the first element in a sub-array has position zero. Hence, the Δpos

value is the difference of the local position of the given item and its parent. To find a parent of a specific node, both the Δpos and the $\Delta item$ value must be known. With help of the $\Delta item$ value it is possible to determine in which sub-array a given node is contained in. Figure 4.3 shows an example of a conversion from a CFP-tree to a CFP-array.

Variable byte encoding is used in order to minimize the memory usage of the array. This means that unnecessary bytes, i.e. leading zero bytes, are not stored for values in the CFP-array. By sacrificing the first bit in each byte it is possible to do this type of compression without a bitmask. The first bit tells if there are more bytes ahead after the given byte. If the first bit is 1, then more bytes are ahead. If, on the other hand, the first bit is 0, then this means that this is the last byte. This is shown in the example below:

1416572 as 32-bit integer:
00000000 00010101 10011101 01111100

Using variable byte encoding:
11010110 10111010 01111100

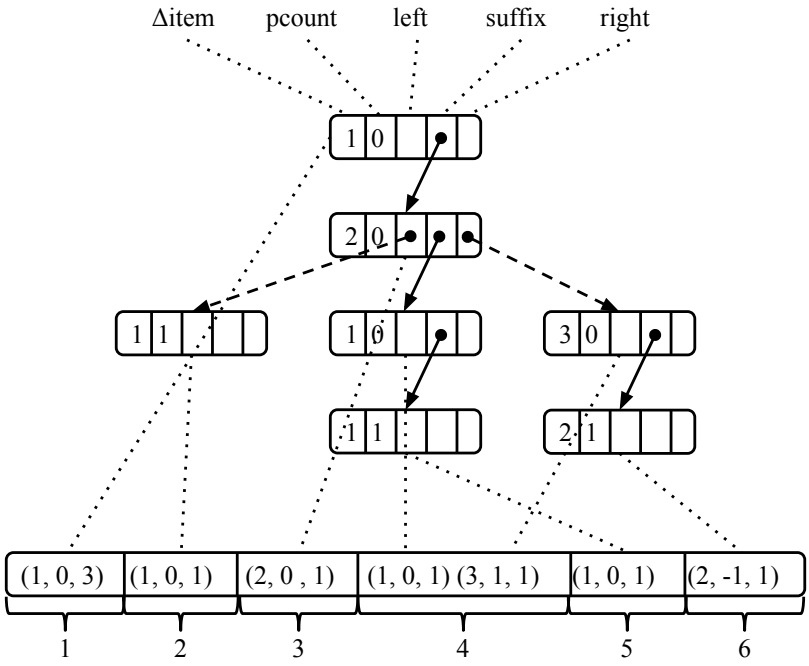


Figure 4.3: CFP-tree to CFP-array conversion. At the bottom of the tree a converted CFP-array is shown. The values in the parentheses are (Δitem, Δpos, count).

The mining procedure starts immediately after the CFP-tree is converted. This is done in a similar fashion as in a regular FP-tree.

4.1.4 Parallel CFP-Growth

In [5] a parallelization method is proposed for creating an FP-tree in shared memory. This is done by partitioning the tree, where each thread is responsible for their own set of partitions. For example, let's say we have four threads available and are using the three most frequent items for partitioning. The three most frequent items in our example are 1, 2, and 3. Load balancing can be accomplished by creating the power set of these items, and divide the resulting set into four main sets. These are then distributed for each thread. Each thread is responsible to process each transaction starting with the items given in its own distributed set of itemsets. Table 4.2 shows the whole partitioning scheme.

Thread	Partitioning scheme	Allowed itemsets	Disallowed itemsets
1	000	{}	{1, 2, 3}
	001	{3}	{1, 2}
2	010	{2}	{1, 3}
	011	{2, 3}	{1}
3	100	{1}	{2, 3}
	101	{1, 3}	{2}
4	110	{1, 2}	{3}
	111	{1, 2, 3}	{}

Table 4.2: Tree build partitioning. Shows which transactions each thread is responsible for. A thread is responsible for a transaction that contains allowed itemsets, and doesn't contain disallowed itemsets.

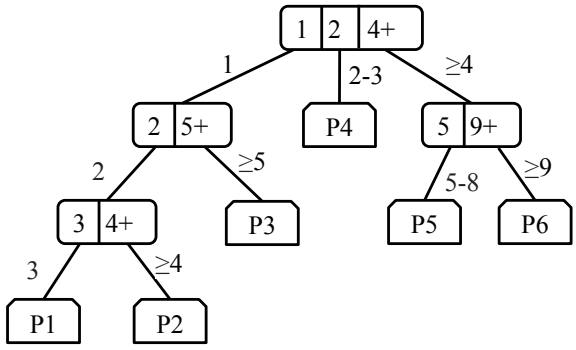


Figure 4.4: Range partitioning with six partitions.

In [26] some improvements are presented for this partitioning algorithm. By strictly using the method shown above, load imbalances can occur for some datasets. If, for example, the itemset {1, 2, 3} occurs in 70% of all transactions, thread 4 will be busy long after thread 1, 2, and 3 have finished building their own parts of the tree. This can be solved by applying tree specific partitioning, where partitioning paths are longer in frequently

occurring parts in the tree. In order to get a more fine grained partitioning it is possible to use range partitioning. Instead of simply checking if a item contains or doesn't contain a given item, each node in the partitioning tree can instead check if a range of items is present in a transaction in order to determine its responsible thread. This type of partitioning is shown in Figure 4.4.

Parallel mining is not as complicated as the parallel tree-building phase. The mining procedure can be parallelized by letting each thread be responsible for their own set of items. These items can be mined independently without any form for synchronization. By using dynamic scheduling and mining the most frequent items first, the performance will be improved.

4.2 Parallel Eclat

Two different parallel Eclat algorithms are presented in this section. A parallel version of Eclat is given in Section 4.2.1, this algorithm is good for a low number of threads. In Section 4.2.2 an Eclat algorithm is presented that is suitable for large quantities of threads.

4.2.1 ParEclat

ParEclat [37] is a parallel Eclat implementation. The core of the algorithm is equal to sequential Eclat, however there are some minor differences. Firstly the algorithm starts off by finding frequent items and frequent 2-itemsets. Then the dataset is converted into vertical layout in parallel. This is done by converting the dataset into equal partitions, and assigning each thread for each partition. After the preprocessing step the mining procedure can start. In order to distribute the load between threads during mining, each equivalence class get its own weight assigned based on cardinality. A greedy based load distribution method is used based on this weight. The mining procedure is run without any form for synchronization, i.e. the threads are run independently of each other. At last the results are combined, and the algorithm is finished.

4.2.2 mcElcat

In [28] a highly scalable Eclat algorithm was presented. The main differences between this algorithm and ParEclat are the internal representation of datasets, the memory management, and the parallel mining procedure.

The TID-lists are represented as bit arrays. Each bit represents a transaction, where the transaction can be identified based on the bit position. If a bit is set to 1 it means that the item is contained in the transaction, if it is zero then it is not in the given transaction. Every item has its own respective TID-bitmap, where the number of bits is equal to the number of transactions that contain at least one frequent item. This makes it easy to calculate frequency of the generated candidates. For example, let's say we have the itemsets {1, 2,

3} and {1, 2, 4}. The respective TID-bitmaps are as follows: 100110011 and 101101010. If we combine these two itemsets we get the following candidate: {1, 2, 3, 4}. To calculate the support count we need to intersect the bitmaps and count the number of one bits in the resulting bitmap. As seen under the support count in this case is three.

$$100110011 \wedge 101101010 = 100100010$$
$$\textit{countOneBits(100100010)} = 3$$

In mcEclat each thread uses its own memory manager. This memory manager works as a stack, which can grow and shrink in memory. Large chunks of memory are allocated in order to reduce expensive heap allocation calls. This results in cheap memory allocation through the memory manager. Since Eclat uses a depth-first type of candidate generation, a memory manager that works as a stack is a good solution. The stack can grow whenever a recursive call is performed, and shrink whenever the recursive call has finished.

There are multiple ways of parallelizing the mining procedure. The simplest one is independent class mining, where each thread gets assigned its own equivalence class which it can mine independently. In this case no thread communication is needed. Scalability can be good if thread count is low, and the number of generated frequent itemsets by each equivalence class is similar. Another way of parallelizing the equivalence class mining is by using shared class mining. In this case multiple threads can mine the same equivalence classes, however thread synchronization is needed. This communication cost between threads can negatively impact the scalability of the algorithm. In mcEclat a combination of these two methods is used. Threads are partitioned into groups, where each group gets assigned its own classes. During the first recursive calls each thread in the group performs shared class mining, in deeper recursive calls each thread employs independent class mining. This results in good scalability for a large number of threads.

4.3 ShaFEM

ShaFEM [35] is a dynamic parallel FIM method, that is capable of handling both sparse and dense datasets in an efficient manner. It is based on the FP-growth algorithm, and uses almost the same data structure. The main difference lies in the mining procedure, where an adaptive method is used. Here the algorithm switches between Eclat and FP-growth type of mining, based on the density of the paths that are being mined. This will be further explained in Section 4.3.2. However, first of all this section starts off by explaining the preprocessing steps in Section 4.3.1. Here the differences between the preprocessing step in FP-growth and ShaFEM are made clear.

4.3.1 Preprocessing

Parallel preprocessing is performed in order to construct the final data structures that are going to be used for the mining. This is accomplished by letting each thread create its own FP-tree, from its own partition in the dataset. These smaller FP-trees are then later merged

together into a XFP-tree that is stored in shared memory. In order to accomplish this, the mining procedure is divided into three main steps. They are as follows:

1. Count the number of occurrences for each item and identify which one is frequent.
2. Each thread constructs its own local FP-tree for its given partition.
3. Merge the local FP-trees into an XFP-tree.

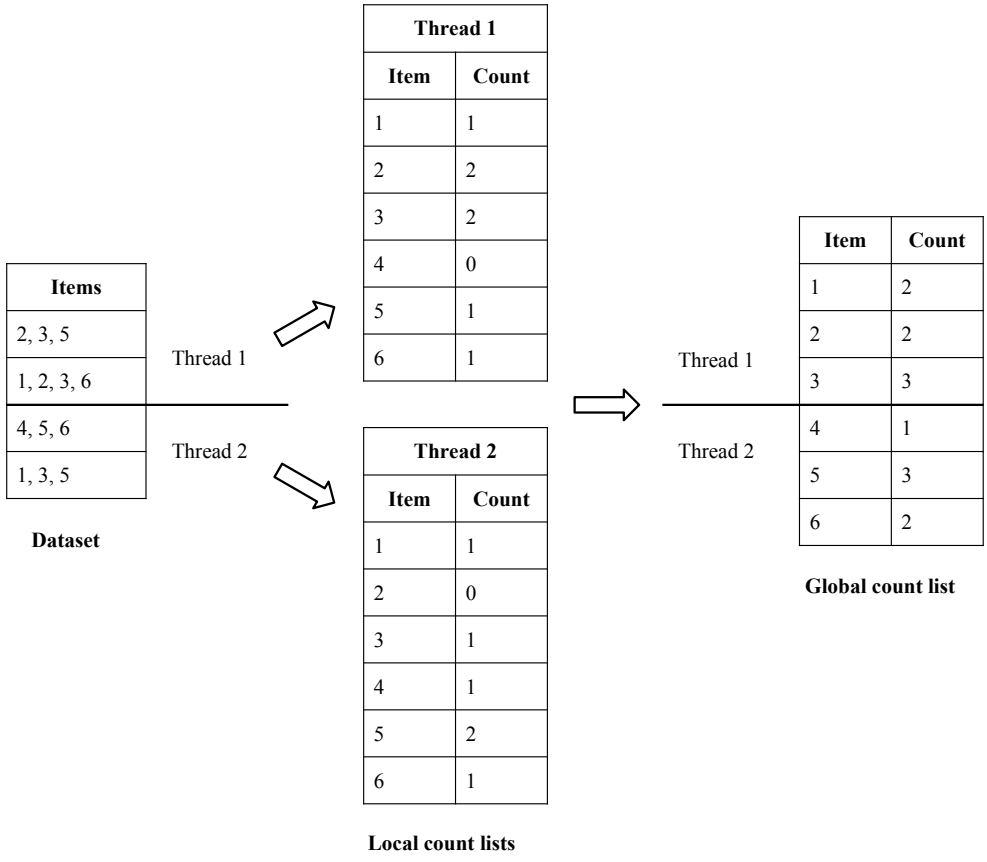


Figure 4.5: Creating local and global count lists in parallel.

All these steps are run in parallel. In the first step the dataset is divided into equal sized partitions, where each thread is responsible to create the local count list for its given partition. The partition count is equal to the number of threads used. Each thread counts the number of items found in its own partition in the dataset, and stores these values in its local count list. After all the local count lists are created, the creation of the global count list can start. The array containing the global counts is divided into equal sized partitions. Each

thread performs a summation of the counts in the local count lists, for its given partition in the global count list. No synchronization is needed when creating the local or global count lists, however all local count lists need to be created first in order to create the global count list. These steps are visualized in Figure 4.5. Lastly all frequent items are identified and sorted in support descending order.

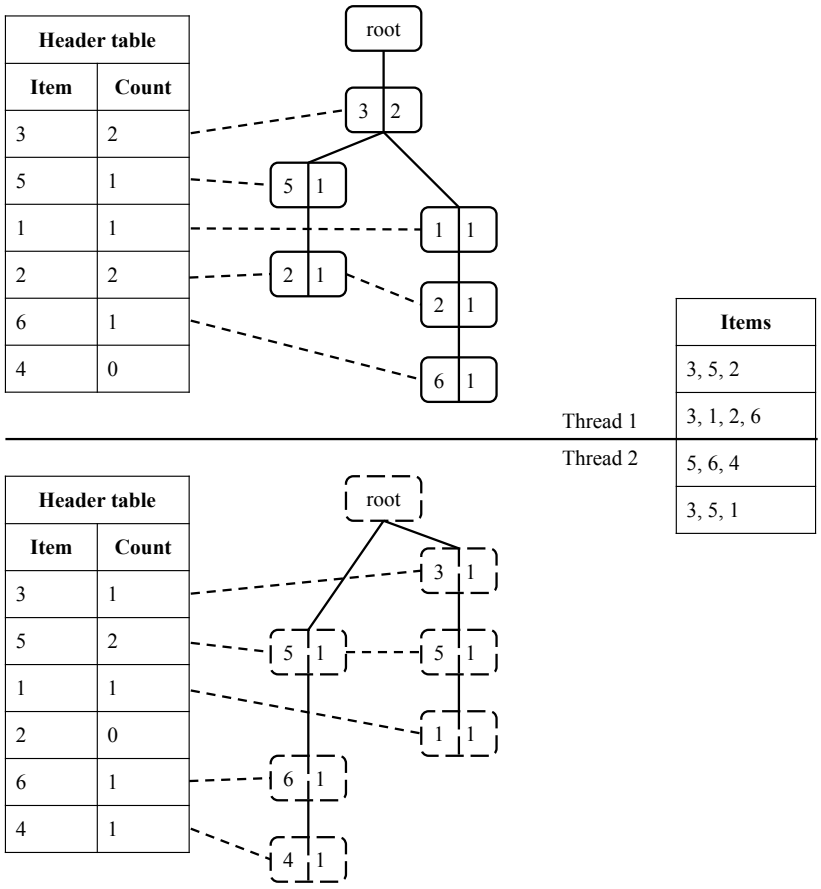


Figure 4.6: Parallel local FP-tree creation.

When the count lists are created, and frequent items are identified, the local FP-trees can be created. Each thread creates its own header table, by using the local count lists created in the previous step. These header tables are used by the threads in order to create their own local FP-tree. The trees are build by using the transactions contained in the threads given partition in the partitioned dataset. This process is illustrated in Figure 4.6. An advantage of using this approach is that no synchronization is needed, and hence better scalability is accomplished. However, path duplication can occur which might cause higher memory consumption, as well as poorer performance of mining frequent itemsets.

The last step of the preprocessing consists of merging together the created FP-trees into an XFP-tree. Firstly a global header table is created with the same count values as in the global count list. The structure of the local FP-trees are not changed, the only difference is that the linked lists from the header tables are connected together. This procedure is not time consuming, since it mostly consists of expanding the linked lists. On top of that this process is done in parallel; each thread can independently join the linked lists of its own partition of items. The final XFP-tree of the local FP-trees from Figure 4.6 is shown in Figure 4.7.

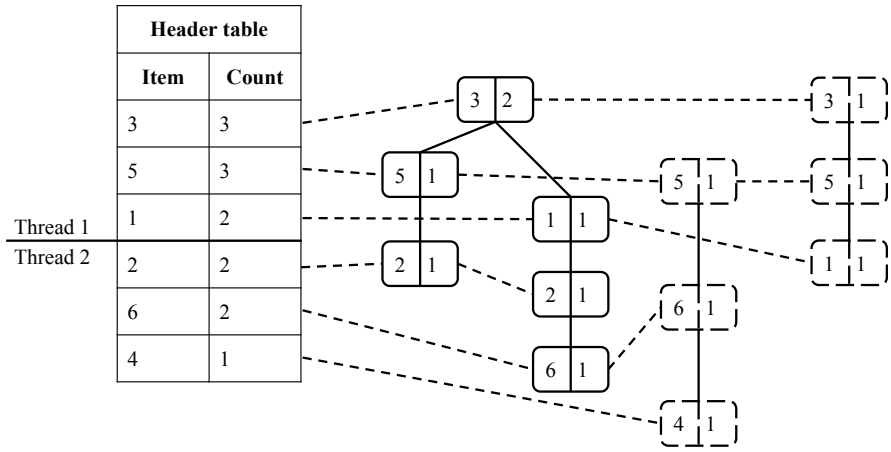


Figure 4.7: Local FP-trees from Figure 4.6 converted into an XFP-tree.

4.3.2 Adaptive mining

The mining procedure in ShaFEM is an adaptive method, where the mining algorithm changes based on the data characteristics. Eclat performs often well for dense datasets, while FP-growth performs well for sparse data. This is exploited in the mining method of ShaFEM by switching between these two methods. In order to determine which mining method to use a variable K is used. Each thread has its own K value which is updated during runtime. The process is parallelized by letting each thread independently finding frequent itemsets for a set of frequent items. The items are dynamically assigned using dynamic scheduling in order to get a good workload distribution. This is illustrated in the function *ParallelMinePattern*, the pseducode for the function is shown in Figure 4.8[1].

The function *MineFPTree*, defined in Figure 4.9[1], is very similar to a regular FP-growth mining algorithm. Paths are recursively mined. However if the size of the conditional pattern base exceeds the value K, an Eclat type of mining is used (i.e. *MineBitVector)*.

[1]The pseudocode is created according to the pseudocode given in [35].

Algorithm 2: ParallelMinePattern

Data: XFP-tree XT, minsup
Result: All frequent itemsets.

```
1  Function ParallelMinePattern(XFP-tree XT, minsup)
2      K_i = 0;
3      Parallel Self-Scheduled for j = 1 to number of items in XT do
4          α = j^th item in XT;
5          Output α;
6          Size = the size of α's conditional pattern base;
7          UpdateK(NumNewPatterns, Size);
8          if Size > K_i then
9              Construct α's private conditional FP-tree T;
10             MineFPTree(T, β, minsup);
11         end
12         else
13             Construct α's private bit vectors V and w;
14             MineBitVector(V, w, β, minsup);
15         end
16     end
```

Figure 4.8: ParallelMinePattern procedure

Algorithm 3: MineFPTree

Data: FP-tree T, suffix, minsup
Result: Frequent itemsets.

```
1  Function MineFPTree(FP-tree T, suffix, minsup)
2      if T contains a single path P then
3          for each combination x of the items in P do
4              Output β = x ∪ suffix;
5              UpdateK(NumNewPatterns, Size);
6          end
7      end
8      else
9          for each item α in the header table of FP-Tree T do
10             Output β = α ∪ suffix;
11             Size = the size of α's conditional pattern base;
12             UpdateK(NumNewPatterns, Size);
13             if Size > K_i then
14                 Construct α's conditional FP-tree T';
15                 MineFPTree(T', β, minsup);
16             end
17             else
18                 Construct α's private bit vectors V and w;
19                 MineBitVector(V, w, β, minsup);
20             end
21         end
22     end
```

Figure 4.9: MineFPTree procedure

MineBitVector is defined in Figure 4.10[1]. It uses vertical data format where a bit vector is used in order to identify in which transactions a given item is contained in. This mining method is similar to Eclat, however there are some differences. The size of the bit-vectors is equal to the number of sets in the conditional pattern base. Let's say that the pattern base consists of the following sets: {1, 2, 3}, {2, 3}, and {1, 3}. Then the bit-vectors will have the following form: **1**: 101, **2**: 110, **3**: 111. A weight vector is used in order to calculate the support value of an itemset. For example if the previous stated sets have the weights **w** = [1 2 4], then the support value of set {1, 3} is (**1** ∧ **3**) · [1 2 4] = ([1 0 1] ∧ [1 1 1]) · [1 2 4] = 1 + 4 = 5 [34].

In order to know when to switch between *MineFPTree* and *MineBitVector*, an update function is used. Each thread has its own *K* value that is updated regularly throughout the mining procedure. The algorithm used for updating this value is shown in Figure 4.11. The input value $Num_{NewPatterns}$ indicates the number of new frequent patterns in the conditional pattern base. While the *Size* parameter tells how many patterns there are in the conditional pattern base. *Step* is a constant that was set to 32 in [35], this is because this value matches most systems word and cache sizes.

Algorithm 4: MineBitVector

Data: vectors V, vec. w, suffix, minsup
Result: Frequent itemsets.

```
1  Function MineBitVector(vectors V, vec. w, suffix, minsup)
2      Sort V in support-descending order of their items;
3      for each vector v_k in V do
4          Output β = item of v_k ∪ suffix;
5          for each vector v_j in V with j < k do
6              u_j = v_k AND v_j;
7              sup_j = support of u_j computed using w;
8              if sup_j ≥ minsup then
9                  U = U + u_j;
10             end
11         end
12         if all u_j in U are identical to v_k then
13             for each combination x of the items in U do
14                 Ouput β' = x ∪ β;
15             end
16         end
17         else if U is not empty then
18             MineBitVector(U, w, β, minsup);
19         end
20     end
```

Figure 4.10: MineBitVector procedure

Algorithm 5: UpdateK

Data: $Num_{NewPatterns}$, Size
Result: Updated K.

```
1  Function UpdateK(Num_NewPatterns, Size)
2      if first time called then
           /* N is the number K_i variables.                    */
3          Create array X with N elements;
4          Initialize all elements of array X to zero;
5      end
6      for j = 0 to N − 1 do
7          if Size > j ∗ Step then
8              X[j] = X[j] + Num_NewPatterns;
9          end
10         else
11             Exit loop;
12         end
13     end
14     K_i = 0;
15     for j = N − 1 to 1 do
16         if X[j − 1] ≥ 2 ∗ X[j] then
17             K_i = (j + 1) ∗ Step;
18             Exit loop;
19         end
20     end
```

Figure 4.11: UpdateK procedure

4.4 Novel CFP-Growth and ShaFEM hybrid

ShaFEM and CFP-Growth have their strengths and weaknesses. The dynamic adaptive mining method of ShaFEM makes it a good choice for mining almost all types of datasets. However, the memory consumption and potential path duplication due to its preprocessing method shows clearly the weakness of this algorithm. The memory usage of CFP-Growth is significantly smaller than ShaFEM, which makes it perform better on large datasets because of better cache utilization. This section presents a novel hybrid algorithm of these two methods that is able to preserve their strengths.

4.4.1 Preprocessing

The preprocessing of the algorithm is equal to the preprocessing stage used in CFP-Growth. This means that the algorithm starts by creating a CFP-tree followed by CFP-array construction. A presentation of these data structures was given in Section 4.1.2 and 4.1.3.

Since these data structures are focusing on memory efficiency, more data will be available in cache. This will result in better cache utilization, and hence better performance.

4.4.2 Mining

One of the strengths of ShaFEM is the dynamic mining procedure. As explained in Section 4.3.2, this procedure is able to adapt its mining process based on the data that is currently being mined. The novel hybrid algorithm uses this mining procedure. With the combination of the data structures from CFP-Growth and the mining process from ShaFEM, this algorithm is able to utilize cache better than ShaFEM and is able to mine data more efficiently than CFP-Growth.

The main difference between the mining procedure in ShaFEM and the hybrid algorithm is the decompression of data. ShaFEM used an XFP-tree during its mining procedure, while the hybrid algorithm uses a CFP-array. The difference is that the pattern base must be extracted from the CFP-array instead of the XFP-tree.

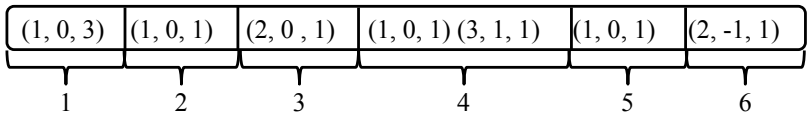


Figure 4.12: A CFP-array. The values in the parentheses are (Δitem, Δpos, count).

In Figure 4.12 we see a CFP-arrray. Let's say that we want to extract the pattern base of item 6. From the figure we see that it consists of the following element:

(Δitem: 2, Δpos: 0, count: 1).

From the explanation of the structure of CFP-array from Section 4.1.3, we are able to decompress the pattern base from these elements. The index position is known and represents the leaf item of the pattern base, i.e. 6 in our case. With this information combined with the Δitem value we are able to get the parent item from the given element.

Parent of (Δitem: 1) is $6 - 2 = 4$.

The position of the parent element, in the index array, can be found by the second value in the CFP-array element combined with the index position. With the definition of Δpos from Section 4.1.3, we can calculate the local position of the parent.

$local\_position\_parent = local\_position\_this - \Delta\text{pos} = 0 - (-1) = 1$

Now we can find the parent element in the CFP-array which is the following:

(Δitem: 3, Δpos: 1, count: 1)

We can follow the same procedure in order to get the parent of this element:

Parent of (Δitem: 3) is $4 - 3 = 1$
$local_position_parent = local_position_this - \Delta$pos$= 1 - 1 = 0$
Parent element: (Δitem: 1, Δpos: 0, count: 3).

Since Δitem is equal to index item we know that this element has no parent. Hence, the conditional pattern base consists of the following items: 1, 4, and 6.

The rest of the mining procedure follows the same pattern as in ShaFEM.

Chapter 5

Frequent inter-transaction itemset mining algorithms

This chapter focuses on inter-transaction itemset mining. Some basic algorithms for this type of problem are explained in Section 5.1. A well known algorithm for this task, i.e. FITI is presented in Section 5.2. One of the newest algorithms, as of this writing, Index-BTFITI is presented in Section 5.3. Lastly, the novel parallel inter-transaction itemset mining method is presented in Section 5.4.

5.1 From intra to inter-transaction mining

Unlike the previously presented methods, this and the following sections present algorithms for *inter*-transaction mining instead of *intra*-transaction itemset mining. The concept of inter-transaction itemset mining was introduced in Section 3.1.4. This section presents adapted Apriori and FP-Growth algorithms for this type of problem statement.

5.1.1 Apriori for inter-transaction mining

E-Apriori and EH-Apriori [18] were the first algorithms created for inter-transaction association rule mining [6]. These are Apriori algorithms extended for inter-transaction mining. Instead of using the regular transactions during candidate generation, megatransactions are used instead as a basis during this procedure. Recall from Section 3.1.4 that a megatransaction was defined as a transaction that contains all items within a sliding window. The Apriori algorithm is dependent on lexicographic order in order to generate candidates, and hence order of extended items needs to be defined. If we have the extended items $e_i(d_i)$ and $e_j(d_j)$, then $e_i(d_i) < e_j(d_j)$ if one of the following conditions is satisfied:

1. $d_i < d_j$

2. $d_i = d_j \wedge e_i < e_j$

By using this definition, when generating candidates of megatransactions, the Apriori algorithm can execute inter-transaction mining in a regular manner. E-Apriori or *Extended*-Apriori uses the above mentioned method. The difference between the above mentioned algorithm and EH-Apriori, is that EH-Apriori uses a similar hashing method proposed in [24]. This is done in order to filter unnecessary candidate 2-itemsets.

5.1.2 FP-Growth for inter-transaction mining

An adaptation of the FP-Growth algorithm for inter-transaction itemset mining was first presented in [20], where the *Extended Frequent Pattern Tree* (EFP-Tree) was introduced. Here the article authors performed empirical tests on the algorithm with real data from a smart home. There have also been created other adaptations of FP-Growth, e.g. in [6] the algorithm InterTARM was presented. This algorithm focuses on inter-transaction mining of stock market data.

Similarly as to E/EH-Apriori, FP-Growth adaptations need to convert its transaction database into a set of megatransactions. After this database transformation, all frequent megatranscation itemsets are inserted into an FP-Tree. The order of extended item insertion is a combination of order of occurrence[1] and frequency. The mining procedure works in a similar fashion as to a regular FP-Growth algorithm.

5.2 FITI: *First Intra Then Inter*

FITI, *First Intra Then Inter*, was introduced in paper [32], and focuses on inter-transaction mining. The algorithm consists of the following three phases:

1. Mine frequent intra-transaction itemsets and store them in a custom made data-structure.
2. Transform the database in order to adapt it for the mining procedure.
3. Mine frequent inter-transaction itemsets.

An in-depth description of these phases is given in this section.

5.2.1 Frequent-itemsets linked table

FITI exploits the property that all subsets of a frequent itemset are also frequent itemsets. This means that all frequent *inter*-transaction itemsets are combinations of frequent *intra*-transaction itemsets. Hence, the first phase of FITI consists of finding frequent intra-transaction itemsts. Recall from Section 3.1.4 that intra-transaction itemsets were defined

[1] This can be seen as lexiocographic order defined in Section 5.1.1.

as itemsets within one transaction. Thus the first phase consists of the same problem as in a regular FIM method, i.e. finding the frequent itemsets. In [32] the Apriori algorithm was used for this task, however other FIM methods are applicable as well.

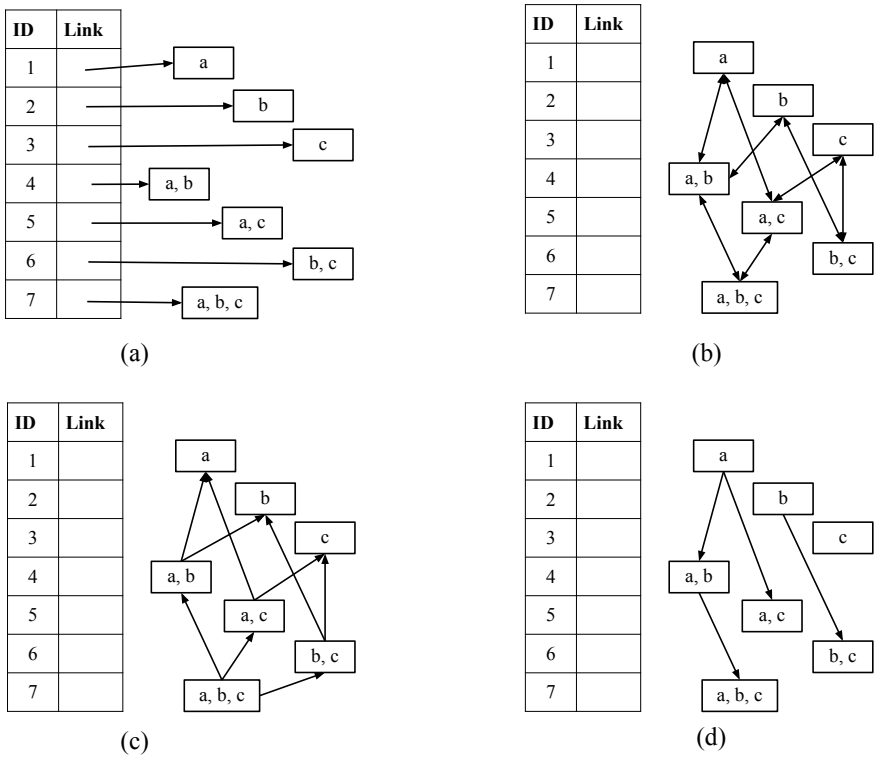


Figure 5.1: Frequent-itemsets linked table. (a) Lookup links, (b) generator and extension links, (c) subset links, and (d) descendant links.

The frequent itemsets found are then stored in a custom data structure called *Frequent-Itemsets Linked Table* or FILT. This data-structure consists of an itemset hash table, where each row points to a node containing a frequent itemset. The nodes contain again multiple different links to other nodes. A description of the different types of links, in the FILT data-structure, is given below:

- **Lookup links**: Each row in the itemset hash table contains a lookup link to a node that contains a frequent intra-transaction itemset. An example is given in Figure 5.1 (a).
- **Generator and extension links**: A node, with a k-itemset, contains links to the two $(k-1)$-itemsets that were combined in order to create the given node in the Apriori algorithm. These links are bidirectional. This is shown in Figure 5.1 (b).

- **Subset links**: A node, with a k-itemset, contains links to all $(k-1)$-itemsets that are a subset of the itemset contained in the given node. This is illustrated in Figure 5.1 (c).
- **Descendant links**: A node, with a k-itemset, contains links to all $(k+1)$-itemsets which it shares its prefix with as shown in Figure 5.1 (d).

5.2.2 Database transformation

The next phase consists of database transformation, where the database is transformed into multiple encoded tables. These tables are called *Frequent-Itemset Tables* or *FIT tables*. There are max_k number of FIT tables, where max_k represents the size of the largest frequent intra-transaction itemset. Each table contains data for each different intra-transaction size, i.e. there is a table for F_1, F_2, ... F_{max_k}. A FIT table consists of two columns, one for the dimensional attribute, and another that contains the IDs for the k-itemsets found in the given transaction. The IDs represents the row in the itemset hash table, i.e. in the FILT data-structure. An example of how the FIT tables are constructed is given in Table 5.2. Here we see the database given in Table 5.1 being transformed into three FIT tables. The FILT table from Figure 5.1 is used.

d	**E**
1	a, b, c
3	c, d
4	a
6	a, b, e

Table 5.1: A dataset with four transactions. Column **d** contains the dimensional value, column **E** contains the transactions.

F_1		F_2		F_3	
d_i	ID_{set_i}	d_i	ID_{set_i}	d_i	ID_{set_i}
1	1, 2, 3	1	4, 5, 6	1	7
3	3	3		3	
4	1	4		4	
6	1, 2	6	4	6	

Table 5.2: FIT table example. Database given in Table 5.1 transformed to three FIT tables. The ID_{set_i} values are given from the FILT table in Figure 5.1.

5.2.3 Finding frequent inter-transaction itemsets

The main bottleneck of the algorithm is the search of frequent inter-transaction itemsets. It finds frequent itemsets by joining already found frequent intra-transaction itemsets. How-

ever, in order to do so an inter-transaction itemset representation needs to be defined. Let's say we have the following set: $A_i = \{e_j | 1 \leq j \leq u, e_j(i) \in F\}$, where $0 \leq i \leq (w-1)$ and F is an inter-transaction itemset. The inter-transaction itemset representation is then given by $I = \{I_0, ..., I_{w-1}\}$, where I_i is the ID of A_i in the FILT data structure. If A_i is empty, then $I_i = 0$. In other words, the representation consists of a set with an item for each dimensional property in the sliding window. Each item has an ID, from the FILT table, that represents the item that occurred with the given dimensional property. If there was no set with the property, then the value is set to 0. For example, let's say we have the inter-transaction itemset $F = \{a(0), b(0), c(0), a(2), b(3), c(3)\}$, and the maxspan is set to four. Then the ID encoding will be $I = \{7, 0, 1, 6\}$, the IDs are given from Figure 5.1.

The mining procedure is similar to Apriori, where it first generates inter-transaction k-itemsets. Then it scans the created FIT-tables in order to update the count value of the candidate. The k value is increased iteratively until all frequent inter-transaction k-itemsets are found. Firstly the algorithm needs to generate frequent inter-transaction two-itemsets. FITI uses a hashing approach introduced in [24], where the technique is used in order to prune generated candidates. The hash function that indicates the bucket number of a candidate is given by the following formula:

$$B = ((I_0 \times w + m) \times N_1 + I_m) \; mod \; Size, m \neq 0 \wedge I_m \neq 0 \tag{5.1}$$

I_i is an item in the candidate given by the ID encoding explained earlier. N_1 is the number of frequent one-itemsets, and $Size$ is simply the size of the hash table. Now each inter-transaction two-itemset can be represented by three values, i.e. I_0, I_m and m.

In order to generate candidates with $k > 2$, two operations are used. These are *intra-transaction join* and *cross-transaction join*, an explanation of these two methods is given below.

- **Intra-transaction join**: If we have two frequent intra-transaction itemsets $I = \{I_0, ..., I_{w-1}\}$ and $J = \{J_0, ..., J_{w-1}\}$, a join can be performed if two criteria are fulfilled. There must exist a p, such that I_p and J_p are generators of a frequent intra-transaction itemset. The second condition is that the two sets, I and J, contain the same items in all positions except for position p. For example if $I = \{1, 3, 0, 4\}$ and $J = \{1, 3, 0, 5\}$ an intra-transaction join will result in $K = \{1, 3, 0, 7\}$, by using Figure 5.1.
- **Cross-transaction join**: This join requires multiple criteria to be satisfied:
  1. $\exists p, I_p \neq 0 \wedge J_p = 0$
  2. $\exists (q \neq p), I_q = 0 \wedge J_q \neq 0$
  3. $\forall r, r \neq p \wedge r \neq q \wedge I_r = J_r$
  4. I_p and J_q are last subwindows that are not zero in I and J respectively.
  5. The subwindows in I and J contain only one-itemsets or are empty.

For example if we have the following itemsets: $I = \{1, 0, 2, 0\}$ and $J = \{1, 0, 0, 3\}$ that satisfy the above criteria. A cross-transaction join will result in the following $K = \{1, 0, 2, 3\}$.

By using the above join functions all frequent inter-transaction candidates can be created. In order to determine the frequency of the generated candidates, the FIT tables are used. This works similarly to the Apriori algorithm.

5.3 Index-BTFITI

This section presents *Index-BTFITI* [14], which uses a compressed bit table and an index array in order to mine frequent inter-transaction itemsets. In [14] tests where run in order to compare this algorithm with ITP-miner [16] and PITP-miner [36]. These tests concluded that Index-BTFITI had better performance for the given test data and configurations. In [16] it was shown that ITP-miner had an order of magnitude better performance than FITI [32]. Hence, we can safely conclude that Index-BTFITI also outperforms this algorithm as well, since the configurations for the tests were similar to those conducted in [16].

5.3.1 Preprocessing

During the preprocessing stage two important data structures needs to be created before the mining procedure can start. This includes a BitTable and an index array [31]. The BitTable consists of multiple bit-vectors where each vector indicates if the item is present in a given transaction. The position in the bit vector represents the transaction, while the value denotes its presence. If the value is one it indicates that the item is in the given transaction, otherwise it is zero. An example of a BitTable is shown in Table 5.4, where the transaction database in Table 5.3 is taken into consideration and maxspan is set to 1.

d	**E**	**Megatransactions** (maxspan=1)			
1	a, b	$(a(0), b(0),$			
2	a	$a(1))$	$(a(0),$		
3	b, c		$b(1), c(1))$	$(b(0), c(0),$	
4	a, b, c			$a(1), b(1), c(1))$	$(a(0), b(0), c(0))$

Table 5.3: A transaction database with its corresponding megatransactions. **d** denotes the dimensional attribute and **E** represents the content of the transaction.

Transaction nr.	$a(0)$	$b(0)$	$c(0)$	$a(1)$	$b(1)$	$c(1)$
1	1	1	0	1	0	0
2	1	0	0	0	1	1
3	0	1	1	1	1	1
4	1	1	1	0	0	0

Table 5.4: The corresponding BitTable according to the transaction database given in Table 5.3.

An *index array* is an array with size equal to the number of frequent items in the database. Each element in the array consists of a tuple of the form (*item*, *subsume*), where *item* represents a unique frequent item and *subsume* the maximal frequent itemset that co-occurs with the given frequent item. The subsume can easily be calculated with help of the BitTable. In order to do so a *bitwise and* operation needs to be performed between the bit rows for the transactions that contain the frequent item. For example the subsume for the item $a(1)$ can be found with the following operation: $bitvector_{row1} \wedge bitvector_{row3} = 110100 \wedge 011111 = 010100$, the bit table from Table 5.4 is used. The resulting bit-vector, i.e. 010100, represents the itemset $(b(0), a(1))$. However, since the frequent item, i.e. $a(1)$ in our example, is already present in the index array it doesn't need to be present in the subsume. The resulting tuple in the index array for $a(1)$ is $(a(1), b(0))$.

5.3.2 Mining procedure

The mining procedure consists of mainly using the index-array to generate itemset candidates. BitTable is used in order to calculate the support count, where this is done in a similar fashion as in Eclat. An intersection between the bitsets for the items in the itemset is performed. Then all non-zero bits are added together and the result represents the support count.

The pseudocodes for the mining procedure is given in Figure 5.2, 5.3 and 5.4. *Index-BTFITI* is the main algorithm that calls *Index-BitTableFI* for each element in the index-array with dimensional attribute set to 0. *Index-BitTableFI* with *Depth-First* calculates the support and outputs the frequent inter-transaction itemsets. The pseudocodes do not cover support counting, since this process is similar to the one used in mcEclat explained in Section 4.2.2.

Algorithm 6: Index-BTFITI

Data: Index-array, BitTable, minsup.
Result: Frequent itemsets.

```
1  Function Index-BTFITI(Index-array, BitTable, minsup)
2      for each element ε_i ∈ Index-array with i = 1, 2, ..., |Index-array| do
3          if ε_i.item.dimensionalAttribute == 0 then
4              Index-BitTableFI(ε_i, BitTable, minsup);
5          end
6          else if ε_i.subsume has item with dimensional attribute == 0 then
               /* i^0 is an item with dimensional attribute == 0          */
7              for each i^0 ∈ ε_i.subsume do
8                  ε_i.item = ε_i.item ∪ i^0;
9                  ε_i.subsume = ε_i.subsume − i^0;
10                 Index-BitTableFI(ε_i, BitTable, minsup);
11                 ε_i.subsume = ε_i.subsume ∪ i^0;
12                 ε_i.item = ε_i.item − i^0;
13             end
14         end
15         else
16             delete ε_i;
17         end
18     end
```

Figure 5.2: Index-BTFITI

Algorithm 7: Index-BitTableFI

Data: ε_i, BitTable, minsup.
Result: Frequent itemsets for the given ε_i.

```
1  Function Index-BitTableFI(ε_i, BitTable, minsup)
2      Add ε_i to frequent itemsets;
3      if ε_i.subsume == ∅ then
           /* function t(itemset) returns all items that have support
              greater than or equal to itemset.                        */
4          Depth-First(ε_i.item, t(ε_i.item));
5      end
6      else
7          for each element s-item ⊆ ε_i.subsume do
8              Add ε_i.item ∪ s-item to frequent itemsets;
9          end
10         tail = t(ε_i.item) - (items in ε_i.subsume);
11         Depth-First(ε_i.item, tail);
12         for each element s-item ⊆ ε_i.subsume do
13             Depth-First(ε_i.item ∪ s-item, tail);
14         end
15     end
```

Figure 5.3: Index-BitTableFI

Algorithm 8: Depth-First

Data: itemset, tail
Result: Frequent itemsets generated from itemset and tail.

```
1  Function Depth-First(itemset, tail, minsup)
2      if tail == ∅ then
3          return;
4      end
5      for each i ∈ tail do
6          f-itemset = itemset ∪ i;
7          if support(f-itemset)≥ minsup then
8              Add f-itemset to frequent itemsets;
9              tail = tail − i;
10             Depth-First (f-itemset, tail, minsup);
11         end
12     end
```

Figure 5.4: Depth-First

5.4 Novel parallel Index-BTFITI

Two novel parallel algorithms for frequent inter-transaction itemset mining are presented in this section. This includes EP-Index-BTFITI and FGP-Index-BTFITI. As of this writing, we believe that these are the first parallel algorithms created for this type of problem statement.

The core of the algorithm is the Index-BTFITI algorithm from [14], which was described in Section 5.3. This algorithm was chosen as foundation, since it is one of the most recent algorithms and from the tests conducted in [14] it also seems to be one of the best algorithms for inter-transaction itemset mining. The best in terms of minimal mining time for high support values. In addition this algorithm also has a good parallelization potential during preprocessing and mining procedure. Data can be easily divided between threads during both operations.

5.4.1 Parallel preprocessing

As described in Section 5.3.1 the preprocessing step consists of building a BitTable followed by Index-array creation. Both of the novel parallel inter algorithms use the same preprocessing procedure. An optimization, described in the next paragraph, was created in order to minimize preprocessing and mining time.

When creating an Index-array multiple *bitwise and* operations are performed between rows

in a BitTable[2]. However, during support counting *bitwise and* operations are performed between columns in a BitTable. If a row can be seen as an array of bytes, then in order to get a column vector more CPU cycles are needed in order to extract bit values from these bytes. These intersection methods require much of the processing time and need to be highly optimized. A way of optimizing this is to create two BitTables were one BitTable is a transposition of the other. This requires more memory usage, however because of the decreased run time this can be seen as a good compromise. If available memory is limited it is possible to remove the non-transposed BitTable after the Index-array is created.

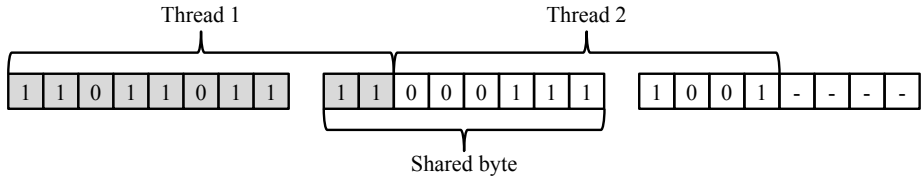


Figure 5.5: Three bytes and two threads. Since both threads have access to the middle byte, it is classified as a shared byte.

When the algorithm first reads the dataset, it is divided into multiple equal sized parts stored into multiple buffers. The buffer count equals the number of threads, where each thread is an owner of a buffer. The BitTables are created in parallel simultaneously from these buffers, and allocated memory in the buffers is also released concurrently as the values in the BitTables are assigned. Each thread is responsible to assign values to its distribution of columns or rows, depending on the BitTable transposition. This operation can be performed independently; however a lock is needed if a shared byte is accessed in the BitTable. A shared byte is a byte where multiple threads can potentially write a value to. For example, let's say that we have 20 transactions and two threads. Then a byte vector representing these transactions will consist of 20 bits. If we assume that each thread is responsible for 10 transactions each, then there is a shared byte that needs to be locked before being accessed. The shared byte is the one in the middle shown in Figure 5.5.

An illustration of how the BitTables can be constructed in parallel is shown in Figure 5.6. Here we see a dataset that is divided into four buffers, and four threads that create two BitTables from the buffers. Note that the two BitTables can be created simultaneously.

[2]Note that we are assuming here that a BitTable is build up the same way as given in Table 5.4.

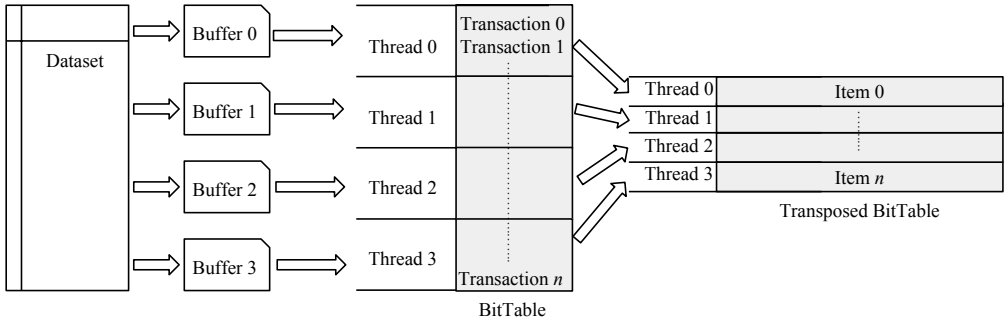


Figure 5.6: Four threads creating two BitTables, one regular (each row represents a transaction) and one transposed (each row represents an item).

The parallelization of the Index-array creation can be done by using a parallel *for loop*, where each thread is assigned an index for each iteration in the loop. The elements in the index-array can safely be created independently without any need of synchronization.

5.4.2 EP-Index-BTFITI

EP-Index-BTFITI, or *Embarrassingly Parallel*-Index-BTFITI, is a parallel Index-BTFITI algorithm that uses a parallel mining procedure with minimal synchronization. In an embarrassingly parallel problem it is possible to parallelize the task without the need of thread communication [11], hence the name EP-Index-BTFITI.

During the mining procedure all frequent items can be iterated through with a *for loop* starting with the most frequent items. This *for loop* can be parallelized by using dynamic scheduling. Although dynamic scheduling adds some overhead, it has an advantage since the time to mine a specific item varies and thus it is not beneficial to divide the work load to equal sized partitions. A thread can mine a given item independently without any need for thread communication.

5.4.3 FGP-Index-BTFITI

The problem with EP-Index-BTFITI is that there are situations where there are more threads than there are frequent items. Another problem is that some items require significantly more time to mine than others. For example, let's say that we have 100 frequent items and two threads. One of the items needs five minutes in order to be mined, all the other items require in total two minutes. The problem here with EP-Index-BTIFITI is that one thread finishes all its work in two minutes, while the other thread needs to work at least another three minutes. In this situation, no matter how many threads are available, the program is still going to need at least five minutes to finish the mining procedure. FGP-Index-BTFITI, or *Fine-Grained Parallel*-Index-BTFITI, is build in order to handle these special situations.

The scalability of EP-Index-BTFITI is good whenever the problems, described above, doesn't occur. Hence, FGP-Index-BTFITI uses the same mining procedure as EP-Index-BTFITI as long as there are still more frequent items to be mined. When a thread finishes mining an item, and there are no more frequent items available, it is marked as free. A thread that is working is able to see that an item is made available, it fetches it and assigns work for the given thread in order to alleviate its own workload.

Threads that have finished mining their set of items are marked as free and added to a list containing available threads. Whenever this list is accessed it needs to be locked with a global lock. Available threads are spinning in a *while loop* and waiting for work. In the loop they are also checking if all threads have finished working, if this is the case then they break the loop. This is illustrated in Figure 5.7. Notice that threads are not responsible to add themselves to the available threads list. This is because a master thread can potentially assign work for multiple threads at a time. Hence, in order to reduce the amount of times a lock needs to be acquired, the master thread is responsible to add its slave threads to the list of available threads.

Algorithm 9: Free threads execution scheme.

```
1 while !everythingFinished do
2     if threads[threadNr].isWorking then
3         threads[threadNr].work(threads[threadNr].args);
4         threads[threadNr].isWorking = false;
5         checkIfAllThreadsFinished();
6     end
7 end
```

Figure 5.7: Free threads execution scheme.

The most resource heavy procedure in Index-BTFITI is the recursive *Depth-First* function (see Figure 5.4). This function consists of a *for loop* that is calling itself recursively in the loop. Here lies a good potential for parallelization, this is because each iteration in this *for loop* is independent and can be mined in parallel. Therefore, for each iteration in this loop the current thread checks if there are any free threads available. This can be done by reading a global volatile boolean variable that denotes if there are any free threads. The variable doesn't need to be locked when read, because the program must handle inaccurate values and the cost of locking is expensive. If the variable indicates that there are available threads, the current thread tries to fetch a global lock. If it doesn't acquire the lock, it continues its execution in order to minimize waiting time. However, if the lock is acquired the list containing available threads is locked. The list must be checked if it actually contains any new items because the volatile bool value can be inaccurate. The current thread retrieves all threads that are needed for the loop, i.e. as many threads as there are estimated iterations left or as many threads as there are available. Estimated iterations that are remaining is item count subtracted by largest assigned index, this value can be inaccurate because a thread can be finished with its iteration before the threads are put to work. A workload is assigned for each thread, and dynamic scheduling is used. When the helper threads have finished

their execution, they are again marked as free and added to the available thread list. The activity diagram for the *Depth-First* procedure is shown in Figure 5.8.

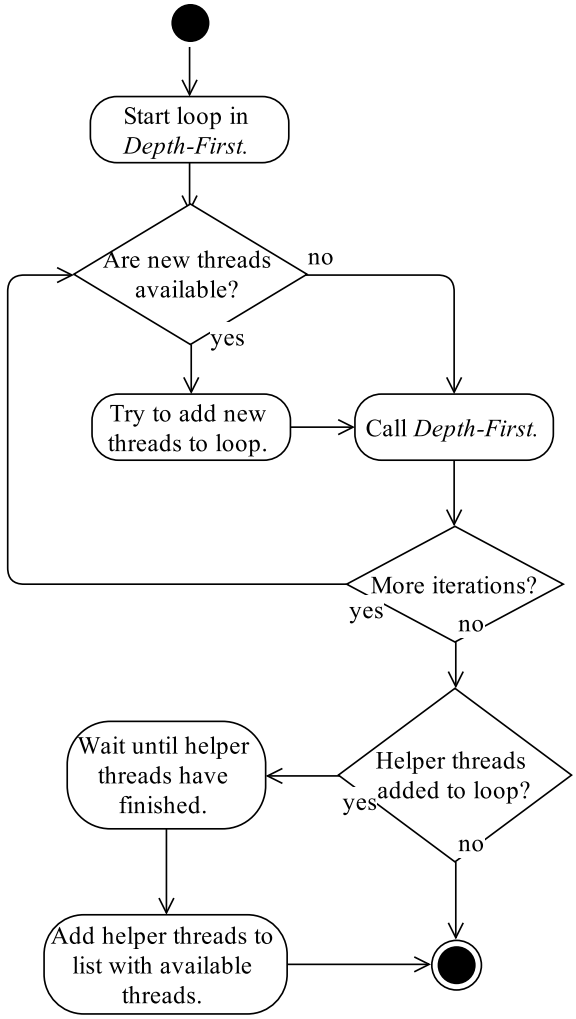


Figure 5.8: Activity diagram for the *Depth-First* procedure in FGP-Index-BTFITI.

In the *Index-BitTableFi* procedure (see Figure 5.3) there are two loops. These can also be parallelized similarly as described above. However, when we know that the workload is equal then static scheduling should be used. This is the case when generating the power set of a given subsume. Static scheduling doesn't need the overhead of locking the index in a *for loop*, hence this can be beneficial when the workload is balanced.

Chapter 6

Implementation details of existing and novel algorithms

Multiple algorithms were implemented for intra- and inter-transaction itemset mining. These were implemented in C++ with OpenMP and Pthreads as parallel programming frameworks. The goal of this chapter is to give a description of the implemented methods. Firstly the tailor made memory manager is described in Section 6.1. Then the existing state-of-the art frequent intra-transaction itemset mining methods that were implemented are presented in Section 6.2. These include the algorithms CFP-Growth, a parallel Eclat, and ShaFEM. The implementation details, of the main algorithm contributions, are presented in Section 6.3 and 6.4. In Section 6.3 the implementation details of the novel intra-transaction method, which is a combination of CFP-Growth and ShaFEM, is presented. Section 6.4 gives the implementation details of the two novel parallel methods for frequent inter-transaction itemset mining.

6.1 Custom-tailored memory manager

All of the algorithms implemented are using a custom-tailored memory manager. This memory manager was implemented based on the memory manager used in [28]. The main concept is that the memory allocation works as a stack, it has two main functions: *grow* in order to allocate memory and *shrink* in order to deallocate memory. The manager allocates large memory blocks of 128 MB whenever it runs out of memory. When the *grow* function is called it allocates data from this chunk of 128MB, an internal pointer is used in order to keep track of how much memory is currently allocated. If the *shrink* function is called the pointer decreases with the given value, and frees up memory allocated by the operating system if necessary. An example of how the memory manager works is illustrated in Figure 6.1.

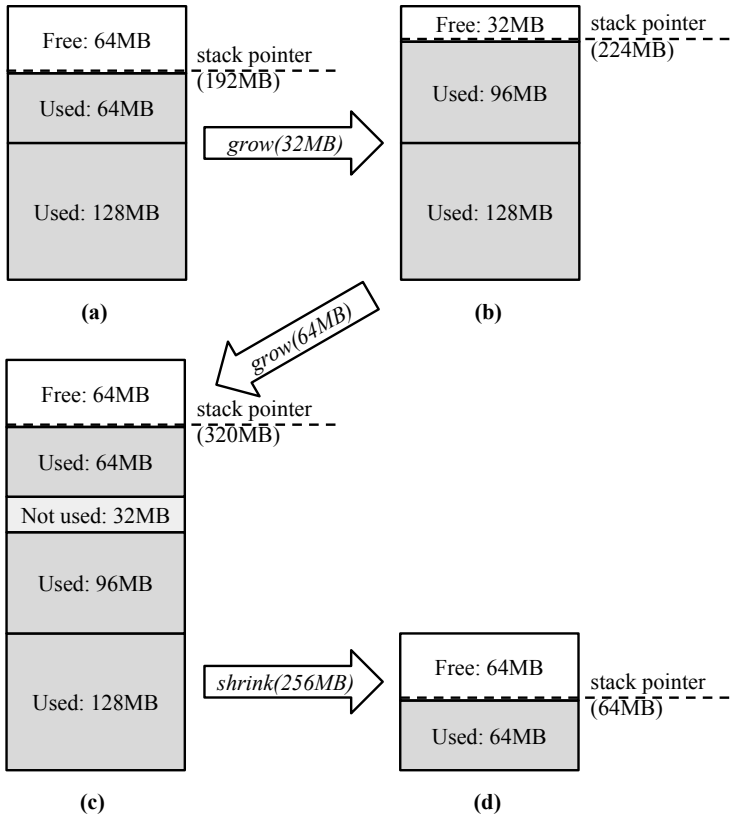


Figure 6.1: Illustration of the implemented memory manager.

In **(a)** the operating system has allocated 256 MB and the memory manager has allocated a total of 192 MB. In **(b)** *grow* has been called and allocated 32 MB, a total of 224 MB has been allocated. Later *grow* is called and tries to allocate 64 MB. The memory stack has only 32 MB available, and the requested 64 MB need to be contiguous. Hence, the operating system needs to allocate another 128 MB to the memory manager. The manager then allocates the 64 MB from the new block, and increases its stack pointer to 320 MB as seen in **(c)**. Notice that 32 MB are wasted here and are not used. In practise the memory wastage is minimal since usually only small memory allocations are performed. 128 MB is a large memory block and the memory manager rarely needs to ask the operating system for more memory, thus memory fragmentation is not needed. In **(d)** the stack frees up 256 MB, the memory manager also deallocates two memory blocks since they are not used.

The implemented CFP-Growth algorithm also uses a more specialized memory manager, similar to the one used in [27], together with the one presented above. This memory manager is used in order to create the CFP-tree, and also works as a stack. It exploits the fact that only a few memory allocation sizes are used, i.e. memory sizes between 7 and 24

bytes. Thus stacks for each memory size are used that stores free memory allocations in order to utilize the memory space efficiently. Before allocating new memory the stack for the given memory size is checked if it contains free memory. If not more memory is allocated. The block size of this memory manager is set to 1 MB, this was sufficient for most datasets.

Memory managers are used in order to minimize memory usage and minimize the need for thread synchronization. All threads usually get their own memory manager where they can allocate and deallocate memory independently. In C++ whenever *new* or *delete* is called an implicit thread synchronization is performed since the heap is shared between all the threads. Therefore, by minimizing calls to *new* and *delete* better scalability can be expected. Another advantage is that less memory is used whenever a large amount of small memory allocations are performed in the custom-tailored memory manager. Usually metadata is stored together with the allocated memory when using *new*, and hence results in more memory usage. For example let's say we run the following code:

```
for (int i = 0; i < 1000000; i++)
    new char;
```

By just looking at the code, a person could quickly come to the conclusion that the memory usage would be 1 MB because a char consists of 1 byte. However in reality the memory consumption is around 62 MB[1]. If on the other hand the following code is run:

```
new char[1000000];
```

The memory usage is approximately 1 MB[1], hence the effect is much smaller when allocating larger chunks of memory. This is exploited in the implemented memory managers by allocating large blocks of memory at a time.

6.2 Parallel intra-transaction itemset mining implementations

Some already existing state-of-the-art algorithms for intra-transaction itemset mining were implemented. These include CFP-Growth, Eclat, and ShaFEM. An overview of how these algorithms work were explained in Section 4.1, 4.2, and 4.3. Therefore a thorough algorithm description is not given here. These algorithms were implemented in C++ and parallelized with help of the OpenMP framework.

6.2.1 Parallel CFP-Growth

The implementation of CFP-Growth is based on [27], the details of this algorithm were given in Section 4.1. An overview of how this algorithm is implemented is shown in the activity diagram given in Figure 6.2.

[1] This was tested on a Linux system with the procedure given in `http://stackoverflow.com/questions/669438/how-to-get-memory-usage-at-run-time-in-c#answer-671389` (Retrieved 15.05.15).

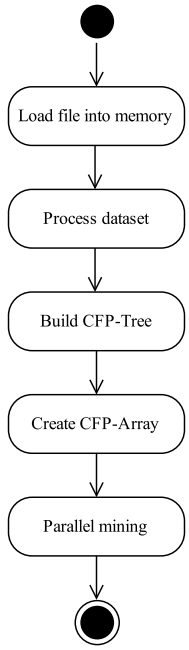


Figure 6.2: CFP-Growth activity diagram.

Firstly the whole dataset file is read into main memory. The dataset is then preprocessed, i.e. support counting is performed and item IDs are remapped by support frequency. After these steps the CFP-tree is build followed by CFP-tree creation. Lastly parallel mining is executed. This step is done by letting each thread mine its respective items independently. All threads have their own stack were they store frequent itemsets that they have found. By doing so no synchronization is needed. OpenMPs dynamic scheduling[2] is used in order to allocate items to different threads. This is done with help of a parallel *for loop* that iterates through all items. Dynamic scheduling works by letting each thread pick their own available iteration variable value for each iteration. In this way all threads are going to be busy until there are no more available iteration variable values.

6.2.2 Parallel Eclat

The Eclat algorithm was described in Section 3.3, and its parallel versions in 4.2. The implementation is mostly based on these descriptions. As in mcEclat the implementation uses its own memory manager and TID-arrays are represented by bit-arrays. Creation of the vertical data representation is done in parallel. This is done by dividing the dataset evenly between threads. Each thread is responsible to assign the correct data, in the vertical data

[2]OpenMP Application Program Interface, Version 3.1. `http://www.openmp.org/mp-documents/OpenMP3.1.pdf`

layout, for its own set of transactions.

The implemented mining procedure uses independent class mining, this was described in Section 4.2.2. In order to minimize mining time the most frequent items are mined first during the mining procedure. This has a positive effect when using dynamic scheduling, since frequent items require greater processing time than less frequent. For example let's say that we have one item that requires 50s to be mined and 100 other items that require in total 60s. If we have two threads and both of them start mining the less time consuming items, the 60s will potentially be reduced down to 30s for each thread. When they are done processing these items one of the threads is going to pick the most time consuming item, while the other is going to do nothing. This will result in a total mining time of 80s. However if we do this the other way around, i.e. letting one thread first start mining the most time consuming item, then the other thread can in parallel mine the less frequent items. After 50s the first thread is done, and can help the other thread mine the less frequent items. By this time a total of 10s mining time is remaining, with optimal load balancing this will be reduced down to 5s per thread. The total mining time will be 55s, which is a 25s improvement.

An activity diagram of the algorithm is shown in Figure 6.3. The data processing step consists of support counting and finding which transactions contain frequent items. This is because non-frequent transactions don't need to take up bit-space in the bit-arrays allocated for the vertical data layout.

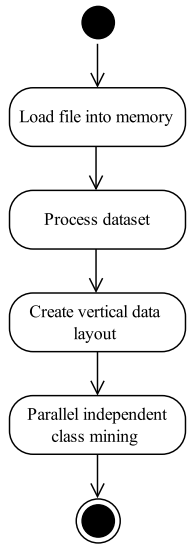


Figure 6.3: Parallel Eclat activity diagram.

6.2.3 Parallel ShaFEM

ShaFEM was implemented according to the description given in 4.3. As in [34] the *Step* variable was set to 32 and *N* was set to 9 in the *UpdateK* function (see Figure 4.11). An activity diagram of the implementation is shown in Figure 6.4. Many of the steps are similar to the previously explained. The dataset is loaded into memory and processed, i.e. support counting is performed and a remap of item IDs by frequency is done. The FP-trees are build in parallel and later merged together. The parallel mining procedure is also implemented with dynamic scheduling were the most frequent items are mined first.

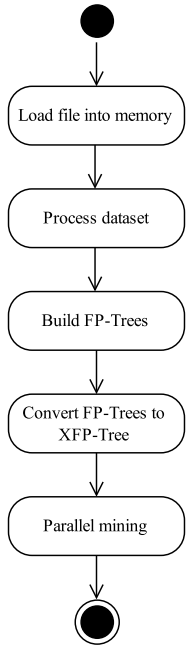


Figure 6.4: ShaFEM activity diagram.

6.3 CFP-Growth and ShaFEM hybrid implemenation

The novel hybrid algorithm of CFP-Growth and ShaFEM was described in Section 4.4. A summary of the algorithm and the implementation details are explained in this section.

In order to implement the hybrid method, a combination was made of the best features of the two algorithms. The strength of CFP-Growth is the memory compression, which results in better cache utilization. Hence, the hybrid method uses the preprocessing steps and data structures from CFP-Growth. This means that all the procedures before the mining step are exactly the same as in CFP-Growth. Firstly a CFP-tree is created and then it is converted into a CFP-array.

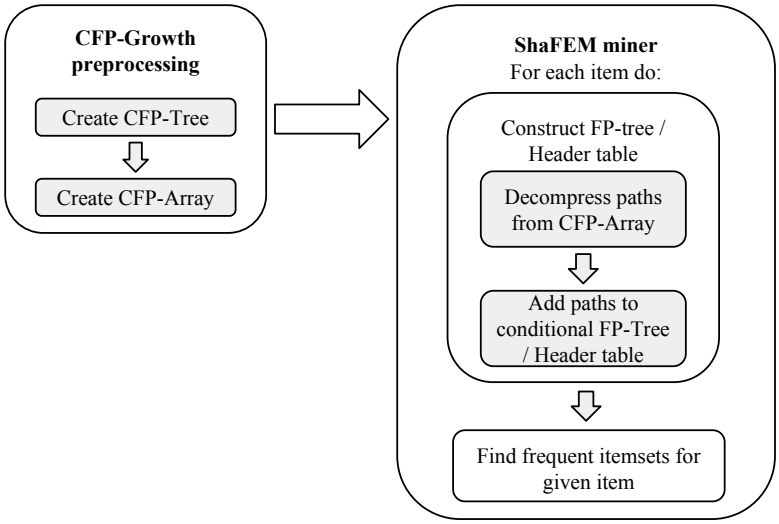


Figure 6.5: Overview of CFP-Growth and ShaFEM hybrid algorithm.

The mining procedure uses the adaptive mining process from ShaFEM. However, it is not exactly the same since it also needs to take decompression of data into consideration. A parallel *for loop* with dynamic scheduling is used in order to mine each item independently. The items are mined in frequency decreasing order. When a thread gets assigned an item to mine, it decompresses the given paths for the item continuously as it builds its conditional FP-tree.

Algorithm 10: ConstructConditionalFpTree

Data: item, cfpArray
Result: Conditional FP-tree.

```
1  Function ConstructConditionalFpTree(item, cfpArray)
2      FPTree fpTree;
3      List<List<int>> conditionalPatternBase;
4      index = 0;
5      for each path in cfpArray[item] do
6          path = cfpArray[item][index].decompressPath();
7          conditionalPatternBase[index].insert(path);
8          index++;
9      end
10     fpTree.headerTable = createHeaderTable(conditionalPatternBase.getUniqueItems());
11     fpTree.constructTree(conditionalPatternBase);
12     return fpTree;
```

Figure 6.6: Simplified construction of conditional FP-tree procedure.

In order to increase scalability all threads get their own memory manager assigned. All memory allocation for each thread happens in their own memory manager or at the local thread stack. An overview of how this algorithm is constructed is shown in Figure 6.5.

An example of how the decompression combined with the mining procedure from ShaFEM works, is shown in Figure 6.6. Here we see a simplified procedure that constructs a conditional FP-tree. Firstly all paths for a given item are decompressed from the CFP-array. The header table, without links, is created from the conditional pattern base. Lastly the FP-tree is constructed by inserting the paths from the conditional pattern base.

6.4 Parallel Index-BTFITI implementation

Two novel parallel methods for frequent inter-transaction itemset mining were implemented, i.e. EP-Index-BTFITI and FGP-Index-BTFITI. These were presented in section 5.4. The algorithms were written in C++ with the parallel programming frameworks OpenMP and Pthreads. An overview of how this algorithm is build up, is shown in Figure 6.7.

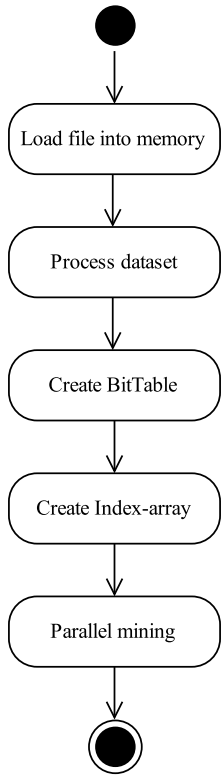


Figure 6.7: Activity diagram for the parallel Index-BTFITI algorithm.

6.4.1 Parallel preprocessing

The parallel preprocessing step is the same as in EP-Index-BTFITI and FGP-Index-BTFITI. In order to parallelize this procedure OpenMP was used. The BitTable creation was done by using a parallel section where the workload was divided between each thread. Since the creation of the index-array consists of a *for loop*, OpenMPs parallel *for loop* with dynamic scheduling was used.

6.4.2 EP-Index-BTFITI

EP-Index-BTFITI was implemented with OpenMP for the mining procedure. When mining the items a parallel for loop was used with dynamic scheduling. As all the previous algorithms implemented, each thread has its own memory manager. Whenever a thread finds a frequent item, it stores it into its own instance of a custom made data structure. This data structure is able to store frequent extended itemsets with its support value, it also contains a memory manager for memory allocation.

6.4.3 FGP-Index-BTFITI

As described in Section 5.4.3, the parallelization technique is more complex than in EP-Index-BTFITI. In order to apply such a parallelization technique a more powerful tool was needed than OpenMP. Thus, Pthreads was used in order to parallelize the mining process in FGP-Index-BTFITI.

Algorithm 11: Dynamic scheduling in Pthreads.

```
Data: data, localLock, frequentItemIndex
1  int i = 0;
2  while true do
3  |   pthread_mutex_lock(localLock);
4  |   i = *frequentItemIndex;
5  |   *frequentItemIndex += 1;
6  |   pthread_mutex_unlock(localLock);
7  |   if i < numFrequentItems then
8  |   |   mine(i, data);
9  |   end
10 |   else
11 |   |   break;
12 |   end
13 end
```

Figure 6.8: Dynamic scheduling in Pthreads.

An implementation of dynamic scheduling in Pthreads was created in order to achieve good workload balancing. This was done by using a *while loop* and letting each thread in that loop get its current iteration variable value. A lock was needed in order to get the current index and to increase its value. Pseudocode for this procedure is given in Figure 6.8. Each thread calls this procedure with a reference to a lock, and a reference to a common index.

The parallelization is implemented as described in Section 5.4.3. Firstly dynamic scheduling is used to mine each item. Depth-first (see Figure 5.4), and the two for loops in IndexBitTableFI (see Figure 5.3) are also parallelized. However, in the two for loops the threads are only assigned work before they are called, and static scheduling is used instead of dynamic. This means that the workload was divided evenly between the threads. The custom memory manager is used during the mining procedure, equally as described in Section 6.4.2.

Chapter 7

Experiments and results

The implemented methods, described in Chapter 6, where thoroughly tested for various datasets. In order to do so multiple datasets and configurations were chosen. Tests were divided into four main categories:

- Experiments for frequent *intra*-transaction itemset algorithms on *real* data.
- Experiments for frequent *intra*-transaction itemset algorithms on *synthetic* data.
- Experiments for frequent *inter*-transaction itemset algorithms on *real* data.
- Experiments for frequent *inter*-transaction itemset algorithms on *synthetic* data.

The results from each category are presented in their own respective section in this chapter. However, firstly a description of the tests conducted is presented in Section 7.1. This is then followed by a description of the datasets used in Section 7.2 After these descriptions the results for the experiments are shown.

7.1 Experimental procedure

Each run for each different configuration was run ten times, where the performance results were written into a log file. The median was chosen between these ten results for each configuration and plotted into a graph with error bars. This was done in order to minimize the effect of outliers in the results. Outliers could occur due to cache advantages, i.e. less cache misses on a specific execution, or simply because another process started in the background. However, this was mitigated by running a cache flush algorithm between all tests that wrote a lot of data into cache in order to clear the previously stored information. It was also made sure, before starting the tests, that unwanted processes would not interfere with the experiments.

The information stored to the log file consisted of preprocessing time, mining time and memory usage. Each algorithm has its own preprocessing phase which includes building of internal data structures that are used for the mining procedure. The size of these structures was logged as the memory usage for the different algorithms. The logged preprocessing time does not include time used for loading file into memory and dataset information gathering, i.e. number of frequent items, number of transactions, and etc. This is because this phase is consistent and equal in all the created algorithms, which would result in equal performance times for these procedures and thus not interesting results. Every algorithm has its own mining procedure, which is the phase were frequent itemsets are found. Since this is the most resource heavy procedure, this is the phase that is most focused on in this chapter.

All algorithms were compiled with the g++ compiler with the optimization flag *-O3*, C++11 was also enabled. The tests were run on a Linux system, where the hardware consisted of two CPUs with 14 cores each. In Figure 7.1 the complete system information is shown.

Operating system	Ubuntu 14.04.2 LTS
Processor	2x Intel® Xeon® CPU E5-2683 v3
Base frequency	2 GHz
Cores per socket	14
Sockets	2
Threads per core	2
Physical cores	28
Logical cores	56
L1d cache	2x14x 32 KB
L1i cache	2x14x 32 KB
L2 cache	2x14x 256 KB
L3 cache	2x 35 MB
Memory	193 GB

Table 7.1: System information

All tests were run with eleven different thread counts separately, i.e. on 1, 2, 4, 7, 14, 21, 28, 42, 56, 112, and 224 threads. The hardware had 28 physical cores, however because of hyper threading the algorithms could get a potential performance boost up to 56 threads. 112 and 224 threads were chosen in order to see how the performance was affected for higher number of threads. Two types of graphs were created regarding variations of thread count. These include mining time, and scalability graphs. The scalability graphs are showing speedup compared to one thread, i.e. $\frac{miningTimeXThreads}{miningTimeOneThread}$. The error bars, for the scalability graphs, are calculated the following way:

$$errorMax = \frac{\max\left(miningTimeXThreads\right)}{\min\left(miningTimeOneThread\right)} \tag{7.1}$$

$$errorMin = \frac{\min\left(miningTimeXThreads\right)}{\max\left(miningTimeOneThread\right)} \tag{7.2}$$

Each test was run with seven different variations of a parameter, e.g. seven different support values. The tests for the intra methods were run on three different datasets, with seven different support values each. Four different algorithms were tested on two of the datasets, and three on the last. In combination with the number of repeated executions and variations of thread counts, the number of individual executions resulted in $10 \cdot 11 \cdot 7 \cdot (2 \cdot 4 + 3) = 8470$ tests.

All four intra methods were also run on synthetic data. Four different parameters were varied: support, average transaction length, number of unique items, and number of transactions. The total number of individual executions for this category of tests resulted in $10 \cdot 11 \cdot 7 \cdot 4 \cdot 4 = 12320$ executions.

The inter tests on real data were run on two different datasets with seven variations of support and seven different variations of maxspan. When tests for support were run the maxspan was set to a constant value, tests for maxspan had a constant support value. Two inter algorithms were tested. The total number of tests for this category was $10 \cdot 11 \cdot (7 + 7) \cdot 2 \cdot 2 = 6160$ tests.

Tests on inter algorithms on synthetic datasets were run with variations on five different parameters: support, maxspan, average transaction length, number of unique items, and number of transactions. Whenever a parameter was chosen to be varied, the other parameters were set to their default values. Therefore the total number of tests for this category was $10 \cdot 11 \cdot 7 \cdot 5 \cdot 2 = 7700$ tests.

In total $8470 + 12320 + 6160 + 7700 = 34650$ individual executions of the different algorithms were run. The most interesting results produced by these executions were chosen, and are presented in this chapter.

7.2 Dataset description

Multiple different datasets were used for the different experiments. These are explained and presented in this section.

7.2.1 Real datasets used for intra experiments

Three different datasets were chosen from the FIMI repository[1] for the experiments on frequent intra-transaction itemset algorithms. The selected datasets are shown in Table 7.2 with their respective attributes. As can be seen from the figure the datasets have different types of characteristics. Chess is a dense and small dataset with few unique items, retail is larger and sparser. Webdocs [19], on the other hand, is a large dataset with many unique

[1] Frequent Itemset Mining Dataset Repository: `http://fimi.ua.ac.be/data/`

items and a large amount of transactions. These datasets were chosen because of their various characteristics, and in order to see the behaviour of the algorithms when using such different datasets. The definition of density is the same as given in Section 3.1.3, i.e.:

$$density = \frac{average\_transaction\_length}{num\_unique\_items} \tag{7.3}$$

Dataset name	Num. trans-actions	Avg. trans. length	Num. unique items	Density	File size
Chess	3196	37.00000	75	0.4933333	334 KB
Retail	88162	10.30576	16470	0.0006257	3.97 MB
Webdocs	5267656	177.2297	1692082	0.0000336	1.37 GB

Table 7.2: Real dataset characteristics for intra-transaction itemset mining.

7.2.2 Real datasets used for inter experiments

Two real datasets were chosen for the tests for frequent inter-transaction itemset mining. The Stock dataset was created by choosing 30 different companies with stock data in the time frame 1986-03-13 - 2015-05-01. Data was downloaded from the historical prices section from *Yahoo! Finance*[2]. The chosen companies are listed in Table A.1 in Appendix A. In order to create the dataset, each individual stock data was assigned the two following states:

- *increased*
- *equal or decreased*

If the difference between the closing and opening price for a particular date was greater than 0, the state was set to *increased* otherwise to *equal or decreased*. Each state combined with the item got its unique item represented in the dataset. For example, let's say we have two companies: *A* and *B*. Then the items can be mapped the same way as shown in Table 7.3.

ID	Company	State
0	*A*	*increased*
1	*A*	*equal or decreased*
2	*B*	*increased*
3	*B*	*equal or decreased*

Table 7.3: Stock dataset item ID mapping example.

[2]http://finance.yahoo.com/

Each transaction in the dataset represents a day, and consists of 30 items representing each separate chosen company. The item ID chosen for a given company indicates if the stock increased, or stayed equal or decreased in price. The resulting dataset is dense and small.

In order to see the effect of the tested algorithms for a larger and sparser dataset, kosarak was chosen from the FIMI repository. This dataset contains anonymized click-stream data from a Hungarian news website. During testing multiple inter-transaction itemsets were found that spanned throughout different transactions. Hence, this dataset was seen as viable for frequent inter-transaction itemset mining. The characteristics of the two datasets are shown in Table 7.4.

Dataset name	Num. trans-actions	Avg. trans. length	Num. unique items	Density	File size
Stock	7346	30.00000	60	0.5000000	631.8 KB
Kosarak	990002	8.099999	41270	0.0001963	30.5 MB

Table 7.4: Real dataset characteristics for inter-transaction itemset mining.

7.2.3 Synthetic dataset description

Similarly as in [39] it was decided to run tests with variations in different dataset parameters. In order to do so synthetic datasets needed to be created. These datasets were created similarly as in [17], and consisted of multiple different steps. A description of these stages is given below.

1. **Unique items**: An array consisting of all unique items was created. This array was then shuffled at random.

2. **Potential megatransactions**: A set of potential megatransactions was created. The number of megatransactions was set to 2% of the total number of transactions. The size of each megatransaction was determined through a Poisson distribution with mean set to 30% of the size of an average transaction length. In order to choose which items to include in the megatransaction, a uniform distribution was used. Items were picked from a set that contained $numUniqueItems \cdot maxspan$ unique identifiers. This set represented all possible items combined with the dimensional attribute. The $maxspan$ variable was set to 6 and indicated that a potential megatransaction could span through 6 transactions. However, in inter-transaction itemset mining often items from the previous transaction occur in the next transaction. This was simulated by using an exponential distribution, with rate set to 2.0, in order determine the fraction of items from the previous transaction that would occur in the next. Items were chosen from the previous transaction with a uniform distribution.

3. **Potential megatransaction modifications**: The dimensional attribute in each potential megatransaction was subtracted by the lowest dimensional attribute present

in the current megatransaction. This was done so the smallest dimensional attribute would always be 0.

Dataset name	Num. trans-actions	Avg. trans. length	Num. unique items	Density	File size
T25K-A50-I1000-M6	25 K	50	1000	0.050	4.9 MB
T50K-A50-I1000-M6	50 K	50	1000	0.050	9.8 MB
100K-A50-I1000-M6	*100 K*	*50*	*1000*	*0.050*	*19.6 MB*
T200K-A50-I1000-M6	200 K	50	1000	0.050	39.1 MB
T400K-A50-I1000-M6	400 K	50	1000	0.050	78.2 MB
T800K-A50-I1000-M6	800 K	50	1000	0.050	156.5 MB
T1600K-A50-I1000-M6	1600 K	50	1000	0.050	312.7 MB
T100K-A35-I1000-M6	100 K	35	1000	0.035	13.7 MB
T100K-A40-I1000-M6	100 K	40	1000	0.040	15.6 MB
T100K-A45-I1000-M6	100 K	45	1000	0.045	17.6 MB
T100K-A55-I1000-M6	100 K	55	1000	0.055	21.5 MB
T100K-A60-I1000-M6	100 K	60	1000	0.060	23.5 MB
T100K-A65-I1000-M6	100 K	65	1000	0.065	25.4 MB
T100K-A50-I2000-M6	100 K	50	2000	0.025	22.3 MB
T100K-A50-I4000-M6	100 K	50	4000	0.0125	23.8 MB
T100K-A50-I8000-M6	100 K	50	8000	0.00625	24.4 MB
T100K-A50-I16K-M6	100 K	50	16 K	0.003125	26.7 MB
T100K-A50-I32K-M6	100 K	50	32 K	0.0015625	28.4 MB
T100K-A50-I64K-M6	100 K	50	64 K	0.00078125	29.2 MB

Table 7.5: Synthetic dataset characteristics. Row in cursive represents the default dataset, i.e. dataset with the default transaction count, average transaction length, and item count.

4. **Megatransaction weight**: All potential megatransaction got a weight assigned with help of a exponential distribution function. The rate of this function was set to 1.0. Then the weights were normalized. This weight was used in order to choose which megatransactions to assign to a transaction.

5. **Transaction lengths**: The length of each transaction was set with a Poisson distribution with mean set to average transaction length. The lengths needed to be adjusted later on, so the actual average transaction length would be the same as specified by the user.

6. **Megatransaction assignment**: Potential megatransactions were assigned to each transaction, where the megatransaction was chosen with help of a uniform distribution. The probability of a megatransaction being chosen was given through its weight. This was done until the current transaction length was equal to the intended transaction length. Megatransactions are not always equal in reality and in order to simulate this, a corruption procedure was used. An item was corrupted if a value

generated from a normal distribution, with mean set to 0.5 and variance to 0.1, was greater than 1.0. Then the item was replaced by a random value from the array containing unique items. It was also made sure that transactions only consisted of unique items.

7. **Non assigned items**: Lastly all non assigned items, from the array containing unique items, were assigned at random positions in the dataset.

The created datasets are shown in Table 7.5.

7.3 Intra-transaction itemset mining tests on real data

This section presents experiments for the implemented intra-transaction itemset mining methods that were described in Section 6.2 and 6.3. Tests were run on four different algorithms: CFP-Growth, ShaFEM, hybrid of ShaFEM and CFP-Growth, and parallel Eclat. Eclat was only tested on two of the datasets, because the execution time was too long compared to the other algorithms on one of the datasets. The tests were run on three different datasets with seven different minimum support configurations. These datasets were described in Section 7.2.1.

7.3.1 Experiments on chess dataset

Firstly tests on the chess dataset were run with all four FIM algorithms, i.e. CFP-Growth, ShaFEM, CFP-Growth and ShaFEM hybrid, and parallel Eclat. Support was varied between 0.25 - 0.37 with an interval of 0.02. Figure 7.1 shows how the different algorithms performed using one thread for these support variations.

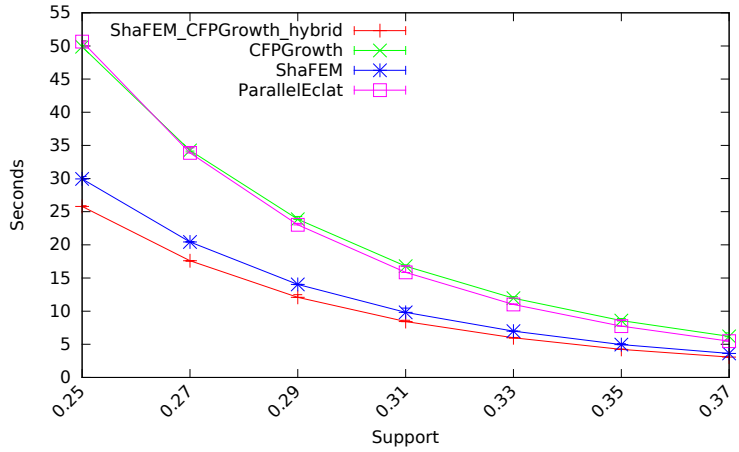
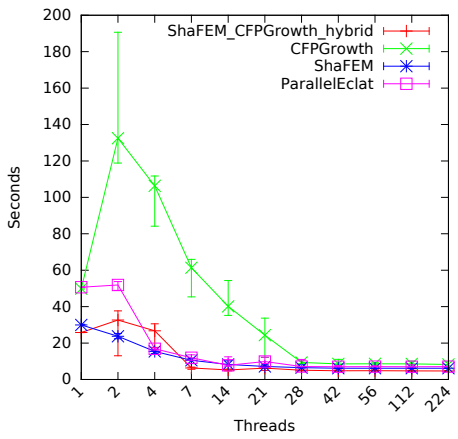
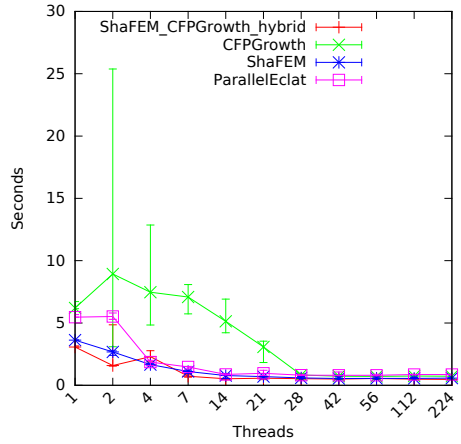


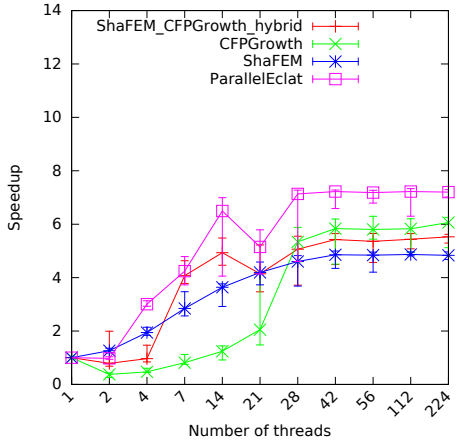
Figure 7.1: Mining time, using one thread, with varying support values on chess dataset.



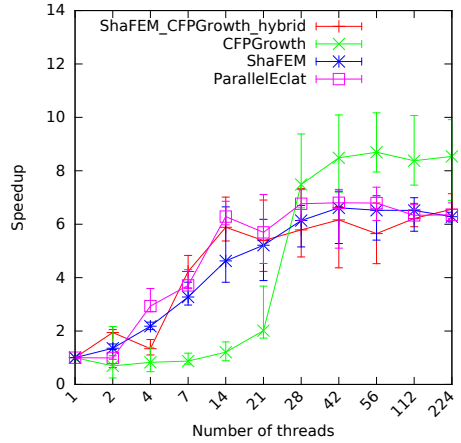
(a) Mining time, support = 0.25.



(b) Mining time, support = 0.37.



(c) Mining scalability, support = 0.25.



(d) Mining scalability, support = 0.37.

Figure 7.2: Mining performance on chess dataset with two different support values.

The overall trend is that the mining time decreases with higher support. This is expected because there are fewer frequent itemsets to be mined with higher support value. We see in the graph that the hybrid method outperforms the other algorithms; this is because of the dynamic mining procedure. It outperforms ShaFEM probably because of better cache utilization. CFP-Growth uses regular FP-Growth type of mining procedure, and hence is not able to perform equally good as ShaFEM or the hybrid method. This is because this is a dense dataset. Even though Eclat, ShaFEM, and the hybrid method are using similar mining procedures for this dataset, we can see that Eclat doesn't perform as good. ShaFEM and the hybrid method are able to reduce the size of the array they are intersecting in order

to calculate the support value. The intersection array size for Eclat, is on the other hand larger and constant, this is probably the reason for the high mining times.

Figure 7.2 shows how the algorithms perform with different thread counts for two different support values, i.e. support set to 0.25 and 0.37. CFP-Growth differs from the other algorithms; we see that the mining time is significantly larger for two threads than with one thread. However, the error bars indicate that the mining time varies significantly. A test was run on another system to see if this problem was possible to replicate, and the results showed otherwise. With two threads CFP-Growth was almost twice as fast as with one thread. Nonetheless the results on both systems indicated that the mining time of CFP-Growth was greater than the other algorithms. This is expected because this is a dense dataset, and an Eclat type of mining method will outperform FP-Growth type of algorithms for these datasets.

In Figure 7.2c and 7.2d we see that the algorithms do not scale very well for this dataset. To some extent the same problem occurs as in EP-Index-BTFITI explained in Section 5.4.3. There are items that require significantly more mining time then others, and results in some threads finishing their work early while others are still mining. Since an embarrassingly parallel technique is used, and not a fine grained method, this results in multiple threads doing nothing. On top of this a large amount of frequent itemsets are generated. With support set to 0.25 almost 100 million frequent itemsets are found, and around 10 million with support set to 0.37. This results in a large amount of data being written into system memory, and hence the system bus becomes the bottleneck.

7.3.2 Experiments on retail dataset

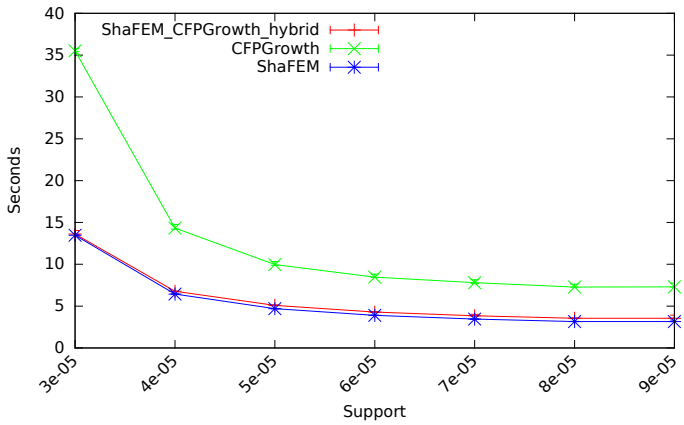
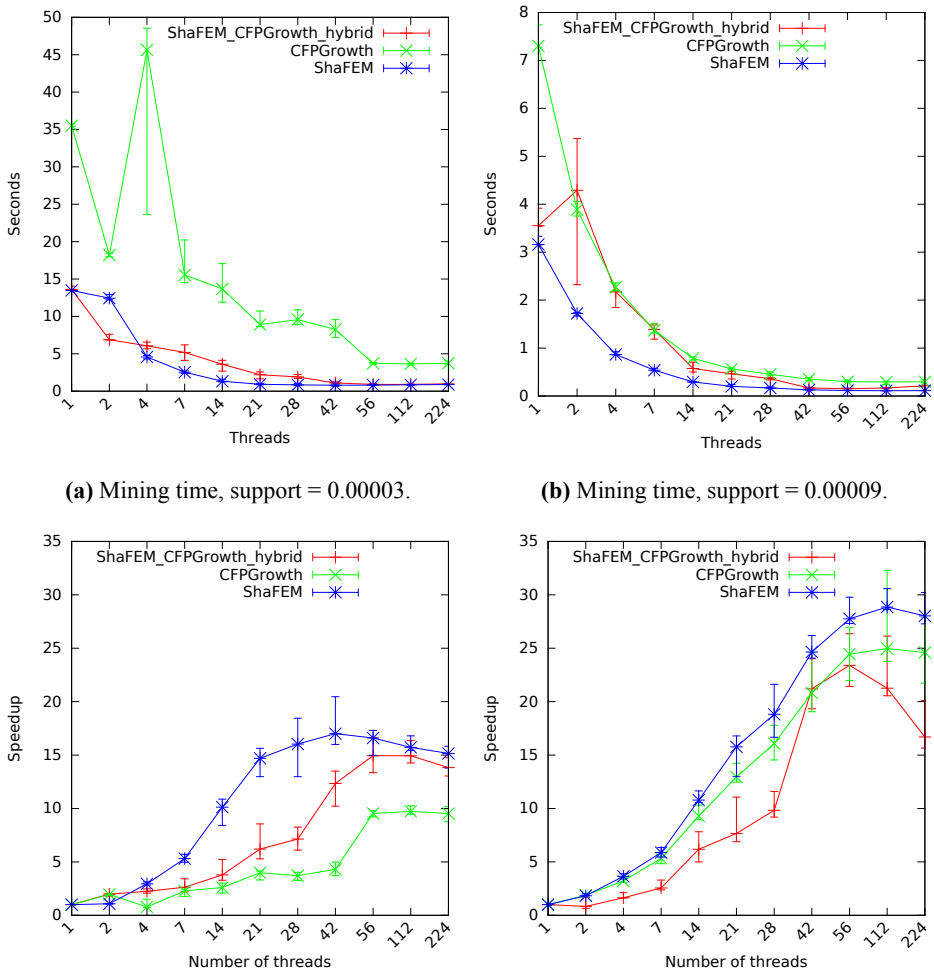


Figure 7.3: Mining time, using one thread, with varying support values on retail dataset.

Because of the amount of tests that had to be run, and restricted time, parallel Eclat was not run on the retail dataset. Even if it was run, the mining time would be so great that the

difference between the other algorithms would not be visible in the mining time graphs. This is a sparse dataset, and algorithms such as Eclat have problems mining such sets especially with such a low support value. The minimum support varied from 0.00003 to 0.00009 with an interval of 0.00001. Results for this are shown in Figure 7.3 where the variations are done with one thread. Even though retail is a sparse dataset we see that both the ShaFEM and the hybrid method get an advantage of their bit vector mining method. The hybrid method shares the best mining time with ShaFEM in this situation.



(a) Mining time, support = 0.00003.

(b) Mining time, support = 0.00009.

(c) Mining scalability, support = 0.00003.

(d) Mining scalability, support = 0.00009.

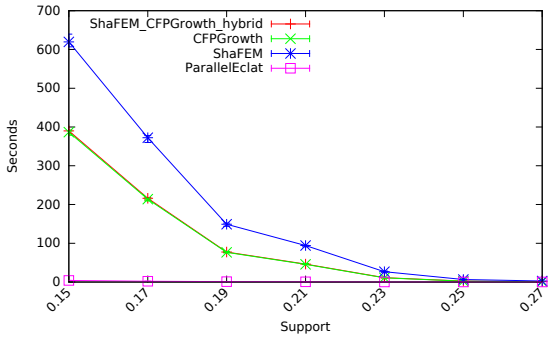
Figure 7.4: Mining performance on retail dataset with two different support values.

The graphs in Figure 7.4 show how the algorithms performed on two different support val-

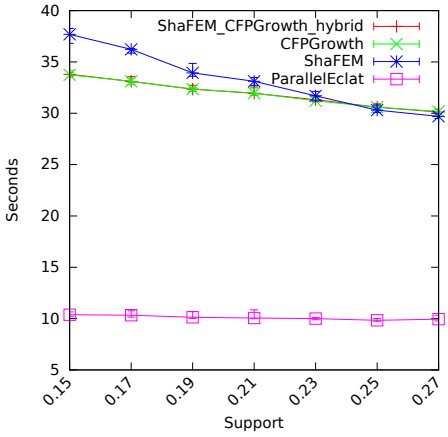
ues, with variations on thread count. From Figure 7.4c and 7.4d we see that the scalability increases with higher support value. This is probably because fewer frequent items are found, and thus fewer accesses to the system bus. With support set to 0.00003 around 20 million frequent items are found, and around 0.3 million for support set to 0.00009. The scalability is good for a high support value, where ShaFEM almost gets 30 times speedup compared to one thread.

With support set to 0.00003 the hybrid method performs almost as good as ShaFEM. There is not a great difference, regarding mining time, between the algorithms with support set to 0.00009. With a large amount of threads the variations are only a few milliseconds.

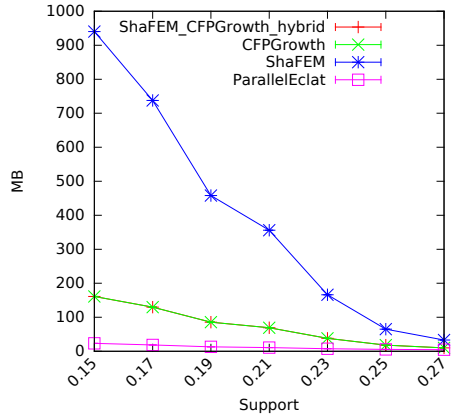
7.3.3 Experiments on webdocs dataset



(a) Mining time.



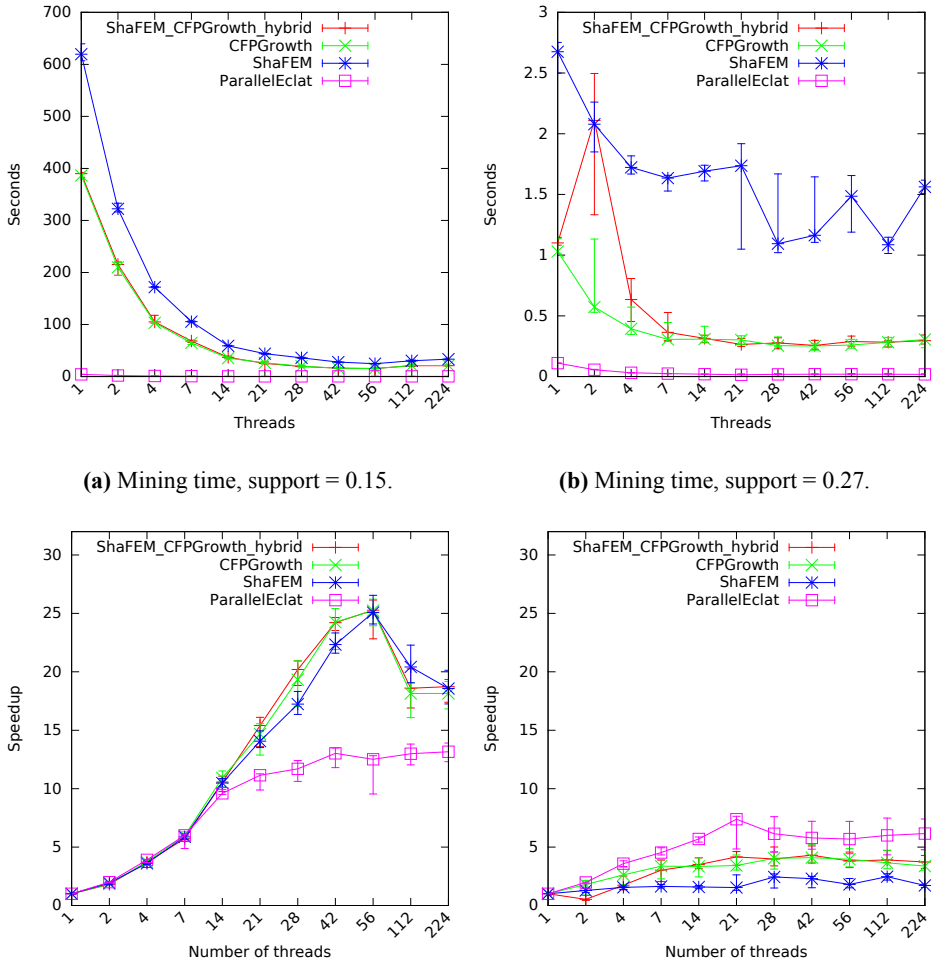
(b) Preprocessing time.



(c) Memory usage.

Figure 7.5: Performance results, using one thread, with varying support values on webdocs dataset.

In the previous tests preprocessing time and memory usage was minimal, and hence not shown in the graphs. Webdocs on the other hand is a large dataset, where the file size is 1.37 GB. This will result in greater memory usage and time to create the different data structures. Figure 7.5 shows different performance results with one thread with varying support. The minimum support was varied from 0.15 to 0.27 with a interval of 0.02.



(a) Mining time, support = 0.15.

(b) Mining time, support = 0.27.

(c) Mining scalability, support = 0.15.

(d) Mining scalability, support = 0.27.

Figure 7.6: Mining performance on webdocs dataset with two different support values.

We see that parallel Eclat performs very good for this dataset in all aspects. The reason for this is that during the preprocessing step Eclat removes all non-frequent items and transactions that don't contain any frequent transactions. This results in a data structure that doesn't

require much memory, and also in low mining time. Even tough ShaFEM, CFP-Growth and the hybrid method are also removing their non-frequent items, their data structure still uses significantly more memory. These algorithms have different data structures and hence is an expected result. However, we see that the memory compression of CFP-Growth and the hybrid method works very well. For support set to 0.15 these algorithms use almost $\frac{1}{6}$ of the memory usage of ShaFEM. This also results in better mining time, because of better cache utilization.

Figure 7.6 shows the performance results with varying thread count and with two different support values. The scalability is good for support set to 0.15, were the speedup is almost 25 times as fast as with one thread except for parallel Eclat. A probable reason for this can be that some threads finish mining before other threads in Eclat. Also the mining time is very small which can again lead to inaccuracies, the same applies for mining scalability with support set to 0.27.

7.3.4 Conclusion

In conclusion we see that all algorithms performed differently under different circumstances. The Eclat algorithm was very good under some cases, e.g. when mining webdocs with high support, and very poor for other datasets like retail. CFP-Growth was able to utilize the cache better than ShaFEM, however ShaFEM performed very well in most cases because of its dynamic mining procedure. The novel hybrid method of ShaFEM and CFP-Growth performed very well for almost all types of configurations. In almost all cases it had the smallest mining time or performed equally as the best result, except for those cases where Eclat was dominating.

7.4 Intra-transaction itemset mining tests on synthetic data

In order to see how the different algorithms performed with variations in dataset parameters, experiments were run on synthetic datasets. These datasets were described in Section 7.2.3. Although these datasets are primarily created for frequent inter-transaction itemset mining algorithms, these are fully applicable for intra methods as well.

A form of parameter isolation was used during the tests. This means that whenever a parameter in the dataset was varied, the other attributes were set to standard constant values. These are the standard values used:

- Transaction count: 100000
- Average transaction length: 50
- Unique item count: 1000
- Support: 0.05, 0.03, 0.02
- Thread count: 1

Note that support varied for each different test. For variations in transaction count support was set to 0.05, when varying average transaction length it was set to 0.03, and when varying item count it was set to 0.02. This was in order to adjust the mining time so it would not be too great or too small. If it would have been too small the results could be inaccurate, if the mining time was to great it could potentially exceed the available time that was made for the experiments.

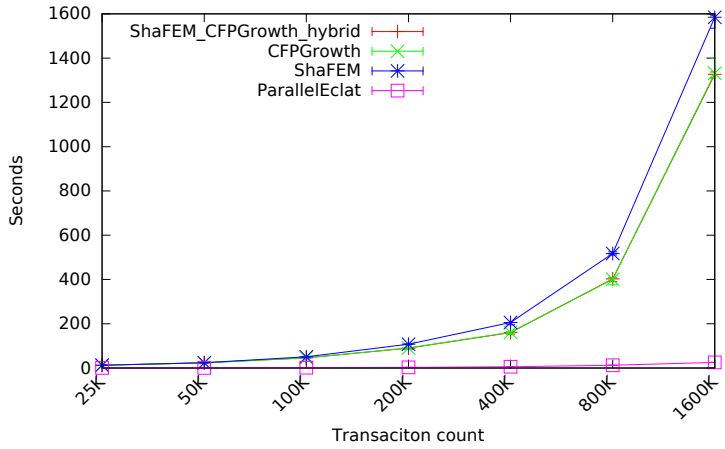
Tests on varying support count are not covered in this section, because variations in support was already thoroughly tested on real datasets. However, the resulting graphs for these tests can be seen in Appendix B.

Parallel Eclat stands out in these experiments because of its low run time combined with the low memory usage. The reason for this is that the synthetic datasets are quite dense, and the support value is high. With the high support value Eclat is able to remove large amounts of unnecessary data during preprocessing, and thus leads to smaller data structures which again leads to smaller run time. Nonetheless, these tests on synthetic data are not supposed to show which algorithm performs good on what type of dataset. This was accomplished with the tests on real data. What these tests are meant to do is to show how the different algorithms perform with variations on different dataset parameters.

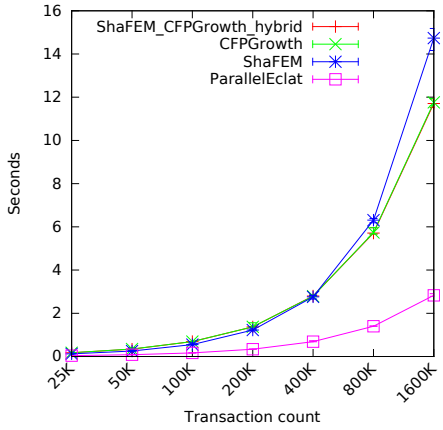
7.4.1 Varying number of transactions

The graphs in Figure 7.7 show how the performance changes with increasing amount transactions. As can be seen in these graphs, the overall trend shows that the run time and memory usage increases with increasing transaction count. This can be expected because the increase of transaction count leads to larger data structures, which again leads to larger run time.

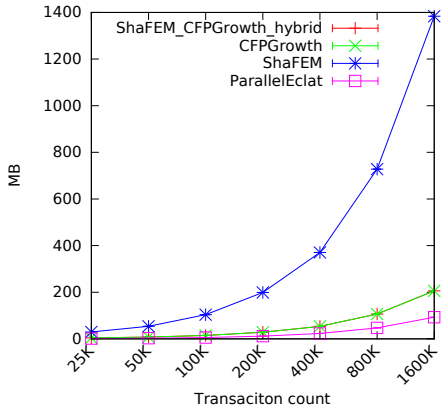
The graphs in Figure 7.8 shows the mining time for two different transaction counts. CFP-Growth and the hybrid method are performing better with 1600 K transactions, because of better cache utilization. The scalability doesn't change much, which is reasonable because the amount of frequent itemsets generated varies little. Figure 7.8c shows a rough graph since the mining time is low. This often leads to small variations in mining time and the scalability graph is very sensitive for this.



(a) Mining time.

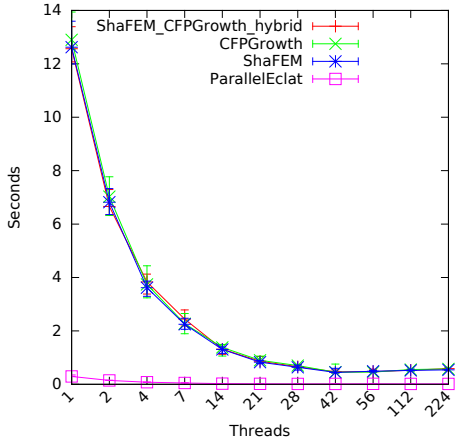


(b) Preprocessing time.

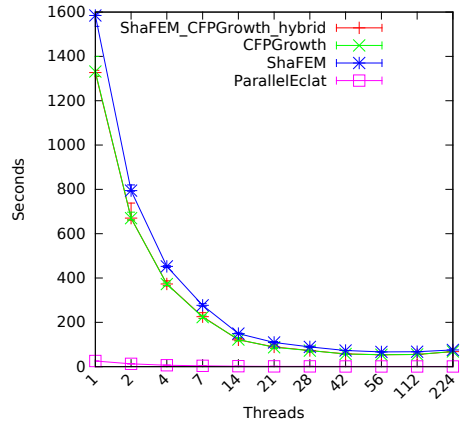


(c) Memory usage.

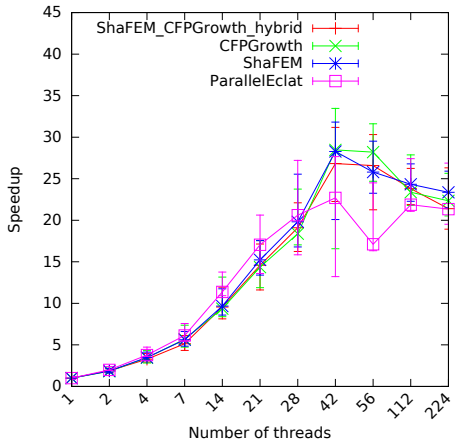
Figure 7.7: Performance results, using one thread, with varying transaction count. Support = 0.05. The hybrid method and CFP-Growth are overlapping in these graphs.



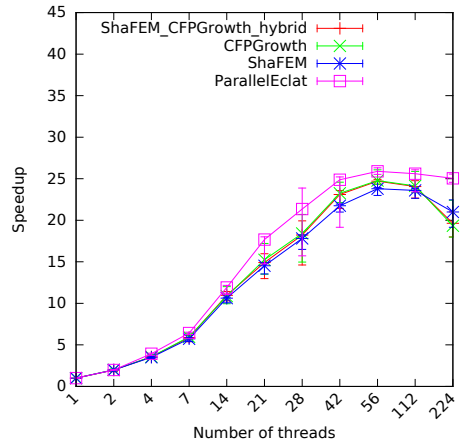
(a) Mining time, transaction count = 25 K.



(b) Mining time, transaction count = 1600 K.



(c) Mining scalability, transaction count = 25 K.

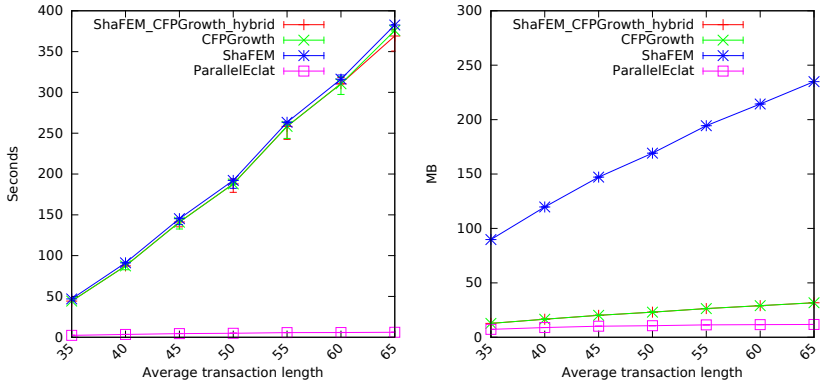


(d) Mining scalability, transaction count = 1600 K.

Figure 7.8: Mining performance with two different transaction counts. Support = 0.05. The hybrid method and CFP-Growth are overlapping in these graphs.

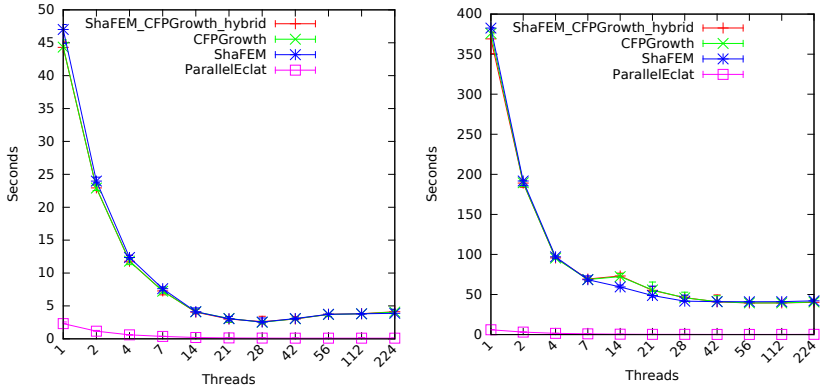
7.4.2 Varying average transaction length

With increasing average transaction length the amount of frequent items and itemsets generated will increase. This will result in larger datasets and larger mining time. As expected this were the results, which can be seen in Figure 7.9. It is not very visible in the graph, however the memory usage and run time of parallel Eclat is also increasing with the average transaction length.

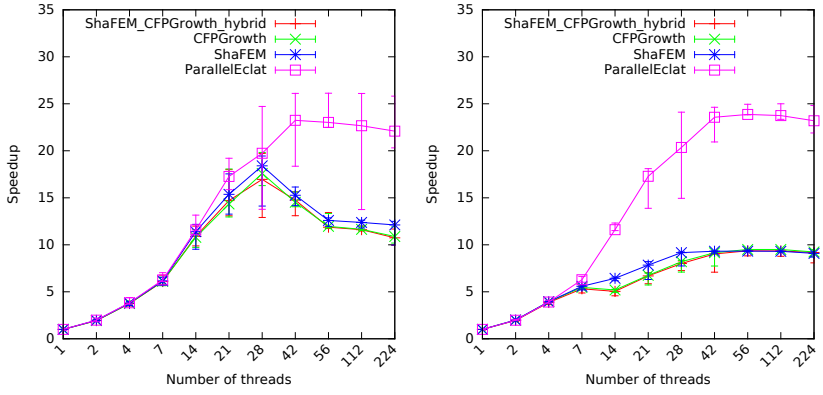


(a) Mining time. **(b)** Memory usage.

Figure 7.9: Performance results, using one thread, with varying transaction count. Support = 0.03.



(a) Mining time, avg. trans. length = 35. **(b)** Mining time, avg. trans. length = 65.



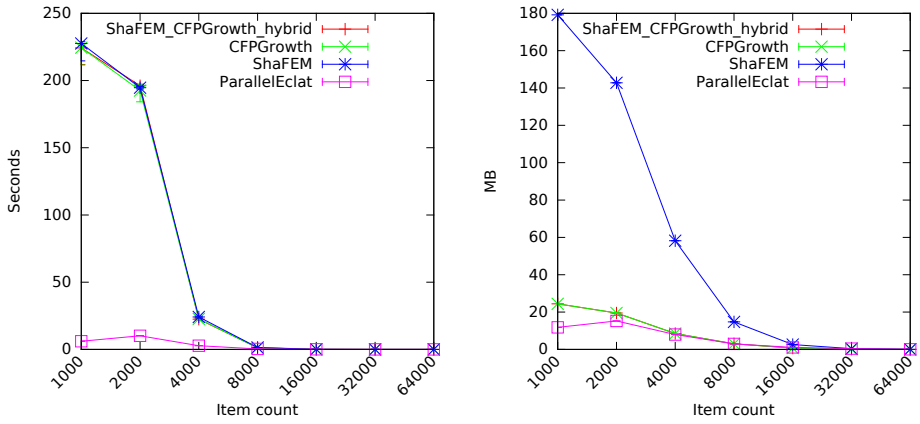
(c) Mining scalability, avg. trans. length = 35. **(d)** Mining scalability, avg. trans. length = 65.

Figure 7.10: Mining performance with two different average transaction lengths. Support = 0.03.

When the number of frequent itemsets found increases, the scalability is expected to decrease. This is because the number of bus accesses increases, and system memory access is slow. This result is confirmed in Figure 7.10. Here we see the mining performance for two different average transaction lengths. The scalability of parallel Eclat doesn't change much. A probable reason for this is that the data structure memory usage is small, which leads to fewer bus accesses. It is also important to keep in mind that the mining time for Eclat is very small, which can lead to inaccurate results.

7.4.3 Varying number of unique items

In Figure 7.11 the performance is shown when the number of unique items increases in the dataset. The increase of items will result in that each item becomes less frequent, because of fewer repetitions of a given item. This is the result because the number of transaction and the average transaction length is constant. As expected this leads to smaller data structures and smaller mining time.

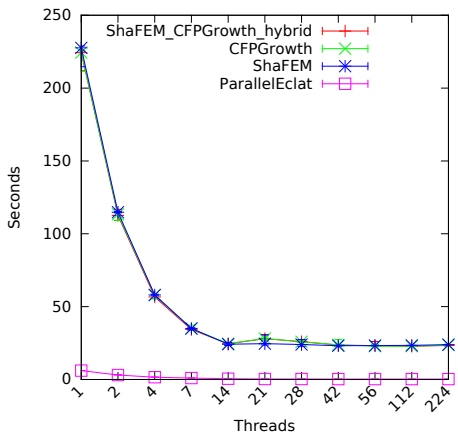


(a) Mining time.

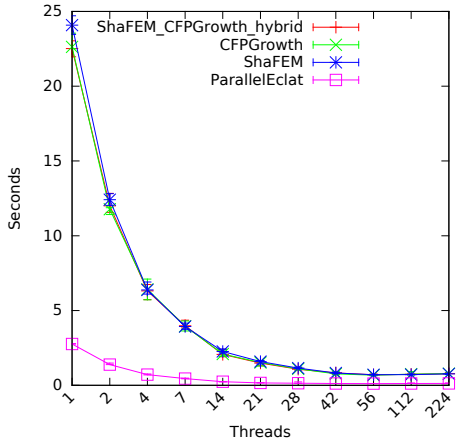
(b) Memory usage.

Figure 7.11: Performance results, using one thread, with varying item count. Support = 0.02.

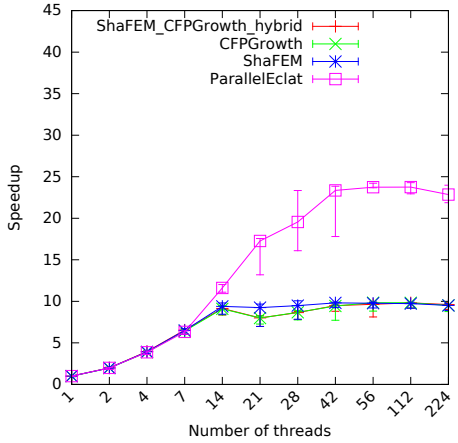
With increasing amount of items, fewer frequent itemsets are generated. This leads to better scalability as can be seen in Figure 7.12. Here we see graphs for datasets with 1000 and 4000 unique items. Even thought the dataset with the maximum amount of unique items had 64000, the dataset with 4000 items was chosen. The reason for this was that the mining time was insignificant for this dataset.



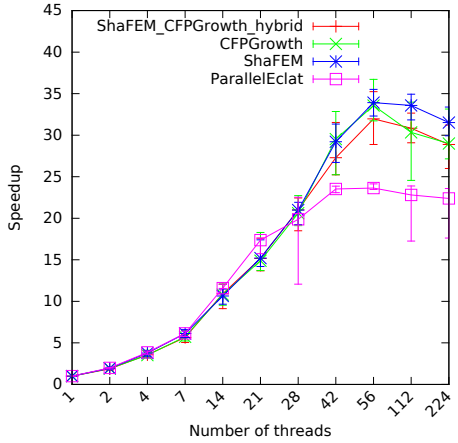
(a) Mining time, item count = 1000.



(b) Mining time, item count = 4000.



(c) Mining scalability, item count = 1000.



(d) Mining scalability, item count = 4000.

Figure 7.12: Mining performance with two different item counts. Support = 0.02.

7.4.4 Conclusion

In summary, what we have learned from these tests is the following:

- Run time and memory usage increases with increasing transactions.
- Run time and memory usage increases with increasing average transaction length.
- Run time and memory usage decreases with increasing item count.

7.5 Inter-transaction itemset mining tests on real data

This section covers tests executed for the two created inter-transaction itemset mining methods described in Section 6.4. Tests were run on two different datasets, which were described in Section 7.2.2. For each dataset two types of tests were run, one in order to see how the algorithms are behaving with variations in support and another test where maxspan was varied. Whenever a parameter was varied, the other parameter was set to a constant default value.

7.5.1 Experiments on stock dataset with varying support

In Figure 7.13 we see the test results with variations in minimum support and number of threads set to 28 and maxspan to 3. The reason 28 threads were chosen was because both the tested algorithms, i.e. EP-Index-BTFITI and FGP-Index-BTFITI, perform equally good with one thread. This is because they are using the same type of mining procedure with one thread, however the parallelization technique differs as explained in Section 6.4. The support was varied from 0.08 to 0.14 with an interval of 0.01. As expected FGP-Index-BTFITI outperforms EP-Index-BTFITI, and the overall mining time decreases with increasing support.

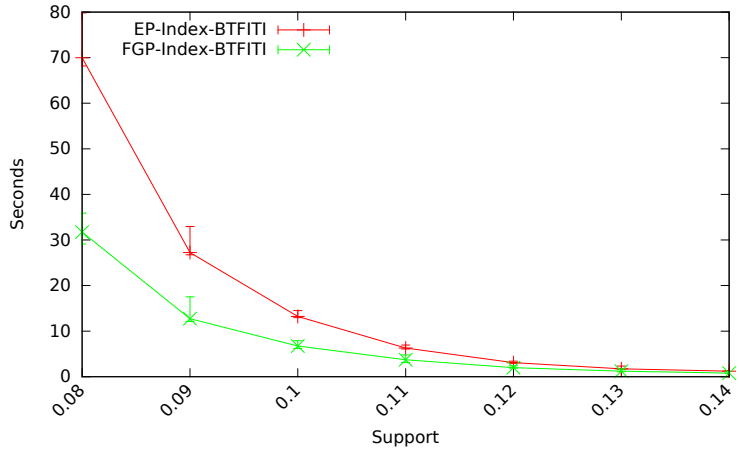
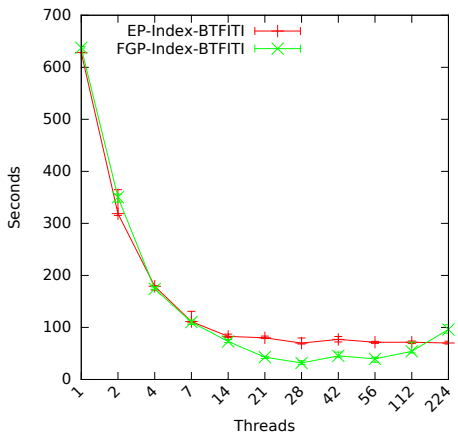
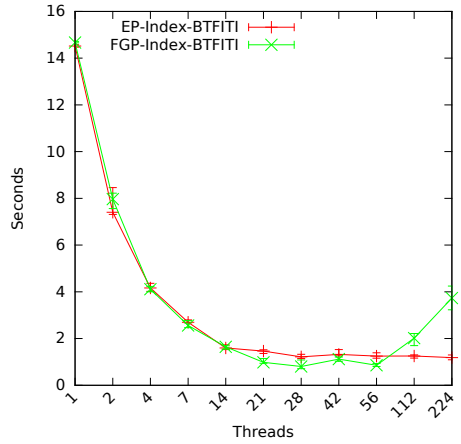


Figure 7.13: Mining time, using 28 threads, with varying support values on stock dataset. Maxspan is set to 3.

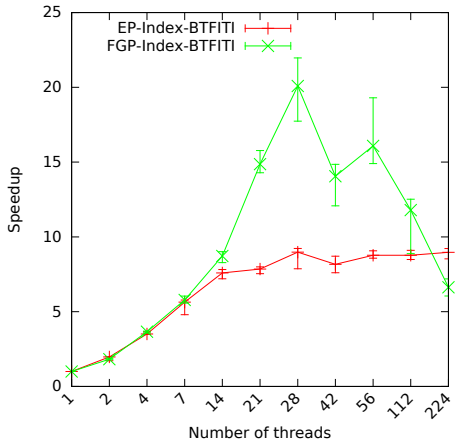
Figure 7.14 shows the performance with varying thread count with two different support values. Based on the scalability we see that the extended parallelization technique, in FGP-Index-BTFITI, is able to improve the performance greatly. With both different support values we see that EP-Index-BTFITI stagnates at around 14 threads, while FGP-Index-BTFITI is able to improve the performance with a greater amount of threads.



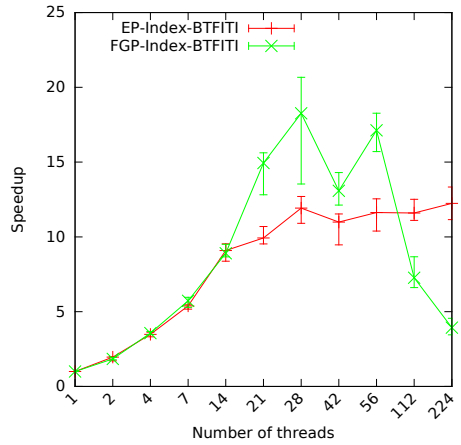
(a) Mining time, support = 0.08.



(b) Mining time, support = 0.14.



(c) Mining scalability, support = 0.08.



(d) Mining scalability, support = 0.14.

Figure 7.14: Mining performance on stock dataset with two different support values. Maxspan is set to 3.

7.5.2 Experiments on stock dataset with varying maxspan

With increasing maxspan it is expected that the mining time also will increase. This behaviour is shown in Figure 7.15. Here we see the mining run time with varying maxspan values, minimum support set to 0.11 and thread count to 28. Maxspan is varied from 0 to 6.

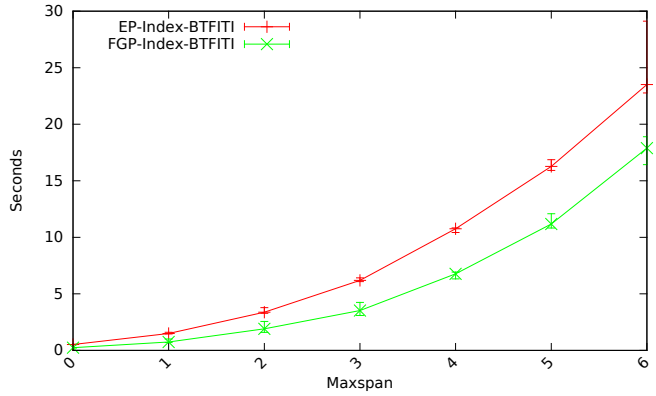
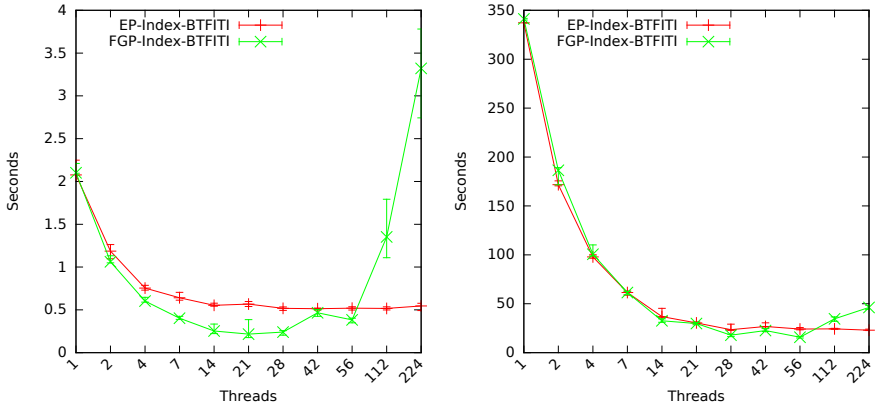
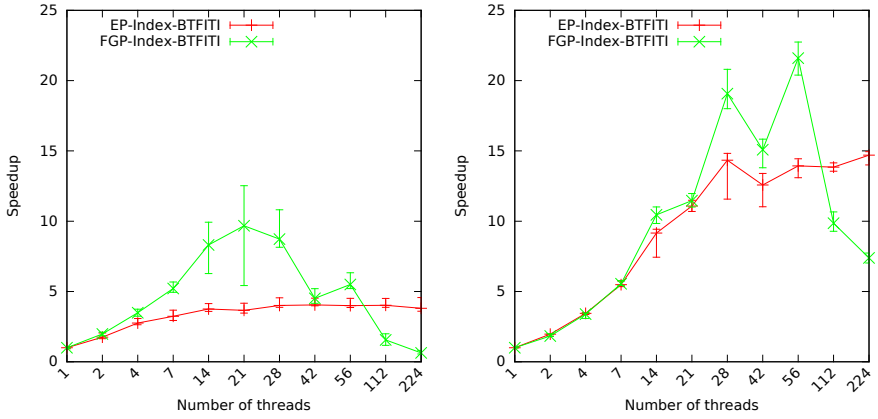


Figure 7.15: Mining time, using 28 threads, with varying maxspan values on stock dataset. Support is set to 0.11.



(a) Mining time, maxspan = 0.

(b) Mining time, maxspan = 6.



(c) Mining scalability, maxspan = 0.

(d) Mining scalability, maxspan = 6.

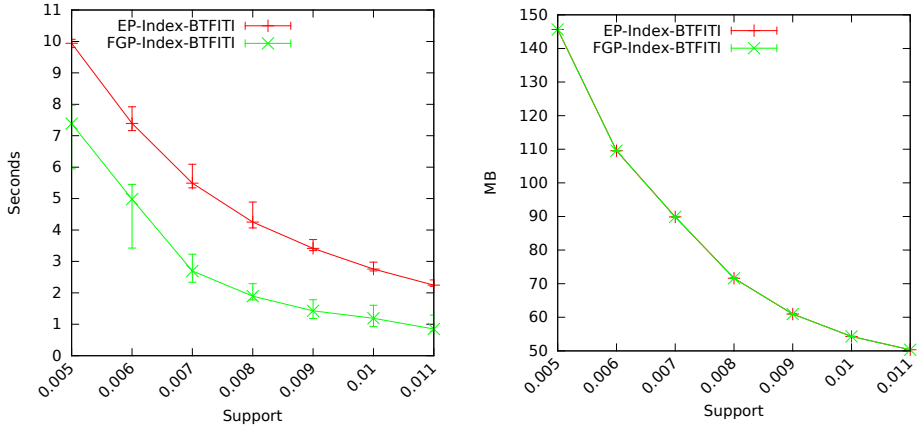
Figure 7.16: Mining performance on stock dataset with two different maxspan values. Support is set to 0.11.

The graphs in Figure 7.16 show run time and scalability variations for different thread counts. For the first column maxspan is set to 0 and in the second column it is set to 6. A common behaviour we see for FGP-Index-BTFITI is that it often reaches a peak value for a given thread, and then it decreases in performance. This is not as common in EP-Index-BTFITI. The reason for this is that FGP-Index-BTFITI must lock data, which again causes other threads to wait. The wait time increases with the thread number. In EP-Index-BTFITI there are no locks in the mining procedure, and therefore this behaviour doesn't occur.

Scalability increases with both algorithms with higher maxspan, as seen from the graphs 7.16c and 7.16d. The reason for this is that the scalability is best when the mining procedure from EP-Index-BTFITI can be used efficiently. We see that this is the case from these graphs, because the difference between the run times for the algorithms is smaller for maxspan set to 6. This means that FGP-Index-BTFITI doesn't need to use its extended parallelization technique to the same degree. In such cases the thread utilization is better.

7.5.3 Experiments on kosarak dataset with varying support

Since kosarak is a larger dataset than stock, a graph showing memory usage is also shown in Figure 7.17b. Figure 7.17a shows the mining time. Both of these figures show performance results with varying support, and maxspan set to 3 while the number of threads is set to 28. Support varies from 0.005 to 0.011 with an interval of 0.001. The memory usage is equal for both of these procedures since they share the same data structures.

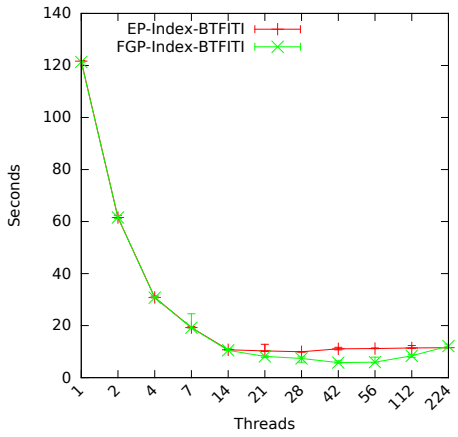


(a) Mining time.

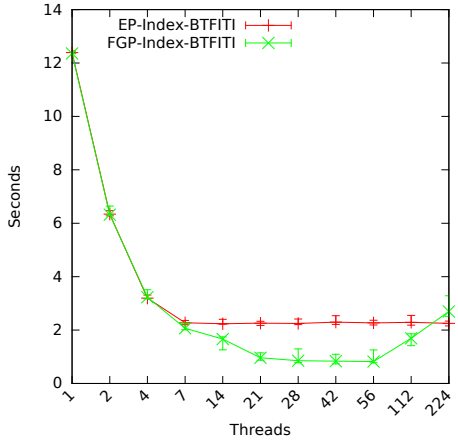
(b) Memory usage.

Figure 7.17: Performance results, using 28 threads, with varying support values on kosarak dataset. Maxspan is set to 3.

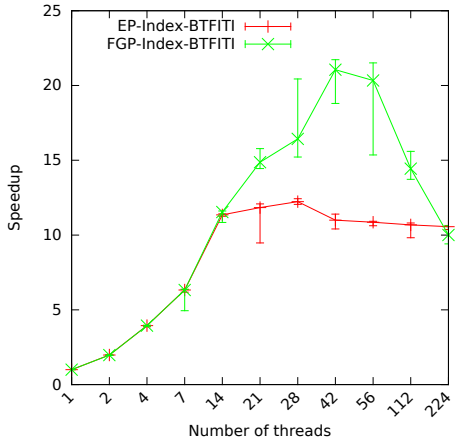
In Figure 7.18 we see the performance results with variations in thread count. Support is set to 0.005 in the first column and 0.011 in the second, maxspan is set to 3. These results are similar to the previously shown, i.e. the graphs of EP-Index-BTFITI stagnates at some point while FGP-Index-BTFITI gets higher performance with higher number of threads.



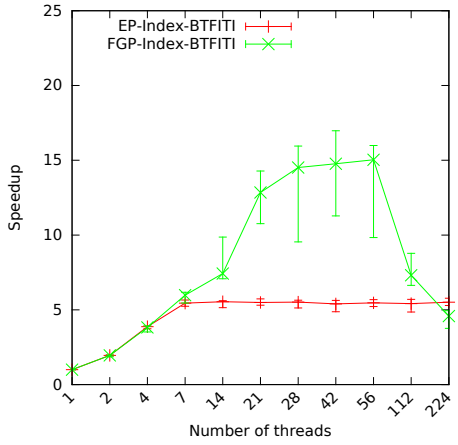
(a) Mining time, support = 0.005.



(b) Mining time, support = 0.011.



(c) Mining scalability, support = 0.005.

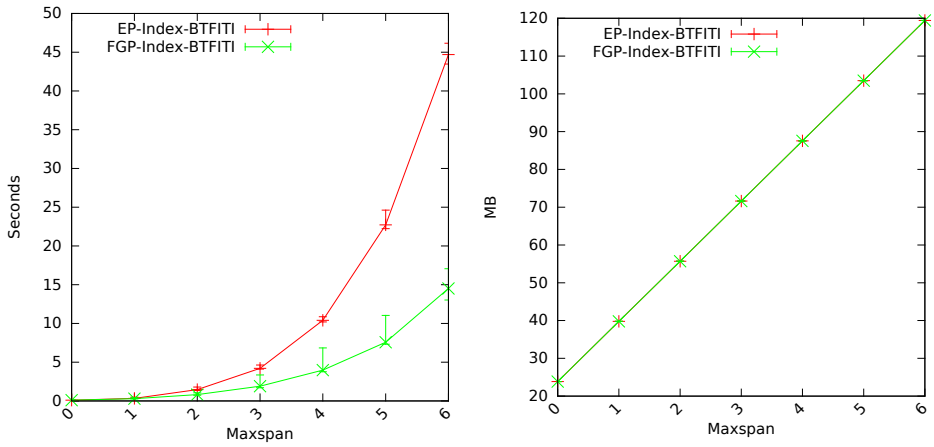


(d) Mining scalability, support = 0.011.

Figure 7.18: Mining performance on kosarak dataset with two different support values. Maxspan is set to 3.

7.5.4 Experiments on kosarak dataset with varying maxspan

The results regarding performance on kosarak with variations in maxspan are shown in Figure 7.19. Maxspan is varied from 0 to 6, while support is set to 0.008 and thread count to 28. We see that the memory usage increases linearly with maxspan, while the mining time increases exponentially. However, the mining time of FGP-Index-BTFITI doesn't increase as steeply as EP-Index-BTFITI.



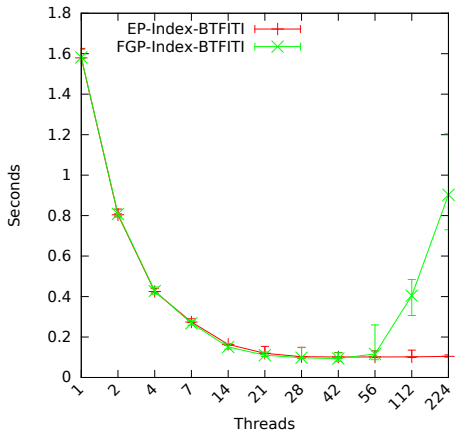
(a) Mining time. **(b)** Memory usage.

Figure 7.19: Performance results, using 28 threads, with varying maxspan values on kosarak dataset. Support is set to 0.008.

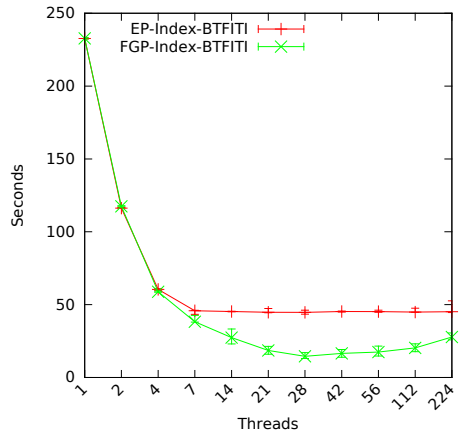
The graphs in Figure 7.20 show the performance results with two different maxspan values with varying threads. The minimum support value is set to 0.008. With maxspan set to zero both algorithms perform similarly, however with higher maxspan we see that FGP-Index-BTFITI clearly outperforms EP-Index-BTFITI.

7.5.5 Conclusion

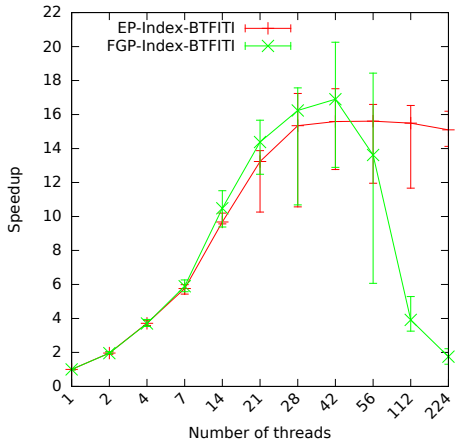
With the test results shown above we can safely conclude that FGP-Index-BTFITI outperforms EP-Index-BTFITI. The only disadvantage with FGP-Index-BTFITI is that it stops scaling at certain thread counts. In many of these situations it is not able to gain an advantage of hyper-threading. Nonetheless, it still dominates EP-Index-BTFITI because of its better thread utilization.



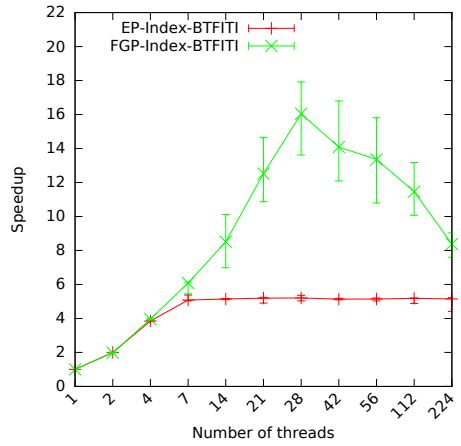
(a) Mining time, maxspan = 0.



(b) Mining time, maxspan = 6.



(c) Mining scalability, maxspan = 0.



(d) Mining scalability, maxspan = 6.

Figure 7.20: Mining performance on kosarak dataset with two different maxspan values. Support is set to 0.008.

7.6 Inter-transaction itemset mining tests on synthetic data

In order to see the performance impact during variations of number of transactions, average transaction length, and number of unique items, synthetic datasets were used. These were described in Section 7.2.3. Whenever a parameter was varied, the other values were set to fixed default values. The standard values used during testing were the following:

- Transaction count: 100000

- Average transaction length: 50
- Unique item count: 1000
- Maxspan: 3
- Support: 0.01
- Thread count: 28

Thread count was set to 28 since both tested algorithms, i.e. EP-Index-BTFITI and FGP-Index-BTFITI, perform equally with one thread. Since the effect upon the algorithms by varying maxspan and support were already tested with two different datasets in Section 7.5, it is not covered in this section. However, the graphs produced by these tests, on the synthetic data, can be found in Appendix C.

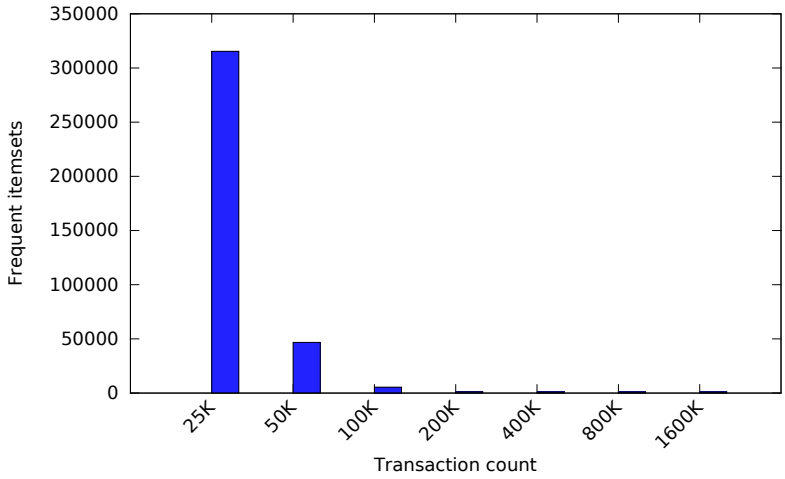


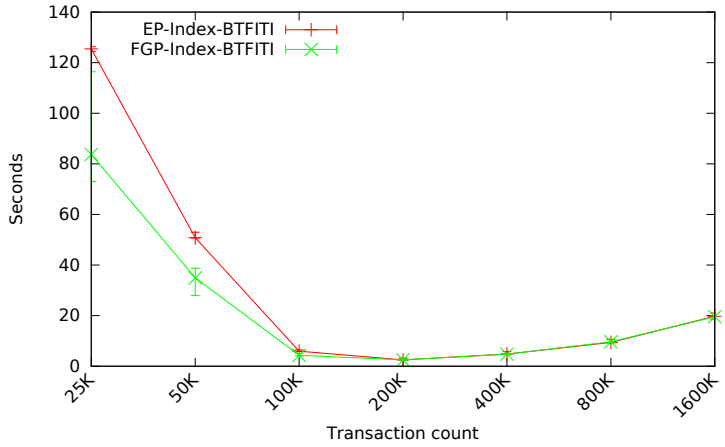
Figure 7.21: Number of frequent itemsets found with varying transaction count.

7.6.1 Varying number of transactions

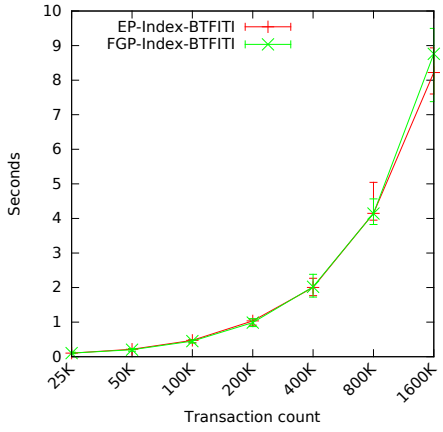
Tests were run on datasets with varying number of transactions. The transaction count varied from 25000 to 1600000, where the transaction count doubled for each test. In order to isolate this variable, it was made sure that the average transaction length and item count was constant for each of these datasets. Maxspan was set to 3, support to 0.01, and the thread count was set to 28.

Figure 7.22 shows the graphs for the mining time, preprocessing time, and memory usage. As expected the preprocessing time and memory usage increases with the transaction count. However, the graph representing the mining time shows some unexpected results. We see in the graph that the mining time is significantly greater with a low number of transactions than with high transaction count. The mining time decreases until around 100K to 200K,

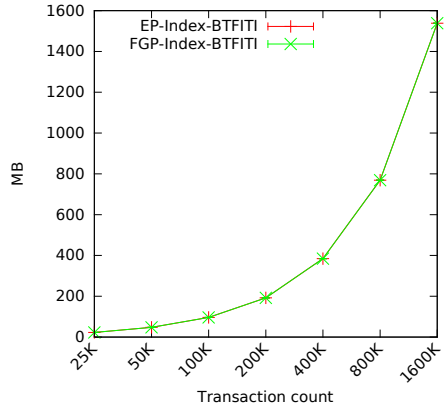
and after that starts slowly increasing. An explanation for this is the amount of frequent itemsets generated from the datasets. The histogram in Figure 7.21 shows that the amount of frequent itemsets generated is decreasing with increasing transaction count. While it is hard to see in the histogram, the lowest amount of frequent itemsets found was around 1000. This explains the large time consumption with the small datasets. Even though the number of frequent itemsets found is low the mining time will still increase with larger datasets. This is because the BitTable will get larger with more transactions, and hence support counting takes longer time.



(a) Mining time.

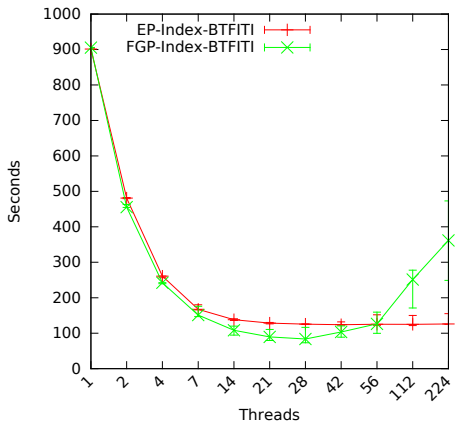


(b) Preprocessing time.

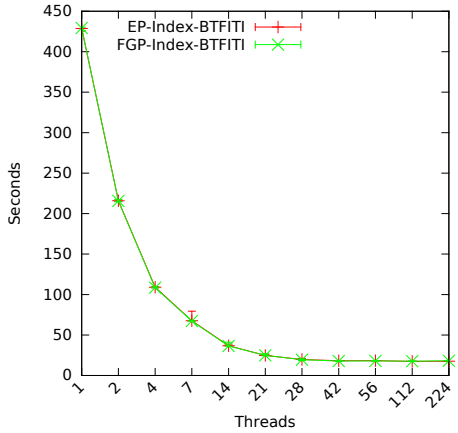


(c) Memory usage.

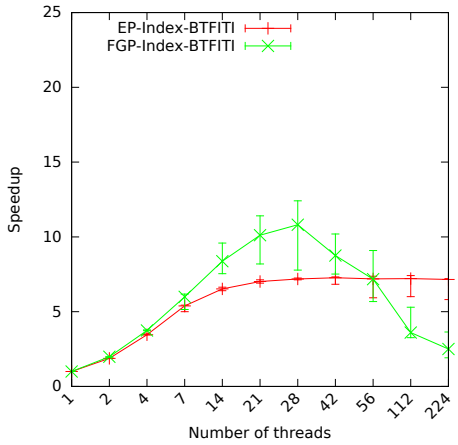
Figure 7.22: Performance results, using 28 threads, with varying transaction count. Maxspan = 3, support = 0.01.



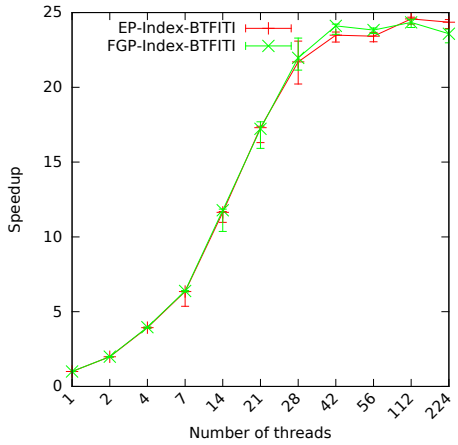
(a) Mining time, transaction count = 25 K.



(b) Mining time, transaction count = 1600 K.



(c) Mining scalability, transaction count = 25 K.



(d) Mining scalability, transaction count = 1600 K.

Figure 7.23: Mining performance with two different transaction counts. Maxspan = 3, support = 0.01.

In Figure 7.23 we see the mining time and scalability with two different datasets, one with 25 K number of transactions and another with 1600 K. We see that the scalability is significantly better for the larger dataset than the small one. There are two major reasons for this. First of the number of transactions mined is greater for the smaller dataset. This results in a larger amount of data that needs to be transferred through the system bus. The second reason is that there are some items in the small dataset that require greater amount of mining time. We can reach this conclusion by looking at how the graph from EP-Index-BTFITI stagnates at around 14 threads. The fine grained parallel mining procedure in

FGP-Index-BTFITI requires thread communication, and contains sequential parts that are not possible to parallelize. This again results in poorer scalability.

7.6.2 Varying average transaction length

Figure 7.24 shows the behaviour of the mining time with increasing average transaction length. The overall trend shows that the mining time increases. This is expected because longer transactions will result in more frequent itemsets found. We also see that the time gap between the two tested algorithms increases with transaction length. This means that the datasets with larger transaction lengths are containing some items that take longer time to mine than others. However, this result is probably occurring due to the distribution function used when creating the datasets.

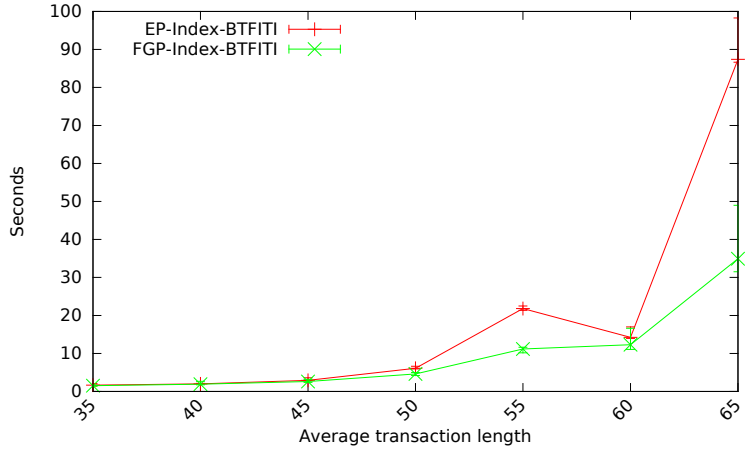
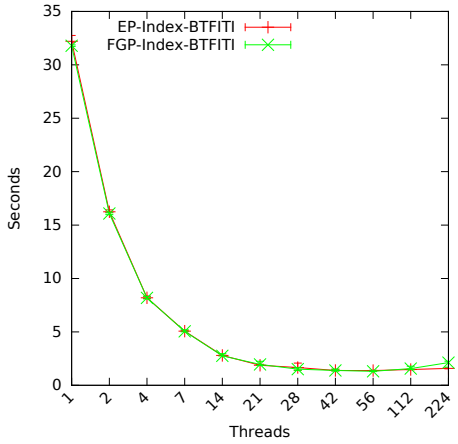
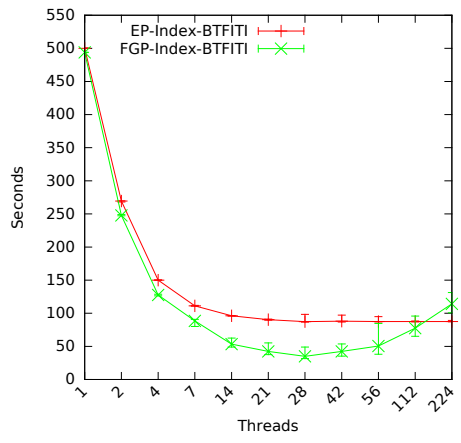


Figure 7.24: Mining time, using 28 threads, with varying average transaction length. Maxspan = 3, support = 0.01.

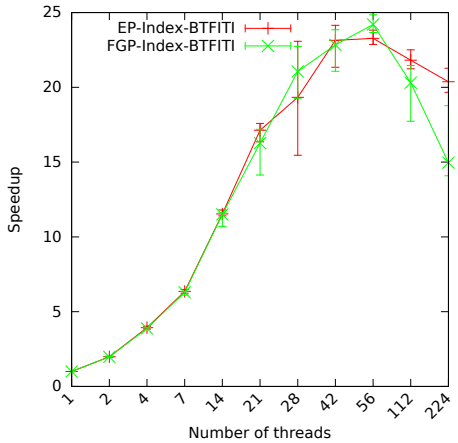
The graphs in Figure 7.25 are showing the mining time and scalability with two different average transaction lengths, i.e. 35 and 65. We can see that the scalability is better for a low number of transaction lengths. Nevertheless it is important to note that these results can be due to the dataset generator used. If a dataset generator was used that was able to distribute the items in such a way that each item took the same amount of time to mine, for each different average transaction length, there would not be such a great difference in scalability. However, with increasing average transaction length the number of frequent itemsets found will be greater. This will result in greater amount of data that needs to be written into memory, and will have a negative impact on scalability.



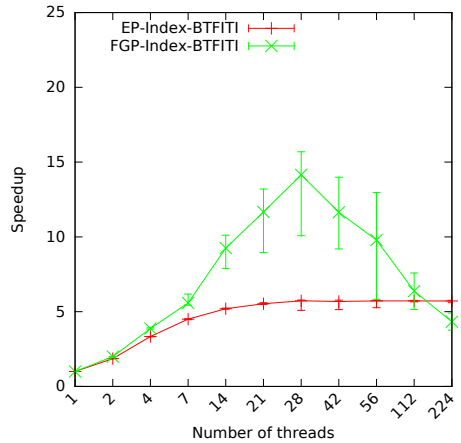
(a) Mining time, avg. trans. length = 35.



(b) Mining time, avg. trans. length = 65.



(c) Mining scalability, avg. trans. length = 35.



(d) Mining scalability, avg. trans. length = 65.

Figure 7.25: Mining performance with two different average transaction lengths. Maxspan = 3, support = 0.01.

7.6.3 Varying number of unique items

Tests with varying number of unique itemsets were run, where the item count was varied between 1000 and 64000. For each test the item count doubled. In Figure 7.27 we see the mining time and memory usage. To understand why we get these results we need to take a look at the data results produced by these tests.

The histogram in Figure 7.26 shows the amount of frequent extended itemsets and items

found. Note that the reason there can be more frequent extended items than itemsets is that an extended item with dimensional attribute greater than zero cannot alone represent a frequent extended itemset. This is according to the definition of an inter-transaction itemset given in Section 3.1.4.

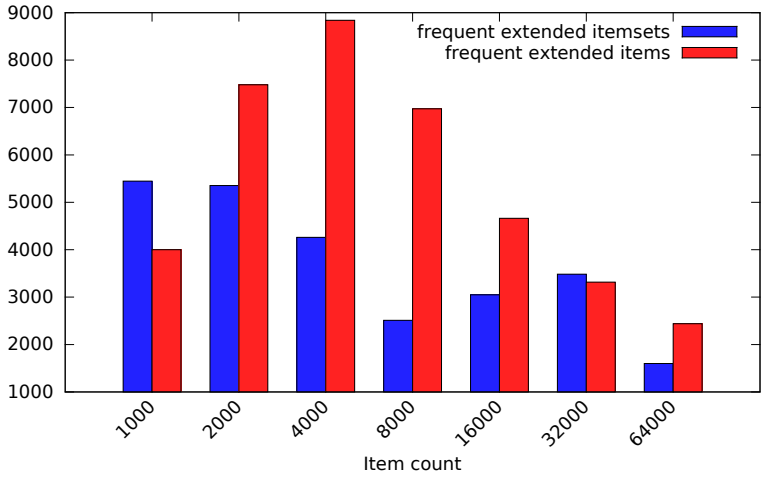
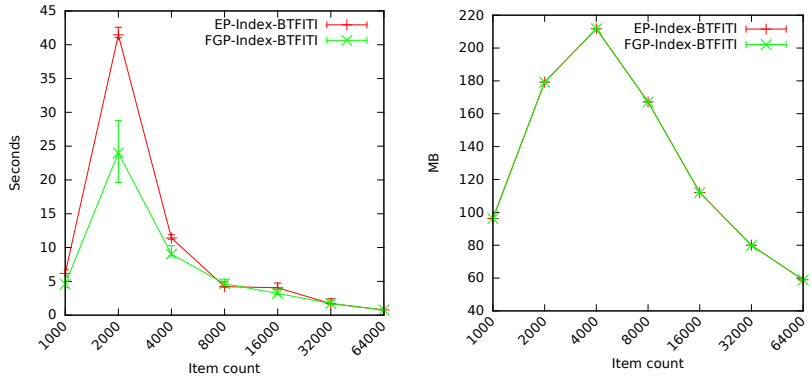


Figure 7.26: Number of frequent extended itemsets and items with varying item count.

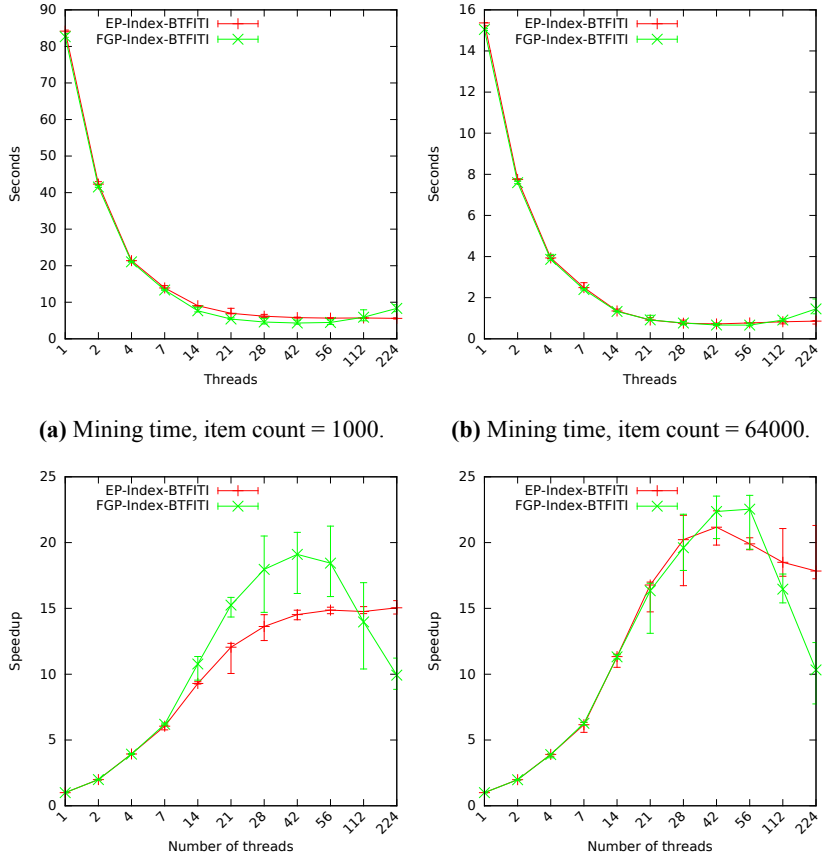
The memory usage graph in 7.27b is directly correlated with the amount of frequent extended items found. This is expected because the size of the BitTable is created according to the number of frequent items. When it comes to the mining time, we have to take into consideration the amount of frequent itemsets and items found. The number of iterations Index-BTFITI has to perform is the same amount as the number of frequent extended items. While the amount of time the algorithm has to perform in each iteration is depended on the amount of frequent extended itemsets. If we take these two variables into consideration and compare the histogram 7.26 and the mining time graph 7.27a, then the results are appropriate.

Figure 7.28 shows the mining time and scalability with two different numbers of unique items. We can see from these graphs that the algorithms have similar behaviour for both different item counts. However, it is seen that the scalability is greater where there are more unique items. The reason for this is that there are fewer frequent itemsets found, and hence less data written into system memory. There are also more frequent extended items found than itemsets, this leads to better load balancing. This is because each extended item can be mined independently by a thread. When there are fewer frequent itemsets found than there are frequent items, then this means that the mining time in each iteration is small compared to the overall time it takes to mine all frequent itemsets.



(a) Mining time. **(b)** Memory usage.

Figure 7.27: Performance results, using 28 threads, with varying item count. Maxspan = 3, support = 0.01.



(a) Mining time, item count = 1000. **(b)** Mining time, item count = 64000.

(c) Mining scalability, item count = 1000. **(d)** Mining scalability, item count = 64000.

Figure 7.28: Mining performance with two different item counts. Maxspan = 3, support = 0.01.

7.6.4 Conclusion

Multiple interesting results were found with these tests. The most interesting are the following:

- When the transaction count is increasing while the item count and average transaction length is constant, the scalability of the tested algorithms increases.
- With increasing average transaction length and constant transaction count and item count, the scalability decreases.
- When the item count increases, and average transaction length with the transaction count is constant, there are fewer itemsets generated and hence better scalability.

Chapter 8

Conclusion and further work

In this chapter the thesis is concluded. Firstly a short summary, of what was covered in this report, is presented in Section 8.1. The section also answers the research questions that were given in the introduction. Lastly, further work is explained in Section 8.2. This includes further work for frequent intra- and inter-transaction itemset mining methods.

8.1 Conclusion

This thesis began with an introduction to different hardware components as well as various parallel programming frameworks. Frequent intra- and inter-transaction itemset mining was presented with several existing algorithms. The existing state-of-the-art methods ShaFEM, CFP-Growth, and parallel Eclat were implemented. In addition a new novel frequent intra-transaction itemset mining method was described and implemented, which is a hybrid method of ShaFEM and CFP-Growth. During testing it was shown to be able to preserve the strengths of these algorithms. Two novel parallel frequent inter-transaction itemset mining methods were constructed and implemented, i.e. EP-Index-BTFITI and FGP-Index-BTFITI. These are believed to be the first parallel frequent inter-transaction itemset mining algorithms. The scalability of these algorithms showed that parallelization of inter-transaction itemset mining methods is definitely advantageous.

A thorough experimental procedure was conducted with variations in different algorithm and dataset parameters. Several synthetic datasets where generated with changes in number of transactions, average transaction length, and number of unique items. All implemented algorithms were executed on these datasets, and also on several real datasets. The real datasets consisted of existing datasets from the FIMI repository (see Section 7.2), and a new stock dataset that was created.

In Section 1.2 several research questions were presented. After implementing multiple intra- and inter-transaction itemset mining methods, and a thorough experimental procedure was executed, we are ready to answer these questions.

RQ1 How can we utilize multi-core processors efficiently in order to minimize mining time for frequent intra- and inter-transaction itemset mining algorithms?

It is important to minimize synchronization costs when constructing parallel algorithms. This is why most of the implemented algorithms are embarrassingly parallel, i.e. they contain no form of synchronization and are able to mine independently. In most cases this was seen as sufficient. However, regarding the implemented inter-transaction itemset mining method this was not always adequate. Therefore, a fine grained algorithm was implemented as well.

Another important aspect of algorithm parallelization is to minimize memory usage, in order to minimize data transfer. This is especially important with parallel algorithms, because multiple threads are trying to write and read data asynchronously.

RQ2 What impact do different dataset characteristics have on the performance for the focused algorithms?

The behaviour of the intra and inter algorithms were similar. All in all the main tendencies of the algorithms were the following:

- **Increase in transaction count**: Run time and memory usage increases. Scalability is either constant or increases.
- **Increase in average transaction length**: Run time and memory usage increases. Scalability decreases.
- **Increase in item count**: Run time and memory usage decreases. Scalability increases.

RQ3 With the increasing amount of cores available on modern computers, how does the data transfer time between primary memory and CPU affect the scalability?

The amount of data that needs to be transferred over the system bus has a huge scalability impact on the algorithms. What was shown, from the experiments conducted, was that the scalability was significantly smaller whenever a large quantity of frequent itemsets was found in a short time interval. The system bus was clearly the bottleneck. However, the viability of parallelization is still advantageous. This result indicates that memory efficient construction of data structures is important.

RQ4 How viable is it to apply compression techniques in order get a better cache utilization? Are the additional CPU cycles worth it?

This was mainly tested with CFP-Growth, and the novel hybrid method implemented. Whenever the datasets were large, the run time of these methods was smaller in comparison to ShaFEM which didn't use data compression. Hence, data compression techniques are viable whenever large datasets are being processed.

8.2 Further work

Regarding frequent intra-transaction itemset mining methods, further work includes examining the viability of finer grained parallelization techniques of the mining procedures for the different intra algorithms. This could be similarly conducted as in FGP-Index-BTFITI. In [28] a fine grained parallel Eclat algorithm was implemented. However, regarding FP-Growth type of algorithms, like ShaFEM and CFP-Growth, we believe that this has not yet been done.

When it comes to frequent inter-transaction itemset mining methods, it would be interesting to adapt the novel hybrid method of ShaFEM and CFP-Growth for this type of task. There are already existing FP-Growth type of algorithms for inter-transaction itemset mining, e.g. [20] and [6]. Nonetheless, a parallel algorithm that focuses on memory efficiency combined with dataset adaptability could be seen as beneficial. In addition an FP-Growth type of algorithm usually performs well with small support values, whereas Index-BTFITI mostly focuses on higher minimum support.

Appendices

Appendix A

Stocks in stock dataset

Company	Abbreviation	Stock exchange
Advanced Micro Devices, Inc.	AMD	NasdaqCM
American Express Company	AXP	NYSE
Apple Inc.	AAPL	NasdaqGS
AT&T, Inc.	T	NYSE
BP p.l.c.	BP	NYSE
Canon Inc.	CAJ	NYSE
Chevron Corporation	CVX	NYSE
Colgate-Palmolive Co.	CL	NYSE
Exxon Mobil Corporation	XOM	NYSE
FedEx Corporation	FDX	NYSE
Ford Motor Co.	F	NYSE
General Mills, Inc.	GIS	NYSE
Hewlett-Packard Company	HPQ	NYSE
Honda Motor Co., Ltd.	HMC	NYSE
Intel Corporation	INTC	NasdaqGS
International Business Machines Corporation	IBM	NYSE
Kellogg Company	K	NYSE
McDonald's Corp.	MCD	NYSE
Microsoft Corporation	MSFT	NasdaqGS
Nike, Inc.	NKE	NYSE
Pepsico, Inc.	PEP	NYSE
Sprint Corporation	S	NYSE
Texas Instruments Inc.	TXN	NasdaqGS
The Clorox Company	CLX	NYSE
The Coca-Cola Company	KO	NYSE
The Procter & Gamble Company	PG	NYSE
Toyota Motor Corporation	TM	NYSE
Verizon Communications Inc.	VZ	NYSE
Wal-Mart Stores Inc.	WMT	NYSE
Xerox Corporation	XRX	NYSE

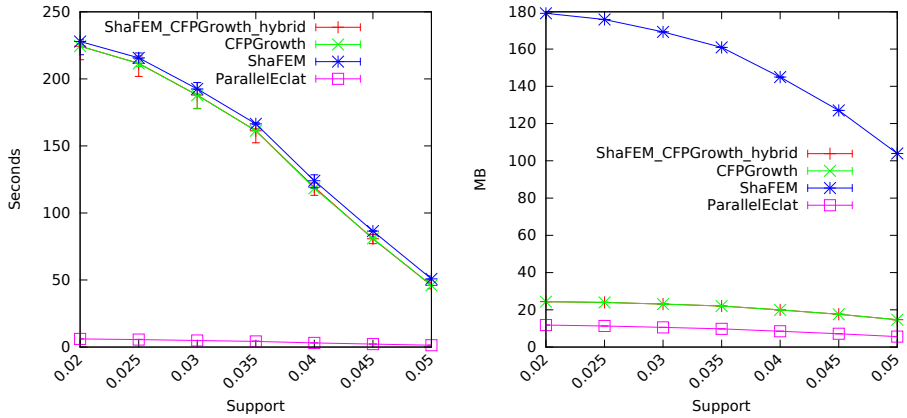
Table A.1: Stocks in stock dataset.

Appendix B

Varying support on synthetic data for intra algorithms

Graphs produced after running varying support values for the implemented intra algorithms are shown in this chapter. The dataset that was used is T100K-A50-I1000-M6, and the characteristics of this dataset can be found in Table 7.5.

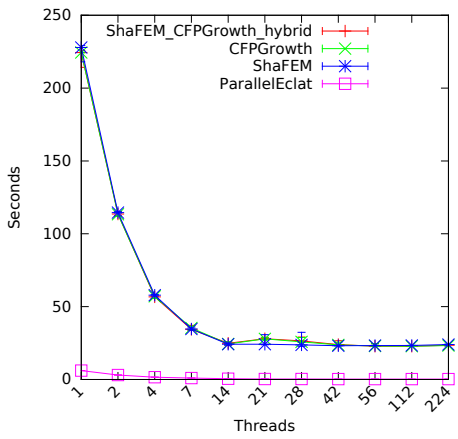
B.1 Varying support



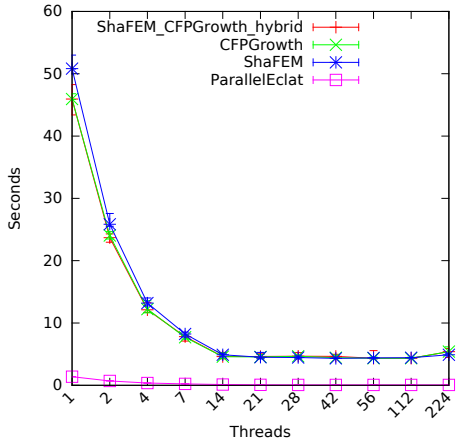
(a) Mining time.

(b) Memory usage.

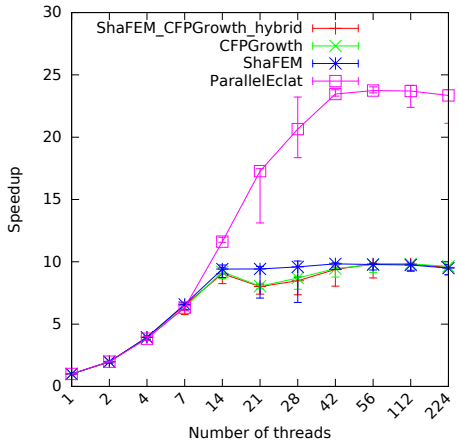
Figure B.1: Performance results, using one thread, with varying support values on T100K-A50-I1000-M6 dataset.



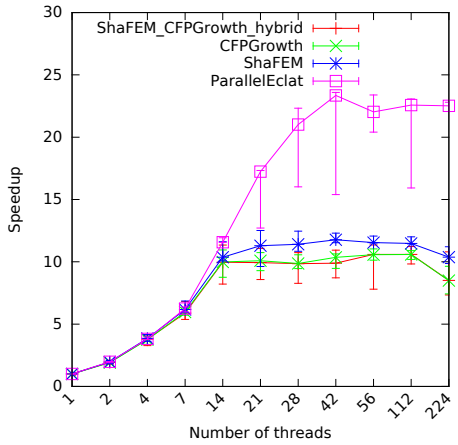
(a) Mining time, support = 0.02.



(b) Mining time, support = 0.05.



(c) Mining scalability, support = 0.02.



(d) Mining scalability, support = 0.05.

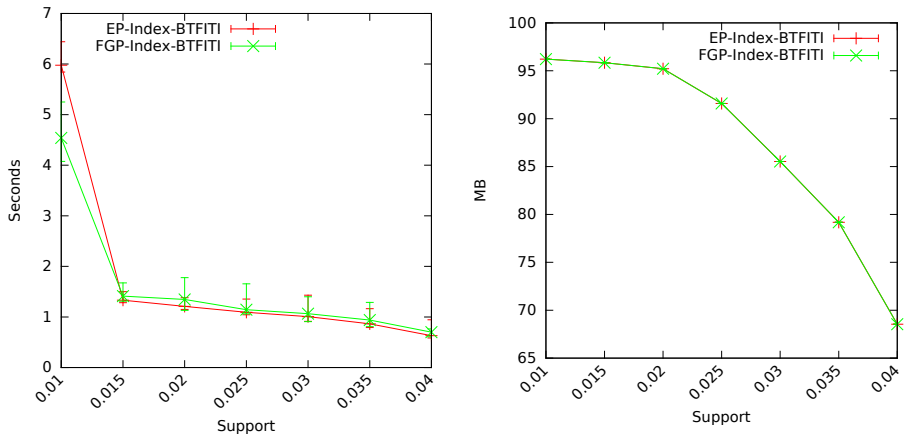
Figure B.2: Mining performance on T100K-A50-I1000-M6 dataset with two different support values.

Appendix C

Varying support and maxspan on synthetic data for inter algorithms

Graphs produced after running varying maxspan and support values for EP-Index-BTFITI and FGP-Index-BTFITI are shown in this chapter. The dataset that was used is T100K-A50-I1000-M6, and the characteristics of this dataset can be found in Table 7.5.

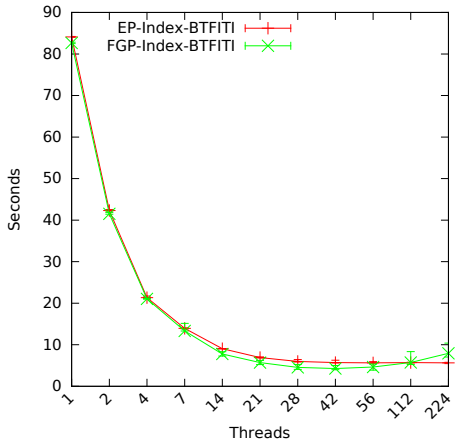
C.1 Varying support



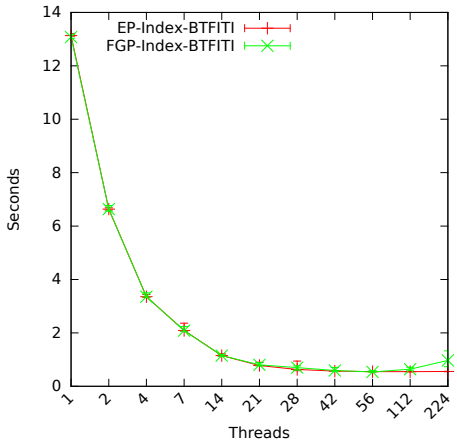
(a) Mining time.

(b) Memory usage.

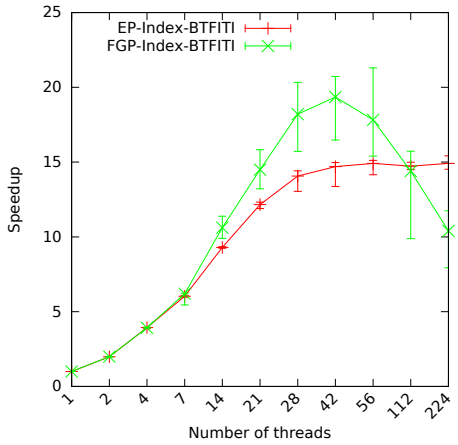
Figure C.1: Performance results, using 28 threads, with varying support values on T100K-A50-I1000-M6 dataset. Maxspan is set to 3.



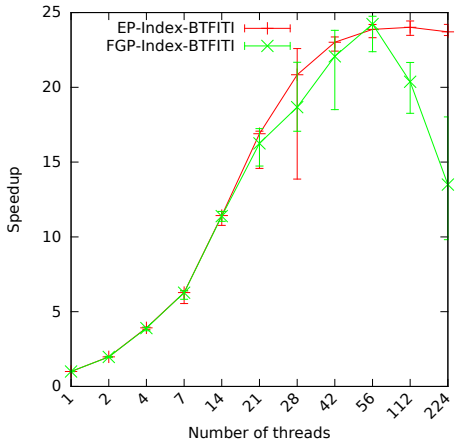
(a) Mining time, support = 0.01.



(b) Mining time, support = 0.04.



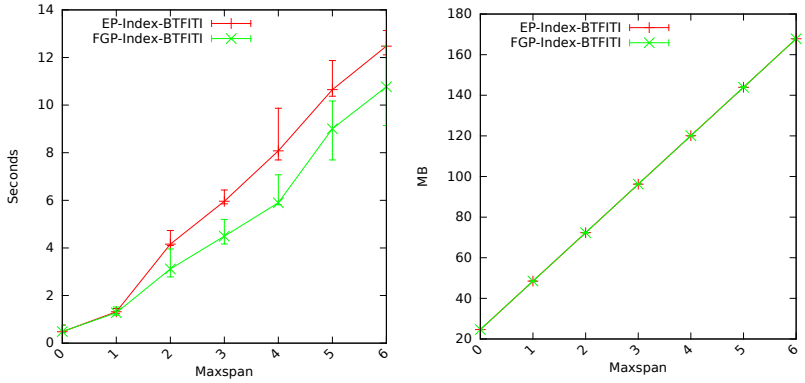
(c) Mining scalability, support = 0.01.



(d) Mining scalability, support = 0.04.

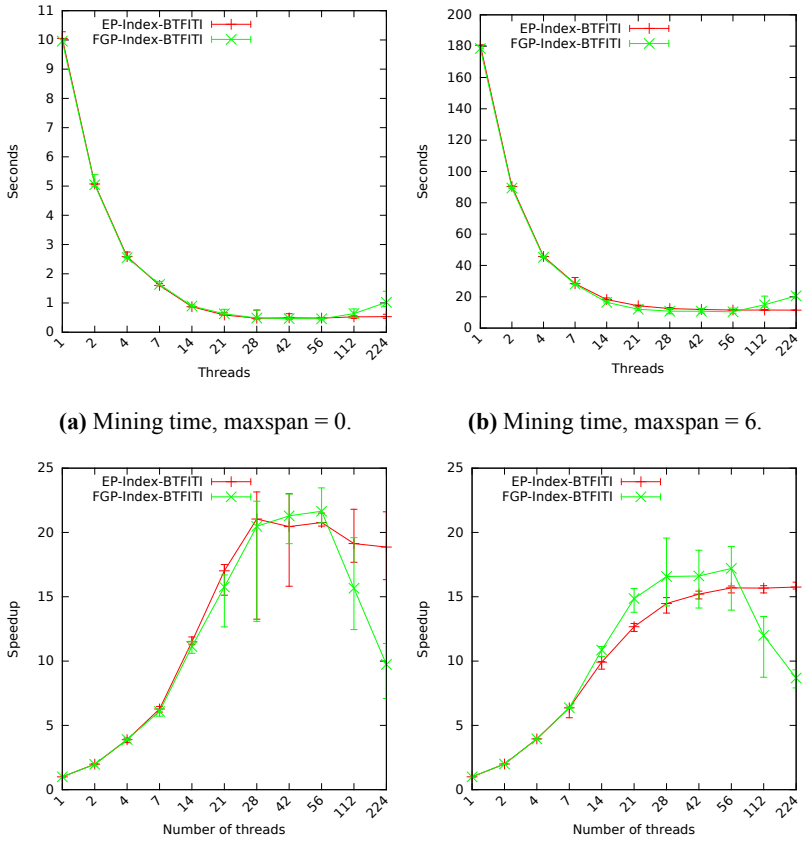
Figure C.2: Mining performance on T100K-A50-I1000-M6 dataset with two different support values. Maxspan is set to 3.

C.2 Varying maxspan



(a) Mining time. **(b)** Memory usage.

Figure C.3: Performance results, using 28 threads, with varying maxspan values on T100K-A50-I1000-M6 dataset. Support is set to 0.01.



(a) Mining time, maxspan = 0. **(b)** Mining time, maxspan = 6.

(c) Mining scalability, maxspan = 0. **(d)** Mining scalability, maxspan = 6.

Figure C.4: Mining performance on T100K-A50-I1000-M6 dataset with two different maxspan values. Support is set to 0.01.

Bibliography

[1] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules. In *Proceedings of VLDB '94*, pages 487--499, 1994.

[2] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of AFIPS '67*, pages 483--485, April 1967.

[3] F. Bodon. A Survey on Frequent Itemset Mining. Technical report, Department of Computer Science and Information Theory Budapest University of Technology and Economics, April 2006.

[4] D. R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.

[5] D. Chen, C. Lai, W. Hu, W. Chen, Y. Zhang, , and W. Zheng. Tree Partition based Parallel Frequent Pattern mining on Shared Memory Systems. In *Proceedings of IPDPS'06*, pages 313--313, 2006.

[6] H. Chhinkaniwala and P. Thilagam. InterTARM: FP-tree based Framework for Mining Inter-transaction Association Rules from Stock Market Data. In *Proceedings of ICCSIT' 08*, pages 513--517, 2008.

[7] L. Dagum and R. Menon. *OpenMP: an industry standard API for shared-memory programming*, volume 5. Computational Science & Engineering, IEEE, 1998.

[8] R. Elmasri and S. B. Navathe. *Database Systems: Models, Languages, Design, and Application Programming*. Pearson Education, 6 edition, 2011.

[9] F. Flouvat, F. D. Marchi, and J.-M. Petit. A new classification of datasets for frequent itemsets. *Journal of Intelligent Information Systems*, 34:1--19, February 2010.

[10] M. J. Flynn. Very high-speed computing systems. In *Proceedings of the IEEE*, volume 54, pages 1901--1909, December 1966.

[11] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley, February 1995.

[12] J. L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31:532--533, May 1988.

[13] J. Han, J. Pei, and Y. Yin. Mining Frequent Patterns without Candidate Generation. In *Proceedings of SIGMOD '00*, page 1–12, 2000.

[14] H. He, S. Feng, and J. Ren. An Algorithm of Mining Frequent Inter-Transaction Itemsets Based on BitTable and Index array. *Journal of Convergence Information Technology*, 6:260--267, October 2011.

[15] B. Jacob, S. W. Ng, and D. T. Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2007.

[16] A. J. Lee and C.-S. Wang. An efficient algorithm for mining frequent inter-transaction patterns. *Information Sciences*, 177:3453--3476, September 2007.

[17] H. Lu, L. Feng, and J. Han. Beyond intratransaction association analysis: mining multidimensional intertransaction association rules. *ACM Transactions on Information Systems (TOIS)*, 18:423--454, October 2000.

[18] H. Lu, J. Han, and L. Feng. Stock Movement Prediction and N-Dimensional Inter-Transaction Association Rules. In *Proceedings of SIGMOD-DMKD'98*, 1998.

[19] C. Lucchese, S. Orlando, R. Perego, and F. Silvestri. WebDocs: a real-life huge transactional dataset. In *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations*, 2004.

[20] S. Lühr, G. West, and S. Venkatesh. An Extended Frequent Pattern Tree for Inter-transaction Association Rule Mining. Technical report, Department of Computing, Curtin University of Technology, Perth, Western Australia, 2005.

[21] R. Micheloni, A. Marelli, and K. Eshghi. *Inside Solid State Drives (SSDs)*. Springer, 2013.

[22] K. Nørvåg, T. Øivind Eriksen, and K.-I. Skogstad. Mining Association Rules in Temporal Document Collections. *Foundations of Intelligent Systems*, 4203:745--754, 2006.

[23] P. Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufmann, 1 edition, 2011.

[24] J. S. Park, M.-S. Chen, and P. S. Yu. An effective hash-based algorithm for mining association rules. In *Proceedings of SIGMOD '95*, pages 175--186, 1995.

[25] A. Pietracaprina and D. Zandolin. Mining Frequent Itemsets using Patricia Tries. In *Proceedings of ICDM*, December 2003.

[26] B. Schlegel. *Frequent Itemset Mining on Multiprocessor Systems*. Dissertation, Technischen Universität Dresden, 2013.

[27] B. Schlegel, R. Gemulla, and W. Lehner. Memory-Efficient Frequent-Itemset Mining. In *Proceedings of EDBT/ICDT '11*, pages 461--472, 2011.

[28] B. Schlegel, T. Karnagel, T. Kiefer, and W. Lehner. Scalable Frequent Itemset Mining on Many-core Processors. In *Proceedings of DaMoN '13*, 2013.

[29] H. Smith. *Data Center Storage: Cost-Effective Strategies, Implementation, and Management*. Auerbach Publications, 1 edition, 2011.

[30] P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. 2006. Published by Addison Wesley.

[31] Y.-J. Tsay, T.-J. Hsu, and J.-R. Yu. FIUT: A new method for mining frequent itemsets. *Information Sciences*, 179:1724--1737, May 2009.

[32] A. K. Tung, H. Lu, J. Han, and L. Feng. Efficient mining of intertransaction association rules. *IEEE Transactions on Knowledge and Data Engineering*, 15:43--56, 2003.

[33] S. Vetter, Y. Aoyama, and J. Nakano. *RS/6000 SP: Practical MPI Programming*. IBM, 1 edition, August 1999.

[34] L. Vu and G. Alaghband. Mining Frequent Patterns Based on Data Characteristics. In *Proceedings of IKE'12*, pages 369--375, 2012.

[35] L. Vu and G. Alaghband. Novel Parallel Method for Mining Frequent Patterns on Multi-core Shared Memory Systems. In *Proceedings of DISCS-2013*, pages 49--54, 2013.

[36] C.-S. Wang and K.-C. Chu. Using a projection-based approach to mine frequent intertransaction patterns. *Expert Systems with Applications*, 38:11024--11031, September 2011.

[37] M. J. Zaki, S. Parthasarathy, and M. Ogihara. Parallel Algorithms for Discovery of Association Rules. *Data Mining and Knowledge Discovery*, 1(4):343--373, December 1997.

[38] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New Algorithms for Fast Discovery of Association Rules. In *Proceedings KDD-97*, 1997.

[39] F. Zhang, Y. Zhang, and J. D. Bakos. Accelerating frequent itemset mining on graphics processing units. *The Journal of Supercomputing*, 66:94--117, October 2013.