

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science and Engineering



Parallel algorithms for mining of frequent itemsets

by

Robert Kessl

A thesis submitted to
the Faculty of Electrical Engineering, Czech Technical University in Prague,
in partial fulfilment of the requirements for the degree of Doctor.

PhD programme: Electrical Engineering and Information Technology
Specialization: Computer Science and Engineering

February 2011

Thesis Supervisor:

Pavel Tvrdík
Department of Computer Systems
Faculty of Information Technologies
Czech Technical University in Prague
Kolejní 550/2
160 00 Praha 6
Czech Republic

Copyright © 2011 by Robert Kessl

Abstract and contributions

In the recent decade companies started collecting of large amount of data. Without a proper analyse, the data are usually useless. The field of analysing the data is called *data mining*. Unfortunately, the amount of data is quite large: the data do not fit into main memory and the processing time can become quite huge. Therefore, we need parallel data mining algorithms.

One of the popular and important data mining algorithm is the algorithm for generation of so called *frequent itemsets*. The problem of mining of *frequent itemsets* can be explained on the following example: customers goes in a store put into theirs baskets some goods; the owner of the store collects the baskets and wants to know the set of goods that are bought together in at least $p\%$ of the baskets.

Currently, the sequential algorithms for mining of frequent itemsets are quite good in the means of performance. However, the parallel algorithms for mining of frequent itemsets still do not achieve good speedup.

In this thesis, we develop a parallel method for mining of frequent itemsets that can be used for an arbitrary depth first search sequential algorithms on a distributed memory parallel computer. Our method achieves speedup of ≈ 6 on 10 processors. The method is based on an approximate estimation of processor load from a database sample – however it always computes the set of frequent itemsets from the whole database. In this thesis, we show a theory underlying our method and show the performance of the estimation process.

Keywords:

frequent itemset mining, parallel algorithms, approximate counting

Acknowledgements

I would like to thanks to Petr Savický. His advises contributed substantially to the quality and completeness of the thesis. I would also like to thanks to Pascal Fleury and Pavel Květoň for proofreading of this text.

This dissertation thesis was supported from the Czech Science Foundation, grant number GA ČR P202/10/1333.

Finally, my greatest thanks to my family for their support.

To Lenka and my daddy.

Contents

1	Introduction	1
2	Mathematical foundation	3
2.1	Basic notions	3
2.2	The monotonicity of support	5
2.3	The lattice of all itemsets	6
2.4	The use of the lattice of frequent itemsets in algorithms	7
2.5	Complexity of mining of frequent itemsets	12
2.5.1	Maximal frequent itemsets	12
3	Contribution of the thesis	14
4	Sequential algorithms for mining of FIs	15
4.1	Taxonomy of sequential algorithms	15
5	Existing parallel algorithms	17
5.1	Example of a shared memory algorithm	18
5.2	Apriori-based parallel DM algorithms	19
5.2.1	The Count distribution algorithm	19
5.2.2	The Fast Parallel Mining algorithm (FPM)	21
5.3	The asynchronous parallel FI mining algorithm	23
5.4	Eclat-based parallel algorithms	25
5.4.1	The bitonic scheduling	25
5.5	FPGrowth-based parallel algorithms	26
5.5.1	A trivial parallelization	26
5.5.2	The Parallel-FPTree algorithm	27
5.5.3	Summary and conclusion	28
6	Approximate counting by sampling	30
6.1	Estimating the support of an itemset from a database sample	31
6.2	Estimating the relative size of a PBEC	32
6.2.1	The coverage algorithm and its modification	33
6.2.2	The reservoir sampling	37
6.3	Estimating the size of a union of PBECs	42
7	Approximate parallel mining of MFIs	43
8	Proposal of a new DM parallel method	49
8.1	Detailed description of Phase 1	52
8.1.1	The modified coverage algorithm based sampling	54
8.1.2	The sampling based on the reservoir algorithm	58

8.2	Detailed description of Phase 2	61
8.3	Detailed description of Phase 3	65
8.4	Detailed description of Phase 4	68
8.5	The summary of the new parallel FIMI methods	69
8.5.1	The PARALLEL-FIMI-SEQ method	69
8.5.2	The PARALLEL-FIMI-PAR method	71
8.5.3	The PARALLEL-FIMI-RESERVOIR method	72
9	Execution of the Eclat algorithm in Phase 4	73
10	The database replication factor	77
10.1	Reduction of the replication factor	77
11	Experimental evaluation	80
11.1	Implementation and experimental setup	80
11.2	Databases	81
11.3	Evaluation of the estimate of the size of PBECs	82
11.4	Evaluation of the speedup	94
11.5	The evaluation of the database replication experiments	116
12	Conclusion and future work	121
12.1	Conclusion	121
12.2	Future work	121
13	Bibliography	123
14	Refereed publications of the author	127
15	Unrefereed publications of the author	128
A	Discrete probability distributions and tails	129
A.1	Chernoff bounds	129
A.2	Hypergeometric distribution and tails	129
A.3	Multivariate binomial distribution	131
B	Selected sequential algorithms	132
B.1	The Apriori algorithm	132
B.1.1	Prefix trie	135
B.1.2	TEST-SUBSET function and the COMPUTE-SUPPORT procedure using prefix trie	136
B.2	The FP-Growth algorithm	139
B.3	The Eclat Algorithm	145
B.3.1	Support counting	145
B.3.2	The depth-first search Eclat algorithm	145

B.4	Possible optimizations of the DFS sequential algorithms	148
B.4.1	The “closed itemsets” optimization	148
B.4.2	Ordering of items in DFS algorithms	148
B.4.3	The concept of diffsets	149
B.5	Discovering rules	150
C	Lists of abbreviations	151
D	Used symbols	152

List of Figures

2.1	Illustration of the mathematical notion	11
7.1	MFIs computed in parallel by a trivial parallelization of a DFS algorithm for mining of MFIs.	47
8.1	Start of the running example.	51
8.2	Example of Phase 1.	53
8.3	The workflow of the coverage algorithm based sampling	54
8.4	The workflow of the reservoir based sampling	58
8.5	Example of Phase 2.	64
11.1	Errors of the estimates of the sizes of union of PBECs, the T500I0.1P50PL10TL40 database	85
11.2	Errors of the estimates of the sizes of union of PBECs, the T500I0.1P50PL20TL40 database	86
11.3	Errors of the estimates of the sizes of union of PBECs, the T500I0.4P250PL20TL80 database	87
11.4	Errors of the estimates of the sizes of union of PBECs, the T500I0.4P50PL10TL40 database	88
11.5	Errors of the estimates of the sizes of union of PBECs, the T500I1P100PL20TL50 database	89
11.6	Probability of errors made in Phases 1 and 2	90
11.7	Probability of errors made in Phases 1 and 2	90
11.8	Probability of errors made in Phases 1 and 2	91
11.9	Probability of errors made in Phases 1 and 2	91
11.10	Probability of errors made in Phases 1 and 2	92
11.11	Probability of errors made in Phases 1 and 2	92
11.12	Probability of errors made in Phases 1 and 2	93
11.13	Speedup measured on the T500I0.1P100PL20TL50	108
11.14	Speedup measured on the T500I0.1P250PL10TL40	109
11.15	Speedup measured on the T500I0.1P50PL10TL40	110
11.16	Speedup measured on the T500I0.1P50PL20TL40	111
11.17	Speedup measured on the T500I0.4P250PL10TL120	112
11.18	Speedup measured on the T500I0.4P250PL20TL80	113
11.19	Speedup measured on the T500I0.4P50PL10TL40	114
11.20	Speedup measured on the T500I1P100PL20TL50	115
B.1	An example execution of the Apriori algorithm	134
B.2	An example of the prefix trie data structure	135
B.3	An example of an FP-Tree	142
B.4	6's conditional tree	143
B.5	The DFS tree of the execution of the Eclat algorithm.	146

List of Tables

6.1	The new notation used to describe the sampling algorithms.	30
11.1	Databases used for measuring of the speedup and used supports values for each database.	82
11.2	Sizes of $ \tilde{\mathcal{D}} $ and $ \tilde{\mathcal{F}}_s $ used in experiments	94
11.3	Best combintation of $ \tilde{\mathcal{D}} $ and $ \tilde{\mathcal{F}}_s $ for $P = 20$	96
11.4	Average speedup of the proposed methods for $ \tilde{\mathcal{D}} = 10000$ and $ \tilde{\mathcal{F}}_s = 19869$	97
11.5	Average speedup of the proposed methods for $ \tilde{\mathcal{D}} = 10000$ and $ \tilde{\mathcal{F}}_s = 26492$	98
11.6	Average speedup of the proposed methods for $ \tilde{\mathcal{D}} = 10000$ and $ \tilde{\mathcal{F}}_s = 33115$	99
11.7	Average speedup of the proposed methods for $ \tilde{\mathcal{D}} = 14450$ and $ \tilde{\mathcal{F}}_s = 13246$	100
11.8	Average speedup of the proposed methods for $ \tilde{\mathcal{D}} = 14450$ and $ \tilde{\mathcal{F}}_s = 19869$	101
11.9	Average speedup of the proposed methods for $ \tilde{\mathcal{D}} = 14450$ and $ \tilde{\mathcal{F}}_s = 26492$	102
11.10	Average speedup of the proposed methods for $ \tilde{\mathcal{D}} = 14450$ and $ \tilde{\mathcal{F}}_s = 33115$	103
11.11	Average speedup of the proposed methods for $ \tilde{\mathcal{D}} = 14450$ and $ \tilde{\mathcal{F}}_s = 39738$	104
11.12	Average speedup of the proposed methods for $ \tilde{\mathcal{D}} = 20000$ and $ \tilde{\mathcal{F}}_s = 19869$	105
11.13	Average speedup of the proposed methods for $ \tilde{\mathcal{D}} = 20000$ and $ \tilde{\mathcal{F}}_s = 26492$	106
11.14	Average speedup of the proposed methods for $ \tilde{\mathcal{D}} = 20000$ and $ \tilde{\mathcal{F}}_s = 33115$	107
11.15	Improvement of the database replication: kosarak	117
11.16	Improvement of the database replication: accidents	117
11.17	Improvement of the database replication: chess	118
11.18	Improvement of the database replication: connect	118
11.19	Improvement of the database replication: mushroom	119
11.20	Improvement of the database replication: pumsb_star	119
11.21	Improvement of the database replication: pumsb	120

List of Algorithms

1	The Multiple Local Frequent Pattern Trees algorithm	18
2	The APRIORI-COUNT-DISTRIBUTION algorithm	21
3	The FPM algorithm (Fast Parallel Mining algorithm)	23
4	The ASYNCHRONOUS-FI-MINING algorithm	24
5	The PARALLEL-ECLAT algorithm	26
6	The PARALLEL-FPTREE algorithm	28
7	The COVERAGE-ALGORITHM algorithm	34
8	The MODIFIED-COVERAGE-ALGORITHM algorithm	37
9	The SIMPLE-RESERVOIR-SAMPLING algorithm	38
10	The schema of a DFS algorithm for mining of MFIs.	44
11	The parallel schema of a DFS algorithm for mining of MFIs.	45
12	The PHASE-1-COVERAGE-SAMPLING-SEQUENTIAL algorithm	55
13	The PHASE-1-COVERAGE-SAMPLING-PARALLEL algorithm	56
14	The PHASE-1-RESERVOIR-SAMPLING algorithm	60
15	The PARTITION algorithm	62
16	The LPT-SCHEDULE algorithm	63
17	The PHASE-2-FI-PARTITIONING algorithm	64
18	The PHASE-3-DB-PARTITION-EXCHANGE algorithm	67
19	The PHASE-4-COMPUTE-FI algorithm	68
20	The PREPARE-TIDLISTS algorithm	75
21	the EXEC-ECLAT algorithm	75
22	The PHASE-4-ECLAT algorithm	76
23	The DB-REPL-MIN algorithm	79
24	The GENERATE-CANDIDATES function	133
25	The APRIORI algorithm	134
26	The INSERT-PREFIXTRIE procedure (prefix trie)	136
27	The TEST-SUBSET function (using prefix trie)	137
28	The TEST-SUBSET function (using prefix trie)	138
29	The COMPUTE-SUPPORT procedure	138
30	The INCREMENT-SUPPORT procedure	139
31	Function CONSTRUCT-FP-TREE	140
32	The FP-GROWTH algorithm	144
33	The FP-GROWTH-COMPUTATION algorithm	144
34	The ECLAT-DFS algorithm	145
35	The ECLAT-DFS-COMPUTATION algorithm	146
36	The GENERATE-ALL-RULES algorithm	150
37	The GENERATE-RULES algorithm	150

1 Introduction

Thanks to the automated data collection, companies collect huge amount of data. It is impossible to manually analyse such amounts of data. Therefore, automatic methods for analysis of the data are developed in *data mining*.

One of the important data mining tasks is the *mining of association rules* or *market basket analysis* [7]. The term market basket analysis comes from the first historical application. The market basket analysis comes from the need to analyse customer baskets of goods bought in a supermarket. The supermarket stores the list of items of the basket, called a *transaction*, into a database. The owner of the supermarket is interested in better shelf organization and wants to analyse the behaviour of customers in the supermarket from the database of the transactions. The result of the process are so called *association rules*, i.e. rules $X \Rightarrow Y$ such that X, Y are sets of goods.

The association rules are mined in a two step process:

1. Mine all *frequent itemsets* (FIs in short): find all sets of items that occur in a fraction of transactions at least of size *min-support**. The *min-support**, called the relative minimal support, is a parameter of the computation. An example of a frequent itemset is the set $U = \{\text{bread}, \text{milk}, \text{butter}\}$ with support $\text{Supp}(U) = 0.3$, i.e., the set U occurs in 30% of transactions.
2. Generate *association rules*: from the FIs generate all association rules with minimal confidence *min-confidence*. An example of an association rule is $\{\text{bread}, \text{milk}\} \Rightarrow \{\text{butter}\}$ with confidence 15%, i.e. the **butter** occurs in 15% of transactions that also contains **bread** and **milk**.

Because the mining of FIs is computationally expensive, we can only mine some subsets of FIs, e.g. the mining of *maximal frequent itemsets* (MFIs in short).

The problem of mining of FIs can be generalized to a wide variety of problems, called frequent substructure mining:

1. mining of frequent subgraphs,
2. mining of frequent sequences,
3. mining of frequent episodes.

An example of the frequent subgraph mining is the mining of the structure of proteins. Proteins are complicated molecules that can be viewed as a combinatorial graph. For a set of proteins, the task is to find frequent subgraphs. The information computed from the database can then help the chemists to search for proteins with similar effect, for example.

2 Mathematical foundation

2.1 Basic notions

Let $\mathcal{B} = \{b_i\}$ be a *base set* of items (items can be numbers, symbols, strings etc.). An arbitrary set of items $U \subseteq \mathcal{B}$ will be further called an *itemset*. Further, we need to view the baseset \mathcal{B} as an ordered set. The items are therefore ordered using an arbitrary order $<$: $b_1 < b_2 < \dots < b_n, n = |\mathcal{B}|$. Hence, we can view an itemset $U = \{b_{u_1}, b_{u_2}, \dots, b_{u_{|U|}}\}$, $b_{u_1} < b_{u_2} < \dots < b_{u_{|U|}}$, as an ordered set denoted by $U = (b_{u_1}, b_{u_2}, \dots, b_{u_{|U|}})$. If it is clear from context, we will not make difference between the set $\{b_{u_1}, b_{u_2}, \dots, b_{u_{|U|}}\}$ and the ordered set $(b_{u_1}, b_{u_2}, \dots, b_{u_{|U|}})$. We denote the i th smallest item of U ordered by the arbitrary order $<$ by $U[i] = b_{u_i}$. We denote the set of all itemsets, the powerset of \mathcal{B} , by $\mathcal{P}(\mathcal{B})$.

First, we define some necessary concepts:

Definition 2.1 (Transaction). *Let $U \subseteq \mathcal{B}$ be an itemset and $id \in \mathbf{Z}$ a natural number, used as an identifier. We call the pair (id, U) a transaction. The id is called the transaction id.*

A subset W of a transaction $t = (id, U)$ will be further denoted by $W \dot{\subseteq} t$, i.e., W is a subset of t if and only if $W \subseteq U$. A superset V of a transaction will be denoted similarly, i.e., $t \dot{\subseteq} V$. Because U can be viewed as an ordered set, we can also view the transaction t as an ordered set and denote i th item of t by $t[i]$.

Definition 2.2 (Database). *A database \mathcal{D} on \mathcal{B} (or database \mathcal{D} if \mathcal{B} is clear from context) is a sequence of transactions $t \dot{\subseteq} \mathcal{B}$. Each transaction t has an unique number in the database, called the transaction id.*

In our algorithms, we need to sample the database \mathcal{D} . A *database sample* is denoted by $\tilde{\mathcal{D}}$.

Definition 2.3 (Itemset cover and support). [9] *Let $U \subseteq \mathcal{B}$ be an itemset. Then the cover of U , denoted by $Cover_{\mathcal{B}}(U, \mathcal{D})$ in a database \mathcal{D} , is the subset of transactions $T = \{(id_i, V_i) | U \subseteq V_i\} \subseteq \mathcal{D}$. The number of transactions in $Cover_{\mathcal{B}}(U, \mathcal{D})$ is called the support of U in \mathcal{D} , denoted by $Supp(U, \mathcal{D}) = |Cover_{\mathcal{B}}(U, \mathcal{D})|$.*

We define the support as the number of transactions containing U , but in some literature, the relative support is defined by $Supp^*(U) = Supp(U)/|\mathcal{D}|$.

Definition 2.4 (Transaction id list). *Let $U \subseteq \mathcal{B}$ be an itemset, \mathcal{D} a database, and $T = \text{Cover}_{\mathcal{B}}(U, \mathcal{D})$ the itemset cover of U . The set $\mathcal{T}(U, \mathcal{D}) = \{id | \exists V, (id, V) \in T\}$ is called the transaction id list or tidlist in short.*

We omit \mathcal{D} from $\mathcal{T}(V, \mathcal{D})$, if clear from context.

Some algorithms use the concept of *vertical representation* of a database. The *vertical representation* of a database \mathcal{D} is the set of pairs $\{(\{b_i\}, \mathcal{T}(\{b_i\}, \mathcal{D})) | b_i \in \mathcal{B}\}$. The database described in Definition 2.2 is sometimes called the *horizontal representation*. The vertical representation holds the same information as the horizontal representation of the database \mathcal{D} . The set of all transaction IDs can be denoted by $\mathcal{T}(\emptyset)$.

Definition 2.5 (Transaction cover). *Let $T \subseteq \mathcal{D}$ be a set of transactions from the database \mathcal{D} . Then the cover of T , denoted by $\text{Cover}_{\mathcal{T}}(T, \mathcal{D})$, is the greatest itemset $U \subseteq \mathcal{B}$ such that for all $t \in T$ it holds that $U \dot{\subseteq} t$.*

Definition 2.6 (Frequent itemset). *Let \mathcal{D} be a database on \mathcal{B} , $U \subseteq \mathcal{B}$ an itemset, and $\text{min_support} \in \mathbf{Z}$ a natural number. We call U frequent in database \mathcal{D} if $\text{Supp}(U, \mathcal{D}) \geq \text{min_support}$.*

We can also define the frequent itemset using the relative support, denoted by min_support^* , $0 \leq \text{min_support}^* \leq 1$, i.e., an itemset is frequent iff $\text{Supp}^*(U, \mathcal{D}) \geq \text{min_support}^*$.

We will denote the set of all frequent itemsets as \mathcal{F} . In our algorithms, we need to sample the set \mathcal{F} . A sample of frequent itemsets is denoted by $\tilde{\mathcal{F}}_s$. In the text, we use \mathcal{D} and min_support (min_support^*) generally, but may be omitted if they are clear from the context.

Definition 2.7 (Maximal Frequent Itemset (MFI in short)). *Let \mathcal{D} be a database on \mathcal{B} , $U \subseteq \mathcal{B}$ an itemset, and $\text{min_support} \in \mathbf{Z}$ a natural number. We call U a maximal frequent itemset if $\text{Supp}(U, \mathcal{D}) \geq \text{min_support}$, and $\text{Supp}(V, \mathcal{D}) < \text{min_support}$ for any V such that $U \subsetneq V$.*

Definition 2.8 (Closure operator). *Let \mathcal{B} be a base set of items. Let $W \subseteq \mathcal{B}$, we define an operator $c : \mathcal{P}(\mathcal{B}) \rightarrow \mathcal{P}(\mathcal{B})$ as $c(W) = (\text{Cover}_{\mathcal{T}} \circ \text{Cover}_{\mathcal{B}})(W) = \text{Cover}_{\mathcal{T}}(\text{Cover}_{\mathcal{B}}(W))$.*

Definition 2.9 (Closed itemset). [38] *An itemset $U \subseteq \mathcal{B}$ is closed, if and only if $U = c(U)$.*

The concept of *closed itemsets* (CIs in short) can be used to reduce the size of the output of an FI algorithm. Additionally, the compound projection $\text{Cover}_{\mathcal{T}} \circ \text{Cover}_{\mathcal{B}}$ can be used

for optimization of depth-first search algorithms (DFS in short) for mining of FIs, see Appendix B.

Definition 2.10 (Association rule). *Let \mathcal{D} be a database on \mathcal{B} and $V, W \subseteq \mathcal{B}$ be itemsets such that $V \cap W = \emptyset$. Then the ordered pair (V, W) , written $V \Rightarrow W$, is called the association rule. The itemset V is called the antecedent and the itemset W is called the consequent.*

Definition 2.11 (Confidence). *Let \mathcal{D} be a database and $V \Rightarrow W$ an association rule. The confidence of $V \Rightarrow W$ is defined as:*

$$\text{Conf}(V, W, \mathcal{D}) = \frac{\text{Supp}(V \cup W, \mathcal{D})}{\text{Supp}(V, \mathcal{D})}$$

If it is clear from context, we omit the database \mathcal{D} from the notation.

The association rules are mined in a two step process: 1) mine all FIs $X = V \cup W, V \cap W = \emptyset$; 2) create association rules $V \Rightarrow W$ from the FIs mined in the first step, such that $\text{Conf}(V, W, \mathcal{D}) \geq \text{min_confidence}$. In our work, we consider only the first phase, i.e., we do not consider the task of creation of association rules from all frequent itemsets.

The values of min_support (or min_support^*) and min_confidence and a database \mathcal{D} are inputs to algorithms for the mining of association rules. These algorithms first find all frequent itemsets, using the min_support , and then generate association rules, using min_confidence .

For the purpose of the description of the parallel algorithm, we denote the number of processors by P . The i th processor, $1 \leq i \leq P$, is denoted by p_i .

At the start of the parallel algorithm, each processor p_i has a database partition D_i . Our parallel algorithms partitions the database at the beginning into disjoint database partitions D_i, D_j such that $\bigcup_i D_i = \mathcal{D}$, $D_i \cap D_j = \emptyset$, and $|D_i| \approx |\mathcal{D}|/P$. In our work, usually, processor p_i loads partition D_i into main memory.

2.2 The monotonicity of support

The basic property of frequent itemsets is the so called *monotonicity of support*. It is an important property for all FIs mining algorithms and is defined as follows:

Theorem 2.12 (Monotonicity of support). *Let $U, V \subseteq \mathcal{B}$ be two itemsets such that $U \subsetneq V$ and \mathcal{D} be a database. Then holds $Supp(U, \mathcal{D}) \geq Supp(V, \mathcal{D})$.*

Proof. If a set U is contained in transactions $\mathcal{T}(U)$, then a superset $V \supseteq U$ is contained in transactions $\mathcal{T}(V) \subseteq \mathcal{T}(U)$. \square

Corollary 2.13. *Let V be a frequent itemset, then all subsets $U \subseteq V$ are also frequent.*

Proof. Let $U, V \subseteq \mathcal{B}$ be two frequent itemsets such that $U \subsetneq V$. Then by using the argument from Theorem 2.12 it holds $\mathcal{T}(V) \subseteq \mathcal{T}(U)$ and therefore $Supp(V) \leq Supp(U)$. Because V is frequent, U must be frequent as well. \square

2.3 The lattice of all itemsets

Zaki [37] uses the set of all items, $\mathcal{P}(\mathcal{B})$, and the underlying lattice for description of DFS algorithms.

Definition 2.14. *Let P be finite ordered set, and let $S \subseteq P$. An element $X \in P$ is an upper bound (lower bound) of S if $s \leq X$ ($s \geq X$) for all $s \in S$. A least upper bound is called join and is denoted by $\bigvee S$, and a greatest lower bound, also called meet, of S is denoted $\bigwedge S$. The greatest element of P , denoted by \top , is called the top element, and the least element of P , denoted by \perp , is called the bottom element.*

We denote the join (meet) of two elements $X, Y \in P$ by $X \vee Y$ ($X \wedge Y$).

Definition 2.15. *Let \mathcal{L} be an ordered set, \mathcal{L} is called a join (meet) semilattice if $X \vee Y$ ($X \wedge Y$) exists for all $X, Y \in \mathcal{L}$. \mathcal{L} is called a lattice if it is both a join and meet semilattice. \mathcal{L} is complete lattice if $\bigvee S$ and $\bigwedge S$ exist for all subsets $S \subseteq \mathcal{L}$. An ordered set $M \subseteq \mathcal{L}$ is a sublattice of \mathcal{L} if $X, Y \in M$ implies $X \vee Y \in M$ and $X \wedge Y \in M$.*

It is well known that for a set S the powerset $\mathcal{P}(S)$ is a complete lattice. The *join* operation is the *set union operation* and *meet* the *set intersection operation*.

For any $\mathcal{S} \subseteq \mathcal{P}(\mathcal{B})$, \mathcal{S} forms a lattice of sets $(\mathcal{S}; \subseteq)$ if it is closed under finite number of unions and intersections.

Lemma 2.16. *The set of all frequent itemsets forms a meet semilattice.*

Proof. The result follows from Corollary 2.13 and the fact that $V \wedge W = V \cap W$. \square

Corollary 2.17. *The set of maximal frequent itemsets delimits the set of all frequent itemsets from above in the sense of set inclusion.*

Definition 2.18. *Let P be an ordered set, and let $X, Y, Z \in P$. We say X is covered by Y , denoted $X \sqsubset Y$, if $X < Y$ and $X \leq Z < Y$ implies $X = Z$, i.e., if there is no element Z of P with $X < Z < Y$.*

Definition 2.19. *Let \mathcal{L} be a lattice with bottom element \perp . Then $a_i \in \mathcal{L}$ is called an atom if $\perp \sqsubset a_i$. The set of atoms of \mathcal{L} is denoted by $\mathcal{A}(\mathcal{L})$.*

A set of all atoms of a lattice $\mathcal{L} = (\mathcal{P}(\mathcal{B}); \subseteq)$ is $\mathcal{A}(\mathcal{L}) = \mathcal{B}$.

2.4 The use of the lattice of frequent itemsets in algorithms

The lattice of frequent itemsets is the basic mathematical structure for the description of sequential FIs mining algorithms. There are many algorithms for mining of FIs, namely the Apriori algorithm, the Eclat algorithm, and the FP-Growth algorithm. All these algorithms are based on the theory described in this section. Additionally, to parallelize the sequential algorithms, we need to partition the set \mathcal{F} of all FIs into disjoint sets. The partitioning is also described in this section.

To decompose $\mathcal{P}(\mathcal{B})$ into disjoint sets, we need to order the items in \mathcal{B} . An equivalence relation partitions the ordered set $\mathcal{P}(\mathcal{B})$ into disjoint subsets called *prefix-based equivalence classes*:

Definition 2.20 (prefix-based equivalence class (PBEC in short)). *Let $U \subseteq \mathcal{B}$, $|U| = n$, be an itemset. We impose some order on the set \mathcal{B} and hence view $U = (u_1, u_2, \dots, u_n)$, $u_i \in \mathcal{B}$ as an ordered set. A prefix-based equivalence class of U , denoted by $[U]_\ell$, is a set of all itemsets that have the same prefix of length ℓ , i.e., $[U]_\ell = \{W = (w_1, w_2, \dots, w_m) | u_i = w_i, i \leq \ell, m = |W| \geq \ell, U, W \subseteq \mathcal{B}\}$*

To simplify the notation, we use $[W]$ for the prefix-based equivalence class $[W]_\ell$ iff $\ell = |W|$. Each $[W], W \subseteq \mathcal{B}$ is a sublattice of $(\mathcal{P}(\mathcal{B}), \subseteq)$.

Definition 2.21 (Extensions). *Let $U \subseteq \mathcal{B}$ be an itemset. We impose some order $<$ on the set $\mathcal{B} = (b_1, b_2, \dots, b_n)$ and view $U = (u_1, u_2, \dots, u_m)$, $u_i \in \mathcal{B}$, as an ordered set. The*

extensions of the prefix-based equivalence class $[U]$ is an ordered set $\Sigma \subseteq \mathcal{B}$ such that $U \cap \Sigma = \emptyset$ and for each $W \in [U]$ holds that $W \setminus U \subseteq \Sigma$. We denote the prefix-based equivalence class together with the extensions Σ by $[U|\Sigma]$.

For example, let have $\mathcal{B} = \{1, 2, 3, 4, 5\}$, a prefix $U = \{1, 2\}$, and the extensions $\Sigma = \{3, 5\}$. Then $[U|\Sigma] = [\{1, 2\}|\{3, 5\}] = \{\{1, 2, 3\}, \{1, 2, 5\}, \{1, 2, 3, 5\}\}$.

Proposition 2.22. *Let $U_i = \{b_i\}$, $b_i \in \mathcal{B}$ for all i , $1 \leq i \leq |\mathcal{B}|$, and $\Sigma_i = \{b | b > b_i; b, b_i \in \mathcal{B}\}$ then $[U_i|\Sigma_i]$ are disjoint.*

Proof. The reason is obvious: each $W \in [U_i|\Sigma_i]$ contains b_i and does not contain $b < b_i$. \square

Proposition 2.23. *Let $Q = \{(U_i, \Sigma_i)\}$ be a set such that $[U_i|\Sigma_i]$ are disjoint and $q = (V, \Sigma_V) \in Q$. Let $W_i = V \cup \{b_i\}$, $b_i \in \Sigma_V$ and $\Sigma_{W_i} = \{b | b_i < b; b_i, b \in \Sigma_V\}$ forms the PBECs $[W_i|\Sigma_{W_i}]$. Let have a new set of pairs $Q' = (Q \setminus \{q\}) \cup (\bigcup_i \{(W_i, \Sigma_{W_i})\})$. Then the pairs $(U'_i, \Sigma'_i) \in Q'$ forms disjoint PBECs $[U'_i|\Sigma'_i]$.*

Proof. It suffices to show that the new PBECs $[W_i|\Sigma_i]$ are disjoint and that the union $(\bigcup_i [W_i|\Sigma_i]) \cup \{V\} = [V|\Sigma_V]$.

The PBECs $[W_i|\Sigma_i]$ are disjoint (using the same argument as in Proposition 2.22), because each $W_i = V \cup \{b_i\}$, $b_i \in \Sigma_V$, contains one $b_i \in \Sigma_V$ and does not contain any $b \in \Sigma_V$ such that $b < b_i$.

Each $X \in [V|\Sigma_V]$ has the form $X = V \cup Y$, $Y \subseteq \Sigma_V$, $Y \neq \emptyset$. We can partition the sets Y on those having a prefix $b_1 \in \Sigma_V$, those having a prefix $b_2 \in \Sigma_V$, etc. But, this is exactly how the new PBECs $[W_i|\Sigma_{W_i}]$ were created. Since we have used all $b \in \Sigma_V$, it must be true that $(\bigcup_i [W_i|\Sigma_{W_i}]) \cup \{V\} = [V|\Sigma_V]$.

\square

We simplify the notation and omit the extensions if clear from context.

Lemma 2.24. *Let $W \subseteq \mathcal{B}$ be an itemset. The equivalence class $[W]$ is a sublattice of the lattice $(\mathcal{P}(\mathcal{B}), \subseteq)$.*

Proof. Let U, V be itemsets in class $[W]$, i.e., U, V share common prefix W . $W \subseteq U \cap V$ implies that $U \wedge V \in [W]$, and $U \vee V \in [W]$. Therefore, $[W]$ is a sublattice of $(\mathcal{P}(\mathcal{B}), \subseteq)$. \square

Definition 2.25. Let $U, W \subseteq \mathcal{B}$ and $[U], [W]$ be prefix-based equivalence classes. Then $[W]$ is a prefix-based equivalence subclass of $[U]$ if and only if $[W] \subsetneq [U]$.

Proposition 2.26. Let $W, U \subseteq \mathcal{B}$. If $[W]$ is a prefix-based equivalence subclass of $[U]$, then $U \subsetneq W$.

From Proposition 2.26, it follows that the prefix-based equivalence classes form a hierarchy. The hierarchy of classes is a tree, where each node corresponds to a prefix W and the children to the supersets $U_i \supseteq W, 1 \leq i \leq n$ for some $n \geq 1$, such that $|U_i| = |W| + 1$. The items $\Sigma = \bigcup_i U_i \setminus W, |\Sigma| = n$, are the *extensions* of $[W]$.

Further, we need to partition \mathcal{F} into n disjoint sets, denoted by F_1, \dots, F_n , satisfying $F_i \cap F_j = \emptyset, i \neq j$, and $\bigcup_i F_i = \mathcal{F}$. This partitioning can be done using the prefix-based equivalence classes. The prefix-based equivalence classes can be collated to a single partition: let have prefix-based equivalence classes $[U_l], (\bigcup_l [U_l]) \cup (\bigcup_l \mathcal{P}(U_l)) = \mathcal{F}, 1 \leq l \leq m$ and sets of indexes of the prefix-based equivalence classes $L_i \subseteq \{k | 1 \leq k \leq m\}, 1 \leq i \leq n$ such that $L_i \cap L_j = \emptyset$ and $\sum_i |L_i| = m$ then $F_i = \bigcup_{l \in L_i} ([U_l] \cap \mathcal{F})$. To create prefix-based equivalence classes, we partition the lattice into sublattices recursively. First, we partition the lattice using prefixes of size 1, i.e., $[(b_i)], b_i \in \mathcal{B}$. Then, we can pick an arbitrary class and partition it further on prefix-based equivalence classes with prefixes of size 2, etc. This recursive decomposition forms a depth-first search (DFS in short) expansion tree, see Example 2.1.

Definition 2.27 (Relative size of a PBEC). Let $[U|\Sigma]$ be a PBEC and a set of itemsets \mathcal{I} . The itemsets in \mathcal{I} are not necessarily frequent. We define the relative size of the PBEC as $\frac{|[U|\Sigma] \cap \mathcal{I}|}{|\mathcal{I}|}$.

By the set \mathcal{I} we usually mean the set of all frequent itemsets \mathcal{F} . But in Chapter 6, we also use other sets then \mathcal{F} .

The prefix-based equivalence classes decompose the lattice into smaller parts that can be processed independently in main memory. That is, for the computation of supports of itemsets in one prefix-based equivalence class, we start with the tidlists of the atoms and recursively construct the tidlists of itemsets belonging to that class by intersecting them. Due to this, the computation of support in different prefix-based equivalence classes is done independently. This is important, because this independence makes parallelization easier. Moreover, we can recursively decompose each equivalence class into smaller prefix-based equivalence subclasses.

For the computation of the support of an itemset $U \subseteq \mathcal{B}$, we can use the tidlists of items:

Lemma 2.28. *Let \mathcal{B} be a baseset and $U \subseteq \mathcal{B}, U = \bigcup_{u_i \in U} \{u_i\}, u_i \in \mathcal{B}$. Then the support of U can be computed by $Supp(U) = |\bigcap_{u_i \in U} \mathcal{T}(\{u_i\})|$.*

Proof. The support of $U = \{u_i | 1 \leq i \leq n, u_i \in \mathcal{B}\}$ is defined by $Supp(U) = |\mathcal{T}(U)|$, i.e., the number of transactions containing all the items u_i . Hence, the set of all transactions containing U is $\mathcal{T}(U) = \bigcap_i \mathcal{T}(u_i)$. \square

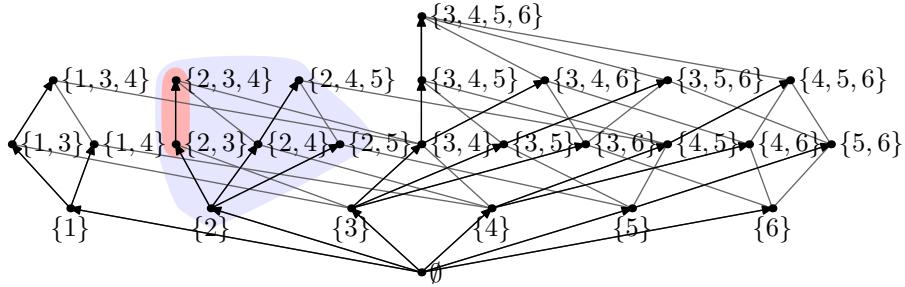
Corollary 2.29. *Let \mathcal{B} be a baseset and $U, W_i \subseteq \mathcal{B}, 1 \leq i \leq n$, for some $n \geq 1$ and $U = \bigcup_i W_i$ then $Supp(U) = |\bigcap_i \mathcal{T}(W_i)|$.*

It follows that for a prefix W and the extensions Σ we can compute the support of $W \cup U, U \subseteq \Sigma$ using the tidlists of items in Σ and the tidlist $\mathcal{T}(W)$.

Example 2.1: Illustration of the mathematical notion

Horizontal representation of the database \mathcal{D} Vertical representation of the database \mathcal{D}

TID	Transaction	itemset	{1}	{2}	{3}	{4}	{5}	{6}
1	{1, 2, 3, 4, 6}	1	1	1	1	1	2	1
2	{3, 5, 6}	3	4	2	3	5	5	2
3	{1, 3, 4}	4	6	3	5	6	4	
4	{1, 2, 6}	5	7	6	6	7	5	
5	{1, 3, 4, 5, 6}	6	8	7	7	8	9	
6	{1, 2, 3, 4, 5}	11	10	8	8	9	12	
7	{2, 3, 4, 5}	15	11	9	9	10	13	
8	{2, 3, 4, 5}		12	10	10	11	11	14
9	{3, 4, 5, 6}			11	11	12	12	15
10	{2, 4, 5}				15	12	13	
11	{1, 2, 4, 5}					13	14	
12	{2, 3, 4, 5, 6}					14	15	
13	{3, 4, 5, 6}							
14	{4, 5, 6}							
15	{1, 3, 4, 5, 6}							



The picture shows the set \mathcal{F} of the database \mathcal{D} with $\text{min_support} = 5$. The grey lines show the subset/superset relationship. The arrows show the DFS expansion tree.

- Prefix-based equivalence class $[(2)] \cap \mathcal{F} = \{\{2\}, \{2, 3\}, \{2, 4\}, \{2, 5\}, \{2, 3, 4\}, \{2, 4, 5\}\}$, marked in blue.
- Prefix-based equivalence class $[(2, 3)] \cap \mathcal{F} = \{\{2, 3\}, \{2, 3, 4\}\}$ is a subclass of $[(2)]$, marked in red.
- The DFS expansion tree is highlighted using thicker lines with arrows. The extensions of the tree node $\{2\}$ is the set of nodes $\{3, 4, 5\}$, i.e., nodes $\{\{2, 3\}, \{2, 4\}, \{2, 5\}\}$.
- The MFIs is the set $\mathcal{M} = \{\{1, 3, 4\}, \{2, 3, 4\}, \{2, 4, 5\}, \{3, 4, 5, 6\}\}$

2.5 Complexity of mining of frequent itemsets

The number of FIs is $2^{|\mathcal{B}|}$ in the worst case. In practice the number of FIs is very small in comparison to $2^{|\mathcal{B}|}$.

Let \mathcal{M} be the set of all maximal frequent itemsets. Let the size of the longest MFI be $s = \max\{|U| : U \in \mathcal{M}\}$. The complexity of mining of all FIs is exponential in s . Let $m_i \in \mathcal{M}$ be an MFI then the computational complexity of mining all FIs is $\mathcal{O}(\sum_i 2^{|m_i|}) = \mathcal{O}(2^s \cdot |\mathcal{M}|)$, where $\mathcal{O}(\cdot)$ denotes the big O notation.

For a good introductory text on the computational complexity, see [8]. The notation used in this Section is based on [8].

2.5.1 Maximal frequent itemsets

We need to assess the computational complexity of the task of enumeration of all MFIs. The NP-Completeness theory is mainly concerned about existence of a solution. Fortunately, there are other complexity classes, such as $\#P$ and $\#P$ -Complete [8] that concerns about counting the number of solutions. The *counting Turing machine* [8] is a standard non-deterministic Turing machine that has an additional tape on which the number of accepting computations is printed. The $\#P$ is a problem that is solved by the counting Turing machine in polynomial time. The $\#P$ -Complete problems are those problems on which all other problems from $\#P$ reduce [8].

The counting problem Π can be solved using an associated enumeration problem Π' : we enumerate the solutions using Π' and then count them. That is: if we know that a counting problem is $\#P$ -Complete then the associated enumeration problem must be NP-Hard [8].

A bipartite graph $G_1 = (U, V, E)$ is a subgraph of another bipartite graph $G_2 = (U', V', E')$ if $U \subseteq U'$, $V \subseteq V'$, and $E \subseteq E'$. A bipartite graph $G_3 = (U_3, V_3, E_3)$ is called *bipartite clique* if and only if $E_3 = U_3 \times V_3$, in particular interest are bipartite cliques that appears as subgraphs in another graph. We will omit E_3 from the notation of a bipartite clique. A *maximal bipartite clique* $G' = (U', V')$ in a given graph $G = (U, V)$ is a clique such that there is no bipartite clique $G'' = (U'', V'')$, $U' \subseteq U''$, and $V' \subseteq V''$.

There is an intuitive correspondence between cliques and transactions. Let have a bipartite graph $G = (\mathcal{B}, \mathcal{T}(\emptyset), E)$, such that $e = (b_i, t) \in E$ if the transaction t contains the item b_i , i.e., an edge of the graph represent the fact that an item is contained in a transaction.

The problem of counting the number of maximal bipartite cliques can be reduced to the problem of counting MFIs [33].

Theorem 2.30. [33] *The problem of counting the number of all bipartite cliques is $\#P$ -complete.*

The previous discussion give us an evidence that mining of maximal frequent itemsets is NP-hard.

3 Contribution of the thesis

In this dissertation thesis, we present a novel method for parallelization of an arbitrary algorithm for mining of all FIs. We are able to parallelize depth-first search algorithms, which is a hard task. Our method statically load-balance the computation using a “double sampling process”. The “double sampling process” first creates a database sample $\tilde{\mathcal{D}}$ and using $\tilde{\mathcal{D}}$ computes a sample of FIs $\tilde{\mathcal{F}}_s$. $\tilde{\mathcal{F}}_s$ is then used for partitioning of \mathcal{F} into disjoint sets F_i such that $\bigcup_i F_i = \mathcal{F}$. The input of the whole process is the database \mathcal{D} , each processor p_i loads a database partition D_i such that $\mathcal{D} = \bigcup_i D_i$, the minimal support *min-support**, and the sampling parameters or the size of the database sample $\tilde{\mathcal{D}}$ and the size of the sample of FIs $\tilde{\mathcal{F}}_s$.

The method consists of four phases: 1) creation of the database sample $\tilde{\mathcal{D}}$ and sample of FIs $\tilde{\mathcal{F}}_s$; 2) creation of the partitioning of \mathcal{F} ; 3) exchanging of database partitions among the processors; 4) the set \mathcal{F} is computed in parallel.

We present three variants of our new method:

1. the PARALLEL-FIMI-SEQ method based on MODIFIED-COVERAGE-ALGORITHM, see Section 6.2.1 and Chapter 8.
2. the PARALLEL-FIMI-PAR method based on parallel execution of MODIFIED-COVERAGE-ALGORITHM, see Section 6.2.1, Chapter 7, and Chapter 8.
3. the PARALLEL-FIMI-RESERVOIR method based on VITTER-RESERVOIR-SAMPLING, see Section 6.2.2 and Chapter 8.

The three variants differ in the way the sample $\tilde{\mathcal{F}}_s$ is constructed in Phase 1. We present theoretical results regarding the accuracy of the static load-balancing: see Corollary 6.5 of Theorem 6.4 and Section 6.3. We experimentally evaluate the theoretical results in Chapter 11. We show that the speedup of our method, in the case of PARALLEL-FIMI-RESERVOIR, is up to 13 on 20 processors. The results are valuable because we apply our method to very fast sequential algorithm: this forces us to make the process of statical load-balancing very efficient.

In order to make the execution of an arbitrary sequential algorithm for mining of FIs efficient, we show how to execute the ECLAT algorithm in parallel in Section 9. The execution of other arbitrary algorithm for mining of FIs is very similar to the algorithm shown in Section 9.

4 Sequential algorithms for mining of FIs

In this section, we show the taxonomy of the sequential algorithms for mining of FIs. The existing algorithms for mining of FIs together with their optimizations are described in Appendix B.

4.1 Taxonomy of sequential algorithms

We can view the algorithms from many different point of views. The basic division of the algorithms is by the way the lattice of FIs is searched on two classes: 1) *depth-first search*; 2) *breadth-first search*. The sequential algorithms can be designed to mine:

1. all frequent itemsets;
2. maximal frequent itemsets;
3. concise representation of frequent itemsets, e.g., *closed itemsets* (CIs in short), see [38].

The algorithms can be also divided by the database representation they use:

1. vertical representation;
2. horizontal representation.

An incomplete list of the algorithms sorted by the expansion strategy is the following:

1. Depth-first search: the Eclat algorithm [41], the FP-Growth algorithm [18], the H-mine [27] algorithm, etc.
2. Breadth-first search: the Apriori algorithm [7], the DCI (Direct Count and Intersect) algorithm [26], etc.

Since, in this dissertation thesis, we are focused on parallel FI mining algorithms, we skip the detailed description of the sequential algorithms. A reader is not familiar with the sequential algorithms, can see Appendix B for the description of the following algorithms:

1. The Apriori algorithm, Section B.1,
2. The FP-Growth algorithm, Section B.2,
3. The Eclat Algorithm, Section B.3.

5 Existing parallel algorithms

We consider basically two categories of parallel computers:

1. shared memory (SM in short) machines;
2. distributed memory (DM in short) machines.

Designing parallel algorithms for mining frequent itemsets on *shared memory machines* is relatively straightforward: the machine hardware supports easy parallelization of the problem. All the processors have access to the shared memory. If we store the database in the shared memory and use a simple stack splitting algorithm with arbitrary distributed termination detection and dynamic load-balancing, the results must be very good. The reason is that each processor has an access to the whole database and to the datastructures created by other processors. To our best knowledge, the parallel algorithms for shared memory machines use the datastructures created by the other processors only for reading. Therefore the memory pages containing the data structures are read by the processors and there is no need for invalidation of the memory pages.

Parallel mining of FIs on DM machines is a hard task for couple reasons:

1. The databases are usually quite large and we want to have the database distributed among the processors so we utilize the main memory of all nodes. Re-distribution of the database due to dynamic load-balancing, i.e., regular exchange of large database parts during the execution, is out of question due to the size of the database.
2. The problem of parallel mining of FIs is highly irregular. For the same reasons as in 1 the dynamic load-balancing is out of question.

In this chapter, we will briefly describe existing parallel algorithms for mining of FIs. In Section 5.1, we show an example of a shared-memory parallel algorithm. Section 5.2 describes Apriori-based DM algorithm, Section 5.3 describes an asynchronous algorithm that does not need a sequential FI mining algorithm, Section 5.4 describes Eclat-based DM algorithms, and Section 5.5 describes FP-Growth-based DM parallel algorithms.

During the whole chapter, we denote disjoint database partitions by $D_i, 1 \leq i \leq P$. D_i has always the size $|D_i| \approx |\mathcal{D}|/P$.

5.1 Example of a shared memory algorithm

An example of an algorithm that is designed for shared memory multiprocessors is the Multiple Local Frequent Pattern Tree algorithm (the MLFPT algorithm for short) [34]. The MLFPT algorithm is a parallelization of the FPGROWTH algorithm. We omit the details of the FPGROWTH algorithm in this section. The details of the FPGROWTH algorithm can be found in Appendix B. The algorithm works as follows:

Algorithm 1 The Multiple Local Frequent Pattern Trees algorithm

MLFPT(**In:** Database \mathcal{D} , **In:** Integer $min_support$, **Out:** Set \mathcal{F})

- 1: **for** all processors p_i **do-in-parallel**
 - /* Parallel FPTree creation */
 - 2: Load i -th partition D_i of the database \mathcal{D} into the main memory.
 - 3: Count local support for each item $b \in \mathcal{B}$.
 - 4: Exchange local supports with other processors to compute global support for each $b \in \mathcal{B}$ (hence an all-to-all broadcast takes place).
 - 5: Prune not frequent items, i.e., remove from \mathcal{B} all items $b \in \mathcal{B}$ such that $Supp(\{b\}, \mathcal{D}) < min_support$.
 - 6: Create FP-Tree T_i from D_i
 - 7: Barrier synchronization¹
 - /* Asynchronous FI mining phase */
 - 8: A modified FPGROWTH algorithm is started: the modified algorithm is almost the same as the original FPGROWTH algorithm but at the beginning it processes each FP-Tree T_i , creating a local FP-Tree that is used for further computations.
 - 9: the computed FIs are put into the set \mathcal{F}
 - 10: **end for**
-

The reported speedup of this algorithm is quite good, e.g., 53.35 at 64 processors, 29.22 at 32 processors, and 7.53 at 8 processors with running time ≈ 25000 seconds on single processor. The experiments used databases of size 1M, 5M, 10M, 25M, and 50M transactions.

5.2 Apriori-based parallel DM algorithms

The first sequential FI mining algorithm was the Apriori algorithm. We omit the details of the sequential Apriori algorithm in this section. The details of the sequential Apriori algorithm can be found in Appendix B.

There are many parallel algorithms based on the Apriori algorithm. The first algorithm was described by Agrawal et al. [6]. Agrawal proposed three parallel algorithms:

1. The Data Distribution algorithm.
2. The Count Distribution algorithm.
3. The Candidate Distribution algorithm.

Because Agrawal evaluated *the count distribution algorithm* as the best of these three algorithms, we will describe this algorithm, see Section 5.2.1. An improvement of the Apriori algorithm, the Fast Parallel Mining algorithm (the FPM algorithm in short) is described in Section 5.2.2.

5.2.1 The Count distribution algorithm

To describe the algorithm, we need to define the candidate itemset:

Definition 5.1 (candidate itemset on frequent itemset). *Let k be an integer, U be an itemset of size k , \mathcal{D} a database, and F_{k-1} the set of all frequent itemsets of size $k - 1$. If each subset $W \subseteq U, |W| = k - 1$ is frequent, $W \in F_k$, then U is called the candidate itemset. The set of all candidates of size k , denoted by C_k , is:*

$$C_k = \{U \mid U \subseteq \mathcal{B}, |U| = k, \text{ and for each } V \subsetneq U, |V| = k - 1 \text{ follows that } V \in F_{k-1}\}.$$

Since the computation of the support is the most computationally expensive part, it computes the support for *candidate itemsets* in parallel. In the following text, we denote the set of all frequent itemsets of size k by F_k and the superset of all FIs, called candidate itemsets, of size k by C_k , i.e., $F_k \subseteq C_k \subseteq \mathcal{P}(\mathcal{B})$.

In the description of the Count Distribution algorithm, we use:

1. The COMPUTE-SUPPORT procedure that computes the support of a set of itemsets from a database, see Algorithm 29 in Section B.1.
2. The GENERATE-CANDIDATES function that generates candidates from a set of frequent itemsets, see Algorithm 24 in Section B.1.

The understanding of the details of the COMPUTE-SUPPORT procedure and the GENERATE-CANDIDATES function are not important in order to understand the details of the Count Distribution algorithm. Therefore, we omit the details in this Section and leave them to Appendix B.

First, each processor p_i loads its part of the database, creates initial set of candidate itemsets $C_1 = \{\{b\} | b \in \mathcal{B}\}$, and computes its support in the database part D_i . The support of candidates can be computed using the COMPUTE-SUPPORT procedure. Since each processor knows \mathcal{B} , each processor has the same set of initial candidate itemsets. Then, the local supports of the initial candidates are broadcast, so each processor can compute the global support of the initial candidates. C_1 is pruned and each processor gets frequent itemsets of size 1, i.e., $F_1 = \{U | U \in C_1 \text{ and } \text{Supp}(U, \mathcal{D}) \geq \text{min_support}\}$. Since each processor has the same initial set of candidates and knows the global supports, then each p_i also has to have the same frequent itemsets of size 1. Thus, the first step is correct. All frequent itemsets of size k will be further denoted by F_k .

In step k , processors create a set of candidates C_k of size k from the previous frequent itemsets F_{k-1} of size $k - 1$. The set C_k can be computed using the GENERATE-CANDIDATES function. The candidates are generated by calling $C_k = \text{GENERATE-CANDIDATES}(F_{k-1})$. Since each processor p_i has the same set of frequent itemsets of size $k - 1$, each processor generates the same set of candidates. Then each processor p_i computes the local support for these candidates within its database part D_i and broadcasts the local supports to each other processor. Each processor updates local support, computing global support for all these candidates, and creates frequent itemsets of size k , i.e., $F_k = \{U | U \in C_k \text{ and } \text{Supp}(U, \mathcal{D}) \geq \text{min_support}\}$. Since each processor has correct frequent itemsets of size $k - 1$ at the beginning of step k , each processor has to have correct candidates C_k . Thus, after exchanging and updating local supports and pruning candidates, all processors have the correct frequent itemsets of size k . Note that only the support values of each $U \in C_k$ must be exchanged, because every processor has exactly the same set of candidates.

The pseudocode for the APRIORI-COUNT-DISTRIBUTION algorithm is shown in Algorithm 2:

Algorithm 2 The APRIORI-COUNT-DISTRIBUTION algorithm

APRIORI-COUNT-DISTRIBUTION(**In:** Database \mathcal{D} , **In:** Integer $min_support$, **Out:** Set \mathcal{F})

```

1:  $k \leftarrow 1$ 
2: for all processors  $p_i$  do-in-parallel
3:   Load the database part  $D_i$ .
4:   if  $k = 1$  then
5:     Generate initial candidates  $C_1 \leftarrow \{\{b_\ell\} | b_\ell \in \mathcal{B}\}$ .
6:   else
7:     Generate candidates  $C_k$  from frequent itemsets  $F_{k-1}$ , by calling  $C_k \leftarrow$ 
      GENERATE-CANDIDATES( $F_{k-1}$ ).
8:   end if
9:   Count the support for candidates  $C_k$  over local database partition using the
      COMPUTE-SUPPORT procedure.
10:  Broadcast the local support of the itemsets in  $C_k$  to each other processor (all-to-all
      broadcast).
11:  Prune candidates, creating  $F_k = \{U | U \in C_k, Supp(U, \mathcal{D}) \geq min\_support\}$ .
12:  if the set of frequent itemsets  $F_k$  is empty then
13:    return all generated frequent itemsets, i.e., return  $\mathcal{F} = \bigcup_k F_k$  and terminate.
14:  end if
15:   $k \leftarrow k + 1$ 
16: end for
```

5.2.2 The Fast Parallel Mining algorithm (FPM)

Cheung [11, 12] proposed two pruning techniques for the Count distribution algorithm. The pruning techniques leverage two important relationships between a partitioned database and frequent itemsets. Let \mathcal{D} be a database partitioned into n disjoint parts D_i of size $|D_i| \approx |\mathcal{D}|/P$, processor p_i having database part D_i . Cheung observed that if an itemset U is frequent in a database \mathcal{D} , i.e., $Supp^*(U, \mathcal{D}) \geq min_support^*$, then U must be frequent in at least one partition D_i , i.e., there exists i such that $Supp^*(U, D_i) \geq min_support^*$. Note that we are using the *relative supports*, instead of the *absolute supports*. Cheung proposed two kind of optimizations: 1) distributed pruning; 2) global pruning.

1) *Distributed pruning*: uses an important relationship between frequent itemsets and the partitioned database: *every (globally) frequent itemset in the whole database \mathcal{D} must also be (locally) frequent on some processors in the database part D_i .*

If an itemset U is globally frequent (i.e. $Supp^*(U, \mathcal{D}) \geq min_support^*$) and locally frequent on some processor p_i (i.e. $Supp^*(U, D_i) \geq min_support^*$), then U is called *gl-frequent*. We will use $GL_{k(i)}$ to denote the gl-frequent itemsets of size k at p_i . As in the Apriori Count-Distribution algorithm, we denote the set of all FIs of size k computed in step k by F_k . Note that $\forall i, 1 \leq i \leq P, GL_{k(i)} \subseteq F_k$.

Lemma 5.2. [12] *If an itemset U is globally frequent, then there exists a processor p_i such that U and all its subsets are gl-frequent at processor p_i .*

For the next theorem, we need a function that creates the set of candidates:

$$CG_{k(i)} = \{U | U \subseteq \mathcal{B}, |U| = k, \text{ and for each } V \subsetneq U, |V| = k-1 \text{ follows that } V \in GL_{k-1(i)}\}.$$

$CG_{k(i)}$ can be computed from $LG_{k(i)}$ using the algorithm GENERATE-CANDIDATES by calling $CG_{k(i)} = \text{GENERATE-CANDIDATES}(GL_{k-1(i)})$, see Appendix B for Algorithm 24.

It follows from Lemma 5.2 that if $U \in F_k$, then there exists a processor p_i , such that all its subsets of size $k-1$ are gl-frequent at processors p_i , i.e., they belong to $GL_{k-1(i)}$.

Theorem 5.3. [12] *For every $k > 1$, the set of all frequent itemsets of size k , F_k , is a subset of $F_k \subseteq CG_{(k)} = \bigcup_{i=1}^n CG_{k(i)}$, where $CG_{k(i)} = \{U | U \subseteq \mathcal{B}, |U| = k, \text{ and for each } V \subsetneq U, |V| = k-1 \text{ follows that } V \in GL_{k-1(i)}\}$.*

In [12] it is shown that CG_k , which is a subset of the Apriori candidates, could be much smaller than the number of the Apriori candidates.

2) *Global pruning*: after the supports of all itemsets are exchanged among the processors, the local support counts $Supp(U, D_i)$ are also available for all processors. Let $|U| = k$. At each partition D_i , the monotonicity principle holds for all itemsets, i.e., $Supp(U, D_i) \leq Supp(V, D_i)$ iff $V \subsetneq U$. Therefore the local support $Supp(U, D_i)$ is bounded by

$$\maxsupp(U, D_i) = \min_V \{Supp(V, D_i) | V \subsetneq U, \text{ and } |V| = |U| - 1\}$$

from above, i.e., $Supp(U, D_i) \leq \maxsupp(U, D_i)$. Because the global support $Supp(U, \mathcal{D}) = \sum_{1 \leq i \leq P} Supp(U, D_i)$ is the sum of its local support counts at all the processors, the value:

$$\sum_{1 \leq i \leq P} \text{maxsupp}(U, D_i)$$

is an upper bound of $\text{Supp}(U, D_i)$. If $\sum_{1 \leq i \leq P} \text{maxsupp}(U, D_i) < \text{min_support}^* \times |\mathcal{D}| = \text{min_support}$, then U can be pruned away. The pseudocode of the FPM algorithm is shown in Algorithm 3:

Algorithm 3 The FPM algorithm (Fast Parallel Mining algorithm)

FPM(**In:** Database \mathcal{D} , **In:** Set \mathcal{B} , **In:** Integer min_support , **Out:** Set \mathcal{F})

- 1: **for** all processors p_i **do-in-parallel**
 - 2: Compute the candidate sets $CG_{(k)} = \bigcup_{i=1}^P \text{GENERATE-CANDIDATES}(GL_{k-1(i)})$.
(distributed pruning)
 - 3: Apply global pruning to prune the candidates in CG_k .
 - 4: Scan partition D_i to find out the local support counts $\text{Supp}(U, D_i)$ for all remaining candidates $U \in CG_k$.
 - 5: Exchange $\{\text{Supp}(U, D_i)\}$ with all other processors to find out the global support counts $\text{Supp}(U, \mathcal{D})$.
 - 6: Compute $GL_{k(i)} = \{U \in CG_k | \text{Supp}^*(U, D) \geq \text{min_support}^* \times |D| \text{ and } \text{Supp}^*(U, D_i) \geq \text{min_support}^* \times |D_i|\}$ and exchange the result with other processors.
 - 7: **end for**
 - 8: **return** $\mathcal{F} \leftarrow \bigcup_{i=1}^P GL_{k(i)}$
-

5.3 The asynchronous parallel FI mining algorithm

Veloso [31] proposed another parallelization of the frequent itemset mining process. This algorithm is based on the fact that if we know MFIs, we are able to mine all frequent itemsets that are subsets of MFIs asynchronously.

Each processor p_i reads its partition of the database D_i and computes the local support for all items in D_i . By exchanging the local supports the processors gets the support of all items in \mathcal{D} .

The algorithm uses the fact that if an itemset is frequent, it must be frequent in at least one partition D_i . Every processor p_i then finds all MFIs in its local database partition D_i

and broadcasts them, together with the support, to other processors. Because the MFIs are MFIs computed using D_i , the processors makes global MFIs. Now the processors know the boundaries of \mathcal{F} (in the whole database) and can proceed in a top-down fashion to compute the support of all itemsets. At the end, the processors exchange counts of the itemsets and prunes infrequent itemsets.

The pseudocode of the algorithm is shown in Algorithm 4:

Algorithm 4 The ASYNCHRONOUS-FI-MINING algorithm

ASYNCHRONOUS-FI-MINING(**In:** Database \mathcal{D} , **In:** Integer $min_support$, **Out:** Set \mathcal{F})

```

1: for all processors  $p_i$  do-in-parallel
   /* Phase 1: computation of MFIs */
2:   Read its local database partition  $D_i$ .
3:   Compute all local MFIs, denoted by  $M_i$ .
   /* Phase 2 */
4:   Broadcast  $M_i$  (hence an all-to-all broadcast takes place).
5:   Compute  $\bigcup_{1 \leq i \leq P} M_i$ .
   /* Phase 3 (every node has  $\bigcup_{1 \leq i \leq P} M_i$ ). */
6:   Enumerate itemsets  $U \subseteq m, m \in M_i$  in a top-down fashion.
   /* Phase 4 (reduction of results) */
7:   Perform sum-reduction operation and removes itemsets  $U, Supp(U) \leq min\_support$ ,
      i.e. processor  $p_i$  sends its frequent itemsets to  $p_{i+1}$  and the last processor removes
      all infrequent itemsets.
8: end for
```

The authors in [31] reports that the speedup range from 5 to 10 on 16 processors. Unfortunately, the paper [31] is missing a table of speedups, therefore we have estimated the speedup from graphs of the running time. Additionally, the problem is that in [31] there is no mention to the algorithm used as a base for the computation of speedup, i.e., a sequential algorithm that is used for computation of the speedup of the method. If the used sequential algorithm is the Apriori algorithm, then we have to argue that the Apriori algorithm itself is slow and the speedup could be much worse if the execution time of the parallel algorithm is compared with some other, quicker, sequential algorithm.

5.4 Eclat-based parallel algorithms

5.4.1 The bitonic scheduling

Zaki et. al. [40] proposed a parallelization of the Eclat algorithm [41]. The algorithm is similar to our method in the sense that it partitions \mathcal{F} into prefix-based equivalence classes. However, it uses the *bitonic scheduling* [39], a heuristic for scheduling the prefix-based classes on the processors that is not able to capture the real size of each prefix-based equivalence class.

The bitonic scheduling works this way: each PBEC with n atoms, see Definition 2.19, is assigned a weight $\binom{n}{2}$, and the equivalence classes are assigned to processors p_i using a best-fit algorithm. The best-fit algorithm is in fact the same algorithm, we use for assigning of the prefix-based equivalence classes, see Algorithm 16 in Section 8.2 and Graham [16] for reference. The problem with this heuristic is that it does not capture the real size of the equivalence classes. This algorithm achieves speedups of $\approx 2.5\text{--}10.5$ on 24 processors, $\approx 2\text{--}10$ on 16 processors, $\approx 1.4\text{--}8$ on 8 processors, and $\approx 3\text{--}3.5$ on 4 processors. The experiments were performed on databases generated by the IBM generator with average transaction size 10 and database sizes 800k, 1.6M, 3.2M, and 6.4M transactions. Our hypothesis is that in many real-world applications, the average size of maximal potentially frequent item is much bigger than 10.

Algorithm 5 The PARALLEL-ECLAT algorithm

PARALLEL-ECLAT(**In:** Database \mathcal{D} , **In:** Integer $min_support$, **Out:** Set \mathcal{F})

- 1: **for** all processors p_i **do-in-parallel**
 - /* Initialization phase */
 - 2: Scan local database partition D_i .
 - 3: Compute local support for all itemsets of size 2,
denoted by $C_2 = \{U | U \subseteq \mathcal{B}, |U| = 2\}$.
 - 4: Broadcast the local support of itemsets in C_2 ,
creating global support of itemsets in C_2 .
 - /* Transformation Phase */
 - 5: Partition C_2 into equivalence classes
 - 6: Schedule the equivalence classes on all processors p_i
 - 7: Transform local database into vertical form
 - 8: Send to each other processor the tidlists, needed by other process for computation
of its assigned portion of the equivalence classes.
 - /* FI computation phase */
 - 9: All processors computes frequent itemsets from the assigned equivalence classes.
 - /* Final Reduction Phase */
 - 10: Aggregate results and output FIs into \mathcal{F}
 - 11: **end for**
-

5.5 FP-Growth-based parallel algorithms

The FP-Growth algorithm is an important sequential FI mining algorithm. In this section, we show two parallel algorithms based on the FP-Growth algorithm. The details of the FP-Growth algorithm are described in Section B.2.

5.5.1 A trivial parallelization

A trivial distributed-memory parallelization of the FP-Growth algorithm is proposed in [28]. The parallelization uses dynamic load-balancing. The idea is that each processor creates its local FP-Tree, broadcast the local FP-Tree to other processors (resulting in global FP-Tree on every processor) and assign prefix-based equivalence classes to processors using a hash function. The problem is that the amount of assigned work is unpredictable

and the resulting computational load could be highly unbalanced. The solution to the unbalanced computation is the use of dynamic load-balancing.

The dynamic load-balancing uses *minimal path-depth* [28] threshold to estimate the granularity of a subtree. We define the *path-depth* as the maximal length of a path from the root to a list in an FP-Tree. Since the path-depth of the FP-Tree is non-increasing during the computation, the dynamic load-balancing works as follows: if a processor finishes its assigned work, it starts requesting work from other, busy, processors. The busy processors sends part of their assigned work to the requesting processor if and only if the path-depth is bigger than the *minimal path-depth* threshold.

The result of this approach is that the aggregate memory is not used efficiently. [28] reports speedup of $\approx 4\text{--}20$ on 32 processors on a *single* database with 100K and maximal potentially frequent itemset size were set to 25, and 20. transactions. However, the speedup of 20 is achieved in only two experiments from five. In the rest of the experiments, the maximum speedup is ≈ 8 at 30 processors. The maximum execution time of the sequential algorithm was ≈ 900 seconds.

5.5.2 The Parallel-FPTree algorithm

The PARALLEL-FPTREE is proposed by Javed and Khokhar in [19]:

Algorithm 6 The PARALLEL-FPTREE algorithm

PARALLEL-FPTREE(**In:** Database D , **In:** Itemset \mathcal{B} , **In:** Integer $min_support^*$)

- 1: **for** all processors p_i **do-in-parallel**
 - 2: Scan its assigned partition and computes the support for single items sets based on items in the local database.
 - 3: Exchange the local supports and compute the global support for each itemset with each other processor.
 - 4: Sort the global support for the single itemsets and discards all the non-frequent items.
 - 5: Scan the assigned partition again and constructs a local FP-Tree.
 - 6: The header table is partitioned into P disjoint sets and each processor is assigned to mine frequent patterns for distinct set of item.
 - 7: Identify the information from its local tree needed by other processors. The prefix paths of the single itemsets assigned to a processor in step 4 constitute the complete information needed for the mining step. This is identified using a bottom up scan of the local FP-Tree.
 - 8: The information in step 6 is communicated in $\log P$ rounds employing a recursive merge of the tree structure over processors. For example, processor p_i communicates with processor $p_{P/2+1}^r \% P$ in round r where $1 \leq i \leq P$ and $0 \leq r \leq \log P$. At the end of each round, a processor simply unpacks the received information into its local FP-tree and prepares a new message for the next round of the merge.
 - 9: Mine FIs in its PBECs with prefix of size 1 constructed from the assigned itemsets.
 - 10: **end for**
-

The problem with this approach is obvious: the computation must be unbalanced. However, in [19] present different results: an almost linear speedup. The reason for such results could be the very small running time of the algorithm (up to couple of seconds) and very small database (10000 transactions).

5.5.3 Summary and conclusion

We have described parallel algorithms based on the Apriori, the FP-Growth and the Eclat algorithm. The biggest problem of the Apriori algorithm is its slowness and memory consumption. Therefore, parallelization of the Apriori algorithm is not practical. The biggest advantage of the parallel Apriori algorithms is that they use the aggregate memory

of the cluster efficiently. That is: every processor has a database partition of size $|\mathcal{D}|/P$. The parallel Apriori algorithms usually works in iterations that correspond to the sequential Apriori iterations, except that they are done in parallel. The authors claim that static load-balancing is used. We must argue that the load is not statically balanced at all: parallel execution of the sequential iterations should not be considered as static load-balancing .

The parallelizations of the Eclat and the FP-Growth algorithms use an estimate of the sizes of the prefix-based classes. However, the estimates are very simple and do not capture the real amount of work assigned to the processors. Dynamic load-balancing on distributed-memory parallel computers also does not work. The reason is that the computation is quite fast and exchanging large portions of the database among processors can be quite time-consuming.

Parallelizations of other algorithms than the Apriori algorithm do not achieve good speedups. But, the Apriori itself is quite slow.

The best solution should:

1. distribute the computation: computation time of each processors should be approximately the same.
2. distribute the database: the database should be distributed among the processors so that processor p_i has database partition of size $|\mathcal{D}|/P$.

All parallel algorithms based on the Apriori algorithm have the previous two properties. However, the sequential Apriori algorithm is very slow and very memory consuming. Therefore, we would like to parallelize faster and less memory consuming algorithm with the described properties.

It seems that the major difference in the sequential algorithms is in the used datastructures. Therefore, we would like have a universal parallelization method for an arbitrary sequential algorithm.

6 Approximate counting by sampling

Our method for parallel mining of FIs is based on efficient estimation of the number of FIs in a given prefix-based equivalence class (PBEC in short). Unfortunately, as discussed in Section 2.5, counting the number of FIs is $\#P$ -Complete problem, i.e., computing the number of FIs in a given PBEC is also $\#P$ -Complete. Fortunately, to estimate the (relative) number of FIs in a PBEC, we do not need to count the relative number of FIs exactly. We can estimate the relative sizes of FIs in PBECs with a sampling algorithm that approximately counts the relative number of FIs in a PBEC. Further, when talking about the relative (absolute) size of a PBEC, we always mean the relative (absolute) number of FIs in the PBEC.

In this chapter, we show two sampling algorithms for estimating the relative size of a given PBEC, or a set of PBECs. Both sampling algorithms need the support of an itemset U to decide whether an itemset is frequent or not. This decision can be made with an *estimate* of the support of U . The support of U is estimated using a *database sample*. Therefore, in this chapter, we also derive minimum sample size needed to achieve a small error of the support estimate with high probability. Finally, we bound the error of the size of a PBEC estimated using $\tilde{\mathcal{F}}_s$.

To describe the sampling methods and our method for parallel mining of FIs, we need an additional notation. We extend the notation introduced in Section 2. A database is denoted by \mathcal{D} and a database sample is denoted by $\tilde{\mathcal{D}}$. The set of all FIs *computed from* \mathcal{D} is denoted by \mathcal{F} . The set of all FIs *computed from* $\tilde{\mathcal{D}}$ is denoted by $\tilde{\mathcal{F}}$. The set of all MFIs *computed from* \mathcal{D} is denoted by \mathcal{M} . The set of all MFIs *computed from* $\tilde{\mathcal{D}}$ is denoted by $\tilde{\mathcal{M}}$. $\tilde{\mathcal{M}}$ is the upper bound on $\tilde{\mathcal{F}}$ in the sense of the set inclusion, i.e., for all $U \in \tilde{\mathcal{F}}$ there exists an $m \in \tilde{\mathcal{M}}$ such that $U \subseteq m$. The sample of $\tilde{\mathcal{F}}$, which is computed using $\tilde{\mathcal{F}}$ or $\tilde{\mathcal{M}}$, is denoted by $\tilde{\mathcal{F}}_s$. The additional notation is summarized in Figure 6.1.

Symbol	Description
$\tilde{\mathcal{D}}$	A database sample computed from \mathcal{D} .
$\tilde{\mathcal{F}}$	The set of all FIs computed from $\tilde{\mathcal{D}}$.
$\tilde{\mathcal{M}}$	The set of all MFIs computed from $\tilde{\mathcal{D}}$. $\tilde{\mathcal{M}}$ also bounds the set $\tilde{\mathcal{F}}$, i.e., for each $U \in \tilde{\mathcal{F}}$ exists $m \in \tilde{\mathcal{M}}$ such that $U \subseteq m$.
$\tilde{\mathcal{F}}_s$	A sample of $\tilde{\mathcal{F}}$, computed using $\tilde{\mathcal{F}}$ or $\tilde{\mathcal{M}}$.

Table 6.1: The new notation used to describe the sampling algorithms.

This chapter is organized as follows: first, in Section 6.1 we show how to estimate support of an itemset from a database sample. In Section 6.2 we show the two methods for estimating the size of a PBEC.

6.1 Estimating the support of an itemset from a database sample

The time complexity of the decision whether an itemset U is frequent or not is in fact the complexity of computing the *relative support* $Supp^*(U, \mathcal{D})$ in the input database \mathcal{D} . If we know the approximate relative support of U , we can decide whether U is frequent or not with certain probability. We can estimate the relative support $Supp^*(U, \mathcal{D})$ from a database sample $\tilde{\mathcal{D}}$, i.e., we can use $Supp^*(U, \tilde{\mathcal{D}})$ instead of $Supp^*(U, \mathcal{D})$.

An approach of estimating the relative support of U was described by Toivonen [30]. Toivonen uses a *database sample* $\tilde{\mathcal{D}}$ for the *sequential* mining of frequent itemsets and for the efficient estimation of theirs supports. Toivonen's algorithm works as follows: 1) create a database sample $\tilde{\mathcal{D}}$ of \mathcal{D} ; 2) compute all frequent itemsets, $\tilde{\mathcal{F}}$, in $\tilde{\mathcal{D}}$; 3) check that all these FIs computed using $\tilde{\mathcal{D}}$ are also FIs in \mathcal{D} and correct the output. If an itemset is frequent in \mathcal{D} and not in $\tilde{\mathcal{D}}$, correct the output using \mathcal{D} , see [30] for details. Toivonen's algorithm is based on an efficient probabilistic estimate of the support of an itemset U .

We reuse this idea of estimating the support of U in our method for parallel mining of FIs, i.e., we use only the first two steps. We define the error of the estimate of $Supp^*(U, \mathcal{D})$ from a database sample $\tilde{\mathcal{D}}$ by:

$$err_{supp}(U, \tilde{\mathcal{D}}) = |Supp^*(U, \mathcal{D}) - Supp^*(U, \tilde{\mathcal{D}})|$$

The database sample $\tilde{\mathcal{D}}$ is sampled with replacement. The estimation error can be analyzed using the Chernoff bound without making other assumptions about the database. The error analysis then holds for a database of arbitrary size and properties.

Theorem 6.1. [30] *Given an itemset $U \subseteq \mathcal{B}$, two real numbers $\epsilon_{\tilde{\mathcal{D}}}, \delta_{\tilde{\mathcal{D}}}, 0 \leq \epsilon_{\tilde{\mathcal{D}}}, \delta_{\tilde{\mathcal{D}}} \leq 1$, and a sample $\tilde{\mathcal{D}}$ drawn from database \mathcal{D} of size*

$$|\tilde{\mathcal{D}}| \geq \frac{1}{2\epsilon_{\tilde{\mathcal{D}}}^2} \ln \frac{2}{\delta_{\tilde{\mathcal{D}}}},$$

then the probability that $err_{supp}(U, \tilde{\mathcal{D}}) > \epsilon_{\tilde{\mathcal{D}}}$ is at most $\delta_{\tilde{\mathcal{D}}}$.

Proof. We denote the probability of an event by $P[\cdot]$. Toivonen used in the original paper the Chernoff bounds described in [25]. We use the same equation, see (A.1) or Appendix A in [25]. The Chernoff bounds gives:

$$\begin{aligned} P\left[err_{supp}(U, \tilde{\mathcal{D}}) > \epsilon\right] &= P\left[|Supp^*(U, \mathcal{D}) - Supp^*(U, \tilde{\mathcal{D}})| > \epsilon_{\tilde{\mathcal{D}}}\right] \\ &= P\left[|Supp^*(U, \mathcal{D}) - Supp^*(U, \tilde{\mathcal{D}})| \cdot |\tilde{\mathcal{D}}| > \epsilon_{\tilde{\mathcal{D}}} \cdot |\tilde{\mathcal{D}}|\right] \leq 2e^{-2(\epsilon_{\tilde{\mathcal{D}}} \cdot |\tilde{\mathcal{D}}|)^2 / |\tilde{\mathcal{D}}|} \end{aligned}$$

give an upper bound for the probability:

$$2e^{-2(\epsilon_{\tilde{\mathcal{D}}} \cdot |\tilde{\mathcal{D}}|)^2 / |\tilde{\mathcal{D}}|} \leq \delta_{\tilde{\mathcal{D}}}$$

$$|\tilde{\mathcal{D}}| \geq \frac{1}{2\epsilon_{\tilde{\mathcal{D}}}^2} \ln \frac{2}{\delta_{\tilde{\mathcal{D}}}}$$

□

Using a database sample $\tilde{\mathcal{D}}$ with size given by the previous theorem, we can estimate $Supp^*(U, \mathcal{D})$ with error $\epsilon_{\tilde{\mathcal{D}}}$ that occurs with probability at most $\delta_{\tilde{\mathcal{D}}}$: it follows from Theorem 6.1 that if we compute the approximation $\tilde{\mathcal{F}}$ of \mathcal{F} from the database sample $\tilde{\mathcal{D}}$ of size $|\tilde{\mathcal{D}}| \geq \frac{1}{2\epsilon_{\tilde{\mathcal{D}}}^2} \ln \frac{2}{\delta_{\tilde{\mathcal{D}}}}$, we should get an estimate of the supports of itemsets $U \in \tilde{\mathcal{F}}$, i.e., potentially, $\tilde{\mathcal{F}}$ closely approximates \mathcal{F} .

6.2 Estimating the relative size of a PBEC

In our parallel method for mining FIs, we need to estimate the relative size of a PBEC $[U]$: $|[U] \cap \tilde{\mathcal{F}}| / |\tilde{\mathcal{F}}|$, see Definition 2.27. This can be estimated using $\tilde{\mathcal{F}}_s$. There are two ways for constructing $\tilde{\mathcal{F}}_s$:

1. Compute $\tilde{\mathcal{M}}$ and get $\tilde{\mathcal{F}}_s$ using the *modified coverage algorithm*;
2. Compute $\tilde{\mathcal{F}}$ and get $\tilde{\mathcal{F}}_s$ using the *reservoir sampling*.

These two algorithms are presented in the next sections.

6.2.1 The coverage algorithm and its modification

Let us have the MFIs $\widetilde{\mathcal{M}}$, computed from $\widetilde{\mathcal{D}}$. The set of all MFIs $\widetilde{\mathcal{M}}$ is the upper bound on the set $\widetilde{\mathcal{F}}$, i.e., $\widetilde{\mathcal{F}} = \bigcup_{m \in \widetilde{\mathcal{M}}} \mathcal{P}(m)$. To construct a sample $\widetilde{\mathcal{F}}_s \subseteq \widetilde{\mathcal{F}}$ of independent and identically distributed (i.i.d.) elements chosen from the uniform distribution, we can use the *coverage algorithm* [24] that uses $\widetilde{\mathcal{M}}$ for construction of the sample $\widetilde{\mathcal{F}}_s$. To make the sampling in our parallel method for mining of FIs faster, we have modified the algorithm, so it does not constructs sample from *uniform* distribution, but it creates only *independently* distributed sample. The coverage algorithm (or its modification) produces only the sample $\widetilde{\mathcal{F}}_s$.

The coverage algorithm estimates the relative size of a set $F \subseteq \widetilde{\mathcal{F}}$. The coverage algorithm takes as input the set $\widetilde{\mathcal{M}}$ of the MFIs computed from the database sample $\widetilde{\mathcal{D}}$, a number $0 \leq \rho \leq 1$ representing the fraction $\rho = \frac{|F|}{|\widetilde{\mathcal{F}}|}$ where in our case $F = [W] \cap \widetilde{\mathcal{F}}$ is the smallest PBEC we want to estimate, and two real numbers $0 \leq \epsilon_{\widetilde{\mathcal{F}}_s}, \delta_{\widetilde{\mathcal{F}}_s} \leq 1$ where $\epsilon_{\widetilde{\mathcal{F}}_s}$ is the error of the approximated size and $\delta_{\widetilde{\mathcal{F}}_s}$ its probability **or** the number of samples $N = |\widetilde{\mathcal{F}}_s|$ instead of the sampling parameters. The output of the coverage algorithm is the sample $\widetilde{\mathcal{F}}_s$ that is used for the estimation of the relative sizes of PBECs.

In our parallel method, the set F represents the set of FIs in some PBECs: let have a set of prefixes $Q = \{U_i | U_i \neq U_j, U_i \subseteq \mathcal{B}, \text{ and } [U_i] \cap [U_j] = \emptyset \text{ for all } i \neq j\}$ be a set of itemsets (prefixes) then in our method $F = (\bigcup_{U \in Q} [U]) \cap \widetilde{\mathcal{F}}$. However, the theory we show holds for an arbitrary set of itemsets F .

The coverage algorithm follows:

Algorithm 7 The COVERAGE-ALGORITHM algorithm

COVERAGE-ALGORITHM(**In:** Set $\widetilde{\mathcal{M}} = \{m_i\}$,
In: Integer N ,
Out: Set $\widetilde{\mathcal{F}}_s$)

```

1:  $\widetilde{\mathcal{F}}_s \leftarrow \emptyset$ 
2: while  $|\widetilde{\mathcal{F}}_s| \neq N$  do
3:   pick index  $i \in [1, |\widetilde{\mathcal{M}}|]$  with probability  $P[i] = \frac{|\mathcal{P}(m_i)|}{\sum_j |\mathcal{P}(m_j)|}$ 
4:   pick a random set  $U \in \mathcal{P}(m_i)$  with probability  $\frac{1}{|\mathcal{P}(m_i)|}$ 
5:   found  $\leftarrow$  false
6:   for  $l \leftarrow i - 1$  to 1 do
7:     if  $U \subseteq m_l$  then
8:       found  $\leftarrow$  true
9:     end if
10:    end for
11:    if found=false then
12:       $\widetilde{\mathcal{F}}_s \leftarrow \widetilde{\mathcal{F}}_s \cup \{U\}$ 
13:    end if
14:  end while
```

Let's have a set of itemsets $F \subseteq \widetilde{\mathcal{F}}$. Now, we analyze the dependency of the error $\epsilon_{\widetilde{\mathcal{F}}_s}$ of the estimated relative size $|F|/|\widetilde{\mathcal{F}}|$ estimated using the sample $\widetilde{\mathcal{F}}_s$ by $|F \cap \widetilde{\mathcal{F}}_s|/|\widetilde{\mathcal{F}}_s|$ on the size of $|\widetilde{\mathcal{F}}_s|$.

First, we define the *multiset* $\mathcal{S} = \biguplus_{m \in \widetilde{\mathcal{M}}} \mathcal{P}(m)$ that contains as many copies of $W \in \widetilde{\mathcal{F}}$ as the number of $\mathcal{P}(m)$ containing W . The sample obtained by uniform sampling of \mathcal{S} is therefore a non-uniform sample of $\widetilde{\mathcal{F}}$.

We can represent every itemset in $s \in \mathcal{S}$ as a pair $s = (W, i)$ that correspond to $W \in \mathcal{P}(m_i)$, that is $\mathcal{S} = \{(W, i) | W \in \mathcal{P}(m_i)\}$. We define a function $g : \mathcal{S} \rightarrow \{\text{true}, \text{false}\}$ as follows:

$$g((W, i)) = \begin{cases} \text{true}, & \text{if } i = \min\{j | W \in \mathcal{P}(m_j)\} \\ \text{false}, & \text{otherwise} \end{cases}$$

To make the sample of $\widetilde{\mathcal{F}}$ uniform, we must sample the set $\mathcal{S}' = \{s | s \in \mathcal{S} \text{ and } g(s) = \text{true}\}$. Each element of \mathcal{S}' corresponds to one element of $\widetilde{\mathcal{F}}$. Therefore, by sampling \mathcal{S}' , we sample

$\tilde{\mathcal{F}}$. It is clear that $|\mathcal{S}| \geq |\mathcal{S}'| = |\tilde{\mathcal{F}}|$.

The coverage algorithm, described in Algorithm 7 in fact samples \mathcal{S}' . To sample \mathcal{S}' , Algorithm 7 picks i with probability proportional to $|\mathcal{P}(m_i)|$ (line 3) and then it picks $W \in \mathcal{P}(m_i)$ uniformly at random (line 4). In order to sample the set \mathcal{S}' instead of \mathcal{S} , the algorithm must assure that we choose only those itemsets $W, g((W, i)) = \text{true}$ for some integer i . That is: we must check that there does not exist $m_j, j < i$, such that $W \in \mathcal{P}(m_j)$. This is performed at line 6.

Theorem 6.2 (estimation error of the size of a subset $F \subseteq \tilde{\mathcal{F}}$). [24] Let $\tilde{\mathcal{M}}$ be the set of MFIs such that $\tilde{\mathcal{F}} = \bigcup_{m_i \in \tilde{\mathcal{M}}} \mathcal{P}(m_i)$, $F \subseteq \tilde{\mathcal{F}}$, $\rho = |F|/|\tilde{\mathcal{F}}|$, two real numbers $\epsilon_{\tilde{\mathcal{F}}_s}, \delta_{\tilde{\mathcal{F}}_s}$ such that $0 \leq \epsilon_{\tilde{\mathcal{F}}_s}, \delta_{\tilde{\mathcal{F}}_s} \leq 1$, and $\tilde{\mathcal{F}}_s$ is the independent and identically distributed sample of $\tilde{\mathcal{F}}$ obtained by the coverage algorithm by calling COVERAGE-ALGORITHM $(\tilde{\mathcal{M}}, N_{\tilde{\mathcal{F}}_s}, \tilde{\mathcal{F}}_s)$. Then the estimate:

$$\frac{|F \cap \tilde{\mathcal{F}}_s|}{|\tilde{\mathcal{F}}_s|}$$

is an estimation of $|F|/|\tilde{\mathcal{F}}|$ with error at most $\epsilon_{\tilde{\mathcal{F}}_s}$ with probability at least $1 - \delta_{\tilde{\mathcal{F}}_s}$ provided

$$N_{\tilde{\mathcal{F}}_s} = |\tilde{\mathcal{F}}_s| \geq \frac{4}{\epsilon_{\tilde{\mathcal{F}}_s}^2 \rho} \ln \frac{2}{\delta_{\tilde{\mathcal{F}}_s}}.$$

Proof. The proof of the theorem is again based on the Chernoff bounds. We know that:

$$P \left[|F \cap \tilde{\mathcal{F}}_s| \geq (1 + \epsilon_{\tilde{\mathcal{F}}_s}) \rho |\tilde{\mathcal{F}}_s| \right] \leq e^{-|\tilde{\mathcal{F}}_s| \rho \epsilon_{\tilde{\mathcal{F}}_s}^2 / 4}$$

and similarly for the lower bound:

$$P \left[|F \cap \tilde{\mathcal{F}}_s| \leq (1 - \epsilon_{\tilde{\mathcal{F}}_s}) \rho |\tilde{\mathcal{F}}_s| \right] \leq e^{-|\tilde{\mathcal{F}}_s| \rho \epsilon_{\tilde{\mathcal{F}}_s}^2 / 4}$$

Therefore:

$$P \left[(1 - \epsilon_{\tilde{\mathcal{F}}_s}) |\tilde{\mathcal{F}}_s| \rho \leq |F \cap \tilde{\mathcal{F}}_s| \leq (1 + \epsilon_{\tilde{\mathcal{F}}_s}) |\tilde{\mathcal{F}}_s| \rho \right] \geq 1 - 2e^{-|\tilde{\mathcal{F}}_s| \rho \epsilon_{\tilde{\mathcal{F}}_s}^2 / 4} \geq 1 - \delta_{\tilde{\mathcal{F}}_s}$$

□

An important part of the coverage algorithm is the for-loop at the line 6. It guarantees that each $U \in \tilde{\mathcal{F}}$ is selected with probability $\frac{1}{|\tilde{\mathcal{F}}|}$. Unfortunately, this loop will prevent many selected samples $U \subseteq m_i$ to not make it into $\tilde{\mathcal{F}}_s$ because the U is contained in an MFI $m_j \in \tilde{\mathcal{M}}, j < i$ with lower index. Additionally, the set $\tilde{\mathcal{M}}$ can be quite large and the loop has the complexity $\mathcal{O}(\tilde{\mathcal{M}})$.

The coverage algorithm runs in polynomial time given 1) the number of samples $N_{\tilde{\mathcal{F}}_s}$ which is a function of $1/\rho, 1/\epsilon_{\tilde{\mathcal{F}}_s}, 1/\delta_{\tilde{\mathcal{F}}_s}$; 2) $|\tilde{\mathcal{M}}|$; and 3) $|\mathcal{B}|$ if the following properties holds:

1. For all $i, |\mathcal{P}(m_i)|, m_i \in \tilde{\mathcal{M}}$ is computable in polynomial time in $|m_i|$.
2. It is possible to sample uniformly from any $\mathcal{P}(m_i), m_i \in \tilde{\mathcal{M}}$.
3. For all $U \in \mathcal{P}(\mathcal{B})$, it can be determined in polynomial time in $|m_i|$ whether $U \in \mathcal{P}(m_i)$.
4. We are using $\tilde{\mathcal{F}}_s$ for estimating the size of a set $F \subseteq \tilde{\mathcal{F}}$. Therefore, $\rho = \frac{|F|}{|\tilde{\mathcal{F}}|}$ must be polynomial in $|\tilde{\mathcal{F}}|$ in order to make the COVERAGE-ALGORITHM polynomial.

The modification of the coverage algorithm: in our parallel method for mining FIs, we need to compute the sample even faster. To do so, we resign on the uniform sampling of $\tilde{\mathcal{F}}$ (i.e. on sampling \mathcal{S}') by omitting the checks against MFIs with lower index, i.e., we omit the for-loop at line 6 and therefore, we sample the set $\mathcal{S} = \biguplus_{m \in \tilde{\mathcal{M}}} \mathcal{P}(m)$. This makes the sampling non-uniform because it prefers samples U that are contained in many $\mathcal{P}(m_i)$, i.e., the sampling prefers sets $\mathcal{P}(m_i) \cap \mathcal{P}(m_j), i \neq j$. Therefore, the estimate of the relative size of a set $F \subseteq \tilde{\mathcal{F}}$ (computed using the modified algorithm) *is just a heuristic!* However, this heuristic is much faster then the COVERAGE-ALGORITHM and the estimates of the relative sizes made using the sample obtained by the modified coverage algorithm are sufficient for our purposes.

Algorithm 8 The MODIFIED-COVERAGE-ALGORITHM algorithm

MODIFIED-COVERAGE-ALGORITHM(**In:** Set $\widetilde{\mathcal{M}} = \{m_i\}$,

In: Integer N ,

Out: Set $\widetilde{\mathcal{F}}_s$)

- 1: $\widetilde{\mathcal{F}}_s \leftarrow \emptyset$
 - 2: **while** $|\widetilde{\mathcal{F}}_s| \neq N$ **do**
 - 3: pick $m \in \widetilde{\mathcal{M}}$ with probability $\frac{|\mathcal{P}(m)|}{\sum_{m' \in \widetilde{\mathcal{M}}} |\mathcal{P}(m')|}$
 - 4: pick subset $S \subseteq m$ with probability $\frac{1}{|\mathcal{P}(m)|}$
 - 5: $\widetilde{\mathcal{F}}_s \leftarrow \widetilde{\mathcal{F}}_s \cup \{S\}$
 - 6: **end while**
-

6.2.2 The reservoir sampling

In this section, we show the *reservoir sampling algorithm* that constructs an uniformly but not independently distributed sample $\widetilde{\mathcal{F}}_s$ of $\widetilde{\mathcal{F}}$ on the contrary of the previous section.

Vitter [32] formulates the problem of *reservoir sampling* as follows: given a stream of records, the task is to construct a sample of size n *without replacement* from the stream of records without any prior knowledge of the length of the stream.

We can reformulate the original problem in the terms of $\widetilde{\mathcal{F}}$ and $\widetilde{\mathcal{F}}_s$: let's consider a sequential algorithm that outputs all frequent itemsets $\widetilde{\mathcal{F}}$ from a database $\widetilde{\mathcal{D}}$. We can view $\widetilde{\mathcal{F}}$ as a stream of FIs. We do not know $|\widetilde{\mathcal{F}}|$ in advance and we need to take $|\widetilde{\mathcal{F}}_s|$ samples of $\widetilde{\mathcal{F}}$. We take the samples $\widetilde{\mathcal{F}}_s$ using the reservoir sampling algorithm. This solves our problem of making a uniform sample $\widetilde{\mathcal{F}}_s \subseteq \widetilde{\mathcal{F}}$. The sampling is done using an array of FIs (a buffer, or in the terminology of [32] a reservoir) that holds $\widetilde{\mathcal{F}}_s$.

The reservoir sampling uses the following two procedures:

1. READNEXTFI(L): reads next FI from an output of an arbitrary sequential algorithm for mining of FIs and stores the itemset at the location L in memory.
2. SKIPFIs(k): skips k FIs from the output of an arbitrary algorithm for mining of FIs.

and the following function:

1. `RANDOM()` which returns an uniformly distributed real number from the interval $[0, 1]$

The simplest reservoir sampling algorithm is summarized in Algorithm 9. It takes as an input an array R (reservoir/buffer) of size $n = |\tilde{\mathcal{F}}_s|$, the function `READNEXTFI(L)` that reads an FI from the output of an FI mining algorithm and stores it in memory at location L , and finally the function `SkipFIs(k)` that skips k FIs. The algorithm samples $|\tilde{\mathcal{F}}_s|$ FIs and stores them in memory into the buffer R .

The SIMPLE-RESERVOIR-SAMPLING follows:

Algorithm 9 The SIMPLE-RESERVOIR-SAMPLING algorithm

SIMPLE-RESERVOIR-SAMPLING(**In/Out:** Array R of size n ,

In: Integer n ,
In: Procedure `ReadNextFI`,
In: Procedure `SkipFIs`)

```

1: for  $j \leftarrow 0$  to  $n - 1$  do
2:   ReadNextFI( $R[j]$ )
3: end for
4:  $t \leftarrow n$ 
5: while not eof do
6:    $t \leftarrow t + 1$ 
7:    $m \leftarrow \lfloor t \times \text{RANDOM}() \rfloor$  {pick uniformly a number from the set  $\{0, \dots, t - 1\}$ }
8:   if  $m < n$  then
9:     ReadNextFI( $R[m]$ )
10:   else
11:     SkipFIs(1)
12:   end if
13: end while

```

The SIMPLE-RESERVOIR-SAMPLING is quite slow, it is linear in the number of input records read by `READNEXTFI(R)`, i.e., it is linear in $|\tilde{\mathcal{F}}|$. Vitter [32] created a *faster* algorithm that has the same parameters as the SIMPLE-RESERVOIR-SAMPLING algorithm. We denote the Vitter's variant of the algorithm by VITTER-RESERVOIR-SAMPLING(**In/Out:** Array R of size n , **In:** Integer n , **In:** Procedure `ReadNextFI`, **In:** Procedure `SkipFIs`).

The VITTER-RESERVOIR-SAMPLING runs with the average running time $\mathcal{O}(|\tilde{\mathcal{F}}_s|(1 + \log \frac{|\tilde{\mathcal{F}}|}{|\tilde{\mathcal{F}}_s|}))$, where $|\tilde{\mathcal{F}}_s|$ is the size of the array R used by VITTER-RESERVOIR-SAMPLING. Vitter in his analyse does not consider the time needed to read the record using the READ-NEXTFI and to skip the records using the SKIPFI. That is: the formula represents only the time needed by the execution of the VITTER-RESERVOIR-SAMPLING algorithm, see [32] for details.

Now, we analyse the relative size of a PBEC using the samples taken by the reservoir algorithm. The reservoir sampling samples the set $\tilde{\mathcal{F}}$ **without replacement**, resulting in $\tilde{\mathcal{F}}_s$. In Theorem 6.2 we analysed the error of the approximation of the relative size of an arbitrary set using an i.i.d. sample using the Chernoff bounds. In the case of the reservoir sampling, we cannot use the Chernoff bounds because the elements of the sample $\tilde{\mathcal{F}}_s$ are *identically* but unfortunately **not independently** distributed due to the use of the reservoir. The reservoir sampling process can be modeled using the *hypergeometric distribution*, see Appendix A or [20]. In the rest of this chapter, we analyze the bounds on the relative size of a set of itemsets using the sample made by the reservoir sampling using a *hypergeometric distribution*.

Using the bounds from Appendix A.2, we can state a theorem similar to Theorem 6.2 (using the Chernoff bounds and an i.i.d sample) but now for the *hypergeometric distribution*, i.e., estimation of the relative size of a PBEC but using a uniformly but not *independently* distributed sample:

Theorem 6.3 (Estimation error of the size of a subset $F \subseteq \tilde{\mathcal{F}}$). *Let $F \subseteq \tilde{\mathcal{F}}$ be a set of itemsets. The relative size of F , $\frac{|F|}{|\tilde{\mathcal{F}}|}$, is estimated with error $\epsilon_{\tilde{\mathcal{F}}_s}$, $0 \leq \epsilon_{\tilde{\mathcal{F}}_s} \leq 1$, with probability $\delta_{\tilde{\mathcal{F}}_s}$, $0 \leq \delta_{\tilde{\mathcal{F}}_s} \leq 1$, from a hypergeometrically distributed sample $\tilde{\mathcal{F}}_s \subseteq \tilde{\mathcal{F}}$ with parameters $N = |\tilde{\mathcal{F}}|$, $M = |F|$ (see Appendix A) of size*

$$|\tilde{\mathcal{F}}_s| \geq -\frac{\log(\delta_{\tilde{\mathcal{F}}_s}/2)}{D(\rho + \epsilon_{\tilde{\mathcal{F}}_s} || \rho)}$$

Where $D(x||y)$ is the Kullback-Leibler divergence of two hypergeometrically distributed variables with parameters x, y and $\rho = |F|/|\tilde{\mathcal{F}}|$.

The expected value of the size $|F \cap \tilde{\mathcal{F}}_s|$ is $E[|F \cap \tilde{\mathcal{F}}_s|] = |\tilde{\mathcal{F}}_s| \cdot \frac{|F|}{|\tilde{\mathcal{F}}|}$.

Proof. The proof is based on bounds provided in [29] which is a summarization of [13], see (A.6) where $p = \rho$, $\epsilon = \epsilon_{\tilde{\mathcal{F}}_s}$, $n = |\tilde{\mathcal{F}}_s|$, and the fact that $D(p + \epsilon || p) > D(p - \epsilon || p)$:

$$1 - (e^{-|\tilde{\mathcal{F}}_s| \cdot D(p-\epsilon||p)} + e^{-|\tilde{\mathcal{F}}_s| \cdot D(p+\epsilon||p)}) \leq 1 - \delta_{\tilde{\mathcal{F}}_s}$$

$$1 - 2e^{-|\tilde{\mathcal{F}}_s| \cdot D(p+\epsilon||p)} \leq 1 - \delta_{\tilde{\mathcal{F}}_s}$$

□

Since F is a union of PBECs and the reservoir sampling algorithm give us identically distributed sample, we are able to bound the error of relative size of a union of PBECs made by the “double sampling process”, i.e., estimating the size of a union of PBECs using a database sample:

Theorem 6.4 (bounds on the size of a set of FIs from a given PBEC). *Let $V_i \subseteq \mathcal{B}, 1 \leq i \leq n, [V_i] \cap [V_j] = \emptyset, i \neq j$. We use two sets of itemsets:*

1. $A = \{U | \text{Supp}^*(U, \mathcal{D}) < \text{min_support}^* \text{ and } \text{Supp}^*(U, \tilde{\mathcal{D}}) \geq \text{min_support}^*\}$, i.e., the collection of itemsets U infrequent in \mathcal{D} and frequent in $\tilde{\mathcal{D}}$ – wrongly added FIs to $\tilde{\mathcal{F}}$.
2. $B = \{U | \text{Supp}^*(U, \mathcal{D}) \geq \text{min_support}^* \text{ and } \text{Supp}^*(U, \tilde{\mathcal{D}}) < \text{min_support}^*\}$, i.e., the collection of itemsets U frequent in \mathcal{D} and infrequent in $\tilde{\mathcal{D}}$ – wrongly removed FIs from $\tilde{\mathcal{F}}$.

The relative size of A is denoted by $a = \frac{|A|}{|\mathcal{F}|}$ and the relative size of B is denoted by $b = \frac{|B|}{|\mathcal{F}|}$. Then for two sets of itemsets $C = \bigcup_i [V_i] \cap \mathcal{F}$ and $\tilde{C} = \bigcup_i [V_i] \cap \tilde{\mathcal{F}}$, we have:

$$\frac{|\tilde{C}|}{|\tilde{\mathcal{F}}|}(1 + a - b) - a \leq \frac{|C|}{|\mathcal{F}|} \leq \frac{|\tilde{C}|}{|\tilde{\mathcal{F}}|} \cdot (1 + a - b) + b$$

Proof. From the assumptions follows: $|\tilde{\mathcal{F}}| = |\mathcal{F}|(1 + a - b)$. Therefore: $\frac{|\tilde{\mathcal{F}}|}{(1+a-b)} = |\mathcal{F}|$.

We know that the fraction a of FIs is not frequent in \mathcal{D} but is frequent in $\tilde{\mathcal{D}}$ are present in $\tilde{\mathcal{F}}$. Therefore, we can compute the lower bound of the relative size of C :

$$|\tilde{C}| \leq |C| + a \cdot |\mathcal{F}| \tag{6.1}$$

$$\frac{|\tilde{C}|}{|\mathcal{F}|} \leq \frac{|C|}{|\mathcal{F}|} + a \tag{6.2}$$

(6.3) follows from (6.2) using the fact that $|\mathcal{F}| = \frac{|\tilde{\mathcal{F}}|}{(1+a-b)}$.

$$\frac{|\tilde{C}|}{|\tilde{\mathcal{F}}|}(1+a-b) \leq \frac{|\tilde{C}|}{|\mathcal{F}|} \leq \frac{|C|}{|\mathcal{F}|} + a \quad (6.3)$$

$$\frac{|\tilde{C}|}{|\tilde{\mathcal{F}}|}(1+a-b) - a \leq \frac{|C|}{|\mathcal{F}|} \quad (6.4)$$

We compute the upper bound of $\frac{|C|}{|\mathcal{F}|}$ using similar computations as for the lower bound. The fraction b of FIs \mathcal{F} was not frequent in $\tilde{\mathcal{D}}$ and frequent in \mathcal{D} and therefore the lower bound of the size $|\tilde{C}|$ is:

$$|C| - b \cdot |\mathcal{F}| \leq |\tilde{C}| \quad (6.5)$$

$$\frac{|C|}{|\mathcal{F}|} - b \leq \frac{|\tilde{C}|}{|\mathcal{F}|} \quad (6.6)$$

$$\frac{|C|}{|\mathcal{F}|} \leq \frac{|\tilde{C}|}{|\tilde{\mathcal{F}}|} \cdot (1+a-b) + b \quad (6.7)$$

□

Corollary 6.5. *If the size of $\frac{|\tilde{C}|}{|\tilde{\mathcal{F}}|}$ is estimated with error $\epsilon_{\tilde{\mathcal{F}}_s}$, $0 \leq \epsilon_{\tilde{\mathcal{F}}_s} \leq 1$, with probability $0 \leq \delta_{\tilde{\mathcal{F}}_s} \leq 1$ then:*

$$\frac{|\tilde{C}|}{|\tilde{\mathcal{F}}|}(1 - \epsilon_{\tilde{\mathcal{F}}_s})(1+a-b) - a \leq \frac{|C|}{|\mathcal{F}|} \leq \frac{|\tilde{C}|}{|\tilde{\mathcal{F}}|}(1 - \epsilon_{\tilde{\mathcal{F}}_s})(1+a-b) + b$$

with probability $\delta_{\tilde{\mathcal{F}}_s}$.

Set C can be viewed as a partition processed by a single processor. We estimate the relative size of $|C|/|\mathcal{F}|$ from $\tilde{\mathcal{F}}_s$ and we are able to bound the error made while estimating the size of a partition. Unfortunately, the bounds are not very tight and making tighter bounds is hard.

Because our modification of the coverage algorithm does not give an identically distributed sample, Corollary 6.5 cannot be used to bound the size of $F \subseteq \tilde{\mathcal{F}}$ using the sample taken by MODIFIED-COVERAGE-ALGORITHM.

6.3 Estimating the size of a union of PBECs

Let $U_i \subseteq \mathcal{B}$, $1 \leq i \leq n$, be prefixes and $[U_i]$ corresponding PBECs. We are constructing the PBECs by recursive splitting and estimating the size using the sample, i.e., $|[U_i] \cap \tilde{\mathcal{F}}| / |\tilde{\mathcal{F}}| \approx |[U_i] \cap \tilde{\mathcal{F}}_s| / |\tilde{\mathcal{F}}_s|$. Let $L \subseteq [1, n]$ be the set of indexes of the PBECs. The set of indexes is chosen in such a way that $|\bigcup_{i \in L} [U_i] \cap \tilde{\mathcal{F}}_s| / |\tilde{\mathcal{F}}_s| \approx 1/P$. That is: the set L is dependent on the constructed sample $\tilde{\mathcal{F}}_s$. Therefore, we are not able to use the Chernoff bounds (or the estimates using the Kullback-Leibler divergence) with the same sample $\tilde{\mathcal{F}}_s$ (used for construction of PBECs) for estimation of the relative size of $F = \bigcup_{j \in L} [U_j] \cap \tilde{\mathcal{F}}$ because the sets $[U_j]$, the set \mathcal{F} and the sample $\tilde{\mathcal{F}}_s$ are not independent. Instead, we must choose $\epsilon_{\tilde{\mathcal{F}}_s}$ such that $\epsilon_{\tilde{\mathcal{F}}_s} \cdot |L|$ is small enough. In Chapter 11, we experimentally show the error and its probability made by a particular choice of the number of samples.

7 Approximate parallel mining of MFIs

In our method, we need to compute an approximation of the maximal frequent itemsets (MFIs), $\widetilde{\mathcal{M}}$ (see Chapter 6 for notation). Because we have P processors at our disposal, we could execute an arbitrary algorithm for mining of MFIs in parallel. Unfortunately, parallel mining of MFIs using a DFS algorithm, is a hard task. We can relax the requirement of computing $\widetilde{\mathcal{M}}$ to a requirement of computing the set M such that $\widetilde{\mathcal{M}} \subseteq M \subseteq \widetilde{\mathcal{F}}$. Recall that we denote a prefix-based equivalence class with prefix U and extensions W by $[U|W]$, to emphasize that the items in the extensions are important as described in Definition 2.21.

We define a *candidate* on an MFI as follows:

Definition 7.1 (candidate itemset on MFI). *Let $U \subseteq \mathcal{B}$ be a frequent itemset and Σ the extensions used by a DFS MFI algorithm for extending U . We call U a candidate itemset (or candidate in short) on an MFI if for each $b \in \Sigma$ the itemset $U \cup \{b\}$ is not frequent, i.e., $\text{Supp}(U \cup \{b\}) < \text{min_support}$.*

A “template” of a DFS algorithm for mining of MFIs is shown in Algorithm 10. The difference between algorithms for mining of MFIs is in the way they implement the depth-first search of the PBECs. The DFS MFI algorithms optimize the search so they visit as small number of FIs as possible. Other difference is in the used datastructures, and the way the algorithms implement the test at line 4.

The candidates on the MFIs are the leafs of the DFS algorithm for mining of MFIs. An example of the candidate on the MFI is the itemset 5, 6 in Example 7.1.

Definition 7.2 (longest subset of a MFI in a PBEC). *Let W be a maximal frequent itemset, $b \in \mathcal{B}$ an item, the set $\Sigma = \{b' \in \mathcal{B} : b < b'\}$, and $[\{b\}|\Sigma]$ the PBEC. We call the set $U = W \cap (\Sigma \cup \{b\})$ the longest subset of W in the PBEC $[\{b\}|\Sigma]$.*

For example, let $U = \{1\}$ be a prefix and $\Sigma = \{2, 3, 5\}$ its extensions. For the MFI $m = \{1, 3, 4, 5\}$ the longest subset of m in $[U|\Sigma]$ is the set $\{1, 3, 5\}$.

The longest subset of a MFI in a PBEC can be a candidate set, but there exists longest subsets that are not candidates. We say that W is a candidate on the MFI U , $W \subsetneq U$ in a PBEC, if it is a *candidate* and a longest subset of U in the PBEC, i.e., it is a leaf of a DFS tree and it is a longest subset.

Recall, that we omit the extensions in a PBEC $[b|\{b'|b < b'; b', b \in \mathcal{B}\}]$, see page 8. We use the extensions in the notation if we want to emphasise them.

A sequential schema for mining of MFIs is shown in Algorithm 10:

Algorithm 10 The schema of a DFS algorithm for mining of MFIs.

DFS-MFI-SCHEMA(**In:** Database $\tilde{\mathcal{D}}$, **In:** $min_support^*$, **In:** Set B , **Out:** Set $\tilde{\mathcal{M}}$)

Require: $\mathcal{B} = \{b_i\}$ to be an ordered set $b_1 \leq \dots \leq b_{|\mathcal{B}|}$ and $Supp^*(\{b_i\}, \tilde{\mathcal{D}}) \geq min_support^*$.

```

1:  $\tilde{\mathcal{M}} \leftarrow \emptyset$ 
2: for each  $b_i \in B$  in ascending order do
3:   perform depth-first search of  $[(b_i)|\{b : b \in \mathcal{B}, b > b_i\}] \cap \tilde{\mathcal{F}}$ 
   (visiting/discovering candidates  $U \in [(b_i)|\{b : b \in \mathcal{B}, b > b_i\}] \cap \tilde{\mathcal{F}}$  on an MFI)
4:   for each candidate to maximal itemset  $U \in [(b_i)|\{b_{i+1}, \dots, b_{|\mathcal{B}|}\}] \cap \tilde{\mathcal{F}}$  do
5:     if exists no  $W \in \tilde{\mathcal{M}}$  such that  $U \subseteq W$  then
6:        $\tilde{\mathcal{M}} \leftarrow \tilde{\mathcal{M}} \cup \{U\}$ 
7:     end if
8:   end for
9: end for
```

A maximal frequent itemset $W = (b_{w_1}, \dots, b_{w_{|W|}})$, $b_{w_1} < \dots < b_{w_{|W|}}$ is visited(discovered) by a DFS MFI algorithm by expanding first $[(b_{w_1})]$, then $[(b_{w_1}, b_{w_2})]$, etc. To our best knowledge, all MFIs DFS mining algorithms follow the schema in Algorithm 10: the algorithm initializes $\tilde{\mathcal{M}} \leftarrow \emptyset$ and starts a depth-first search on the lattice of all FIs, skipping some FIs. The PBECs are expanded in the order of b_i , i.e., $[(b_1)|(b_2, b_3, \dots, b_{|\mathcal{B}|})]$ is processed first, then $[(b_2)|(b_3, \dots, b_{|\mathcal{B}|})]$ is processed, etc. Therefore, if $U = (b_{u_1}, \dots, b_{u_{|U|}})$ is an MFI and $W \subseteq U$ be a candidate itemset. W are visited *after* visiting U . If the algorithm finds a candidate itemset W , it looks into $\tilde{\mathcal{M}}$ and if $\tilde{\mathcal{M}}$ contains a superset of W , the algorithm skips W (*not* storing W in $\tilde{\mathcal{M}}$). If $\tilde{\mathcal{M}}$ does not contain a superset of W , it is an MFI and is stored into $\tilde{\mathcal{M}}$ (see line 6).

Proposition 7.3. *Let $\tilde{\mathcal{M}}$ be a set of all MFIs mined with some value of $min_support^*$ in a database $\tilde{\mathcal{D}}$, $U \in \tilde{\mathcal{M}}$ be an MFI, and W be a candidate on the MFI such that $W \subsetneq U$. Then W is visited by Algorithm 10 after visiting the MFI U .*

Proof. The proposition follows from the fact that the items in the baseset \mathcal{B} are ordered and Algorithm 10 processes $[(b_i)]$ and its extensions in the order of the items in \mathcal{B} . \square

We can execute Algorithm 10 in parallel with dynamic load-balancing as shown in Algo-

rithm 11.

Algorithm 11 The parallel schema of a DFS algorithm for mining of MFIs.

PARALLEL-DFS-MFI-SCHEMA(**In:** Database $\tilde{\mathcal{D}}$,

In: $min_support^*$,

In: Set \mathcal{B} ,

Out: Set M)

Require: $\mathcal{B} = \{b_i\}$ to be an ordered set $b_1 \leq \dots \leq b_{|\mathcal{B}|}$ and $Supp^*(\{b_i\}, \tilde{\mathcal{D}}) \geq min_support^*$.

```

1: for each  $p_i$  do-in-parallel
2:    $M_i \leftarrow \emptyset$ 
3:    $S_i \leftarrow \{b_j | b_j \in \mathcal{B}, i = \lceil j \cdot P / |\mathcal{B}| \rceil\}$ .
4:   for each  $b_k \in S_i$  in ascending order do
5:     perform depth-first search of  $[(b_k) | \{b_{k+1}, \dots, b_{|\mathcal{B}|}\}] \cap \tilde{\mathcal{F}}$ , visiting/discovering can-
       didates  $U \in [(b_k) | \{b_{k+1}, \dots, b_{|\mathcal{B}|}\}] \cap \tilde{\mathcal{F}}$  on an MFI by calling DFS-MFI-
       SCHEMA( $\tilde{\mathcal{D}}, min\_support^*, \{b_k\}, M'_i$ )
6:     Dynamic load-balancing: during the depth-first search we have to perform
       dynamic-load balancing. Each  $p_i$  has to check if it has work and if not it asks
       other processors for a PBEC. The processors can send to other processors only a
       PBEC with prefix of size 1. Therefore, at this point the set  $S_i$  can be modified,
       removing  $b \in S_i$  if it has been processed or scheduled to other processor and adding
        $b \in \mathcal{B}$  to  $S_i$  if it was send by another processor. We omit other details from the
       description of the algorithm.
7:     for each maximal itemset in  $U \in M'_i$  do
8:       if there is no  $W \in M_i$  such that  $U \subseteq W$  then
9:          $M_i \leftarrow M_i \cup \{U\}$ 
10:      end if
11:    end for
12:  end for
13: end for
```

Algorithm 11 works in the following way: because $\tilde{\mathcal{D}}$ is much smaller than the whole database \mathcal{D} , the processors replicates $\tilde{\mathcal{D}}$, i.e., every processor has a copy of the database sample $\tilde{\mathcal{D}}$ and knows the items that are frequent in the database \mathcal{D} (note that \mathcal{D} is distributed among the processors). All processors partition the base set \mathcal{B} to P blocks of size

$\approx |\mathcal{B}|/P$. Processor p_i runs a sequential DFS MFI algorithm in the i -th part of \mathcal{B} , where the items b_i are interpreted as 1-prefixes, i.e., prefix-based equivalence classes $[(b_i)]$. When a processor finishes its assigned items, it asks other processors for work. The computation is terminated using the Dijkstra's token termination detection algorithm. The output of the algorithm is a superset of all MFIs.

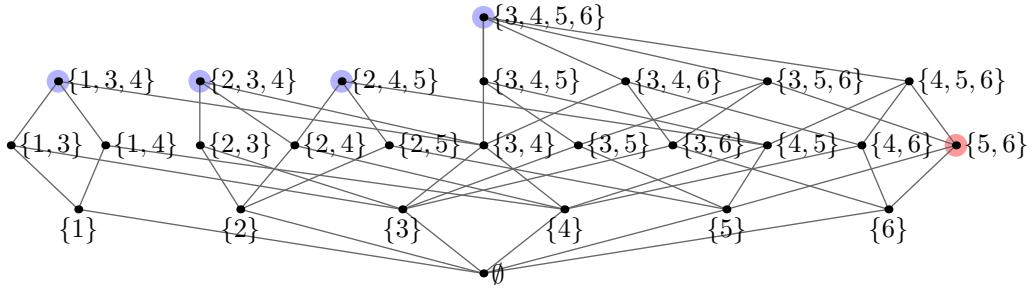
The approach described in the Algorithm 11 computes the set $M = \bigcup_i M_i$ such that $\widetilde{\mathcal{M}} \subseteq M$. The reason is the following:

1. every processor has its copy of the database sample $\widetilde{\mathcal{D}}$;
2. an arbitrary algorithm for mining of MFIs always correctly computes the support of an arbitrary itemset.

We demonstrate the parallel execution (the parallel processing of assigned PBECs) of a sequential DFS algorithm for mining of MFIs on the following example (for simplicity without dynamic load balancing): because the computation is distributed, the algorithm is unable to check the candidate against all already computed MFIs which results in a superset of all MFIs. Let $B = \{1, 2, 3, 4, 5, 6\}$ and $P = 3$ and assume that the prefix-based equivalence classes $[(1)|(2, 3, 4, 5, 6)], [(2)|(3, 4, 5, 6)]$ were assigned to p_1 ; the prefix-based equivalence classes $[(3)|(4, 5, 6)], [(4)|(5, 6)]$ were assigned to p_2 ; and the classes $[(5)|(6)], [(6)|\emptyset]$ to p_3 . The MFIs $\{\{1, 3, 4\}, \{2, 3, 4\}, \{2, 4, 5\}\}$ are correctly computed by p_1 . The processor p_2 correctly computes the MFI $\{3, 4, 5, 6\}$, but processor p_3 computes also the itemset $\{5, 6\}$ as an MFI. The reason is that p_3 does not know that the MFI $\{3, 4, 5, 6\}$ was already computed by processor p_2 . In Figure 7.1 the FIs, MFIs, and the additional itemset computed as MFI are shown.

Example 7.1: MFIs computed in parallel by a trivial parallelization of a DFS algorithm for mining of MFIs.

The FIs are computed from the database from Example 8.1 are marked by a dot, except the set \emptyset which is not an FI. The MFIs are marked in blue, the additionally computed itemset $\{5, 6\}$, which is a candidate on the MFI $\{3, 4, 5, 6\}$ in the PBEC $[(5)]$, is marked in orange. In this case the itemset $\{5, 6\}$ is also the longest subset of the MFI $\{3, 4, 5, 6\}$ in the PBEC $[(5)]$. The MFIs are computed with dynamic load-balancing on $P = 3$ processors. The processor p_1 is scheduled with $[(1)], [(2)]$; p_2 with $[(3)], [(4)]$; and p_3 with $[(5)], [(6)]$. The following picture shows the lattice of all FIs.



Lemma 7.4. Let $W = (b_{w_1}, \dots, b_{w_{|W|}})$ be an MFI, $b \in W$ any of its element. There exists at most one candidate on the MFI W in the PBEC $[(b)]$. If such candidate exists then it is the longest subset $S_W = \{b' | b' \in W, b \leq b'\}$ of the MFI W in the PBEC $[(b)]$.

Proof. In each PBEC $[(b)]$ all frequent sets $X \in [(b)]$ such that $X \subseteq W$ are always subsets of S_W . Consider sets $X \subsetneq S_W$: X cannot be a candidate because there exists an item $b \in S_W$ such that $X \cup \{b\}$ is frequent, due to the monotonicity of the support, see Theorem 2.12. \square

Note that S_W is a candidate if and only if there is no frequent itemset in $[(b)]$, which is a proper superset of S_W .

As stated in the proof of the lemma, in some cases the longest subset S_W is not a candidate on the MFI W . Let have an arbitrary other MFI $U = (b_{u_1}, \dots, b_{u_{|U|}})$, the item $b \in U, W$ and $S_U = \{b' | b, b' \in U; b \leq b'\}$ be the longest subset of the MFI U in the PBEC $[(b)]$. We discuss the cases of the MFI W and its longest subset S_W in the PBEC $[(b)]$. If for S_W holds $S_W \subsetneq S_U$ then the candidate on the MFI W does not exist in the PBEC $[(b)]$ because there exists $b' \in S_U$ such that $S_W \cup \{b'\}$ is frequent. If for S_W holds $S_W = S_U = S$ then there is a candidate S on both MFIs W, U . Therefore, the number of candidates of the MFI W depends on all other mined MFIs and subset/superset relations of the longest

subsets of all MFIs.

The following theorem is a corollary of Lemma 7.4:

Theorem 7.5. *Let have a baseset \mathcal{B} and $1 < P < |\mathcal{B}|$ processors p_1, \dots, p_P , a database $\tilde{\mathcal{D}}$, M_i be a set of itemsets computed by p_i in Algorithm 11, and $M = \bigcup_{1 \leq i \leq P} M_i$. Let W be the longest MFI, i.e., for all $U, W \in \tilde{\mathcal{M}}$ holds that $|U| \leq |W|$. An arbitrary DFS algorithm for mining MFIs that is executed in parallel, e.g., in Algorithm 11, computes a set of itemsets M , such that $\tilde{\mathcal{M}} \subseteq M$, of size:*

$$|\tilde{\mathcal{M}}| < |M| = \left| \bigcup_{1 \leq i \leq P} M_i \right| \leq |W| \cdot |\tilde{\mathcal{M}}|.$$

Proof. The proof of this theorem follows from the Lemma 7.4 and the fact that for each MFI U there are at most $|U|$ PBECs that contains some subsets of U , i.e., in the worst case the dynamic load-balancing causes that Algorithm 11 discovers all candidates on a single MFI. \square

If we do not use dynamic load-balancing and assign the items statically (each processor processing $|\mathcal{B}|/P$ PBECs), for an MFI $U = (b_{u_1}, \dots, b_{u_{|U|}})$ each p_i computes the candidate on the MFI U , if it exists, in each of its assigned PBECs with prefix of size 1. The **if** condition at line 8 of Algorithm 11 assures that from these candidates will be picked the longest candidate on the MFI U (in the sense of the cardinality of the candidates). Denote the longest MFI by W , as in the previous theorem. If we statically assign the PBECs to each processor and do not use dynamic load-balancing, the upper bound on $|M|$ is $|M| < P \cdot |\tilde{\mathcal{M}}|$. The two bounds can be combined: $|M| < \min(P, |W|) \cdot |\tilde{\mathcal{M}}|$.

8 Proposal of a new DM parallel method

In this section, we present our *new method*, called PARALLEL-FIMI that has three variants: PARALLEL-FIMI-SEQ [42], PARALLEL-FIMI-PAR [23], and PARALLEL-FIMI-RESERVOIR. The method provides parallelizations of the DFS (or BFS) sequential frequent itemsets mining algorithm. The method has the following advantages over current existing algorithms:

1. *It is universal*: with our method it is possible to parallelize any DFS algorithms for mining of frequent itemsets. It is even possible to parallelize BFS algorithms, though the performance of the Apriori algorithm could suffer in the candidate pruning phase.
2. *The computation is balanced statically*: if the database is very large, the dynamic load-balancing is out of question as the overhead of exchanging large partitions of a database and/or large data structures during dynamic-load balancing is too expensive.

Static load-balancing of the computation is not easy, as the amount of work for each prefix-based equivalence class is unknown.

In our approach, the static load-balancing is based on a heuristic and a sampling algorithm for estimating the size of the PBECs. The PBECs are then assigned to the processors, so that the processors perform approximately the same amount of work.

Our method also has the following property:

Result distribution: at the end of the execution of our parallel method, the frequent itemsets are distributed among the processors. This is an advantage, if we need to query for particular frequent itemsets. For example, we need to find all frequent itemsets containing the set $\{5, 8\}$ as a subset. Each processor gets the set $\{5, 8\}$, finds the FIs and sends them to the querying processor. In some cases, we need to send the FIs to a particular processor for further processing. The FIs distributed among processors could help for parallel computation of association rules. However, parallel computation of association rules goes beyond the scope of our work.

In this chapter, if we talk about *size of a PBEC* or *relative size of a PBEC*, we mean the *relative number of FIs* in the particular PBEC. If we talk about a partition $F \subseteq \mathcal{F}$ or $F \subseteq \tilde{\mathcal{F}}$ then the relative size of F is $|F|/|\mathcal{F}|$ or $|F|/|\tilde{\mathcal{F}}|$.

Our new method is called *Parallel Frequent Itemset MIning* (Parallel-FIMI in short). This method works for any number of processors $P \ll |\mathcal{B}|$. The basic idea is to partition all FIs into P disjoint sets F_i , using PBECs, of relative size $\frac{|F_i|}{|\mathcal{F}|} \approx \frac{1}{P}$. Each processor p_i then processes partition F_i .

The input and the parameters of the whole method are the following:

1. Minimal support: the real number $min_support^*$, see Definition 2.3.
2. The sampling parameters: real numbers $0 \leq \epsilon_{\tilde{\mathcal{D}}}, \delta_{\tilde{\mathcal{D}}}, \epsilon_{\tilde{\mathcal{F}}_s}, \delta_{\tilde{\mathcal{F}}_s} \leq 1$, see Section 6.
3. The relative size of a smallest PBEC: the parameter $\rho, 0 \leq \rho \leq 1$, see Sections 6.
4. Partition parameter: real number $\alpha, 0 \leq \alpha \leq 1$, see Section 8.2.
5. Database parts $D_i, 1 \leq i \leq P$: processor p_i loads its database partition D_i to a local memory. The database partitions D_i has the following properties: $D_i \cap D_j = \emptyset, i \neq j$, and $|D_i| \approx \frac{|\mathcal{D}|}{P}$.

Additionally, without loss of generality, we expect that each $b_i \in \mathcal{B}$ is frequent. Otherwise, each processor p_i computes local support of all items $b_j \in \mathcal{B}$ in its database part D_i . The support is then broadcast and each p_i removes all b_j that are not globally frequent.

The whole method consists of four phases. The first three phases are designed in such a way that they statically balance the load of the computation of all FIs. Phases 1–2 prepare the PBECs and its assignment to the processors for Phase 4, i.e., the static load-balancing is precompute in Phases 1–2. In the Phase 3, we redistribute the database partitions so each processor can proceeds independently with the assigned PBECs. In the Phase 4, we execute an arbitrary algorithm for mining of FIs and the processors computes the FIs in it assigned PBECs. To speed-up Phases 1–2, we can execute each of Phase 1–2 in parallel. The four phases are summarized below:

Phase 1 (sampling of FIs): the input of Phase 1 is the minimal support $min_support^*$, a partitioning of the database \mathcal{D} into P disjoint partitions D_i , and the real numbers $0 \leq \epsilon_{\tilde{\mathcal{D}}}, \delta_{\tilde{\mathcal{D}}}, \epsilon_{\tilde{\mathcal{F}}_s}, \delta_{\tilde{\mathcal{F}}_s} \leq 1$. Output of Phase 1 is a sample of frequent itemsets $\tilde{\mathcal{F}}_s$. Generally, the purpose of the first phase is to compute a sample $\tilde{\mathcal{F}}_s$ and create the database sample $\tilde{\mathcal{D}}$. First, each processor samples D_i (in parallel) and creates part \mathcal{D}'_i and broadcasts them to other processors (all-to-all broadcast). Each processor p_i then creates $\tilde{\mathcal{D}} = \bigcup_i \mathcal{D}'_i$. Then from $\tilde{\mathcal{D}}$ is computed $\tilde{\mathcal{F}}_s$. We propose three methods for creation of $\tilde{\mathcal{F}}_s$.

Phase 2 (lattice partitioning): the input of this phase is the sample $\tilde{\mathcal{F}}_s$, the database sample $\tilde{\mathcal{D}}$ (both computed in Phase 1) and the parameter α . In Phase 2, the algorithm creates prefixes $U_i \subseteq \mathcal{B}$ and the extensions Σ_i of disjoint PBECs $[U_i|\Sigma_i]$, and estimates the size of $[U_i|\Sigma] \cap \mathcal{F}$ using $\tilde{\mathcal{F}}_s$. p_1 assigns the PBECs $[U_i|\Sigma_i]$ to all processors and the PBECs together with the assignment are broadcast to all processors.

Phase 3 (data distribution): the input of this phase is the assignment of the prefixes U_i and the extensions Σ_i to the processors p_i and the database partitioning $D_i, i = 1, \dots, P$. Now, the processors exchange database partitions: processor p_i sends $S_{ij} \subseteq D_i$ to processor p_j such that S_{ij} contains transactions needed by p_j for computing support of the itemsets of its assigned PBECs.

Phase 4 (computation of FIs): as the input to each processor are the prefixes $U_i \subseteq \mathcal{B}$, the extensions Σ_i , and the database parts needed for computation of supports of itemsets $V \in [U_i] \cap \mathcal{F}$ and the original D_i . Each processor computes the FIs in $[U_i] \cap \mathcal{F}$ by executing an arbitrary sequential algorithm for mining of FIs. Additionally, each processor computes support of $W \subseteq U_i$ in D_i , i.e., $Supp(W, D_i)$. The supports are then send to p_1 and p_1 computes $Supp(W, \mathcal{D}) = \sum_{1 \leq i \leq P} Supp(W, D_i)$

In this chapter, we use the database in Example 8.1 to demonstrate Phases 1–4.

Example 8.1: (start of the running example)

The four phases of our method will be further demonstrated on the following database \mathcal{D} with $min_support = 5$ and $\mathcal{B} = \{1, 2, 3, 4, 5, 6\}$ (or equivalently $min_support^* = 0.3$):

TID	Transaction
1	{1, 2, 3, 4, 6}
2	{3, 5, 6}
3	{1, 3, 4}
4	{1, 2, 6}
5	{1, 3, 4, 5, 6}
6	{1, 2, 3, 4, 5}
7	{2, 3, 4, 5}
8	{2, 3, 4, 5}
9	{3, 4, 5, 6}
10	{2, 4, 5}
11	{1, 2, 4, 5}
12	{2, 3, 4, 5, 6}
13	{3, 4, 5, 6}
14	{4, 5, 6}
15	{1, 3, 4, 5, 6}

8.1 Detailed description of Phase 1

In Phase 1, we create a sample $\tilde{\mathcal{F}}_s$ of all frequent itemsets. The input of this phase, for processor p_i , are the database partitions D_i such that $D_i \cap D_j = \emptyset, i \neq j$, $|D_i| \approx |\mathcal{D}|/P$, the relative minimal support $min_support^*$, and the real numbers $0 \leq \epsilon_{\tilde{\mathcal{D}}}, \epsilon_{\tilde{\mathcal{F}}_s}, \delta_{\tilde{\mathcal{D}}}, \delta_{\tilde{\mathcal{F}}_s} \leq 1$. The output of this phase is the sample of FIs $\tilde{\mathcal{F}}_s$ and the database sample $\tilde{\mathcal{D}}$. We propose three methods for creation of $\tilde{\mathcal{F}}_s$. The input and the output is the same for all of the three proposed variants of Phase 1. For the details on sampling, see the Sections 6.1 and 6.2.

Without the knowledge of the process that creates the sample $\tilde{\mathcal{F}}_s$, we can demonstrate the purpose of $\tilde{\mathcal{F}}_s$ and the idea behind Phase 1 and the consequences of Phase 1 on the whole process of mining of FIs. The idea of Phase 1 is to create the sample $\tilde{\mathcal{F}}_s$ so that we can estimate the relative size of PBECs using $\tilde{\mathcal{F}}_s$ and making a set of PBECs that can be processed by a single processor, see Example 8.2.

This section is organized as follows: first, in Section 8.1.1 we propose two variants based on our modification of the *coverage algorithm*. Then, in Section 8.1.2, we propose a variant based on the reservoir sampling algorithm, i.e., we propose three variants of the first phase:

1. Compute the boundary M of $\tilde{\mathcal{F}}$, in the sense of set inclusion:
 - (a) Sequentially: the boundary in this case is the set $M = \tilde{\mathcal{M}}$, see [42]. This variant of Phase 1 is denoted by PHASE-1-COVERAGE-SAMPLING-SEQUENTIAL, resulting in the PARALLEL-FIMI-SEQ method.
 - (b) In parallel: the boundary in this case is a set M , such that $\tilde{\mathcal{M}} \subsetneq M \subsetneq \tilde{\mathcal{F}}$, see [23]. This variant of Phase 1 is denoted by PHASE-1-COVERAGE-SAMPLING-PARALLEL, resulting in the PARALLEL-FIMI-PAR method.

Using the boundary M , we create a sample $\tilde{\mathcal{F}}_s$ using the *modified coverage algorithm*, see Section 8.1.1. The details of parallel mining of MFIs and the boundary M are in Chapter 7.

2. Create the sample $\tilde{\mathcal{F}}_s$ by putting together an arbitrary sequential algorithm for mining of FIs and the so called *reservoir sampling*, see Section 8.1.2. For the details on the reservoir sampling algorithm, see Section 6.2.2. This variant of Phase 1 is denoted by PHASE-1-RESERVOIR-SAMPLING, resulting in the PARALLEL-FIMI-RESERVOIR method.

Example 8.2: (the running example) Example of Phase 1.

In this part of the example, we will show the sample obtained in Phase 1. The pictures show only the FIs \mathcal{F} of the database \mathcal{D} and the FIs $\tilde{\mathcal{F}}$ of $\tilde{\mathcal{D}}$ with $\text{min_support}^* = 0.3$. The red circles mark the sampled frequent itemsets $\tilde{\mathcal{F}}_s$ and the blue circles mark the MFIs $m \in \mathcal{M}$ or $m \in \tilde{\mathcal{M}}$.

Horizontal representation of the database \mathcal{D} :

TID	Transaction
1	{1, 2, 3, 4, 6}
2	{3, 5, 6}
3	{1, 3, 4}
4	{1, 2, 6}
5	{1, 3, 4, 5, 6}
6	{1, 2, 3, 4, 5}
7	{2, 3, 4, 5}
8	{2, 3, 4, 5}
9	{3, 4, 5, 6}
10	{2, 4, 5}
11	{1, 2, 4, 5}
12	{2, 3, 4, 5, 6}
13	{3, 4, 5, 6}
14	{4, 5, 6}
15	{1, 3, 4, 5, 6}

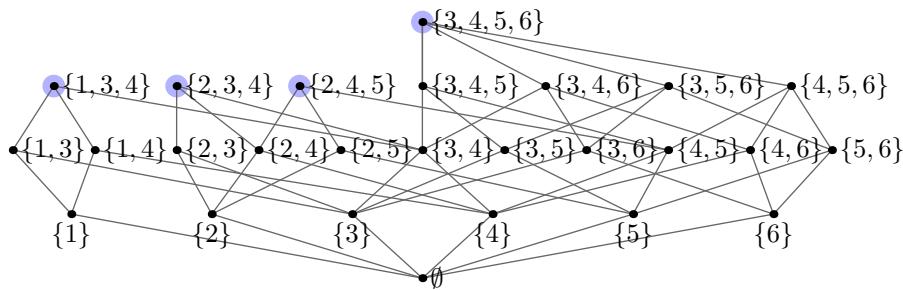
Horizontal representation of the database sample $\tilde{\mathcal{D}}$

TID	Transaction
1	{1, 2, 3, 4, 6}
7	{2, 3, 4, 5}
8	{2, 3, 4, 5}
9	{3, 4, 5, 6}
10	{2, 4, 5}
11	{1, 2, 4, 5}
12	{2, 3, 4, 5, 6}
13	{3, 4, 5, 6}
14	{4, 5, 6}
15	{1, 3, 4, 5, 6}

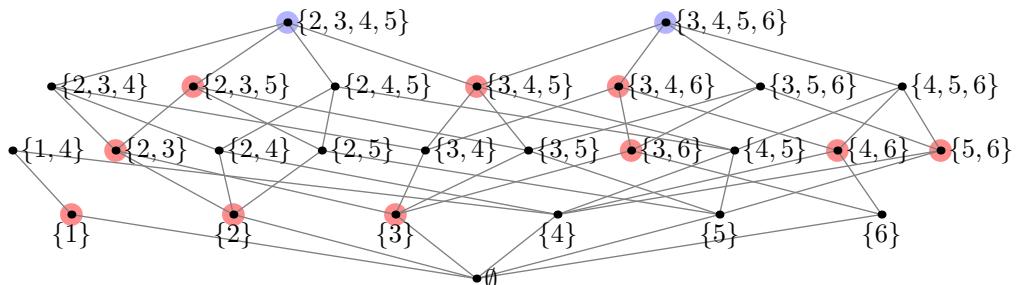
Prefix based equivalence classes with its relative sizes:

Prefix	Real relative size computed from \mathcal{D}	Real relative size computed from $\tilde{\mathcal{D}}$	Estimated relative size
{1}	4/25 = 0.1600	2/25 = 0.08	1/10 = 0.1
{2}	6/25 = 0.2399	8/25 = 0.32	3/10 = 0.3
{3}	8/25 = 0.3200	8/25 = 0.32	4/10 = 0.4
{4}	4/25 = 0.1600	4/25 = 0.16	1/10 = 0.1
{5}	2/25 = 0.0800	2/25 = 0.08	1/10 = 0.1
{6}	1/25 = 0.0400	1/25 = 0.04	0

The lattice representing the FIs \mathcal{F} and the MFIs \mathcal{M} in the database \mathcal{D} :



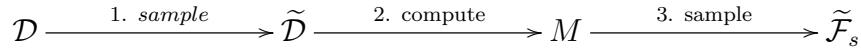
The lattice representing the FIs $\tilde{\mathcal{F}}$ and $\tilde{\mathcal{M}}$ in the database sample $\tilde{\mathcal{D}}$:



8.1.1 The modified coverage algorithm based sampling

In this section, we propose two variants of Phase 1 based on our modification of the *coverage algorithm*, see Algorithm 8. Additionally, we put together the fragments of the algorithms shown in previous chapters.

The workflow of the Phase 1 is summarized in Figure 8.3:



1. The sampling produces a database sample $\tilde{\mathcal{D}}$ of size $|\tilde{\mathcal{D}}| \geq \frac{1}{2\epsilon_{\tilde{\mathcal{D}}}^2} \ln \frac{2}{\delta_{\tilde{\mathcal{D}}}}$. For details see Section 6.1.
2. Computation of the boundary M of the set $\tilde{\mathcal{F}}$ using $\tilde{\mathcal{D}}$ is described in Chapter 7. The boundary M is then used for creation of the sample $\tilde{\mathcal{F}}_s \subseteq \tilde{\mathcal{F}} = \bigcup_{m_i \in M} \mathcal{P}(m_i)$. The boundary is created:
 - (a) sequentially, producing $M = \tilde{\mathcal{M}}$;
 - (b) in parallel, producing $\tilde{\mathcal{M}} \subseteq M \subsetneq \tilde{\mathcal{F}}$.
3. Creation of the sample $\tilde{\mathcal{F}}_s$ using M is performed using the MODIFIED-COVERAGE-ALGORITHM. For details see Section 6.2.1. The sample $\tilde{\mathcal{F}}_s$ is an independently but **not identically** distributed sample. Therefore, the estimates of a size of a PBEC using this sample is a *heuristic* for estimating the size of a prefix-based equivalence class.

Figure 8.3: The workflow of the coverage algorithm based sampling

(a) $\widetilde{\mathcal{M}}$ is computed sequentially [42]: the $\widetilde{\mathcal{M}}$ is computed on processor p_1 using an arbitrary algorithm for mining of MFIs. The sampling is performed sequentially by processor p_1 using the MODIFIED-COVERAGE-ALGORITHM. Phase 1 based on the sequential computation of MFIs. The pseudocode of Phase 1 is given in Algorithm 12:

Algorithm 12 The PHASE-1-COVERAGE-SAMPLING-SEQUENTIAL algorithm

PHASE-1-COVERAGE-SAMPLING-SEQUENTIAL(**In:** Database D_i ,

In: Double $\min_support^*$,

In: Double $\epsilon_{\tilde{\mathcal{D}}}$,

In: Double $\delta_{\tilde{\mathcal{D}}}$,

In: Double $\epsilon_{\tilde{\mathcal{F}}_s}$,

In: Double $\delta_{\tilde{\mathcal{F}}_s}$,

In: Double ρ ,

Out: Set $\tilde{\mathcal{F}}_s$,

Out: Database $\tilde{\mathcal{D}}$)

1: **for** all p_i **do-in-parallel**

$$2: \quad N_{\tilde{\mathcal{D}}} \leftarrow \frac{1}{2\epsilon_{\tilde{\mathcal{D}}}^2} \ln \frac{2}{\delta_{\tilde{\mathcal{D}}}}$$

3: $D'_i \leftarrow$ an i.i.d. sample of D_i of size $N_{\tilde{\mathcal{D}}}/P$

4: send D'_i to p_1 (an all-to-one gather)

5: **end for**

6: **processor** p_1 **executes:**

$$7: \quad \tilde{\mathcal{D}} \leftarrow \bigcup_{1 \leq j \leq P} D'_j$$

8: compute the approximation of MFIs $\widetilde{\mathcal{M}}$ from $\tilde{\mathcal{D}}$ using an arbitrary algorithm for mining of MFIs .

// The modified coverage algorithm

$$9: \quad N_{\tilde{\mathcal{F}}_s} \leftarrow \frac{4}{\epsilon_{\tilde{\mathcal{F}}_s}^2 \rho} \ln \frac{2}{\delta_{\tilde{\mathcal{F}}_s}}$$

10: call MODIFIED-COVERAGE-ALGORITHM($\widetilde{\mathcal{M}}, N_{\tilde{\mathcal{F}}_s}, \tilde{\mathcal{F}}_s$)

11: **end of** p_1 **execution**

At the lines 1–5, the PHASE-1-COVERAGE-SAMPLING-SEQUENTIAL algorithm creates the database sample $\tilde{\mathcal{D}}$ from the database partitions D_i that are collected by processor p_1 at the line 7. Then at line 8, p_1 executes an arbitrary algorithm for mining of MFIs. The creation of the sample $\tilde{\mathcal{F}}_s$ is performed at lines 6–11. The sampling is a heuristic based on the *modified coverage algorithm*, see Section 6.2.1.

(b) The set $M, \widetilde{M} \subseteq M \subsetneq \widetilde{\mathcal{F}}$ (\widetilde{M} plus some additional frequent itemsets) is computed in parallel [23]: the parallel variant of the MFI based sampling is summarized in Algorithm 13.

Algorithm 13 The PHASE-1-COVERAGE-SAMPLING-PARALLEL algorithm

PHASE-1-COVERAGE-SAMPLING-PARALLEL(**In:** Database D_i ,

In: Double $\min_support^*$,

In: Double $\epsilon_{\tilde{D}}$,

In: Double $\delta_{\tilde{D}}$,

In: Double $\epsilon_{\widetilde{\mathcal{F}}_s}$,

In: Double $\delta_{\widetilde{\mathcal{F}}_s}$,

In: Double ρ ,

Out: Set $\widetilde{\mathcal{F}}_s$,

Out: Database \widetilde{D})

1: **for** all p_i **do-in-parallel**

$$2: \quad N_{\tilde{D}} \leftarrow \frac{1}{2\epsilon_{\tilde{D}}^2} \ln \frac{2}{\delta_{\tilde{D}}}$$

$$3: \quad N_{\widetilde{\mathcal{F}}_s} \leftarrow \frac{4}{\epsilon_{\widetilde{\mathcal{F}}_s}^2 \rho} \ln \frac{2}{\delta_{\widetilde{\mathcal{F}}_s}}$$

4: $D'_i \leftarrow$ an i.i.d. sample of D_i of size $N_{\tilde{D}}/P$

5: broadcast D'_i (an all-to-all broadcast).

$$6: \quad \widetilde{D} \leftarrow \bigcup_{1 \leq j \leq P} D'_j.$$

7: Execute an arbitrary algorithm for mining of MFIs in parallel, p_i computing M_i from \widetilde{D} , e.g., call PARALLEL-DFS-MFI-SCHEMA($\widetilde{D}, \min_support^*, M_i$).

// Create the sample using the modified coverage algorithm

8: broadcast $s_i = \sum_{m \in M_i} |\mathcal{P}(m)|$ (hence an all-to-all scatter takes place).

$$9: \quad s \leftarrow \sum_{1 \leq i \leq P} s_i$$

10: $F_i \leftarrow \emptyset$.

11: call MODIFIED-COVERAGE-ALGORITHM($M_i, N_{\widetilde{\mathcal{F}}_s} \cdot \frac{s_i}{s}, F_i$)

12: send F_i to p_1 .

13: **end for**

14: processor p_1 computes $\widetilde{\mathcal{F}}_s = \bigcup_{1 \leq i \leq P} F_i$.

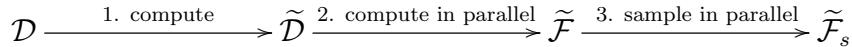
In Algorithm 13, an arbitrary modified DFS sequential algorithm for mining of MFIs is executed in parallel, see Chapter 7. The modified algorithm for mining of MFIs does not compute \widetilde{M} , but instead it computes a set $M = \bigcup_{1 \leq i \leq P} M_i$ such that $\widetilde{M} \subseteq M \subsetneq \widetilde{\mathcal{F}}$. The

computed sets are distributed among the processors and the number of these sets can be large. Therefore, we perform the sampling in parallel. For details of parallel mining of MFIs, see Chapter 7.

The parallel sampling of $\tilde{\mathcal{F}}$ using M , steps 8–12, is performed in the following way: every processor p_i broadcasts the sum $s_i = \sum_{m \in M_i} |\mathcal{P}(m)|$ of sizes of powersets of its local MFIs (hence, an all-to-all broadcast takes place), creates a fraction of sample $\tilde{\mathcal{F}}_s$ of size $|\tilde{\mathcal{F}}_s| \cdot \frac{s_i}{\sum_{1 \leq j \leq P} s_j}$, and finally sends them to p_1 . We should pick the number of samples chosen by each processor from a multivariate binomial distribution with parameters $p_i = \frac{s_i}{\sum_{1 \leq j \leq P} s_j}$ and $n = \sum_{1 \leq j \leq P} s_j$, see Appendix A.3 in order to be able to give guarantees on the error of the estimate. However, using the modified coverage algorithm makes from the sample just a heuristic. Therefore, we do not have any guarantees and p_i takes the number of samples $|\tilde{\mathcal{F}}_s| \cdot \frac{s_i}{\sum_{1 \leq j \leq P} s_j}$.

8.1.2 The sampling based on the reservoir algorithm

In the previous section, we have proposed a variant of Phase 1, based on the *modified coverage algorithm*, that samples \mathcal{F} non-uniformly. In this Section, we propose another variant of Phase 1: a sampling process based on the *reservoir sampling* [32] that samples $\tilde{\mathcal{F}}$ uniformly, i.e., it creates an identically distributed sample of $\tilde{\mathcal{F}}$. The workflow of the reservoir sampling algorithm is shown in Figure 8.4.



1. As in the previous section, we first need to produce the database sample $\tilde{\mathcal{D}}$ of size $|\tilde{\mathcal{D}}| = \frac{1}{2\epsilon_{\tilde{\mathcal{D}}}^2} \ln \frac{2}{\delta_{\tilde{\mathcal{D}}}}$. For details see Section 6.1.
2. From the database sample $\tilde{\mathcal{D}}$, we compute all FIs $\tilde{\mathcal{F}}$, using an arbitrary sequential algorithm for mining of FIs.
3. The output of the sequential algorithm for mining of FIs is sampled using the *reservoir sampling*, we produce $\tilde{\mathcal{F}}_s$ of size $|\tilde{\mathcal{F}}_s| = -\frac{\log(\delta_{\tilde{\mathcal{F}}_s}/2)}{D(\rho + \epsilon_{\tilde{\mathcal{F}}_s} || \rho)}$. For details see Section 6.2.2.

Figure 8.4: The workflow of the reservoir based sampling

In our parallel method, we are using the VITTER-RESERVOIR-SAMPLING Algorithm, the faster reservoir sampling algorithm. To speedup the sampling phase of our parallel method, we execute the reservoir sampling in parallel. The database sample $\tilde{\mathcal{D}}$ is distributed among the processors – each processor having a copy of the database sample $\tilde{\mathcal{D}}$. The baseset \mathcal{B} is partitioned into P parts $B_i \subseteq \mathcal{B}$ of size $|B_i| \approx |\mathcal{B}|/P$ such that $B_i \cap B_j = \emptyset, i \neq j$. Processor p_i then takes part B_i and executes an arbitrary sequential DFS algorithm for mining of FIs, enumerating $[(b_j)] \cap \tilde{\mathcal{F}}, b_j \in B_i$. The output, the itemsets $[(b_j)] \cap \tilde{\mathcal{F}}$, of the sequential DFS algorithm are read by the reservoir sampling algorithm. If a processor finished its part B_i , it asks other processors for work, hence performing dynamic load-balancing. For terminating the parallel execution, we use the Dijkstra's token termination algorithm.

The task of the Phase 1 is to take $|\tilde{\mathcal{F}}_s| = -\frac{\log(\delta_{\tilde{\mathcal{F}}_s}/2)}{D(\rho + \epsilon_{\tilde{\mathcal{F}}_s} || \rho)}$ samples, see Theorem 6.3. Because the reservoir algorithm and the sequential algorithm is executed in parallel, it is not known how many FIs is computed by each processor. Denote the unknown number of FIs computed on p_i by f_i , the total number of FIs is denoted by $f = \sum_{1 \leq i \leq P} f_i$. Because, we do not know f_i in advance, each processor samples $|\tilde{\mathcal{F}}_s|$ frequent itemsets using the reservoir sampling algorithm, producing $\tilde{\mathcal{F}}_s$, and counts the number of FIs computed by the sequen-

tial algorithm. When the reservoir sampling finishes, processor p_i sends f_i to p_1 . p_1 picks P random variables $X_i, 1 \leq i \leq P$ from multivariate hypergeometrical distribution, see Appendix A, with parameters $M_i = f_i$. The value of X_i is send to p_i . p_i then choose X_i itemsets $U \in \tilde{\mathcal{F}}_s$ at random out of the $|\tilde{\mathcal{F}}_s|$ sampled frequent itemsets computed by p_i . The samples are then send to processor p_1 . p_1 stores the received samples in $\tilde{\mathcal{F}}_s$. This process is summarized in Algorithm 14.

Algorithm 14 The PHASE-1-RESERVOIR-SAMPLING algorithm

PHASE-1-RESERVOIR-SAMPLING(**In:** Database D_i ,

In: Double $\min_support^*$,

In: Double $\epsilon_{\tilde{D}}$,

In: Double $\delta_{\tilde{D}}$,

In: Double $\epsilon_{\tilde{\mathcal{F}}_s}$,

In: Double $\delta_{\tilde{\mathcal{F}}_s}$,

In: Double ρ ,

Out: Set $\tilde{\mathcal{F}}_s$,

Out: Database \tilde{D})

1: **for** all processors p_i **do-in-parallel**

$$2: \quad N_{\tilde{\mathcal{F}}_s} \leftarrow -\frac{\log(\delta_{\tilde{\mathcal{F}}_s}/2)}{D(\rho + \epsilon_{\tilde{\mathcal{F}}_s} || \rho)}$$

$$3: \quad N_{\tilde{D}} \leftarrow \frac{1}{2\epsilon_{\tilde{D}}^2} \ln \frac{2}{\delta_{\tilde{D}}}$$

4: $D'_i \leftarrow$ an i.i.d. sample of D_i of size $N_{\tilde{D}}/P$

5: broadcast D'_i to p_1 (an all-to-all scatter).

$$6: \quad \tilde{D} \leftarrow \bigcup_{1 \leq j \leq P} D_j.$$

$$7: \quad R \leftarrow \text{array of size } N_{\tilde{\mathcal{F}}_s}$$

8: Partition \mathcal{B} on P parts \mathcal{B}_i , such that $\mathcal{B}_i \cap \mathcal{B}_j = \emptyset, i \neq j$.

9: Execute VITTER-RESERVOIR-SAMPLING($R, N_{\tilde{\mathcal{F}}_s}$, READNEXTFI, SKIPFIs) and the READNEXTFI(R) reads the output of an arbitrary sequential algorithm A_{FI} for mining of FIs with minimal support $\min_support^*$ and SKIPFIs(n) skips n FIs from the output of the algorithm. A_{FI} at processor p_i processes $[(b_k)], b_k \in \mathcal{B}_i$. If A_{FI} finishes its \mathcal{B}_i it asks other processors for work, performing dynamic load-balancing. The algorithm terminates using the Dijkstra's token termination algorithm.

// The number of **all** FIs computed by p_i is denoted by f_i .

10: f_i is broadcast to other processors (all-to-all-broadcast)

11: p_1 picks the random numbers X_i from the multivariate hypergeometric distribution with parameters $M_i = f_i$.

12: p_1 broadcasts X_i to other processors and each processor creates a sample $S_i \subseteq R, |S_i| = X_i$.

13: S_i is send to processor p_1 .

14: p_1 creates $\tilde{\mathcal{F}}_s \leftarrow \bigcup_{1 \leq j \leq P} S_j$.

15: **end for**

8.2 Detailed description of Phase 2

In Phase 2 the method partitions \mathcal{F} sequentially on processor p_1 . As an input of the partitioning, we use the samples $\tilde{\mathcal{F}}_s$, the database $\tilde{\mathcal{D}}$ (computed in Phase 1), and a real number $\alpha, 0 < \alpha \leq 1$. Recall that, we denote the prefixes by U_k , the extensions of U_k by Σ_k , i.e., U_k and Σ_k forms a PBEC $[U_k|\Sigma_k]$. In the following text, we omit Σ_k from the notation, i.e., a PBEC $[U_k|\Sigma_k]$ is denoted by $[U_k]$ if clear from context or if Σ_k is unnecessary. The set of the indexes of the PBECs assigned to processor p_i is denoted by L_i , and the set of all FIs assigned to processor p_i is denoted by F_i . Each F_i is the union of FIs in one or more PBECs $[U_k|\Sigma_k]$, i.e., $F_i = \bigcup_{k \in L_i} ([U_k|\Sigma_k]) \cap \mathcal{F}$. Each processor p_i then in Phase 4 processes the FIs contained in F_i . The output of Phase 2 are the index sets L_i of PBECs, computed on p_1 , and the PBECs $[U_k|\Sigma_k]$.

The partitioning of \mathcal{F} is a two step process:

- (1) p_1 creates a list of prefixes U_k such that the estimated relative size of the PBEC $[U_k] \cap \mathcal{F}$ satisfies $\frac{|[U_k] \cap \tilde{\mathcal{F}}_s|}{|\tilde{\mathcal{F}}_s|} \leq \alpha \cdot \frac{1}{P}$, where $0 < \alpha < 1$ is a parameter of the computation set by the user. The reason for making the PBECs of relative size $\leq \alpha \cdot \frac{1}{P}$ is to make the PBECs small enough so that they can be scheduled and the schedule is balanced, i.e., each processor having a fraction $\approx 1/P$ of FIs. Smaller number of large PBECs could make the scheduling unbalanced.
- (2) p_1 creates set of indexes L_i such that $|F_i|/|\mathcal{F}| \approx 1/P$.

(1) The creation of the prefixes U_k proceeds as follows: processor p_1 initially set $U_k = \{b_k\}, b_k \in \mathcal{B}$ and estimate the size of $[U_k] \cap \mathcal{F}$ using $\tilde{\mathcal{F}}_s$. The extensions of the initial U_k are the sets $\Sigma_k = \{b_i | b_k, b_i \in \mathcal{B}, b_k \in U_k \text{ and } b_k < b_i\}$. After the construction of U_k and Σ_k is finished, we estimate the relative size of $[U_k|\Sigma_k] \cap \mathcal{F}$ by $\frac{|[U_k|\Sigma_k] \cap \tilde{\mathcal{F}}_s|}{|\tilde{\mathcal{F}}_s|}$. If some of the PBEC $[U_k|\Sigma_k]$ is too big, i.e., $\frac{|[U_k|\Sigma_k] \cap \tilde{\mathcal{F}}_s|}{|\tilde{\mathcal{F}}_s|} > \alpha \cdot \frac{1}{P}$, the algorithm recursively partitions $[U_k|\Sigma_k]$ into smaller disjoint prefix-based equivalence subclasses with prefix $U_k \cup \{b_i\}, b_i \in \Sigma_k$ with extensions $\Sigma'_i = \{b_j | b_j \in \Sigma_k \text{ and } b_i < b_j\}$, i.e., the PBEC $[U_k \cup \{b_i\}|\Sigma'_i]$. The size of the parameter α influence the granularity of the partitioning.

The result of this process are the PBECs $[U_i|\Sigma_i]$ that are assigned to the processors and used in Phase 3 and 4. The pseudocode of the partitioning of the PBEC $[U|\Sigma]$, i.e., partitioning a prefix U and its extensions Σ , is summarized in Algorithm 15.

Algorithm 15 The PARTITION algorithm

PARTITION(**In:** Prefix U ,

In: Extensions Σ ,

In: Database $\tilde{\mathcal{D}}$,

In: Sample $\tilde{\mathcal{F}}_s$,

Out: Set Q)

1: sort $b_i \in \Sigma_k$ by the support in ascending order, i.e.,

$$\text{Supp}(U \cup \{b_1\}, \tilde{\mathcal{D}}) < \text{Supp}(U \cup \{b_2\}, \tilde{\mathcal{D}}) < \dots < \text{Supp}(U \cup \{b_{|\Sigma_k|}\}, \tilde{\mathcal{D}})$$

2: **for** $b \in \Sigma_k$ **do**

3: $U' \leftarrow U \cup \{b\}$

4: $\Sigma' \leftarrow \{b_i | b_i \in \Sigma_k, b < b_i\}$ – use the ordering created at line 1.

5: $s \leftarrow |[U'|\Sigma'] \cap \tilde{\mathcal{F}}_s|$, i.e., estimate the number of FIs in $[U'|\Sigma']$

6: $Q \leftarrow Q \cup \{(U', \Sigma', s)\}$.

7: **end for**

Proposition 8.1. Let $W \subseteq \mathcal{B}$ be a prefix and $\Sigma \subseteq \mathcal{B}$ its extensions and $\tilde{\mathcal{D}}$ a database sample and $\tilde{\mathcal{F}}_s$ a sample of FIs. Let $Q = \{(U_k, \Sigma_k, s_k)\}$ be the PBECS created by the PARTITION algorithm by calling PARTITION($W, \Sigma, \tilde{\mathcal{D}}, \tilde{\mathcal{F}}_s, Q$). The PBECS $[U_k|\Sigma_k], (U_k, \Sigma_k, s_k) \in Q$, are disjoint.

Proof. Trivially follows from the fact that the prefixes U_k are distinct, i.e., $U_i \cap U_j = \emptyset$, and the process of creation of the extensions. \square

In Chapter 2, we defined without loss of generality a single order of $b_i \in \mathcal{B}$: $b_1 < b_2 < \dots < b_{|\mathcal{B}|}$. But: a sequential DFS algorithms (like Eclat and FP-Growth) expands every prefix W_k using the extensions Σ_k sorted by the support in ascending order by the support of $b, b' \in \Sigma_k$ and $b < b'$ if and only if $\text{Supp}(W \cup \{b\}, \mathcal{D}) < \text{Supp}(W \cup \{b'\}, \mathcal{D})$, i.e., each prefix W_k can have different order of the extensions Σ_k . The dynamic re-ordering of items can significantly reduce the execution time of the sequential algorithm executed in Phase 4. To make the parallel algorithm fast, we have to use the same order as the sequential algorithm for mining of FIs, see Section B.4.2. To make the order the same as the sequential algorithm, we estimate the order of extensions Σ_k for prefix W_k using the supports from $\tilde{\mathcal{D}}$, i.e., $\text{Supp}(W \cup \{b\}, \tilde{\mathcal{D}}), \text{Supp}(W \cup \{b'\}, \tilde{\mathcal{D}})$. The different order of items for different prefix does not influence the output of a sequential algorithm for mining of

FIs. The details of the influence of the order of the items of \mathcal{B} in sequential algorithms are discussed in the Section B.4.2.

(2) The creation of the assignment, i.e., the index sets L_i of the prefix-based classes $[U_k]$ proceeds as follows: we need to create index sets L_i , such that $F_i = \bigcup_{k \in L_i} ([U_k] \cap \mathcal{F})$ and $\max_i |F_i|/|\mathcal{F}|$ is minimized, i.e., we want to schedule $\sum_i |L_i|$ tasks on P equivalent processors. The scheduling task is known NP-complete problem with known approximation algorithms. We use the LPT-SCHEDULE algorithm (LPT stands for least processing time). The LPT-SCHEDULE algorithm (see [16] for the proofs) is a best-fit algorithm, see Algorithm 16:

Algorithm 16 The LPT-SCHEDULE algorithm

LPT-SCHEDULE(**In:** Set $S = \{(U_i, \Sigma_i, s_i)\}$, **Out:** Sets L_i)

- 1: Sort the set S such that $s_i < s_j, i \neq j$.
 - 2: Assign each (U_i, Σ_i, s_i) (in decreasing order by s_i) to the least loaded processor p_k . The indexes assigned to p_k , are stored in L_k .
-

Lemma 8.2. [16] LPT-SCHEDULE is $4/3$ -approximation algorithm.

Let OPT be the time of the optimum schedule. The lemma says that the LPT-SCHEDULE algorithm finds a schedule with the time at most $4/3 \cdot \text{OPT}$.

The index sets L_i together with U_k and Σ_k are then broadcast to the remaining processors.

The pseudocode of Phase 2 is summarized in Algorithm 17. An example of the partitioning process is in Figure 8.5.

Algorithm 17 The PHASE-2-FI-PARTITIONING algorithm

PHASE-2-FI-PARTITIONING(**In:** Set $\tilde{\mathcal{F}}_s$,

In: Database $\tilde{\mathcal{D}}$,

In: Double α ,

Out: Set Q ,

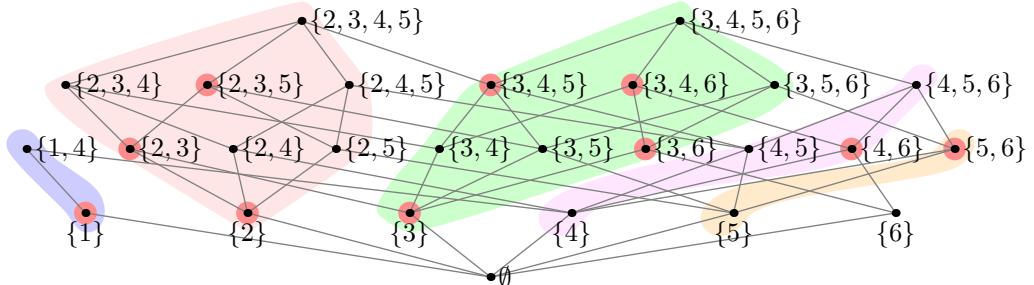
Out: Sets L_i)

- 1: create initial prefixes and extensions
 $Q \leftarrow \left\{ (U_k, \Sigma_k, s) \mid U_k = \{b_k\}, b_k \in \mathcal{B} \text{ and } \Sigma_k = \{b_i \mid b_k < b_i\}, s = \frac{|[U_k] \cap \tilde{\mathcal{F}}_s|}{|\tilde{\mathcal{F}}_s|} \right\}$
- 2: **while** exists $q = (U, \Sigma, s) \in Q$ such that $s > \alpha \cdot \frac{1}{P} \cdot |\tilde{\mathcal{F}}_s|$ **do**
- 3: select $q = (U, \Sigma, s) \in Q$ such that $s > \alpha \cdot \frac{1}{P} \cdot |\tilde{\mathcal{F}}_s|$
- 4: PARTITION($U, \Sigma, \tilde{\mathcal{D}}, Q'$)
- 5: $Q \leftarrow (Q \setminus \{q\}) \cup Q'$
- 6: **end while**
- 7: $L_i \leftarrow \emptyset, i = 1, \dots, P$
- 8: call LPT-SCHEDULE(Q, L_i)

Example 8.5: (the running example) Example of Phase 2.

This example shows the prefix-based classes, samples created using the modified coverage algorithm or the reservoir sampling, and the final assignment of the PBECs to the processors. The samples are marked by a red color.

The lattice partitioning computed from the database sample $\tilde{\mathcal{D}}$:



Processor	Assigned prefix-based classes	The estimated amount of work	Real amount of work
p_1	$[(3)]$	0.4	0.3076
p_2	$[(2)]$	0.3	0.2307
p_3	$[(1)], [(4)], [(5)], [(6)]$	0.3	0.4228

8.3 Detailed description of Phase 3

The input of Phase 3 for a processor p_i is the set of indexes of the assigned PBECs L_i together with the prefixes U_k and its extensions Σ_k . The processor p_i needs for the computation of $F_i = \bigcup_{k \in L_i} ([U_k] \cap \mathcal{F})$ a database partition D'_i that contain all the information needed for computation of F_i . At the beginning of this phase, the processors has disjoint database partitions D_i such that $|D_i| \approx \frac{|\mathcal{D}|}{P}$. For the description of the algorithm of Phase 3, we expect that we have a distributed memory machine whose nodes are interconnected using a network such as Myrinet [2] or Infiniband [1], i.e., a network that is not congested while an arbitrary permutation of two nodes communicates with each other. The problem is the congestion of the network in Phase 3.

To construct D'_i on processor p_i , every processor $p_j, i \neq j$, has to send a part of its database partition D_j needed by the other processors to all other processors (an all-to-all scatter takes place¹). That is: processor p_i send to processor p_j the set of transactions $\{t | t \in D_i, k \in L_j, \text{ and } U_k \dot{\subseteq} t\}$, i.e., all transactions that contain at least one $U_k, k \in L_j$ as a subset. Each processor then has the database part $D'_j = \bigcup_i \{t | t \in D_i, k \in L_j, \text{ and } U_k \dot{\subseteq} t\} = \{t | t \in \mathcal{D}, \exists k \in L_j, U_k \dot{\subseteq} t\}$.

Each round of the all-to-all scatter is done in $\lfloor \frac{P}{2} \rfloor$ parallel communication steps. We can consider the scatter as a round-robin tournament of P players [3]. Creating the schedule for the tournament is the following procedure: if P is odd, a dummy processor(player) can be added, whose scheduled opponent waits for the next round and the processors(player) performs P communication rounds(games). If P is even, then we perform $P - 1$ rounds of the parallel communication steps(games). For example let have 14 processors, in the first round the following processors exchange their database partitions:

1	2	3	4	5	6	7
14	13	12	11	10	9	8

The processors are paired by the numbers in the columns. That is, database parts are exchanged between processors p_1 and p_{14} , p_2 and p_{13} , etc.

In the second round one processor is fixed (number one in this case) and the other are rotated clockwise:

¹all-to-all scatter is a well known communication operation: each processor p_i sends a message m_{ij} to processor p_j such that $m_{ij} \neq m_{ik}, i \neq k$

1	14	2	3	4	5	6
13	12	11	10	9	8	7

This process is iterated until the processors are almost in the initial position:

1	3	4	5	6	7	8
2	14	13	12	11	10	9

In the description of the DB-PARTITION-EXCHANGE, Algorithm 18, we borrow the *ternary operator* ?: from C. The ternary operator has the form “`test ? value1 : value2`” that means: if `test` is true then return `value1` else return `value2`. The shift operation works like a bit shift operator, e.g., given an array $A = (1, 2, 3, 4, 5)$, the result of A shifted to the right is the array $(\text{undef}, 1, 2, 3, 4)$.

Algorithm 18 The PHASE-3-DB-PARTITION-EXCHANGE algorithm

PHASE-3-DB-PARTITION-EXCHANGE(**In:** Integer P , **In:** Prefixes $\{U_k\}$, **In:** Indexsets L_i , **Out:** Database parts D'_i)

```

1: for all processors  $p_i$  do-in-parallel
2:   if  $P$  is odd then
3:      $A_s \leftarrow (P - 1)/2$ 
4:   else
5:      $A_s \leftarrow P/2$ 
6:   end if
7:    $A_1 \leftarrow$  new array of size  $A_s$ ;  $A_2 \leftarrow$  new array of size  $A_s$ 
8:    $D'_i \leftarrow$  empty database
9:   Rounds  $\leftarrow P$  is odd ?  $P : P - 1$ 
10:  for  $q \leftarrow 1$  to  $A_s$  do
11:     $A_1[q] \leftarrow q$ ;  $A_2[A_s - q + 1] \leftarrow A_s + q$ 
12:  end for
13:  for  $m \leftarrow 1$  to Rounds do
14:     $\ell \leftarrow$  index  $\ell$  such that  $A_1[\ell] = i$  or  $A_2[\ell] = i$ 
15:    opponent  $\leftarrow A_1[\ell] = i ? A_2[\ell] : A_1[\ell]$ 
16:     $T \leftarrow$  all transactions  $t \in D_i$  such that  $U_k \dot{\subseteq} t$  and  $k \in L_{\text{opponent}}$ 
17:    if ( $P$  is odd and  $i \neq P + 1$ ) or  $P$  is even then
18:      if  $i <$  opponent then
19:        send transactions  $T$  to  $p_{\text{opponent}}$ 
20:        receive transactions  $T$  from  $p_{\text{opponent}}$  and store them in  $D'_i$ 
21:      else
22:        receive transactions  $T$  from  $p_{\text{opponent}}$  and store them in  $D'_i$ 
23:        send transactions  $T$  to  $p_{\text{opponent}}$ 
24:      end if
25:    end if
26:     $\text{tmp}_1 \leftarrow A_1[A_s]; \text{tmp}_2 \leftarrow A_2[1]$ 
27:    shift  $A_1[2..A_s]$  to the right
28:    shift  $A_2$  to the left
29:     $A_1[2] \leftarrow \text{tmp}_2$ ;  $A_2[A_s] \leftarrow \text{tmp}_1$ 
30:  end for
31: end for

```

8.4 Detailed description of Phase 4

The input to this phase, for processor $p_q, 1 \leq q \leq P$, is the database partition D_q (the database partition that is the input of the whole method, the database partition), the set $Q = \{(U_k, \Sigma_k) | U_k \subseteq \mathcal{B}, \Sigma_k \subseteq \mathcal{B}, U_k \cap \Sigma_k = \emptyset\}$ of prefixes U_k and the extensions Σ_k , and the sets of indexes L_q of prefixes U_k and extensions Σ_k assigned to processor p_q , and $D'_q = \bigcup_{1 \leq i \leq P} \{t | t \in \mathcal{D}, \text{ such that for each } k \in L_i \text{ holds } U_k \dot{\subseteq} t\}$ (the database received in Phase 3 from other processors).

In Phase 4, we execute an arbitrary algorithm for mining of FIs. The sequential algorithm is run on processor p_q for every prefix and extensions $(U_k, \Sigma_k) \in Q, k \in L_q$ assigned to the processor, i.e., p_q enumerates all itemsets $W \in [U_k|\Sigma_k], k \in L_q$. Therefore, the datastructures used by a sequential algorithm, must be prepared in order to execute the sequential algorithm for mining of FIs with particular prefix and extensions. To make the parallel execution of a DFS algorithm fast, we prepare the datastructures by simulation of the execution of the sequential DFS algorithm, e.g., to enumerate all FIs in a PBEC $[U_k|\Sigma_k]$ Phase 4 simulates the sequential branch of a DFS algorithm for mining of FIs up to the point the sequential algorithm can compute the FIs in $[U_k|\Sigma_k]$. An example of such a simulation is in Chapter 9. We describe Phase 4 in Algorithm 19 (D_q is the database partition loaded in Phase 2, D'_q is the database partition received in Phase 3):

Algorithm 19 The PHASE-4-COMPUTE-FI algorithm

PHASE-4-COMPUTE-FI(**In:** Set of prefixes $Q = \{(U_k, \Sigma_k)\}$,

In: Indexsets L_q ,
In: Database D_q ,
In: Database D'_q ,
In: Integer $\min_support$,
Out: Set \mathcal{F}_q)

- 1: **for** all processors p_i **do-in-parallel**
 - 2: compute support of itemsets $W \subseteq U_k$ in D_q , i.e., $Supp(W, D_q)$
 - 3: send $Supp(W, D_q)$ to p_1
 - 4: **end for**
 - 5: p_1 outputs W such that $\sum_{1 \leq i \leq P} Supp(W, D_i) \geq \min_support$.
 - 6: all p_q execute an arbitrary algorithm for mining of FIs in parallel that computes supports of $Supp(W, D'_q), W \in \bigcup_{k \in L_q} [U_k|\Sigma_k], (U_k, \Sigma_k) \in Q$ and adds them to \mathcal{F}_q .
-

8.5 The summary of the new parallel FIMI methods

From the previous discussion it follows that we can create three parallel FIMI methods. Two of the methods are based on the *modified coverage algorithm*. The third method leverages the *reservoir sampling*. The methods are described in such a way that they can be parametrized using an arbitrary algorithm for mining of MFIs and/or an arbitrary algorithm for mining of FIs. In this section, we show pseudocodes for the three methods.

8.5.1 The Parallel-FIMI-Seq method

The PARALLEL-FIMI-SEQ method is based on the *coverage algorithm* and computes the MFIs sequentially. It samples $\tilde{\mathcal{F}}$ non-uniformly, see Method 1.

Method 1 The PARALLEL-FIMI-SEQ method

PARALLEL-FIMI-SEQ(**In:** Double $\min_support^*$,

In: Double $\epsilon_{\tilde{\mathcal{D}}}$,

In: Double $\delta_{\tilde{\mathcal{D}}}$,

In: Double $\epsilon_{\tilde{\mathcal{F}}_s}$,

In: Double $\delta_{\tilde{\mathcal{F}}_s}$,

In: Double ρ ,

In: Double α ,

Out: Sets \mathcal{F}_i)

- 1: // Phase 1: sampling.
 - 2: Each processor p_i reads D_i .
 - 3: p_1 calls PHASE-1-COVERAGE-SAMPLING-SEQUENTIAL($D_i, \min_support^*, \epsilon_{\tilde{\mathcal{D}}}, \delta_{\tilde{\mathcal{D}}}, \epsilon_{\tilde{\mathcal{F}}_s}, \delta_{\tilde{\mathcal{F}}_s}, \rho, \tilde{\mathcal{F}}_s, \tilde{\mathcal{D}}$).
 - 4: // Phase 2: partitioning.
 - 5: p_1 does: $L_i \leftarrow \emptyset, 1 \leq i \leq P$.
 - 6: p_1 calls PHASE-2-FI-PARTITIONING($\tilde{\mathcal{F}}_s, \tilde{\mathcal{D}}, \alpha, Q$, sets L_i).
 - 7: p_1 broadcasts Q and the sets L_i to each other processor.
 - 8: **for** all processors p_i **do-in-parallel**
 - 9: // Phase 3: data distribution.
 - 10: PHASE-3-DB-PARTITION-EXCHANGE($P, \{U_k | u = (U_k, \Sigma_k, s) \in Q\}, L_i, D'_i$).
 - 11: // Phase 4: execution of arbitrary sequential algorithm for computation of FIs.
 - 12: $Q' \leftarrow \{(U_k, \Sigma_k) | u = (U_k, \Sigma_k, s) \in Q\}$.
 - 13: PHASE-4-COMPUTE-FI($Q', L_i, D_i, D'_i, \min_support^* \cdot \sum_i |D_i|, \mathcal{F}_i$).
 - 14: **end for**
-

8.5.2 The Parallel-FIMI-Par method

The PARALLEL-FIMI-PAR method is based on the coverage algorithm and computes the MFIs in parallel. It samples $\tilde{\mathcal{F}}$ non-uniformly, see Method 2.

Method 2 The PARALLEL-FIMI-PAR method

PARALLEL-FIMI-PAR(**In:** Double $min_support^*$,

In: Double $\epsilon_{\tilde{\mathcal{D}}}$,

In: Double $\delta_{\tilde{\mathcal{D}}}$,

In: Double $\epsilon_{\tilde{\mathcal{F}}_s}$,

In: Double $\delta_{\tilde{\mathcal{F}}_s}$,

In: Double ρ ,

In: Double α ,

Out: Sets \mathcal{F}_i)

- 1: **for** all processors p_i **do-in-parallel**
 - 2: // Phase 1: sampling.
 - 3: Read D_i .
 - 4: PHASE-1-COVERAGE-SAMPLING-PARALLEL($D_i, min_support^*, \epsilon_{\tilde{\mathcal{D}}}, \delta_{\tilde{\mathcal{D}}}, \epsilon_{\tilde{\mathcal{F}}_s}, \delta_{\tilde{\mathcal{F}}_s}, \tilde{\mathcal{F}}_s, \rho, \tilde{\mathcal{D}}$).
 - 5: **end for**
 - 6: // Phase 2: partitioning.
 - 7: p_1 does: $L_i \leftarrow \emptyset, 1 \leq i \leq P$.
 - 8: p_1 calls PHASE-2-FI-PARTITIONING($\tilde{\mathcal{F}}_s, \tilde{\mathcal{D}}, \alpha, Q$, sets L_i).
 - 9: p_1 broadcasts Q and the sets L_i to each other processor.
 - 10: **for** all p_i **do-in-parallel**
 - 11: // Phase 3: data distribution.
 - 12: PHASE-3-DB-PARTITION-EXCHANGE($P, \{U_k | u = (U_k, \Sigma_k, s) \in Q\}, L_i, D'_i$).
 - 13: // Phase 4: execution of arbitrary sequential algorithm for computation of FIs.
 - 14: $Q' \leftarrow \{(U_k, \Sigma_k) | u = (U_k, \Sigma_k, s) \in Q\}$, in parallel.
 - 15: PHASE-4-COMPUTE-FI($Q', L_i, D_i, D'_i, min_support^* \cdot \sum_i |D_i|, \mathcal{F}_i$).
 - 16: **end for**
-

8.5.3 The Parallel-FIMI-Reservoir method

This method samples $\tilde{\mathcal{F}}$ using the *reservoir sampling*. The reservoir sampling samples $\tilde{\mathcal{F}}$ uniformly. To make the reservoir sampling algorithm faster, the reservoir sampling is executed in parallel. The PARALLEL-FIMI-RESERVOIR method follows:

Method 3 The PARALLEL-FIMI-RESERVOIR method

PARALLEL-FIMI-RESERVOIR(**In:** Double $min_support^*$,

In: Double $\epsilon_{\tilde{\mathcal{D}}}$,
In: Double $\delta_{\tilde{\mathcal{D}}}$,
In: Double $\epsilon_{\tilde{\mathcal{F}}_s}$,
In: Double $\delta_{\tilde{\mathcal{F}}_s}$,
In: Double ρ ,
In: Double α ,
Out: Sets \mathcal{F}_i)

- 1: **for** all processors p_i **do-in-parallel**
 - 2: // Phase 1: sampling.
 - 3: Read D_i .
 - 4: PHASE-1-RESERVOIR-SAMPLING($D_i, min_support^*, \epsilon_{\tilde{\mathcal{D}}}, \delta_{\tilde{\mathcal{D}}}, \epsilon_{\tilde{\mathcal{F}}_s}, \delta_{\tilde{\mathcal{F}}_s}, \rho, \tilde{\mathcal{F}}_s, \tilde{\mathcal{D}}$)
 - 5: **end for**
 - 6: // Phase 2: partitioning.
 - 7: p_1 does: $L_i \leftarrow \emptyset, 1 \leq i \leq P$.
 - 8: p_1 calls PHASE-2-FI-PARTITIONING($\tilde{\mathcal{F}}_s, \tilde{\mathcal{D}}, \alpha, Q$, sets L_i).
 - 9: p_1 broadcasts Q and the sets L_i to each other processor.
 - 10: **for** all processors p_i **do-in-parallel**
 - 11: // Phase 3: data distribution.
 - 12: PHASE-3-DB-PARTITION-EXCHANGE($P, \{U_k | u = (U_k, \Sigma_k, s) \in Q\}, L_i, D'_i$).
 - 13: // Phase 4: execution of arbitrary sequential algorithm for computation of FIs.
 - 14: $Q' \leftarrow \{(U_k, \Sigma_k) | u = (U_k, \Sigma_k, s) \in Q\}$.
 - 15: PHASE-4-COMPUTE-FI($Q', L_i, D_i, D'_i, min_support^* \cdot \sum_i |D_i|, \mathcal{F}_i$).
 - 16: **end for**
-

9 Execution of the Eclat algorithm in Phase 4

In Chapter 11, we are evaluating the method with the sequential Eclat algorithm used in Phase 4. Therefore, in this Chapter, we show how to *efficiently execute the Eclat algorithm* on a processor p_q , $1 \leq q \leq P$ in Phase 4, so it efficiently process the assigned PBECs. The Eclat algorithm is a DFS algorithm that works with the tidlists, see Definition 2.4 and Section 2.4 for details. We omit the details of the Eclat algorithm, they are described in Section B.3. To execute the Eclat algorithm for one assigned PBEC, we have to prepare the tidlists for every assigned prefix and its extensions, by simulating one branch of the Eclat algorithm.

We denote the set of indexes of PBECs assigned to processor p_q by L_q . The task is to efficiently prepare the tidlists used by the Eclat algorithm in order to enumerate FIs $W \in \bigcup_{i \in L_q} [U_i | \Sigma_i] \cap \mathcal{F}$ (and its supports). We denote the database partition that was received by processor p_q in Phase 3 by D'_q , i.e., database D'_q contains all the necessary information needed to compute the support of itemsets $W \in \bigcup_{i \in L_q} [U_i | \Sigma_i]$.

In order to explain the execution of the Eclat algorithm in Phase 4, we need to define the lexicographical order:

Definition 9.1 (lexicographical order of two itemsets). *Let $U = (b_{u_1}, \dots, b_{u_{|U|}})$, $W = (b_{w_1}, \dots, b_{w_{|W|}})$ be two itemsets. We say that $U < W$ (U is lexicographically smaller than W) if and only if:*

1. $b_{u_i} = b_{w_i}$ for each $1 \leq i < k$ and $b_{u_k} < b_{w_k}$ for some $k \leq \min(|U|, |W|)$.
2. $|U| < |W|$ and $b_{u_i} = b_{w_i}$ for all $1 \leq i \leq |U|$.

At the start of Phase 4, processor p_q , $1 \leq q \leq P$, creates tidlists $\mathcal{T}(b_i, D'_q)$, $b_i \in \mathcal{B}$, i.e., p_q creates tidlists for each item b_i in its partition of the database D'_q .

Processor p_q has been assigned a set of prefixes. Let one PBEC, assigned to p_q , be $[U_i | \Sigma_i]$. The Eclat algorithm uses the tidlists for computation of supports. To prepare the execution of the Eclat algorithm for processing of $[U_i | \Sigma_i]$, we have to compute the tidlists of each itemset $U_i \cup \{b\}$, $b \in \Sigma_i$, i.e., $\mathcal{T}(U_i \cup \{b\}, D'_q)$. Each processor p_q has been assigned with a set of such prefixes. We denote the prefix an itemset X of size $k < |X|$ by X^{k-1} . To make the preparation of $\mathcal{T}(U_i \cup \{b\}, D'_q)$ efficient, we sort the PBECs $[U_i | \Sigma_i]$, $i \in L_q$ lexicographically in ascending order by U_i and prepare the tidlist of each prefix of U_i of length $\leq |U_i|$, i.e.,

$U_i^j, j \leq |U_i|$. We reuse the tidlists for preparation of processing of subsequent PBECs $[U_k|\Sigma_k], k > i$. The sorting of prefixes allows reuse of the already prepared tidlists. This is preformed using an array C that serves as a cache of tidlists. The array C contains at position j a pair $C[j] = (b, T)$, where $b \in \mathcal{B}$ is the j -th item in the prefix U_k , and $T = \{(b', \mathcal{T}(\{b'\} \cup U_k^j))\}, b' \in \Sigma_k$, is the set of pairs of the extensions of the prefix Σ_k and its tidlist in the database D'_q . We omit the details of the preparation of the tidlists, the details can be found in Section 2.4 and in Appendix B.

The PBECs are then processed sequentially one by one: when preparing the tidlists for the next prefix, U_{i+1} , we reuse the elements in the cache C that represents the longest common prefix of U_i and U_{i+1} .

Further, we denote the i th item of an itemset $U = (b_{u_1}, \dots, b_{u_{|U|}})$ by $U[i] = b_{u_i}$. The algorithm PREPARE-TIDLISTS, see Algorithm 20, summarizes the preparation of the tidlists for the sequential run of the Eclat algorithm. The algorithm EXEC-ECLAT, see Algorithm 21, summarizes the execution of the Eclat algorithm needed to processes the assigned PBECs. The EXEC-ECLAT is executed in parallel on each processor p_q . The Phase 4 parametrized with the Eclat algorithm is summarized in the PHASE-4-ECLAT algorithm.

Algorithm 20 The PREPARE-TIDLISTS algorithm

PREPARE-TIDLISTS(In/Out:** Array C of size $|\mathcal{B}|$, **In:** Pair (U, Σ))**

Notation: $U = (b_{u_1}, \dots, b_{u_{|U|}})$, $U[i] = b_{u_i}$.
 $C[i] = (b_i, T_i)$, $C[i].item = b_i$, $C[i].tidlists = T_i$.

```

1:  $n \leftarrow -1$ 
2: for  $i \leftarrow 1, \dots, |U|$  do
3:   if  $C[i].item \neq U[i]$  then
4:      $n \leftarrow i$ 
5:     break
6:   end if
7:    $C[i].tidlist \leftarrow$  prepare tidlists using  $\Sigma$  and  $C[i-1].tidlist$ 
8: end for
9: for  $i \leftarrow n, \dots, |U| - 1$  do
10:   $C[i] \leftarrow$  create new array element from  $C[i-1]$  using  $\Sigma$  and  $C[i-1].tidlists$ 
11: end for
12: for  $i \leftarrow |C|, \dots, |\mathcal{B}| - 1$  do
13:    $C[i] \leftarrow null$ 
14: end for

```

Algorithm 21 the EXEC-ECLAT algorithm

EXEC-ECLAT(In:** Prefixes and extensions $Q = \{(U_k, \Sigma_k)\}$,**

In: Integer $min_support$,

In: Database \mathcal{D} ,

Out: Set F)

```

1: sort  $Q$  lexicographically by  $U_k$ , i.e.,  $(U_i, \Sigma_i), (U_j, \Sigma_j) \in Q$  and  $U_i < U_j, i < j$ 
2:  $Q_{tidlists} \leftarrow$  array of size  $|\mathcal{B}|$  with  $Q_{tidlists}[i] \leftarrow null$ 
3:  $Q_{tidlists}[0] \leftarrow (\emptyset, \{(b_i, \mathcal{T}(b_i, \mathcal{D})) | b_i \in \mathcal{B}\})$ 
4: for all  $q = (U_i, \Sigma_i) \in Q$  such that  $i = 1, \dots, |Q|$  do
5:   PREPARE-TIDLISTS( $Q_{tidlists}, q$ )
6:   run the Eclat algorithm with prepared tidlists and extensions that are stored in
       $Q_{tidlists}[|U_i|]$  with support value  $min\_support$ . Output FIs into  $F$ .
7: end for

```

Algorithm 22 The PHASE-4-ECLAT algorithm

PHASE-4-ECLAT(**In:** Set of prefixes $S = \{(U_k, \Sigma_k)\}$,

In: Indexsets L_q ,

In: Database D_q ,

In: Database D'_q ,

In: Integer $\min_support$,

Out: Set F)

1: **for** all p_i **do-in-parallel**

2: computes support of itemsets $W \subset U_k$ in D_q , i.e., $Supp(W, D_q)$

3: send $Supp(W, D_q)$ to p_1

4: **end for**

5: p_1 puts all W into F' such that $\sum_i Supp(W, D_i) > \min_support$

6: each p_q executes EXEC-ECLAT($\{u = (U_k, L_k) | u \in S \text{ and } k \in L_q\}$, $\min_support$, D'_q , F_q) in parallel.

7: $F \leftarrow (\cup_{1 \leq i \leq P} F_q) \cup F'$

10 The database replication factor

In Phase 3 of the PARALLEL-FIMI methods, the processors must exchange database partitions in order to start Phase 4, i.e., Phase 3 re-distributes the database in such a way that the sequential algorithm for mining of FIs used in Phase 4 can compute the FIs. After Phase 3, the database must not be distributed evenly among the processors. To measure how the database is distributed among the processors, we use the *database replication factor*.

We define the database replication factor as a real number that determines the number of copies of a database that is spread among the processors. Let D'_i be the database partition received by p_i in Phase 3. The database replication factor is defined as:

$$\frac{\sum_{i=1}^P |D'_i|}{|\mathcal{D}|}$$

The database replication factor measures memory efficiency of our method by measuring the number of replications of \mathcal{D} among processors. We can handle the database replication factor in two ways:

1. hope that the database replication factor will be small;
2. reduce the database replication factor.

In this section, we will describe how to reduce the database replication factor. The measurements of the database replication factor of our method is given in Chapter 11.

10.1 Reduction of the replication factor

The LPT-SCHEDULE algorithm assigns the prefix-based equivalence classes to the processors based solely on their sizes. If we want to reduce the database replication, we have to consider the mutual sharing of the database partitions among the prefix-based classes. In this section, we will show how the problem of scheduling of the prefix-based classes with respect to the mutual share of transactions is related to the Quadratic Knapsack Problem (QKP in short). For a good source of information on knapsack problems, see [21].

The QKP can be defined as follows: let have n items and the j -th item having a positive integer weight w_j , and a limit on the total weight of the chosen items is given by a positive integer knapsack capacity c . In addition, we have a $n \times n$ *profit matrix* $S = (S_{ij})$, where S_{ij} is the profit of having item i together with item j in the knapsack. Additionally, we have indicator variables $x_i \in \{0, 1\}$ where $x_i = 1$ if the item i was selected to the knapsack and 0 otherwise. The QKP selects subset of items that fit in the knapsack and have maximal profit. The problem can be stated in the following way:

$$\begin{aligned} & \text{maximize} && \sum_i \sum_j S_{ij} x_i x_j \\ & \text{subject to} && \sum_j w_j x_j \leq c \end{aligned}$$

We can reformulate the QKP in the terms of our problem: let have a list of prefixes $P = \{U_i | U_i \subseteq \mathcal{B}\}$. The *profit matrix* S , contains the number of shared transactions for every two PBECs, i.e., $S_{ij} = |\mathcal{T}(U_i \cup U_j)|$, $i \neq j$ and $S_{ii} = 0$. The weight w_i is defined as the size of the prefix-based class $[U_i] \cap \mathcal{F}$. The size $|[U_i] \cap \mathcal{F}|$ is determined by the relative number of samples $\tilde{\mathcal{F}}_s$ belonging to $[U_i]$, i.e., $|[U_i] \cap \tilde{\mathcal{F}}_s| / |\tilde{\mathcal{F}}_s|$. The task is to put prefix-based equivalence classes into the knapsack, such that the size of the knapsack $c = \sum_i s_i / P$ while maximizing the share of transactions. This task is the same as solving the QKP. When we have a set of prefixes, we assign them to a processor, remove them from the set Q , update the matrix and the weight vector, and repeat the process until we assign all the prefix-based classes.

For the purpose of the database replication reduction algorithm, we denote the prefix-based equivalence class by a tuple (U_i, p_j^S) , where U_i is a prefix and p_j^S is the scheduled processor. The algorithm is summarized in Algorithm 23.

Algorithm 23 The DB-REPL-MIN algorithm

DB-REPL-MIN(**In/Out**: Prefixes $Q = \{(U_i, p_i^S)\}$, **In**: Profit matrix S)

```

1:  $p \leftarrow 1$ 
2: for all  $i$  do
3:    $p_i^S \leftarrow 0$ 
4: end for
5: for  $p \leftarrow 1, \dots, P$  do
6:    $S' \leftarrow$  a submatrix of  $S$  such that for all columns  $j$  and rows  $i$   $p_i^S = 0$  and  $p_j^S = 0$ 
7:   Using a QKP: find a subset of prefix-based classes,  $x_i = 1$ , with:  $\sum x_i \cdot w_i \leq c = \sum_i s_i / P$ .
8:   for all  $i$ ,  $x_i = 1$  do
9:      $p_i^S = p$ 
10:  end for
11: end for

```

This algorithm will not give the optimal solution, however, it should have better results, from the replication point of view, than the LPT-SCHEDULE algorithm because LPT-SCHEDULE does not consider the sharing of transactions.

11 Experimental evaluation

Our proposed method is a two step sampling process: 1) sampling of the database, creation of $\tilde{\mathcal{D}}$; 2) creation of a sample of FIs, $\tilde{\mathcal{F}}_s$, from the sampled database. Since the whole process is quite complicated and, as shown in the previous sections, theoretically we can make big error of the estimate of the size of a PBEC, we must experimentally show the performance of our method.

This chapter is organized as follows: in Section 11.1 we describe our implementation and the experimental setup, in Section 11.2 we describe the databases, in Section 11.3, we experimentally show the error of the estimate of the size of a set of PBECs, in Section 11.4 we experimentally evaluate the speedup of the proposed method and in Section 11.5 we evaluate the database replication factor.

11.1 Implementation and experimental setup

We have implemented our methods using the C++ language and the g++ compiler version 4.4.3 with the -O4 option (highest optimizations on speed of the resulting code). As the sequential algorithm, we have used the Eclat algorithm [37]. As the algorithm for mining of MFIs, we have chosen the *fpmax** [17] algorithm. The choice of the two algorithms is not accidental: we choose very fast algorithms. This makes our result more valuable because it is harder to achieve good speedup results: a very fast algorithm for mining of FIs and MFIs forces us to make the process of statical load-balancing more efficient. If we have used the Apriori algorithm for computation of FIs, we could have better speedup.

We have to modify the Eclat algorithm so it can be executed in parallel and the output could be read by the reservoir algorithm. We had also modified the *fpmax** algorithm so it runs in parallel. Both algorithms utilize the dynamic load-balancing with the Dijkstra's token termination algorithm. The dynamic load-balancing is limited to PBECs with prefix of size 1, see Section 7. As the implementation of the *fpmax** algorithm, we have used the implementation from the FIMI workshop [15]. The implementation of the Eclat algorithm was downloaded from [14].

We have preformed all the experiments with our methods on a cluster of workstations interconnected with the Infiniband network. Every node in the cluster has two dual-core 2.6GHz AMD Opteron processors and 8GB of main memory.

11.2 Databases

The experiments were performed on databases generated using the IBM database generator – which is a standard way for assessing the algorithms for mining of all FIs. We would like to use real datasets, however the standart datasets used as benchmarks are too small. We have used databases with 500k transactions and supports for each database such that the sequential run of the Eclat algorithm is between 100 and 12000 seconds (≈ 3.3 hours) and two cases with running time 33764 seconds (9.37 hours) and 132186 seconds (36.71 hours). The IBM generator is parametrized by the average transaction length TL (in thousands), the number of items I (in thousands), by the number of patterns P used for creation of the parameters, and by the average length of the patterns PL . To clearly differentiate the parameters of a database we are using the string `T[number in thousands]I[items count in 1000]P[number]PL[number]TL[number]`, e.g. the string `T500I0.4P150PL40TL80` labels a database with 500K transactions 400 items, 150 patterns of average length 40 and with average transaction length 80. All speedup experiments were performed with various values of the support parameter on 2, 4, 6, 10, 16, and 20 processors. The databases and supports used for evaluation of our methods is summarized in Table 11.2. *We have chosen the parameters of the IBM generator so that the distribution of the lengths of FI, the lengths of intersections of MFI, and of length of MFIs are similar to the same characteristics of some of the real databases.* However, mimicking the real dataset using the IBM generator is a hard task. The database characteristics are the following:

1. The distribution of intersections of MFIs: let have a set of MFIs \mathcal{M} . We have measured $|m_i \cap m_j|$, $m_i, m_j \in \mathcal{M}$ for particular choice of $min_support^*$ and compared the histograms of real databases and databases generated by the IBM generator.
2. The distribution of FIs of certain length: let have a set of FIs \mathcal{F} . We have measured $|U|$, $U \in \mathcal{F}$. We have measured the lengths for various values of $min_support^*$: we have split the interval $[0, 1]$ on $n = 1000$ values $i \cdot \frac{1}{n}$ for $i = 0, \dots, n - 1$ and compared histograms of real databases and databases generated by the IBM generator for each value of $i \cdot \frac{1}{n}$.
3. The distribution of lengths of MFI: let have a set of MFIs \mathcal{M} . We have drawn the histograms of $|m|$, $m \in \mathcal{M}$ for various values of $min_support^*$ and compared the histograms of the databases generated by the IBM generator to the histograms of real databases.

Database	Supports
T500I0.1P100PL20TL50	0.11, 0.12, 0.13, 0.14, 0.15, 0.16, 0.17, 0.18
T500I0.1P250PL10TL40	0.05, 0.07, 0.09, 0.1
T500I0.1P50PL10TL40	0.09, 0.1, 0.13, 0.15, 0.18
T500I0.1P50PL20TL40	0.05, 0.07, 0.09, 0.1
T500I0.4P250PL10TL120	0.2, 0.25, 0.26, 0.27, 0.3
T500I0.4P250PL20TL80	0.02, 0.03, 0.05, 0.07, 0.09
T500I0.4P50PL10TL40	0.02, 0.05, 0.07, 0.09
T500I1P100PL20TL50	0.02, 0.03, 0.05, 0.07, 0.09

Table 11.1: Databases used for measuring of the speedup and used supports values for each database.

We have chosen the datasets so these characteristics are close to the characteristics of real datasets, e.g., `connect`, `pumsb`, etc. The only exception to this choice is the T500I1P100PL20TL50 dataset. We omit details of the measurements because they are out of the scope of this thesis.

11.3 Evaluation of the estimate of the size of PBECs

In the previous chapters, we have shown that the parallel mining of FIs is a two stage sampling process. Some of the shown theorems suggest that the results of the double sampling process can be very bad, e.g., Theorem 6.4, Corollary 6.5, and Section 6.3. In this section, we show that the results are not that pessimistic, as shown in Section 6.3. The estimates are always made only using the samples taken by the VITTER-RESERVOIR-SAMPLING algorithm. The reason why we do not consider the sample taken by the MODIFIED-COVERAGE-ALGORITHM algorithm is that the estimates using the sample are just heuristics and we consider the PARALLEL-FIMI-RESERVOIR as the *major* result of this thesis.

We use the notation from our previous chapters: by U_j , we denote the prefixes of PBECs, by L_i we denote a set of indexes of prefixes assigned to processor p_i . The indexsets L_i are chosen as described in Section 8.2, i.e., $\frac{|\cup_{j \in L_i} [U_j] \cap \tilde{\mathcal{F}}_s|}{|\tilde{\mathcal{F}}_s|} \approx 1/P$.

We have made two experiments on each database:

1. *Experiment 1.* Measuring the error of a union of PBECs: The probability of the error of the estimation of the sizes of a union of PBECs: we show the probability of the error $\left| \frac{|\cup_{j \in L_i} [U_j] \cap \tilde{\mathcal{F}}|}{|\tilde{\mathcal{F}}|} - \frac{|\cup_{j \in L_i} [U_j] \cap \tilde{\mathcal{F}}_s|}{|\tilde{\mathcal{F}}_s|} \right|$. We have chosen $P = 5$ and $P = 10$.

2. *Experiment 2.* Error of the estimate of the amount of work per processor: for a set of prefixes $\{U_i\}$ and for P processors such that $\frac{|\bigcup_{j \in L_i} [U_j] \cap \tilde{\mathcal{F}}_s|}{|\tilde{\mathcal{F}}_s|} \approx 1/P$, we show a graph of probability of the error $\left| \frac{1}{P} - \frac{|\bigcup_{j \in L_i} [U_j] \cap \mathcal{F}|}{|\mathcal{F}|} \right|$. This graph is the most important for our work. See Figures 11.6–11.12

Detailed description of experiment 1: We have performed the measurements with the following parameters: 1) we have chosen $P = 5$ processors and $|\tilde{\mathcal{F}}_s| = 1'001'268$ samples; 2) we have chosen $P = 5$ and $|\tilde{\mathcal{F}}_s| = 26492$. For both measurements, we have chosen $|\tilde{\mathcal{D}}| = 42586$ and $|\tilde{\mathcal{D}}| = 14450$, various values of *min_support**. These parameters were mixed resulting into four graphs: two graphs for $P = 5$: 1) $|\tilde{\mathcal{F}}_s| = 1'001'268$; 2) $|\tilde{\mathcal{F}}_s| = 26492$ and two graphs for $P = 10$: 1) $|\tilde{\mathcal{F}}_s| = 1'001'268$; 2) $|\tilde{\mathcal{F}}_s| = 26492$. We show typical results of the measurements 1: one figure for measurements 1, see Figures 11.1–11.5. Note that each experiment is performed on a single database with *various values of min_support**

The graphs in Figures 11.1–11.5 show the typical results of the measurement of the probability of the error (experiment 1). We have measured the probability $\delta_{\tilde{\mathcal{F}}_s}$ of the error $\epsilon_{\tilde{\mathcal{F}}_s}$ of the estimation of the union of PBECs that were scheduled in Phase 2 for particular number of samples, see Algorithm 17 (the PHASE-2-FI-PARTITIONING Algorithm). We denote the error of the estimation of *single* PBEC by $\epsilon_{\tilde{\mathcal{F}}_s}$, its probability by $\delta_{\tilde{\mathcal{F}}_s}$, and the number of PBECs by N , the number of items by $|\mathcal{B}|$ and the length of the longest prefix by ℓ . The figures show four lines: black is the measured probability of the error; red is the probability of the error computed as $\epsilon_{\tilde{\mathcal{F}}_s} \cdot N$ with probability $\delta_{\tilde{\mathcal{F}}_s}$; violet line is the probability of the error computed as $\epsilon_{\tilde{\mathcal{F}}_s} \cdot N$ with probability $\delta_{\tilde{\mathcal{F}}_s} \cdot N$; the blue line is the probability of the error computed as $\epsilon_{\tilde{\mathcal{F}}_s} \cdot \binom{|\mathcal{B}|}{\ell}$ and the green line is the error $\epsilon_{\tilde{\mathcal{F}}_s} \cdot \binom{|\mathcal{B}|}{\ell}$ with probability $\delta_{\tilde{\mathcal{F}}_s} \cdot \binom{|\mathcal{B}|}{\ell}$. The green line is the real theoretical upper bound on the probability. The reason is that we have to consider *independent* PBECs, however the PBECs are dependent on the sample $\tilde{\mathcal{F}}_s$, see the PHASE-2-FI-PARTITIONING Algorithm. In the figures, some of the lines are missing: the reason is that the lines are out of the graph on the right. The lines should always be in the following order: (from left to right) red, violet, blue. The green line is the correct theoretical upper bound. We can view all PBECs with prefix size ℓ as independent, the number of such prefixes is $\binom{|\mathcal{B}|}{\ell}$. The other lines are shown to see how big is the influence of each factor and the dependence of PBECs on the sample.

The result of the experiment is: the theoretical upper bound is too loose and the probability of the error is usually reasonable for practical purposes.

Detailed description of experiment 2: Figures 11.6–11.12 show the results of the double

sampling process, i.e., the size of the union of the PBECs created in Phase 1 and 2, processed by one processor. There are combination of dashed and solid line with two colors: red and blue. That is: four lines per graph. The red color indicates measurement with $|\tilde{\mathcal{D}}| = 42856$ and the blue color indicates measurements with $|\tilde{\mathcal{D}}| = 14450$. The solid line shows the probability of the error with $|\tilde{\mathcal{F}}_s| = 1001268$ and the dashed line shows the probability of the error with $|\tilde{\mathcal{F}}_s| = 26492$. The left hand graph shows the measurements for $P = 5$ and the right hand graph show the measurements for $P = 10$. It can be seen from the graphs that the larger database sample the smaller the probability of the error. The probability of the error is almost the same for different size of $|\tilde{\mathcal{F}}_s|$. The exception of this is Figure 11.7: in this figure the probability of error is lower for larger database size and bigger for smaller database size (and the size of the sample almost does not matter).

In addition to the measurements, we have computed for each database the number of PBECs that make 96% of the total number of FIs. We denote the set of the prefixes of the 96% of PBECs by $S = \{U\}$, i.e., $\sum_{U \in S} |[U] \cap \mathcal{F}| \geq 0.96 \cdot |\tilde{\mathcal{F}}_s|$. We have discovered that 96% of all samples are contained in $\approx 100\text{--}200$ PBECs (the number of all PBECs varies between $\approx 300\text{--}3000$). Let $V_{min} = \arg \min_{W \in S} |[W] \cap \mathcal{F}|$ be the prefix of the smallest PBEC created in Phase 2, we have measured the relative size of the smallest PBEC $|[V_{min}] \cap \mathcal{F}| \approx 0.0007\text{--}0.003$. Therefore, the value of ρ can be chosen between 0.0007-0.003, depending on the database.

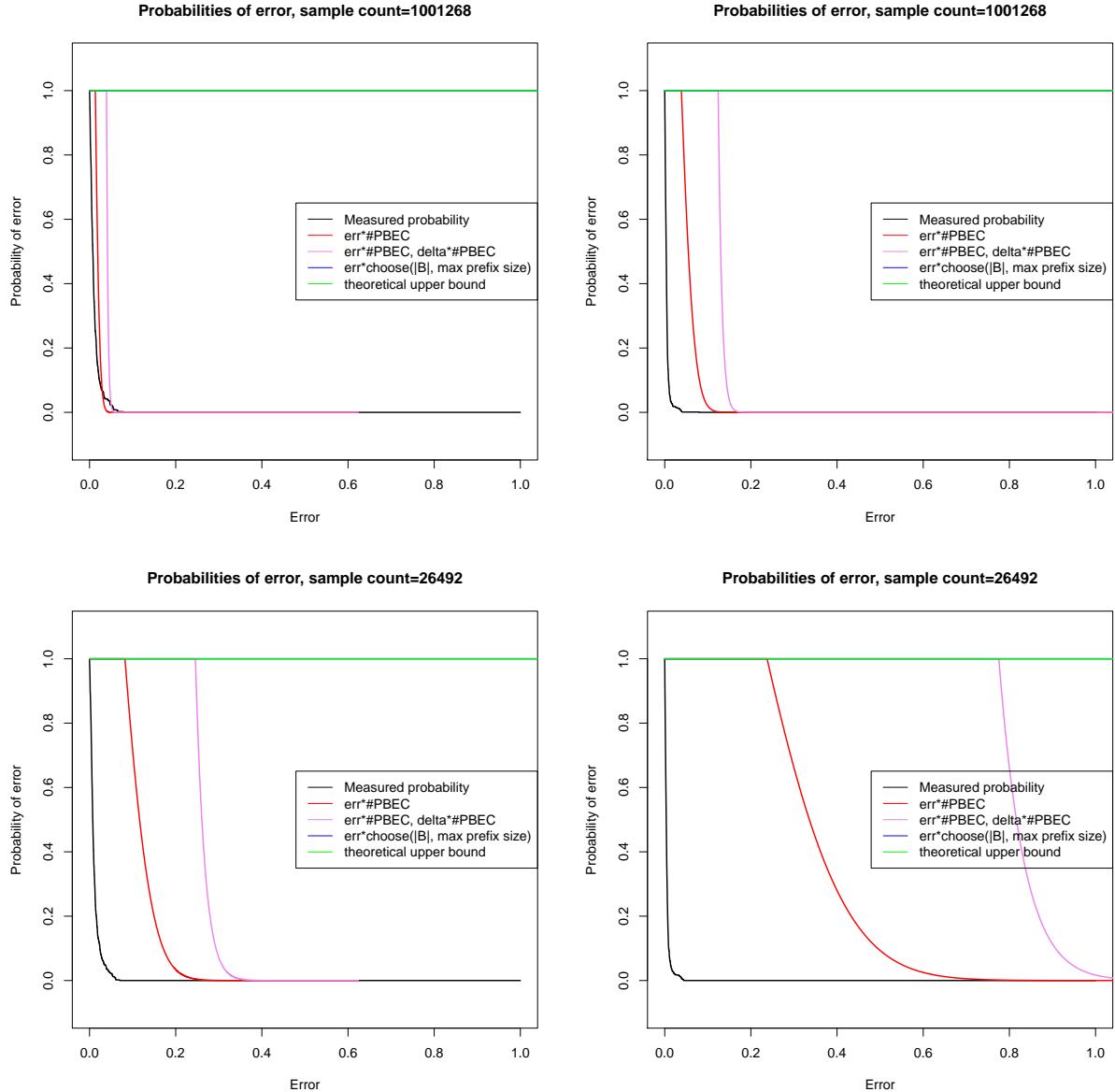


Figure 11.1: The T500I0.1P50PL10TL40 database: probability of error of the estimation of the union of PBECs created in Phase 2 for $P = 5$ on the left hand graphs and for $P = 10$ on the right hand graphs.

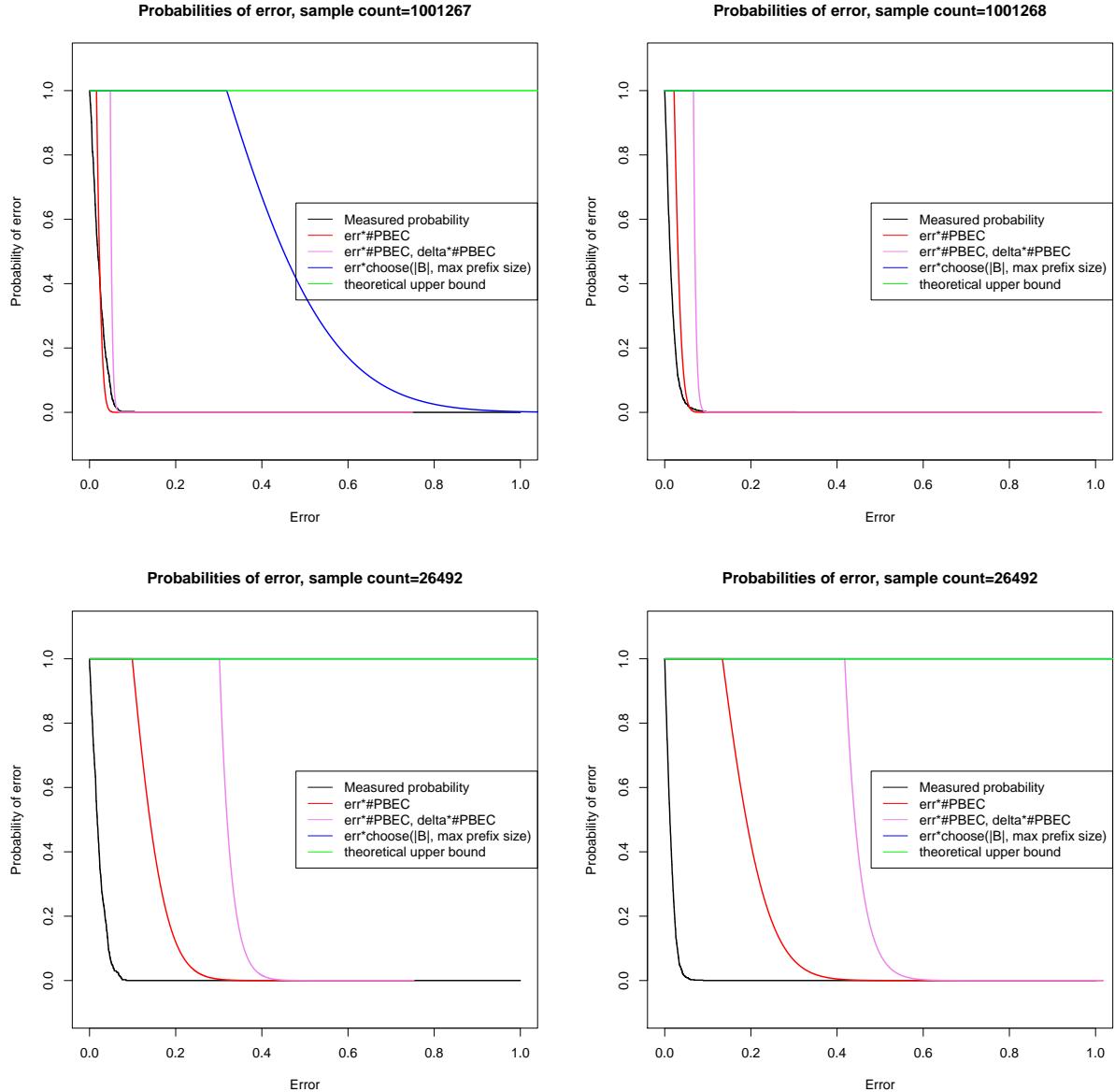


Figure 11.2: The T500I0.1P50PL20TL40 database: probability of error of the estimation of the union of PBECs created in Phase 2 for $P = 5$ on the left hand graphs and for $P = 10$ on the right graphs.

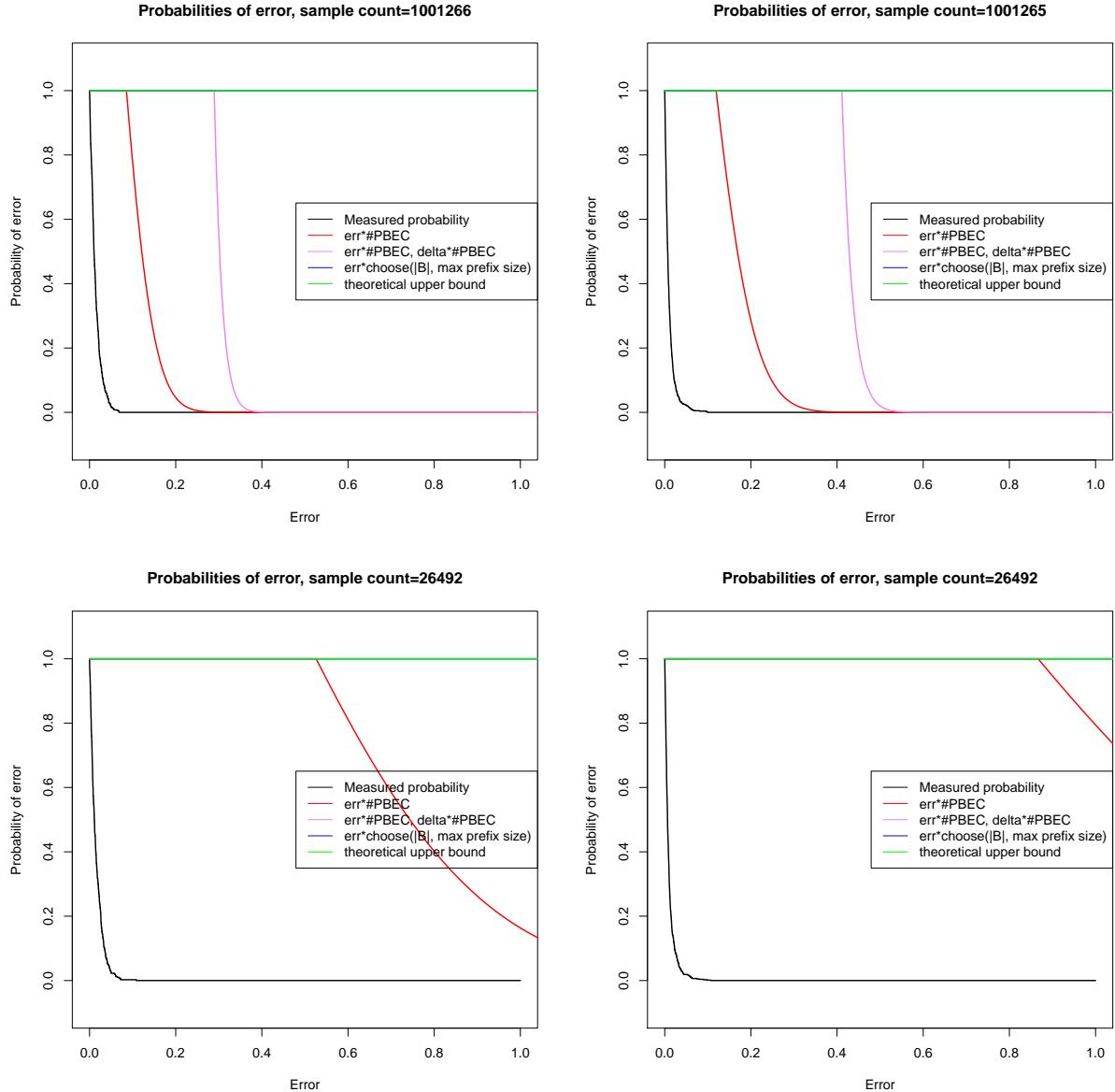


Figure 11.3: The T500I0.4P250PL20TL80 database: probability of error of the estimation of the union of PBECs created in Phase 2 for $P = 5$ on the left hand graphs and for $P = 10$ on the right hand graphs.

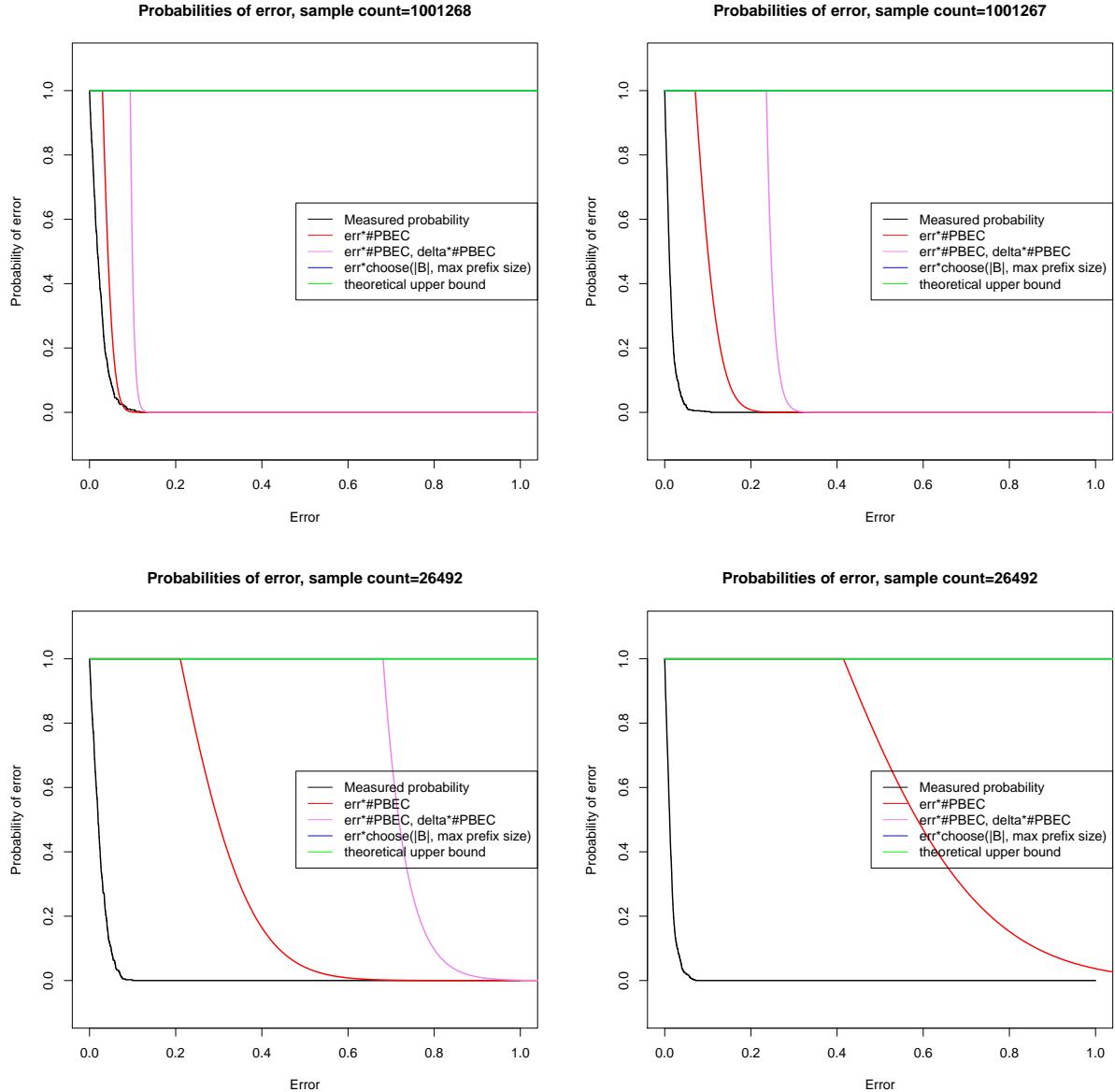


Figure 11.4: The T500I0.4P50PL10TL40 database: probability of error of the estimation of the union of PBECs created in Phase 2 for $P = 5$ on the left hand graphs and for $P = 10$ on the right hand graphs.

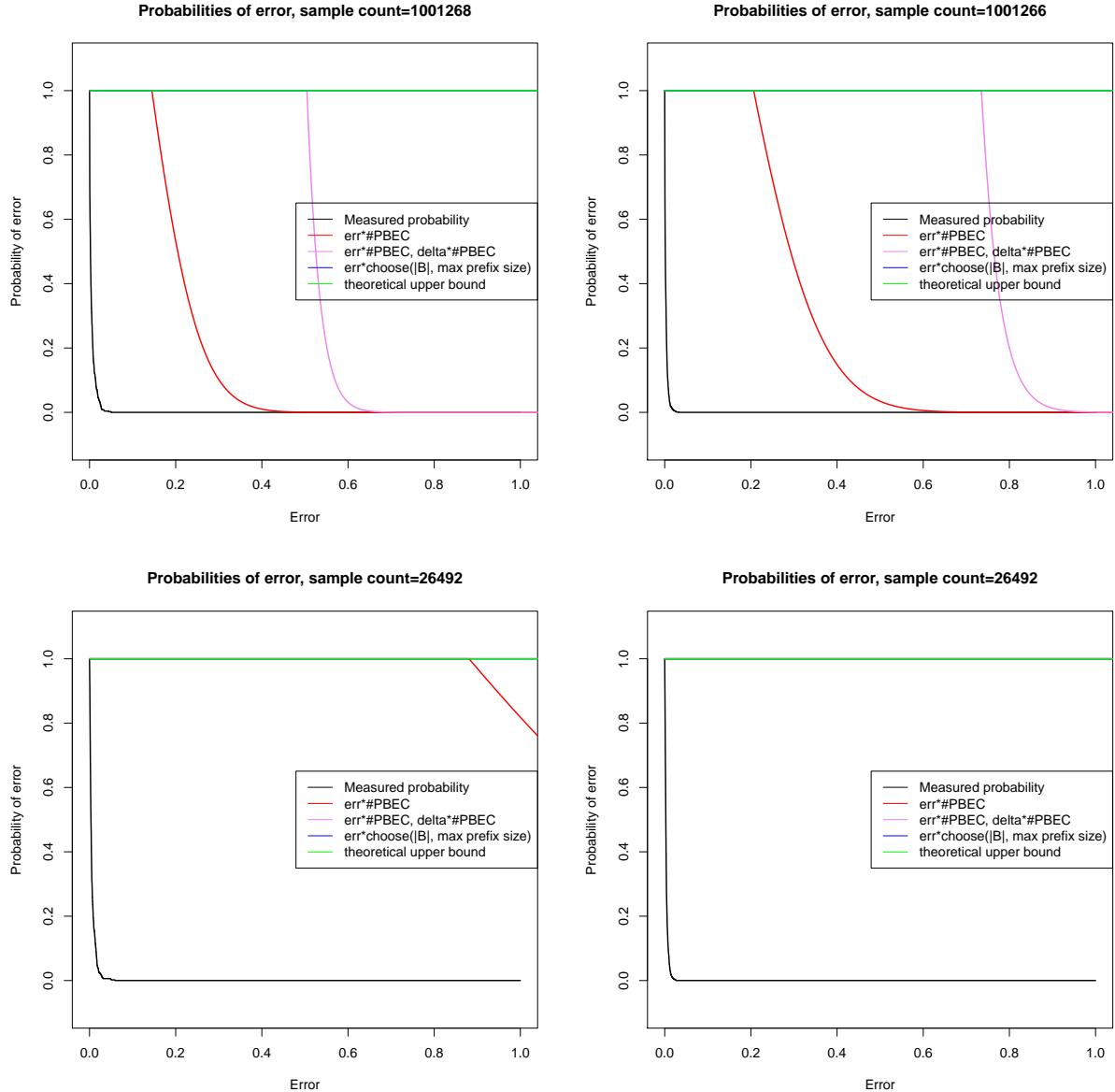


Figure 11.5: The T500I1P100PL20TL50 database: probability of error of the estimation of the union of PBECs created in Phase 2 for $P = 5$ on the left hand graphs and for $P = 10$ on the right hand graphs.

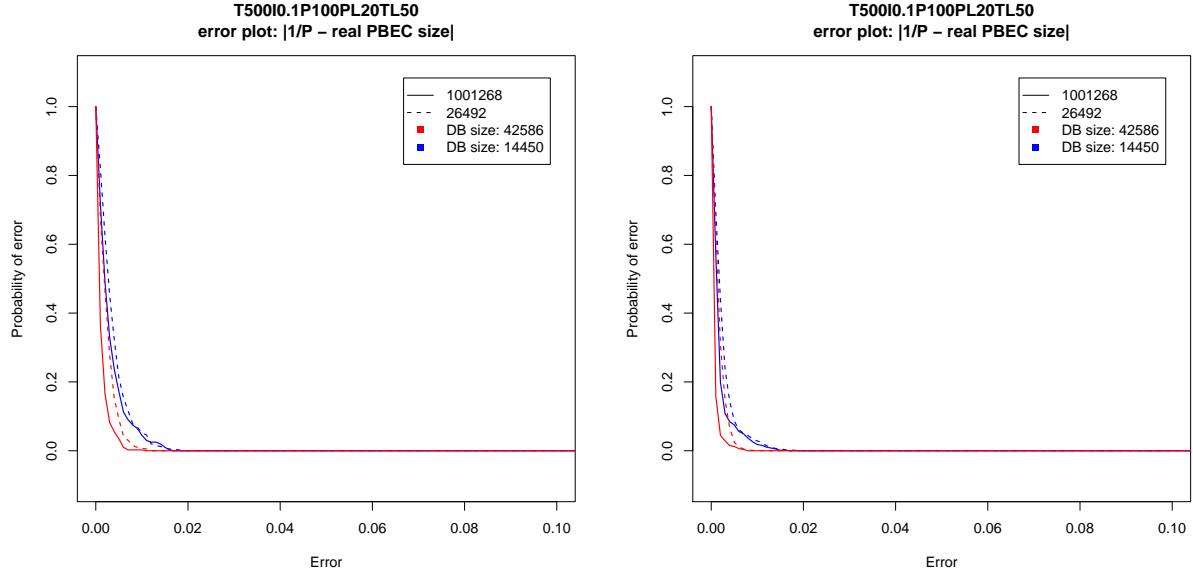


Figure 11.6: Probability of error of the estimation of the union of PBECs using a database sample created in Phase 1 and 2. Experiments made using $P = 5$ processors (left) and $P = 10$ processors (right). The T500I0.1P100PL20TL50 database.

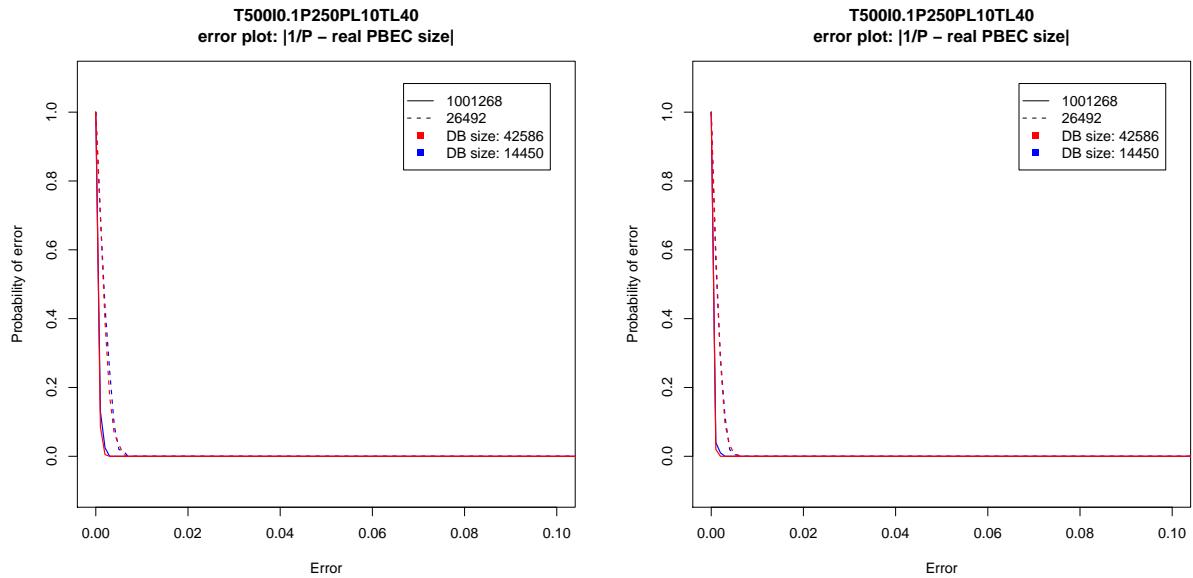


Figure 11.7: Probability of error of the estimation of the union of PBECs using a database sample created in Phase 1 and 2. Experiments made using $P = 5$ processors (left) and $P = 10$ processors (right). The T500I0.1P250PL10TL40 database.

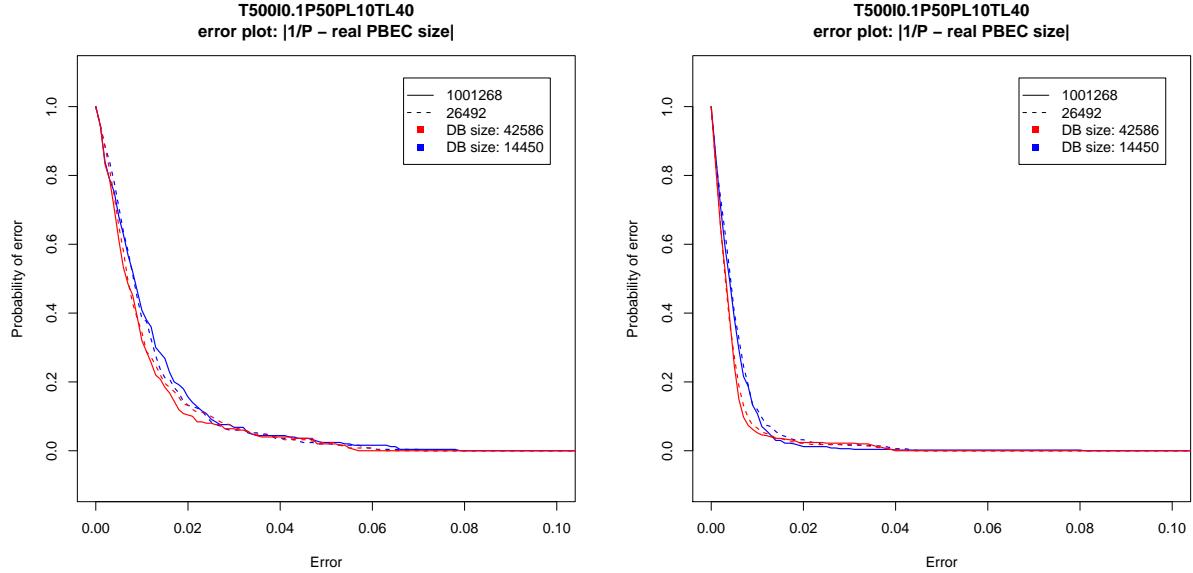


Figure 11.8: Probability of error of the estimation of the union of PBECs using a database sample created in Phase 1 and 2. Experiments made using 5 processors (left) and 10 processors (right). The T500I0.1P50PL10TL40 database.

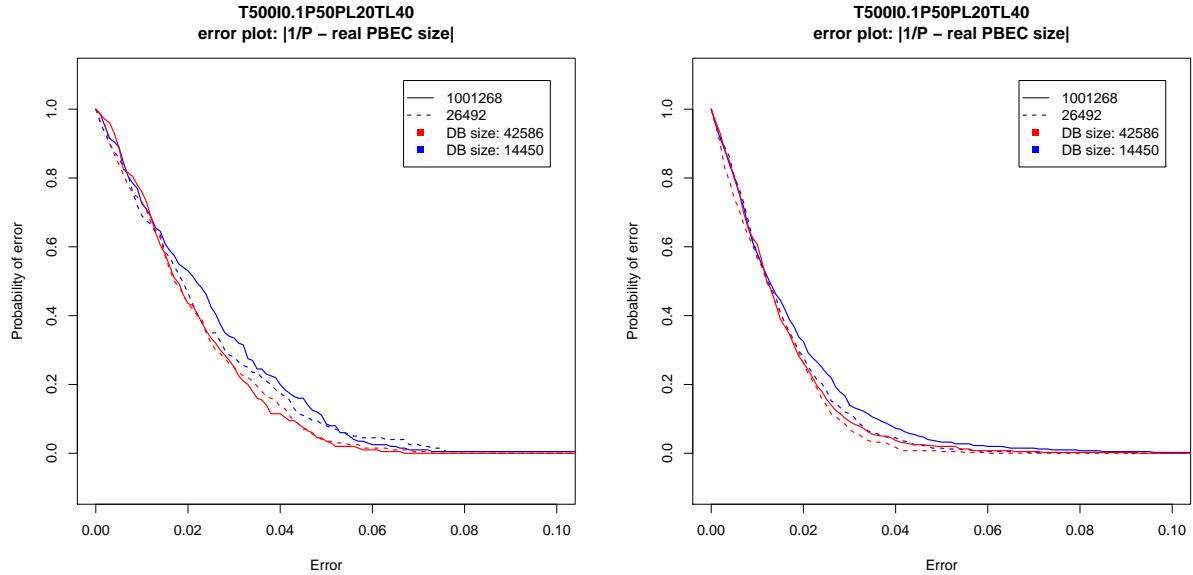


Figure 11.9: Probability of error of the estimation of the union of PBECs using a database sample created in Phase 1 and 2. Experiments made using $P = 5$ processors (left) and $P = 10$ processors (right). The T500I0.1P50PL20TL40 database.

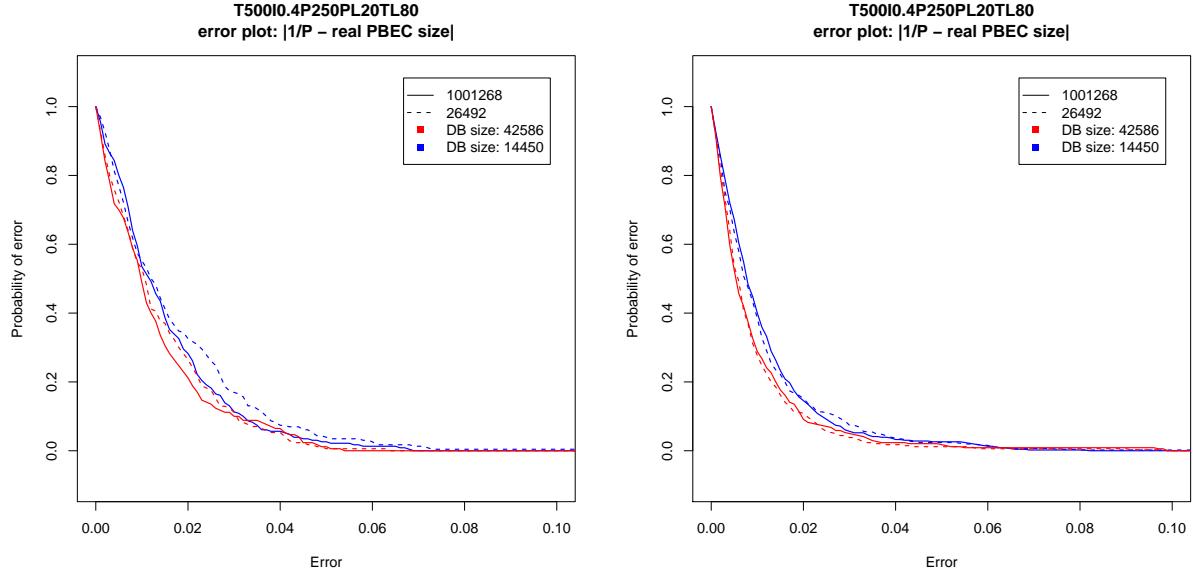


Figure 11.10: Probability of error of the estimation of the union of PBECs using a database sample created in Phase 1 and 2. Experiments made using $P = 5$ processors (left) and $P = 10$ processors (right). The T500I0.4P250PL20TL80 database.

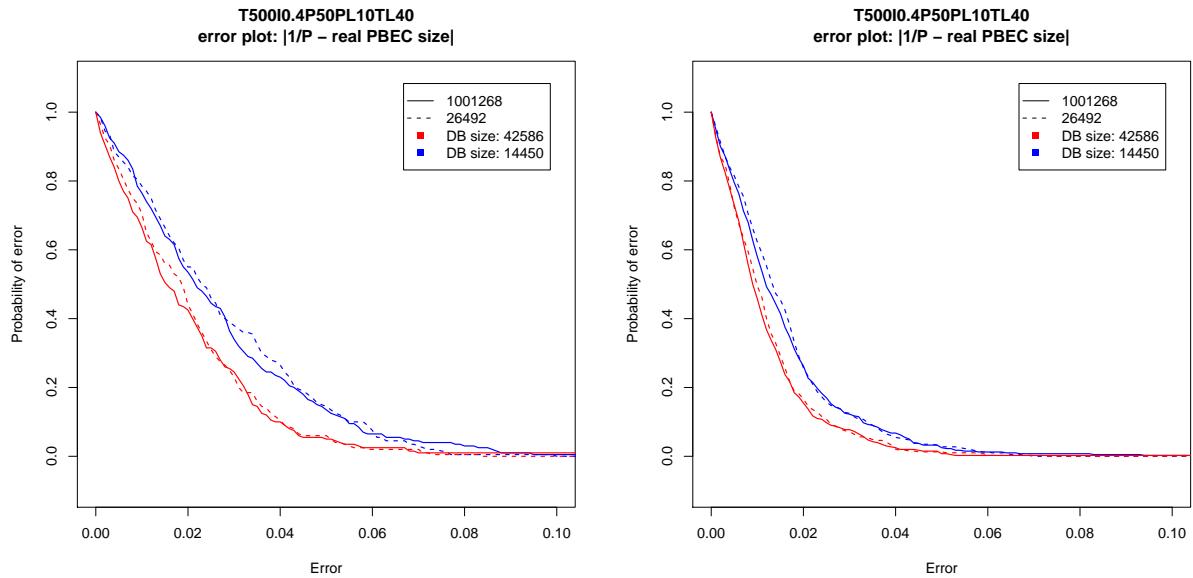


Figure 11.11: Probability of error of the estimation of the union of PBECs using a database sample created in Phase 1 and 2. Experiments made using $P = 5$ processors (left) and $P = 10$ processors (right). The T500I0.4P50PL10TL40 database.

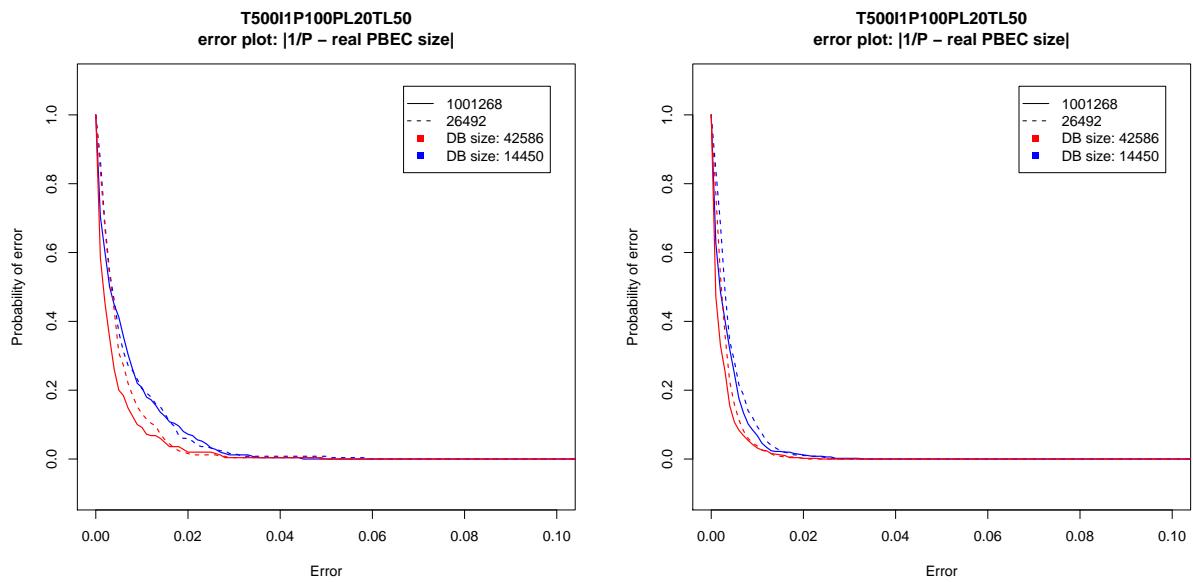


Figure 11.12: Probability of error of the estimation of the union of PBECs using a database sample created in Phase 1 and 2. Experiments made using $P = 5$ processors (left) and $P = 10$ processors (right). The T500I1P100PL20TL50 database.

11.4 Evaluation of the speedup

Two of the proposed parallel methods, namely the PARALLEL-FIMI-SEQ method (Method 1) and the PARALLEL-FIMI-PAR method (Method 2), need to compute the MFIs $\widetilde{\mathcal{M}}$ from a database sample $\widetilde{\mathcal{D}}$. In the experiments, in Phase 1, we use the $fpm\max^*$ [17] algorithm that computes the MFIs. In the case of the PARALLEL-FIMI-SEQ the $fpm\max^*$ algorithm is executed sequentially on processor p_1 . In the case of the PARALLEL-FIMI-PAR, we execute the $fpm\max^*$ algorithm in parallel, see Chapter 7.

In Phase 4 in our experiments, we use the ECLAT algorithm for mining of FIs. In the case of the PARALLEL-FIMI-RESERVOIR method (Method 3), the ECLAT algorithm is also used in Phase 1, i.e., the reservoir sampling samples the output of the ECLAT algorithm.

As the parameters of our method, we use the number of samples $|\widetilde{\mathcal{D}}|$ and $|\widetilde{\mathcal{F}}_s|$. The used parameters are summarized in Table 11.4.

$ \widetilde{\mathcal{D}} $	10000	10000	10000	14450	14450	14450	14450	14450	20000	20000	20000
$ \widetilde{\mathcal{F}}_s $	19869	26492	33115	13246	19869	26492	33115	39738	19869	26492	33115

Table 11.2: Sizes of $|\widetilde{\mathcal{D}}|$ and $|\widetilde{\mathcal{F}}_s|$ used in experiments

Because the number of graphs with speedups is prohibitive, we show the graphs for $|\widetilde{\mathcal{D}}| = 10000, |\widetilde{\mathcal{F}}_s| = 19869$. Figures 11.13–11.19 clearly demonstrate that for reasonably large and reasonably structured databases, the speedup is up to ≈ 13 on 20. The PARALLEL-FIMI-SEQ achieves speedup up to ≈ 8 on 20 processors, the PARALLEL-FIMI-PAR method achieves maximal speedup up to ≈ 11 on 20 processors, and the PARALLEL-FIMI-RESERVOIR method achieves speedup up to ≈ 13 on 20 processors and in one case up to ≈ 15 on 20 processors, see Figure 11.19 with measurements for database T500I0.4P50PL10TL40 . The speedups 0 indicates that the program run out of memory. The reason of the memory exhaustion is the large amount of MFIs and the effect of Theorem 7.5: due to the dynamic load balancing each processor can found all candidates in each assigned PBEC. That is: if the program implementing the PARALLEL-FIMI-SEQ method runs out of memory, then the program implementing the PARALLEL-FIMI-PAR method usually also runs out of main memory. The program implementing the PARALLEL-FIMI-RESERVOIR method never runs out of memory: the sample needs approximately the same amount of memory independently of the value of the minimal support and the database. The evaluation of the sampling process in Section 11.3 shows that the estimates

are quite good. The question is, why the speedup is not almost linear? The answer to this question is obvious: making the sample takes some time. Additionally, we can observe that lower values of *min_support** makes better speedup with the two cases for the T500I0.4P250PL20TL80 database having a very good speedup of ≈ 13 on $P = 20$ processors. The reason is obvious: the sampling process taking the same number of sample on the database of the same size makes better speedup, i.e., if it takes more time to compute sequentially the FIs for given support in the given database, then the speedup is usually better.

Tables 11.4–11.14 contain average values of the speedup for particular combination of database and number of processors. Some of the numbers in these tables are typed in bold:

1. First consider the tables for PARALLEL-FIMI-PAR and PARALLEL-FIMI-SEQ: the *bigger* value of average speedup corresponding to the same database and the same number of processors is typed in bold, e.g., PARALLEL-FIMI-SEQ has average value of speedup 1.354 and PARALLEL-FIMI-PAR has average value 1.407 for the database T500I0.4P50PL10TL40 and $P = 2$. The value 1.407 is typed in bold because it is the maximum of the two values.
2. A value in the table for PARALLEL-FIMI-RESERVOIR is bold if the value is the *biggest* value of average speedup corresponding to the same database and the same number of processors for all three methods, e.g., the average speedup of PARALLEL-FIMI-RESERVOIR for the T500I0.4P50PL10TL40 database and $P = 2$, the value 1.543, is typed in bold because it is the biggest of the three values: 1.354, 1.407, 1.543.

From the graphs on Figures 11.13–11.20 and tables on Table 11.9 it follows that the PARALLEL-FIMI-PAR is usually faster then the PARALLEL-FIMI-SEQ for the number of processors $P \leq 20$. The PARALLEL-FIMI-RESERVOIR performs better then: a) the PARALLEL-FIMI-PAR and b) the PARALLEL-FIMI-SEQ method. Our hypothesis is that the PARALLEL-FIMI-RESERVOIR makes better estimates of the relative size of the union of PBECs, see Section 6 for discussion of the sampling process. Still, there is a possibility to improve the speedup of the PARALLEL-FIMI-RESERVOIR method, for discussion see Chapter 12.2. Additionally, there is an advantage of the PARALLEL-FIMI-RESERVOIR over the two other methods: it is not necessary to compute the MFIs. The number

of MFIs can be very large and the program implementing the PARALLEL-FIMI-SEQ method or the PARALLEL-FIMI-PAR can run out of main memory. This happens for some supports of some databases, e.g., T500I0.4P250PL20TL80, T500I0.4P50PL10TL40, and T500I1P100PL20TL50 (indicated by the speedup value 0).

Very low speedups were obtained for the database with 1000 items in Figure 11.20, the T500I1P100PL20TL50 database. The reason for such a bad speedup lies in Phase 1 and 2. There is always a processor that has much bigger running time in Phase 4. For example, for $\min_support^* = 0.02$ and for $P = 10$ the execution time of Phase 4 is (in seconds): 194, 1199, 319, 245, 536, 357, 477, 212, 332, 212. A sum of these times is 4087 seconds, the sequential algorithm runs ≈ 3800 seconds. The probability of error of the estimates made in Phase 2 of T500I1P100PL20TL50 are competitive to other databases, see Figure 11.5. The best speedup that achieved by PARALLEL-FIMI-RESERVOIR is ≈ 8 on 20 processors for $\min_support^* = 0.02$. In other cases the speedup is not so good. The reason of such behaviour is unknown.

Tables 11.4–11.14 show the average speedup for the parameters shown in Table 11.4. The best combination of values for the PARALLEL-FIMI-RESERVOIR algorithm, e.g., the best values of $|\tilde{\mathcal{D}}|$ and $|\tilde{\mathcal{F}}_s|$ is the following:

Variant of our method	$ \tilde{\mathcal{D}} $	$ \tilde{\mathcal{F}}_s $
PARALLEL-FIMI-RESERVOIR	10000	19869
PARALLEL-FIMI-PAR	10000	33115
PARALLEL-FIMI-SEQ	10000	19869

Table 11.3: Best combintation of $|\tilde{\mathcal{D}}|$ and $|\tilde{\mathcal{F}}_s|$ for $P = 20$

We have made some experiments with the parameter α and chosen $\alpha = 0.3$: this value of α assures good granularity of the partitioning of \mathcal{F} using the PBECs, i.e., the PBECs are small enough so the LPT-MAKESPAN algorithm makes partitions of size $1/P$. The value of ρ can be chosen between 0.0007-0.003, see Section 11.3. Even that there is large number of parameters, our experiments show that there is not so big difference between the values of $|\tilde{\mathcal{D}}|$ and $|\tilde{\mathcal{F}}_s|$. Our hypothesis is the value of α can be set to $\alpha = 0.3$ and the value of ρ can be set to $\rho = 0.001$. These setting of parameters seems to be sufficient for all our experiments.

datafile/PARALLEL-FIMI-SEQ	2	4	6	10	16	20
T500I0.4P50PL10TL40	1.349	1.918	2.423	3.037	3.532	3.578
T500I0.1P50PL20TL40	1.417	2.399	3.040	4.280	6.113	6.761
T500I0.4P250PL20TL80	0.771	1.471	1.906	2.928	4.108	4.703
T500I1P100PL20TL50	1.020	1.520	1.824	1.939	2.281	2.273
T500I0.1P50PL10TL40	1.345	2.264	3.103	4.538	6.386	7.385
T500I0.4P250PL10TL120	0.759	1.471	2.101	3.044	4.159	4.985
T500I0.1P100PL20TL50	1.062	1.832	2.413	3.660	5.349	6.110
T500I0.4P150PL40TL80	0.965	1.635	2.282	3.163	4.121	4.658
T500I0.1P250PL10TL40	0.985	1.724	2.285	3.513	4.830	5.778
Total average	1.075	1.804	2.375	3.345	4.542	5.137

datafile/PARALLEL-FIMI-PAR	2	4	6	10	16	20
T500I0.4P50PL10TL40	1.524	2.209	2.889	2.649	2.972	3.255
T500I0.1P50PL20TL40	1.502	2.553	3.452	4.829	6.372	7.973
T500I0.4P250PL20TL80	0.945	1.838	2.882	3.679	4.813	5.374
T500I1P100PL20TL50	1.116	1.498	1.980	1.697	2.763	2.051
T500I0.1P50PL10TL40	1.371	2.244	2.633	4.862	6.194	7.461
T500I0.4P250PL10TL120	0.879	2.030	2.549	4.011	5.553	6.273
T500I0.1P100PL20TL50	1.122	1.932	2.237	3.869	5.583	5.885
T500I0.4P150PL40TL80	1.076	1.955	2.591	3.630	4.776	3.156
T500I0.1P250PL10TL40	1.100	1.898	2.791	3.954	5.525	6.239
Total average	1.182	2.018	2.667	3.687	4.950	5.296

datafile/PARALLEL-FIMI-RESERVOIR	2	4	6	10	16	20
T500I0.4P50PL10TL40	1.628	2.347	3.372	4.776	6.202	6.893
T500I0.1P50PL20TL40	1.455	2.550	3.093	4.076	5.670	6.342
T500I0.4P250PL20TL80	1.453	2.757	3.965	5.761	6.208	8.430
T500I1P100PL20TL50	1.121	1.923	2.374	2.841	4.026	5.753
T500I0.1P50PL10TL40	1.380	2.521	3.281	4.464	6.772	8.860
T500I0.4P250PL10TL120	1.216	2.249	3.117	4.584	6.179	7.067
T500I0.1P100PL20TL50	1.172	2.071	2.767	3.802	6.840	8.967
T500I0.4P150PL40TL80	1.211	2.155	2.858	3.789	4.976	5.423
T500I0.1P250PL10TL40	1.243	2.161	2.930	4.606	6.863	7.793
Total average	1.320	2.304	3.084	4.300	5.971	7.281

Table 11.4: Average speedup of the proposed methods for $|\tilde{\mathcal{D}}| = 10000$ and $|\tilde{\mathcal{F}}_s| = 19869$.

datafile/PARALLEL-FIMI-SEQ	2	4	6	10	16	20
T500I0.4P50PL10TL40	1.350	2.028	2.446	2.978	3.686	3.583
T500I0.1P50PL20TL40	1.415	2.388	3.301	4.523	5.895	6.811
T500I0.4P250PL20TL80	0.781	1.372	1.960	2.953	3.914	3.855
T500I1P100PL20TL50	1.052	1.511	1.602	2.119	2.228	2.165
T500I0.1P50PL10TL40	1.312	2.287	3.112	4.364	6.190	7.308
T500I0.4P250PL10TL120	0.712	1.353	2.089	3.103	4.172	5.170
T500I0.1P100PL20TL50	1.078	1.845	2.672	3.589	5.354	6.220
T500I0.4P150PL40TL80	0.893	1.658	2.252	3.135	4.205	4.555
T500I0.1P250PL10TL40	0.969	1.765	2.325	3.387	5.213	5.999
Total average	1.062	1.801	2.418	3.350	4.540	5.074

datafile/PARALLEL-FIMI-PAR	2	4	6	10	16	20
T500I0.4P50PL10TL40	1.504	2.100	2.874	2.584	2.969	3.197
T500I0.1P50PL20TL40	1.456	2.605	3.469	5.105	6.153	6.739
T500I0.4P250PL20TL80	0.950	1.985	2.528	3.966	5.153	5.580
T500I1P100PL20TL50	0.969	1.662	1.913	1.507	2.190	2.012
T500I0.1P50PL10TL40	1.305	2.613	3.485	4.846	5.478	7.338
T500I0.4P250PL10TL120	0.860	1.927	2.624	3.943	5.491	5.783
T500I0.1P100PL20TL50	1.125	2.041	2.724	3.723	5.490	6.278
T500I0.4P150PL40TL80	1.079	1.986	2.645	3.774	4.761	5.045
T500I0.1P250PL10TL40	1.090	2.020	2.740	3.926	5.100	6.577
Total average	1.149	2.104	2.778	3.708	4.754	5.395

datafile/PARALLEL-FIMI-RESERVOIR	2	4	6	10	16	20
T500I0.4P50PL10TL40	1.551	2.463	3.107	4.546	5.588	5.466
T500I0.1P50PL20TL40	1.405	2.235	3.127	3.674	5.568	6.561
T500I0.4P250PL20TL80	1.525	2.755	3.695	5.460	6.556	8.632
T500I1P100PL20TL50	1.207	1.713	2.429	2.627	3.756	4.702
T500I0.1P50PL10TL40	1.413	2.377	2.759	4.557	7.476	8.220
T500I0.4P250PL10TL120	1.232	2.306	2.978	4.725	6.363	7.016
T500I0.1P100PL20TL50	1.167	2.074	2.700	3.955	7.386	8.905
T500I0.4P150PL40TL80	1.184	2.147	1.569	3.897	4.760	5.201
T500I0.1P250PL10TL40	1.220	2.111	2.492	4.293	7.112	8.342
Total average	1.323	2.242	2.762	4.193	6.063	7.005

Table 11.5: Average speedup of the proposed methods for $|\tilde{\mathcal{D}}| = 10000$ and $|\tilde{\mathcal{F}}_s| = 26492$.

datafile/PARALLEL-FIMI-SEQ	2	4	6	10	16	20
T500I0.4P50PL10TL40	1.296	1.904	2.541	2.931	3.546	3.701
T500I0.1P50PL20TL40	1.339	2.354	3.164	4.520	5.903	6.697
T500I0.4P250PL20TL80	0.788	1.402	1.946	2.528	3.779	4.643
T500I1P100PL20TL50	1.090	1.459	1.757	1.934	2.041	2.384
T500I0.1P50PL10TL40	1.390	2.262	3.191	4.472	6.263	6.981
T500I0.4P250PL10TL120	0.790	1.287	2.070	3.162	4.289	5.027
T500I0.1P100PL20TL50	1.079	1.855	2.482	3.628	5.407	6.258
T500I0.4P150PL40TL80	0.989	1.687	2.340	3.069	4.052	4.313
T500I0.1P250PL10TL40	0.980	1.719	2.340	3.415	5.035	5.770
Total average	1.082	1.770	2.426	3.295	4.480	5.086

datafile/PARALLEL-FIMI-PAR	2	4	6	10	16	20
T500I0.4P50PL10TL40	1.436	2.189	3.113	2.644	2.997	3.335
T500I0.1P50PL20TL40	1.422	2.603	2.546	5.020	6.261	7.438
T500I0.4P250PL20TL80	0.986	1.889	2.891	4.091	4.720	4.922
T500I1P100PL20TL50	1.086	1.562	1.302	1.564	2.396	1.693
T500I0.1P50PL10TL40	1.366	2.483	3.383	4.992	5.939	7.696
T500I0.4P250PL10TL120	0.857	1.892	2.760	4.042	5.420	6.282
T500I0.1P100PL20TL50	1.122	1.953	2.637	3.782	5.600	6.421
T500I0.4P150PL40TL80	1.087	1.980	2.663	3.576	4.793	5.132
T500I0.1P250PL10TL40	1.021	2.061	2.775	3.806	5.898	6.064
Total average	1.154	2.068	2.674	3.724	4.892	5.443

datafile/PARALLEL-FIMI-RESERVOIR	2	4	6	10	16	20
T500I0.4P50PL10TL40	1.662	2.227	3.100	3.362	5.583	6.690
T500I0.1P50PL20TL40	1.426	2.478	2.782	4.578	4.885	6.674
T500I0.4P250PL20TL80	1.491	2.635	3.745	5.280	7.219	8.103
T500I1P100PL20TL50	1.188	1.706	2.347	2.792	3.632	5.037
T500I0.1P50PL10TL40	1.429	2.452	2.907	4.964	7.401	8.508
T500I0.4P250PL10TL120	1.240	2.285	2.999	4.556	6.181	7.108
T500I0.1P100PL20TL50	1.158	2.076	2.787	3.764	7.085	8.843
T500I0.4P150PL40TL80	1.201	2.021	2.852	3.952	4.970	5.501
T500I0.1P250PL10TL40	1.235	2.199	2.923	4.736	7.157	8.508
Total average	1.337	2.231	2.938	4.221	6.013	7.219

Table 11.6: Average speedup of the proposed methods for $|\tilde{\mathcal{D}}| = 10000$ and $|\tilde{\mathcal{F}}_s| = 33115$.

datafile/PARALLEL-FIMI-SEQ	2	4	6	10	16	20
T500I0.4P50PL10TL40	1.345	2.047	2.315	2.680	3.493	3.590
T500I0.1P50PL20TL40	1.391	2.430	3.236	3.778	5.983	6.931
T500I0.4P250PL20TL80	0.778	1.431	2.004	2.940	3.750	4.362
T500I1P100PL20TL50	1.131	1.531	1.734	1.880	2.001	2.287
T500I0.1P50PL10TL40	1.350	2.327	3.128	4.398	6.417	7.121
T500I0.4P250PL10TL120	0.797	1.432	2.084	2.914	4.096	5.065
T500I0.1P100PL20TL50	1.066	1.785	2.561	3.773	5.359	6.247
T500I0.4P150PL40TL80	0.969	1.704	2.280	3.155	4.059	4.391
T500I0.1P250PL10TL40	1.025	1.842	2.346	3.516	4.739	5.629
Total average	1.095	1.836	2.410	3.226	4.433	5.069

datafile/PARALLEL-FIMI-PAR	2	4	6	10	16	20
T500I0.4P50PL10TL40	1.512	2.358	2.718	2.711	2.873	3.170
T500I0.1P50PL20TL40	1.430	2.715	3.416	5.149	6.644	6.130
T500I0.4P250PL20TL80	0.949	1.794	2.903	3.602	4.729	5.293
T500I1P100PL20TL50	1.026	1.144	2.015	1.731	2.362	2.017
T500I0.1P50PL10TL40	1.340	2.451	3.463	4.981	6.204	7.237
T500I0.4P250PL10TL120	0.911	1.894	2.672	3.939	5.196	5.896
T500I0.1P100PL20TL50	1.099	1.965	2.737	3.765	4.698	5.872
T500I0.4P150PL40TL80	1.055	2.013	2.600	3.665	4.456	3.495
T500I0.1P250PL10TL40	1.087	1.900	2.730	3.956	5.044	6.895
Total average	1.157	2.026	2.806	3.722	4.690	5.112

datafile/PARALLEL-FIMI-RESERVOIR	2	4	6	10	16	20
T500I0.4P50PL10TL40	1.523	2.358	3.184	4.582	5.544	6.730
T500I0.1P50PL20TL40	1.451	2.571	2.869	4.228	5.341	5.930
T500I0.4P250PL20TL80	1.502	2.715	3.564	5.009	6.656	7.965
T500I1P100PL20TL50	1.195	1.881	2.227	2.845	3.779	3.848
T500I0.1P50PL10TL40	1.341	2.466	3.187	4.322	7.177	8.273
T500I0.4P250PL10TL120	1.218	2.230	3.068	4.660	6.234	6.955
T500I0.1P100PL20TL50	1.175	1.987	2.589	4.181	6.752	8.382
T500I0.4P150PL40TL80	1.201	2.065	2.769	3.855	4.732	5.341
T500I0.1P250PL10TL40	1.204	2.189	2.860	4.685	6.393	8.155
Total average	1.312	2.274	2.924	4.263	5.845	6.842

Table 11.7: Average speedup of the proposed methods for $|\tilde{\mathcal{D}}| = 14450$ and $|\tilde{\mathcal{F}}_s| = 13246$.

datafile/PARALLEL-FIMI-SEQ	2	4	6	10	16	20
T500I0.4P50PL10TL40	1.375	1.877	2.424	3.003	3.377	3.746
T500I0.1P50PL20TL40	1.355	2.454	3.307	4.515	5.888	6.738
T500I0.4P250PL20TL80	0.781	1.405	1.807	2.848	4.041	4.494
T500I1P100PL20TL50	1.025	1.472	1.624	1.742	2.183	2.216
T500I0.1P50PL10TL40	1.399	2.284	3.115	4.367	6.258	6.912
T500I0.4P250PL10TL120	0.791	1.464	1.920	2.855	4.070	5.085
T500I0.1P100PL20TL50	1.058	1.858	2.443	3.579	5.311	6.104
T500I0.4P150PL40TL80	0.945	1.699	2.178	2.581	3.915	4.480
T500I0.1P250PL10TL40	0.969	1.623	2.295	3.445	4.848	5.931
Total average	1.078	1.793	2.346	3.215	4.432	5.078

datafile/PARALLEL-FIMI-PAR	2	4	6	10	16	20
T500I0.4P50PL10TL40	1.488	2.126	2.804	2.660	2.720	3.186
T500I0.1P50PL20TL40	1.490	2.584	3.438	5.201	5.542	6.274
T500I0.4P250PL20TL80	0.959	1.857	2.712	4.042	5.248	5.628
T500I1P100PL20TL50	1.117	1.576	1.805	1.643	2.398	1.978
T500I0.1P50PL10TL40	1.286	2.354	3.382	5.016	5.869	7.517
T500I0.4P250PL10TL120	0.848	1.897	2.620	3.883	5.451	6.101
T500I0.1P100PL20TL50	1.119	1.979	2.825	3.854	4.928	6.737
T500I0.4P150PL40TL80	1.082	2.008	2.267	3.616	4.630	5.152
T500I0.1P250PL10TL40	1.073	2.110	2.734	4.105	5.232	6.089
Total average	1.163	2.055	2.732	3.780	4.669	5.407

datafile/PARALLEL-FIMI-RESERVOIR	2	4	6	10	16	20
T500I0.4P50PL10TL40	1.575	2.472	3.123	4.695	4.878	6.525
T500I0.1P50PL20TL40	1.380	2.340	2.925	4.095	4.786	6.015
T500I0.4P250PL20TL80	1.482	2.803	3.770	5.525	6.059	8.403
T500I1P100PL20TL50	1.195	1.900	2.340	2.741	4.367	4.374
T500I0.1P50PL10TL40	1.429	2.380	2.729	4.566	6.926	8.419
T500I0.4P250PL10TL120	1.228	2.271	2.996	4.483	6.040	6.983
T500I0.1P100PL20TL50	1.158	1.965	2.596	3.644	6.791	7.951
T500I0.4P150PL40TL80	1.163	2.169	2.809	3.870	4.803	5.031
T500I0.1P250PL10TL40	1.228	2.126	2.884	4.848	6.956	8.288
Total average	1.315	2.270	2.908	4.274	5.734	6.888

Table 11.8: Average speedup of the proposed methods for $|\tilde{\mathcal{D}}| = 14450$ and $|\tilde{\mathcal{F}}_s| = 19869$.

datafile/PARALLEL-FIMI-SEQ	2	4	6	10	16	20
T500I0.4P50PL10TL40	1.354	1.817	1.881	2.754	2.573	3.474
T500I0.1P50PL20TL40	1.313	2.455	3.253	4.533	4.807	6.695
T500I0.4P250PL20TL80	0.744	1.445	1.935	2.503	3.764	4.532
T500I1P100PL20TL50	1.100	1.625	1.695	1.896	2.153	2.270
T500I0.1P50PL10TL40	1.413	2.350	3.243	4.549	6.127	7.109
T500I0.4P250PL10TL120	0.783	1.418	1.962	2.960	4.379	4.917
T500I0.1P100PL20TL50	1.030	1.773	2.410	3.662	5.397	6.176
T500I0.4P150PL40TL80	0.964	1.693	2.272	3.128	3.957	4.327
T500I0.1P250PL10TL40	1.008	1.633	2.413	3.478	4.914	5.653
Total average	1.079	1.801	2.340	3.274	4.230	5.017

datafile/PARALLEL-FIMI-PAR	2	4	6	10	16	20
T500I0.4P50PL10TL40	1.407	2.014	2.760	2.850	2.246	3.501
T500I0.1P50PL20TL40	1.469	1.978	3.600	5.185	5.718	6.319
T500I0.4P250PL20TL80	0.987	2.025	3.130	3.630	4.386	5.519
T500I1P100PL20TL50	0.991	1.641	1.799	1.616	2.685	2.031
T500I0.1P50PL10TL40	1.259	2.432	3.346	4.838	5.742	7.503
T500I0.4P250PL10TL120	0.949	1.955	2.648	3.906	5.402	6.147
T500I0.1P100PL20TL50	1.101	1.892	2.426	3.694	5.510	6.348
T500I0.4P150PL40TL80	1.069	2.020	1.408	3.692	4.580	4.954
T500I0.1P250PL10TL40	1.080	1.916	2.413	4.096	5.412	6.085
Total average	1.146	1.986	2.615	3.723	4.631	5.379

datafile/PARALLEL-FIMI-RESERVOIR	2	4	6	10	16	20
T500I0.4P50PL10TL40	1.543	2.510	3.219	3.264	4.625	6.006
T500I0.1P50PL20TL40	1.398	2.514	3.211	3.904	4.670	5.486
T500I0.4P250PL20TL80	1.403	2.665	3.810	5.079	6.741	8.457
T500I1P100PL20TL50	1.207	1.588	1.969	2.760	3.240	4.466
T500I0.1P50PL10TL40	1.397	2.382	3.104	4.453	6.714	8.209
T500I0.4P250PL10TL120	1.218	2.162	3.001	4.557	6.084	7.103
T500I0.1P100PL20TL50	1.167	1.999	2.614	3.787	6.646	8.001
T500I0.4P150PL40TL80	1.203	2.065	2.834	3.805	4.729	5.132
T500I0.1P250PL10TL40	1.224	2.131	2.788	4.443	6.142	8.334
Total average	1.307	2.224	2.950	4.006	5.510	6.799

Table 11.9: Average speedup of the proposed methods for $|\tilde{\mathcal{D}}| = 14450$ and $|\tilde{\mathcal{F}}_s| = 26492$.

datafile/PARALLEL-FIMI-SEQ	2	4	6	10	16	20
T500I0.4P50PL10TL40	1.236	1.934	2.325	2.778	3.513	3.330
T500I0.1P50PL20TL40	1.315	2.414	3.200	4.454	6.058	6.459
T500I0.4P250PL20TL80	0.787	1.503	1.985	2.910	4.046	4.757
T500I1P100PL20TL50	1.084	1.452	1.779	1.934	2.177	1.924
T500I0.1P50PL10TL40	1.356	2.384	3.068	4.532	6.329	7.184
T500I0.4P250PL10TL120	0.779	1.494	1.945	2.967	4.128	4.979
T500I0.1P100PL20TL50	1.086	1.842	2.377	3.678	5.386	5.873
T500I0.4P150PL40TL80	0.938	1.655	2.170	3.027	3.986	4.480
T500I0.1P250PL10TL40	0.999	1.683	2.284	3.425	5.015	5.946
Total average	1.064	1.818	2.348	3.300	4.515	4.993

datafile/PARALLEL-FIMI-PAR	2	4	6	10	16	20
T500I0.4P50PL10TL40	1.433	2.200	3.031	2.676	3.084	3.152
T500I0.1P50PL20TL40	1.447	2.550	3.017	4.936	5.700	7.993
T500I0.4P250PL20TL80	0.973	2.038	2.995	3.603	4.586	5.222
T500I1P100PL20TL50	1.127	1.431	1.920	1.575	2.227	2.009
T500I0.1P50PL10TL40	1.391	2.360	3.069	5.107	6.054	6.917
T500I0.4P250PL10TL120	0.936	1.963	2.596	3.933	5.425	6.211
T500I0.1P100PL20TL50	1.068	1.944	2.596	3.744	5.041	5.899
T500I0.4P150PL40TL80	1.063	1.973	2.526	3.562	4.734	4.924
T500I0.1P250PL10TL40	1.074	2.047	2.685	4.037	5.339	6.543
Total average	1.168	2.056	2.715	3.686	4.688	5.430

datafile/PARALLEL-FIMI-RESERVOIR	2	4	6	10	16	20
T500I0.4P50PL10TL40	1.557	2.376	2.825	4.181	4.919	5.290
T500I0.1P50PL20TL40	1.359	2.487	2.954	4.224	5.190	5.364
T500I0.4P250PL20TL80	1.484	2.772	3.782	5.120	6.231	8.514
T500I1P100PL20TL50	1.201	1.664	2.170	2.456	3.966	4.878
T500I0.1P50PL10TL40	1.421	2.423	3.074	4.549	6.927	8.365
T500I0.4P250PL10TL120	1.238	2.264	3.062	4.607	6.129	7.075
T500I0.1P100PL20TL50	1.175	1.998	2.530	4.040	6.156	8.572
T500I0.4P150PL40TL80	1.191	2.067	2.720	3.756	4.952	5.139
T500I0.1P250PL10TL40	1.199	2.179	2.676	3.860	6.896	8.114
Total average	1.314	2.248	2.866	4.088	5.707	6.812

Table 11.10: Average speedup of the proposed methods for $|\tilde{\mathcal{D}}| = 14450$ and $|\tilde{\mathcal{F}}_s| = 33115$.

datafile/PARALLEL-FIMI-SEQ	2	4	6	10	16	20
T500I0.4P50PL10TL40	1.389	1.877	2.457	2.883	3.268	3.681
T500I0.1P50PL20TL40	1.388	2.417	3.211	4.453	5.939	6.709
T500I0.4P250PL20TL80	0.777	1.477	1.868	2.562	3.993	4.497
T500I1P100PL20TL50	1.087	1.529	1.785	2.028	2.224	2.283
T500I0.1P50PL10TL40	1.367	2.304	3.147	4.280	6.253	7.184
T500I0.4P250PL10TL120	0.749	1.467	1.950	2.611	4.020	4.997
T500I0.1P100PL20TL50	1.077	1.881	2.495	3.521	5.156	6.283
T500I0.4P150PL40TL80	0.929	1.672	2.211	2.196	3.892	4.546
T500I0.1P250PL10TL40	0.996	1.763	2.258	3.389	5.011	5.851
Total average	1.084	1.821	2.376	3.103	4.417	5.115

datafile/PARALLEL-FIMI-PAR	2	4	6	10	16	20
T500I0.4P50PL10TL40	1.433	2.244	3.097	2.868	2.880	3.399
T500I0.1P50PL20TL40	1.407	2.510	3.450	4.849	5.785	7.211
T500I0.4P250PL20TL80	0.914	1.951	2.921	3.960	4.808	5.347
T500I1P100PL20TL50	1.094	1.679	1.920	1.700	2.348	1.975
T500I0.1P50PL10TL40	1.396	2.492	3.208	4.930	5.920	7.487
T500I0.4P250PL10TL120	0.943	1.993	2.703	4.212	5.455	5.671
T500I0.1P100PL20TL50	1.120	1.974	2.687	4.198	5.320	6.394
T500I0.4P150PL40TL80	1.059	2.005	2.660	3.539	4.326	4.960
T500I0.1P250PL10TL40	1.112	1.838	2.729	4.276	5.629	6.186
Total average	1.164	2.076	2.820	3.837	4.719	5.403

datafile/PARALLEL-FIMI-RESERVOIR	2	4	6	10	16	20
T500I0.4P50PL10TL40	1.570	2.497	2.998	4.003	5.733	6.238
T500I0.1P50PL20TL40	1.399	2.478	2.488	4.478	5.074	5.954
T500I0.4P250PL20TL80	1.474	2.652	3.548	5.381	6.187	8.493
T500I1P100PL20TL50	1.226	1.788	2.113	2.597	3.536	4.414
T500I0.1P50PL10TL40	1.393	2.415	3.095	4.109	7.039	8.142
T500I0.4P250PL10TL120	1.223	2.297	3.035	4.548	6.077	6.995
T500I0.1P100PL20TL50	1.144	1.969	2.645	4.218	6.588	8.010
T500I0.4P150PL40TL80	1.189	2.019	2.807	3.667	4.883	5.292
T500I0.1P250PL10TL40	1.220	2.155	2.781	4.229	7.065	8.137
Total average	1.315	2.252	2.834	4.137	5.798	6.853

Table 11.11: Average speedup of the proposed methods for $|\tilde{\mathcal{D}}| = 14450$ and $|\tilde{\mathcal{F}}_s| = 39738$.

datafile/PARALLEL-FIMI-SEQ	2	4	6	10	16	20
T500I0.4P50PL10TL40	1.348	2.115	2.571	2.970	3.542	3.453
T500I0.1P50PL20TL40	1.409	2.395	2.996	4.579	6.050	6.612
T500I0.4P250PL20TL80	0.757	1.372	2.042	2.737	3.921	4.712
T500I1P100PL20TL50	1.171	1.546	1.581	1.968	2.159	2.297
T500I0.1P50PL10TL40	1.354	2.351	3.143	4.447	6.313	7.193
T500I0.4P250PL10TL120	0.757	1.362	1.959	3.092	4.259	5.012
T500I0.1P100PL20TL50	1.049	1.872	2.314	3.704	5.278	6.321
T500I0.4P150PL40TL80	0.965	1.633	2.206	3.041	4.078	4.533
T500I0.1P250PL10TL40	0.958	1.743	2.332	3.495	5.118	5.674
Total average	1.085	1.821	2.349	3.337	4.524	5.090

datafile/PARALLEL-FIMI-PAR	2	4	6	10	16	20
T500I0.4P50PL10TL40	1.467	2.244	3.151	2.930	2.720	3.182
T500I0.1P50PL20TL40	1.474	2.461	3.226	5.050	6.198	7.335
T500I0.4P250PL20TL80	0.969	1.820	2.992	3.345	4.360	5.501
T500I1P100PL20TL50	1.043	1.609	1.899	1.614	2.644	1.847
T500I0.1P50PL10TL40	1.392	2.519	3.335	5.038	5.853	7.333
T500I0.4P250PL10TL120	0.944	1.915	2.780	3.822	5.649	6.162
T500I0.1P100PL20TL50	1.116	2.057	2.653	3.867	5.438	6.039
T500I0.4P150PL40TL80	1.058	1.985	2.652	3.379	4.455	4.742
T500I0.1P250PL10TL40	1.077	1.992	2.612	3.840	5.288	5.750
Total average	1.171	2.067	2.811	3.654	4.734	5.321

datafile/PARALLEL-FIMI-RESERVOIR	2	4	6	10	16	20
T500I0.4P50PL10TL40	1.566	2.396	3.182	4.254	4.867	5.211
T500I0.1P50PL20TL40	1.354	2.303	3.261	3.946	4.987	6.029
T500I0.4P250PL20TL80	1.493	2.628	3.761	4.900	6.112	7.940
T500I1P100PL20TL50	1.180	1.745	2.361	2.826	3.870	3.701
T500I0.1P50PL10TL40	1.387	2.505	3.184	4.514	6.259	8.128
T500I0.4P250PL10TL120	1.217	2.199	2.959	4.309	6.049	6.718
T500I0.1P100PL20TL50	1.163	2.067	2.590	4.052	6.846	8.401
T500I0.4P150PL40TL80	1.173	1.939	2.734	3.826	4.604	5.096
T500I0.1P250PL10TL40	1.209	2.137	2.849	4.906	6.857	7.984
Total average	1.305	2.213	2.987	4.170	5.606	6.579

Table 11.12: Average speedup of the proposed methods for $|\tilde{\mathcal{D}}| = 20000$ and $|\tilde{\mathcal{F}}_s| = 19869$.

datafile/PARALLEL-FIMI-SEQ	2	4	6	10	16	20
T500I0.4P50PL10TL40	1.220	2.110	2.269	3.017	3.073	3.714
T500I0.1P50PL20TL40	1.353	2.412	3.302	4.609	5.839	6.583
T500I0.4P250PL20TL80	0.784	1.407	1.970	2.687	3.766	4.826
T500I1P100PL20TL50	1.152	1.502	1.670	1.952	2.163	2.160
T500I0.1P50PL10TL40	1.373	2.264	3.081	4.682	6.259	7.066
T500I0.4P250PL10TL120	0.785	1.429	1.986	3.052	4.092	5.001
T500I0.1P100PL20TL50	1.075	1.771	2.474	3.632	5.297	6.340
T500I0.4P150PL40TL80	0.978	1.584	2.274	3.131	4.061	4.413
T500I0.1P250PL10TL40	1.000	1.722	2.361	3.354	4.846	5.783
Total average	1.080	1.800	2.376	3.346	4.377	5.098

datafile/PARALLEL-FIMI-PAR	2	4	6	10	16	20
T500I0.4P50PL10TL40	1.421	2.254	3.252	2.832	3.047	3.055
T500I0.1P50PL20TL40	1.428	2.555	3.434	4.997	6.425	7.296
T500I0.4P250PL20TL80	0.936	1.927	2.928	3.677	4.970	5.156
T500I1P100PL20TL50	1.037	1.575	2.041	1.573	2.413	1.893
T500I0.1P50PL10TL40	1.436	2.313	3.352	4.950	6.082	7.778
T500I0.4P250PL10TL120	0.919	1.884	2.641	3.547	5.214	6.040
T500I0.1P100PL20TL50	1.109	1.996	2.630	3.726	5.142	5.993
T500I0.4P150PL40TL80	1.077	1.908	2.521	3.454	4.473	4.767
T500I0.1P250PL10TL40	1.004	1.933	2.635	4.071	5.201	6.659
Total average	1.152	2.038	2.826	3.647	4.774	5.404

datafile/PARALLEL-FIMI-RESERVOIR	2	4	6	10	16	20
T500I0.4P50PL10TL40	1.546	2.347	3.060	4.299	5.165	5.331
T500I0.1P50PL20TL40	1.290	2.380	2.948	4.238	5.704	6.591
T500I0.4P250PL20TL80	1.434	2.649	3.615	5.243	5.917	7.824
T500I1P100PL20TL50	1.122	1.828	2.044	2.475	3.216	4.309
T500I0.1P50PL10TL40	1.419	2.345	2.905	4.345	6.761	8.257
T500I0.4P250PL10TL120	1.212	2.197	2.971	4.376	6.064	6.611
T500I0.1P100PL20TL50	1.154	1.982	2.493	3.636	7.013	7.989
T500I0.4P150PL40TL80	1.168	2.110	2.650	3.765	4.780	5.314
T500I0.1P250PL10TL40	1.199	2.030	2.794	4.484	6.686	7.955
Total average	1.283	2.208	2.831	4.096	5.701	6.687

Table 11.13: Average speedup of the proposed methods for $|\tilde{\mathcal{D}}| = 20000$ and $|\tilde{\mathcal{F}}_s| = 26492$.

datafile/PARALLEL-FIMI-SEQ	2	4	6	10	16	20
T500I0.4P50PL10TL40	1.356	1.893	2.665	2.955	3.526	3.887
T500I0.1P50PL20TL40	1.415	2.464	3.363	4.488	5.484	6.851
T500I0.4P250PL20TL80	0.795	1.444	1.803	2.709	3.564	4.650
T500I1P100PL20TL50	1.020	1.393	1.672	2.061	2.254	2.318
T500I0.1P50PL10TL40	1.374	2.280	3.129	4.445	6.360	7.242
T500I0.4P250PL10TL120	0.792	1.312	1.966	3.022	4.138	5.053
T500I0.1P100PL20TL50	1.028	1.848	2.343	3.730	5.404	6.253
T500I0.4P150PL40TL80	0.950	1.641	2.165	3.096	3.950	4.130
T500I0.1P250PL10TL40	0.980	1.706	2.339	3.528	4.947	5.741
Total average	1.079	1.776	2.383	3.337	4.403	5.125

datafile/PARALLEL-FIMI-PAR	2	4	6	10	16	20
T500I0.4P50PL10TL40	1.429	2.064	3.170	2.814	3.033	2.825
T500I0.1P50PL20TL40	1.426	2.298	3.457	4.414	6.078	7.625
T500I0.4P250PL20TL80	0.953	1.942	2.592	3.399	3.990	5.315
T500I1P100PL20TL50	1.107	1.369	1.840	1.687	2.209	2.082
T500I0.1P50PL10TL40	1.356	2.501	3.551	5.117	5.981	7.708
T500I0.4P250PL10TL120	0.911	1.939	2.637	3.921	5.450	5.934
T500I0.1P100PL20TL50	1.096	1.959	2.711	3.916	5.364	6.242
T500I0.4P150PL40TL80	1.070	1.950	2.526	3.568	4.633	4.774
T500I0.1P250PL10TL40	1.057	2.134	2.751	3.674	5.541	6.217
Total average	1.156	2.017	2.804	3.612	4.698	5.413

datafile/PARALLEL-FIMI-RESERVOIR	2	4	6	10	16	20
T500I0.4P50PL10TL40	1.608	2.415	3.295	3.333	5.588	6.022
T500I0.1P50PL20TL40	1.423	2.381	3.029	4.046	5.472	6.304
T500I0.4P250PL20TL80	1.464	2.667	3.483	4.560	6.548	7.946
T500I1P100PL20TL50	1.190	1.876	1.862	2.528	3.838	4.414
T500I0.1P50PL10TL40	1.438	2.430	3.146	4.725	6.836	7.702
T500I0.4P250PL10TL120	1.220	2.187	3.031	4.405	6.073	6.528
T500I0.1P100PL20TL50	1.146	2.040	2.227	3.705	6.315	7.501
T500I0.4P150PL40TL80	1.178	2.041	2.693	3.784	4.687	5.363
T500I0.1P250PL10TL40	1.181	2.188	2.831	4.785	6.929	7.336
Total average	1.316	2.247	2.844	3.986	5.810	6.569

Table 11.14: Average speedup of the proposed methods for $|\tilde{\mathcal{D}}| = 20000$ and $|\tilde{\mathcal{F}}_s| = 33115$.

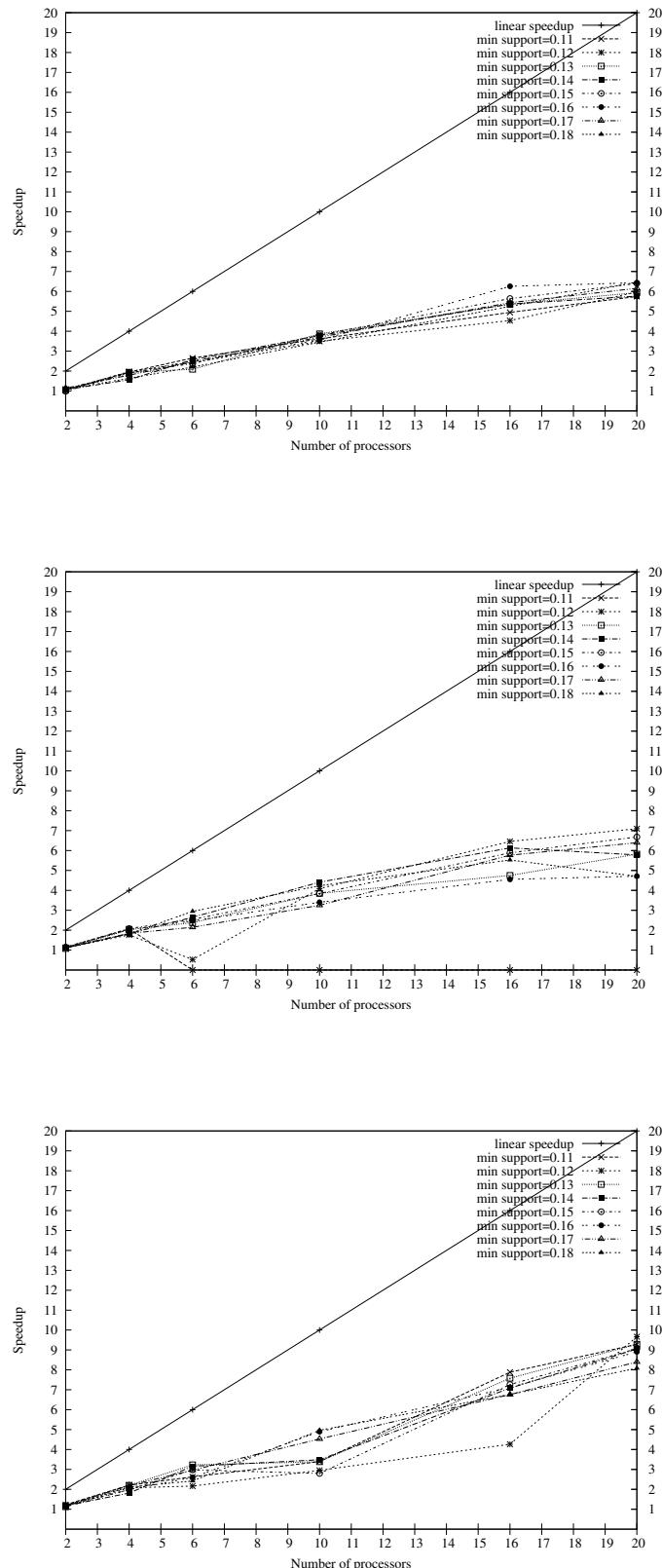


Figure 11.13: Speedups of the PARALLEL-FIMI-SEQ, PARALLEL-FIMI-PAR, and PARALLEL-FIMI-RESERVOIR methods (from top to bottom) on the T500I0.1P100PL20TL50 database.

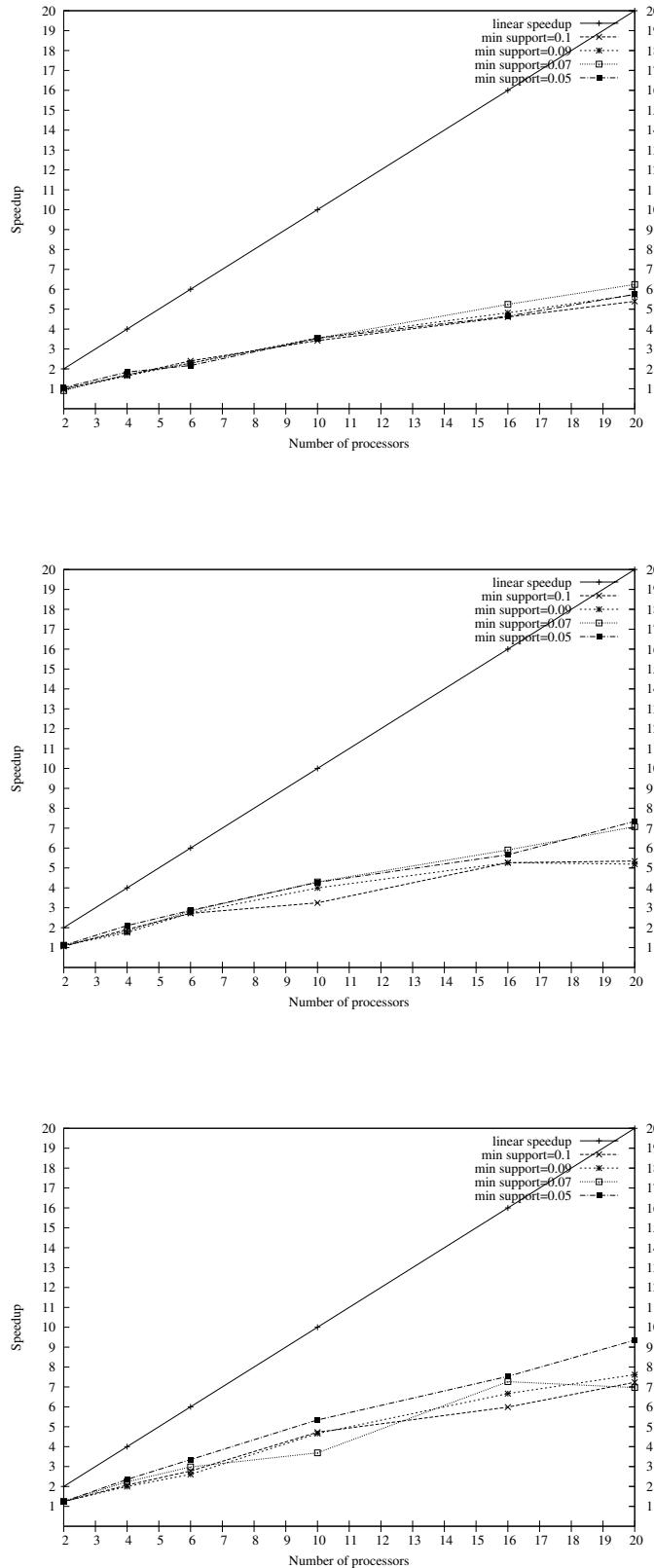


Figure 11.14: Speedups of the PARALLEL-FIMI-SEQ, PARALLEL-FIMI-PAR, and PARALLEL-FIMI-RESERVOIR methods (from top to bottom) on the T500I0.1P250PL10TL40 database.

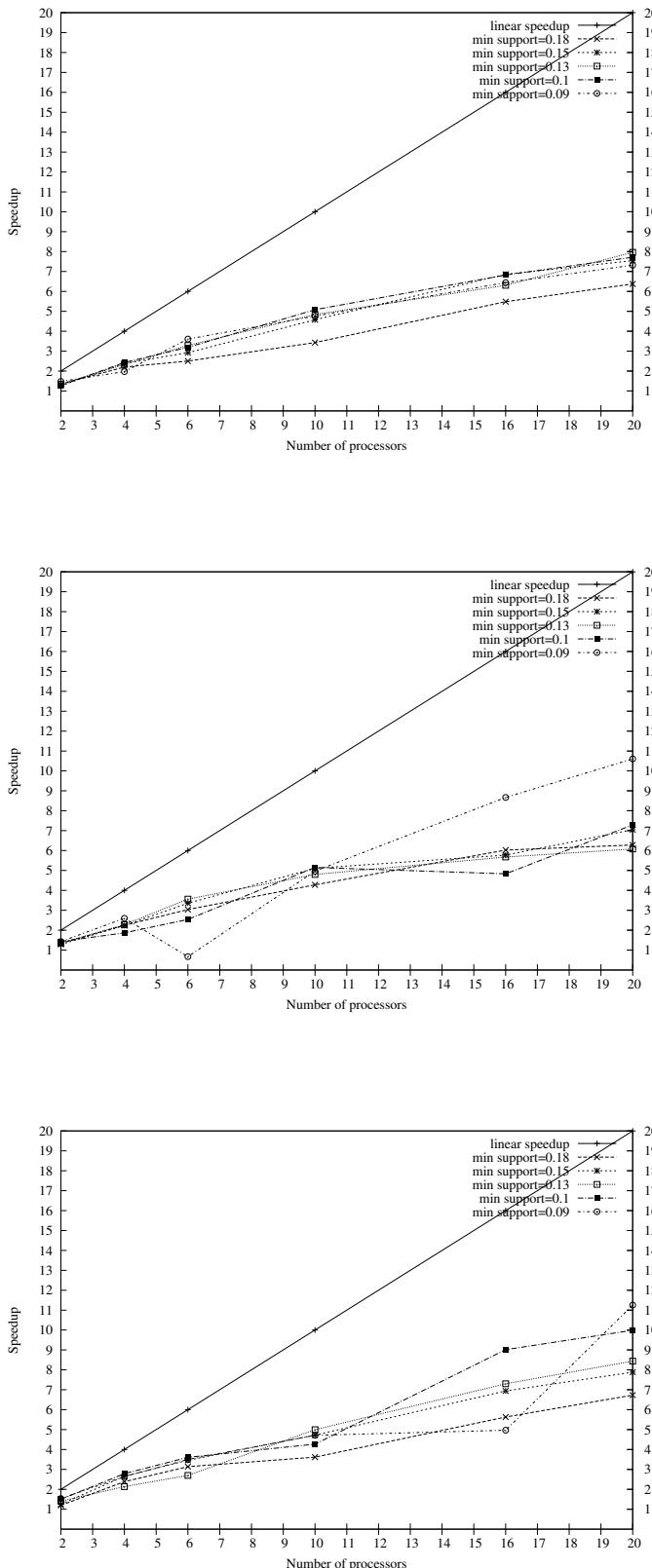


Figure 11.15: Speedups of the PARALLEL-FIMI-SEQ, PARALLEL-FIMI-PAR, and PARALLEL-FIMI-RESERVOIR methods (from top to bottom) on the T500I0.1P50PL10TL40 database.

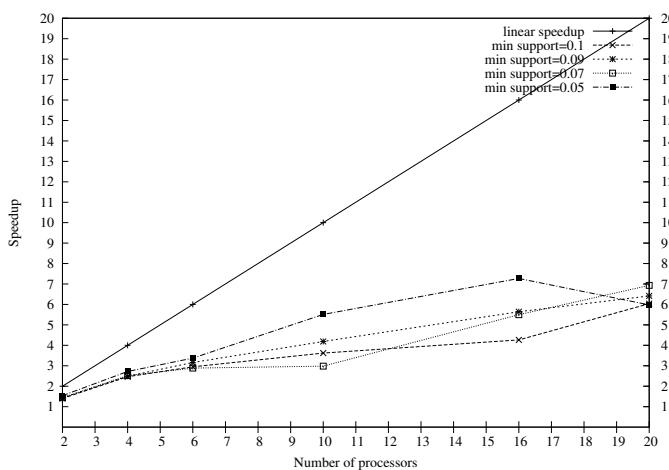
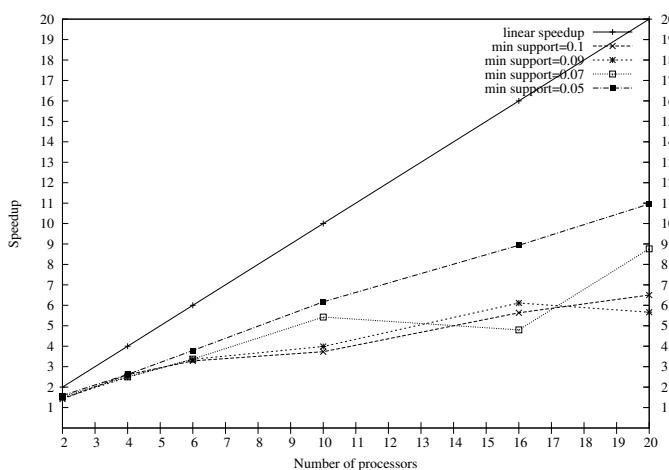
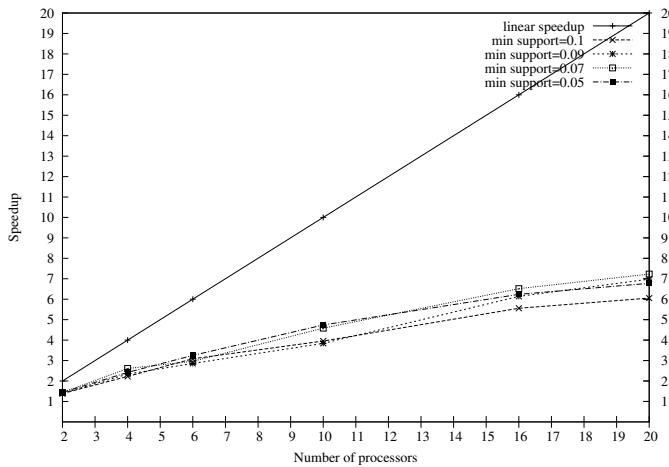


Figure 11.16: Speedups of the PARALLEL-FIMI-SEQ, PARALLEL-FIMI-PAR, and PARALLEL-FIMI-RESERVOIR methods (from top to bottom) on the T500I0.1P50PL20TL40 database.

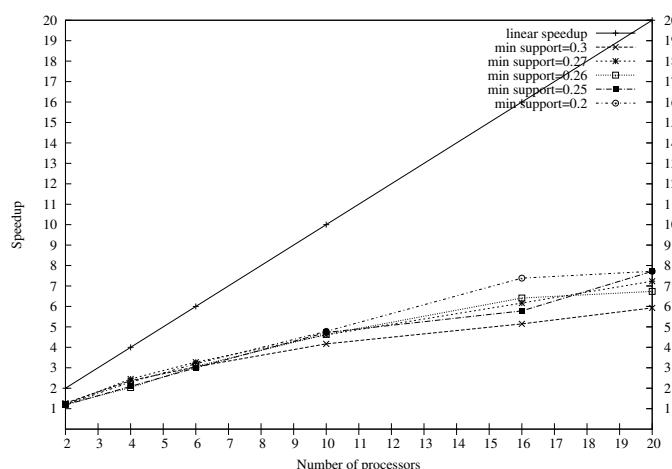
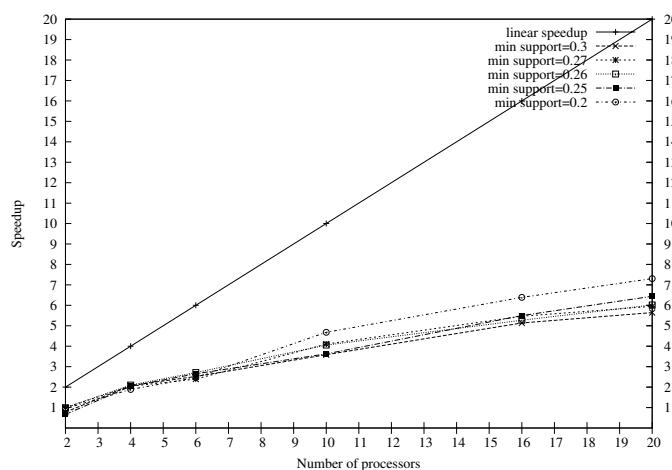
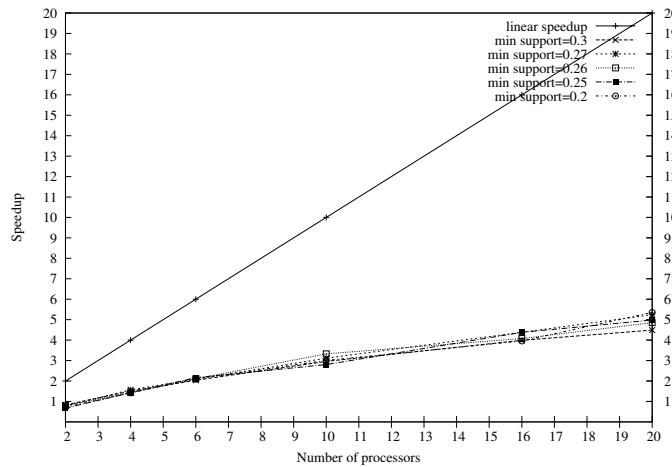


Figure 11.17: Speedups of the PARALLEL-FIMI-SEQ, PARALLEL-FIMI-PAR, and PARALLEL-FIMI-RESERVOIR methods (from top to bottom) on the T500IO.4P250PL10TL120 database.

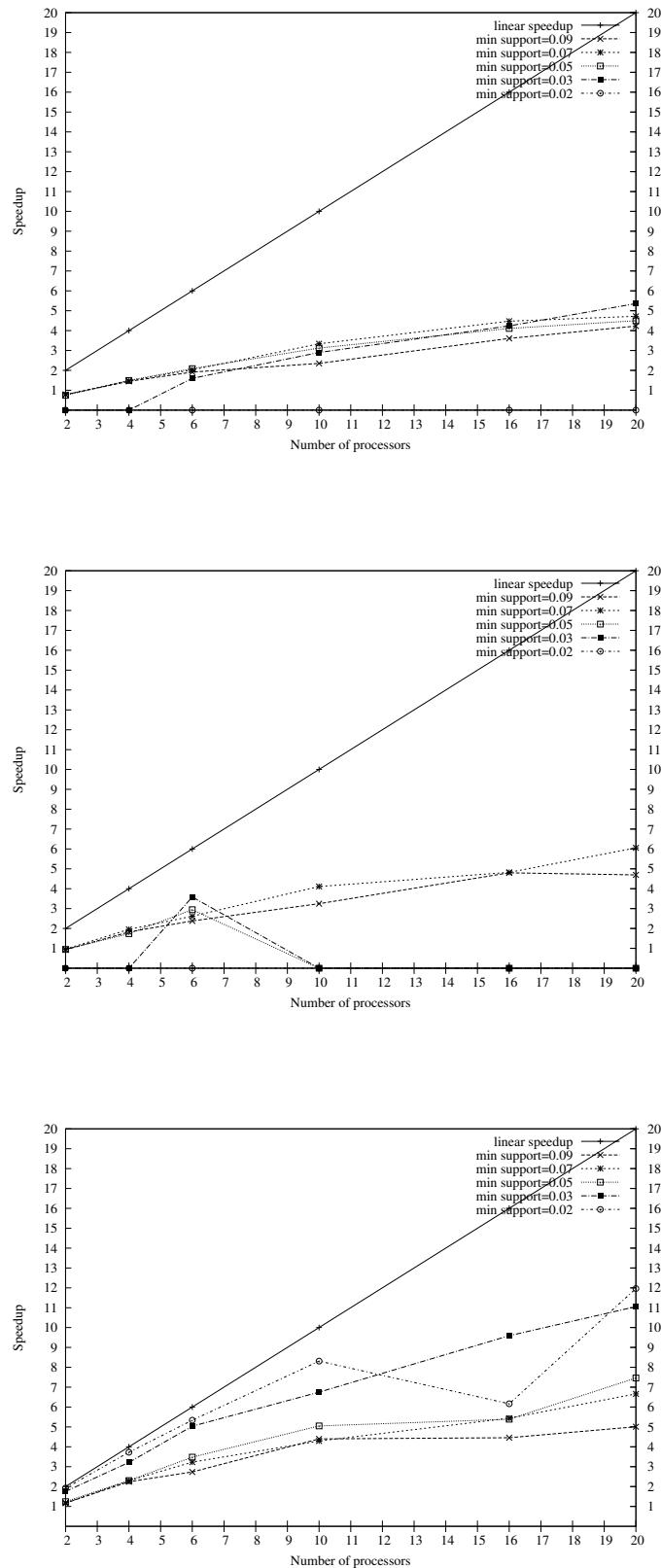


Figure 11.18: Speedups of the PARALLEL-FIMI-SEQ, PARALLEL-FIMI-PAR, and PARALLEL-FIMI-RESERVOIR methods (from top to bottom) on the T500I0.4P250PL20TL80 database.

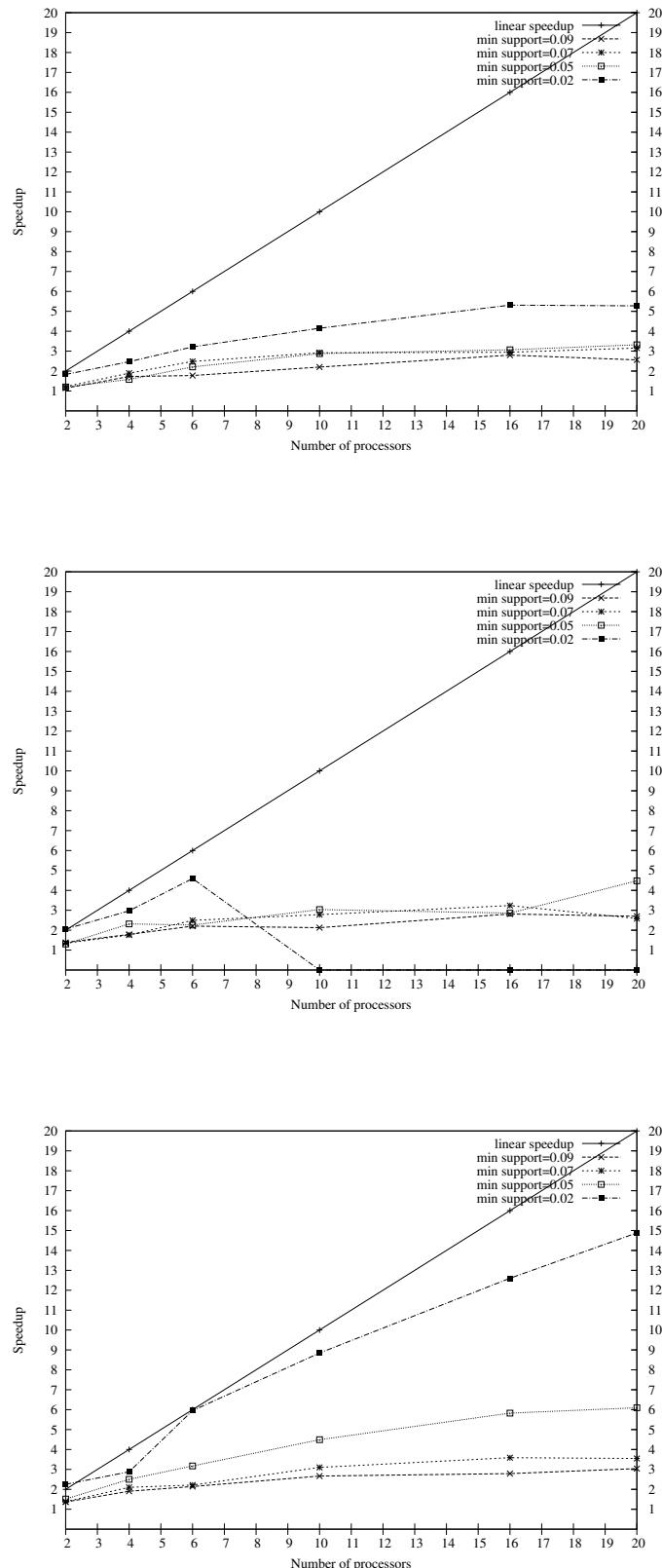


Figure 11.19: Speedups of the PARALLEL-FIMI-SEQ, PARALLEL-FIMI-PAR, and PARALLEL-FIMI-RESERVOIR methods (from top to bottom) on the T500I0.4P50PL10TL40 database.

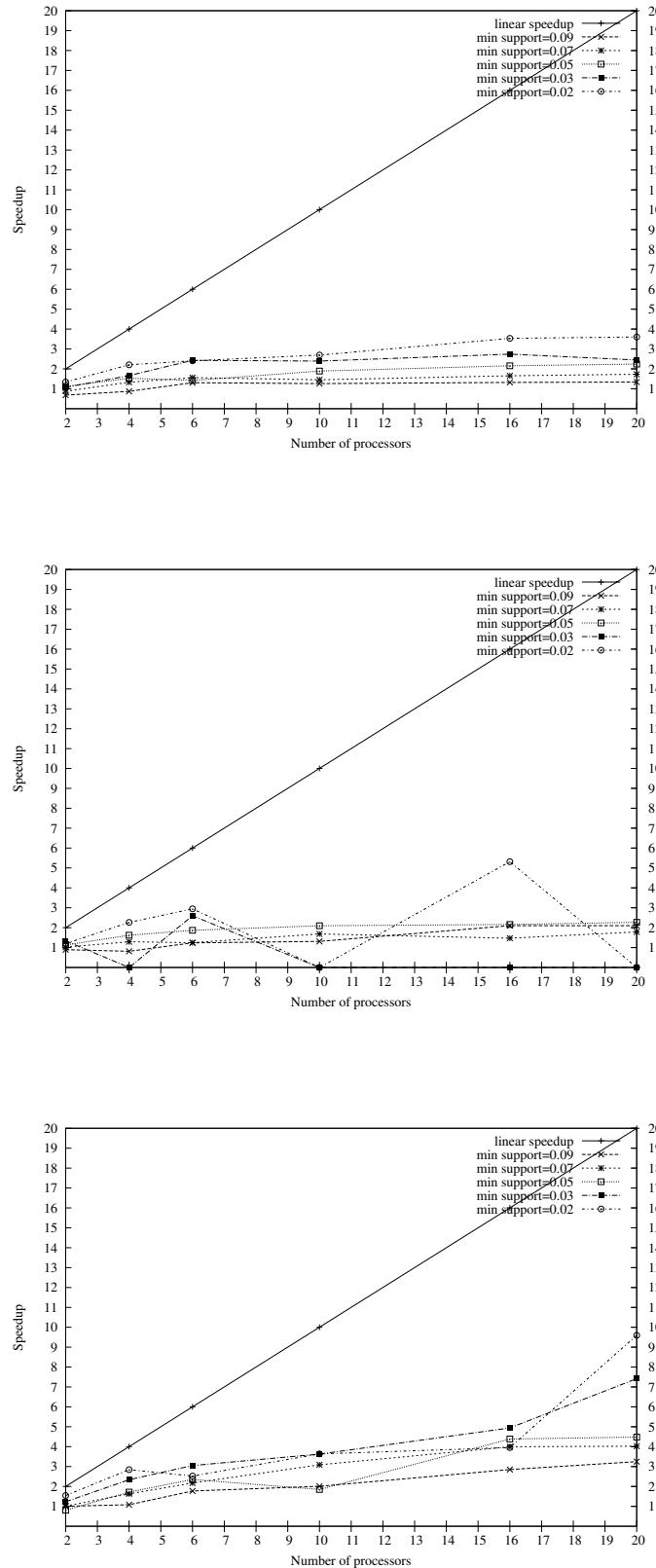


Figure 11.20: Speedups of the PARALLEL-FIMI-SEQ, PARALLEL-FIMI-PAR, and PARALLEL-FIMI-RESERVOIR methods (from top to bottom) on the T500I1P100PL20TL50 database.

11.5 The evaluation of the database replication experiments

We have evaluated the improvement of the database replication and the database replication itself on the real databases. We have not used the data generated by the IBM generator, the replication factor in this data is almost P . The reason for the bad replication factor is the randomness of the data. We have used the following real databases [15]: `kosarak`, `accidents`, `chess`, `connect`, `mushroom`, `pumsb_star`, and `pumsb`. As the implementation of the QKP algorithm, we have downloaded the source code from [5], an implementation of the algorithm described in [10].

The results of the experiments are summarized in tables. For each database there are three tables: improvement of QKP scheduling against the greedy scheduling, see Algorithm 16, the database replication using the greedy scheduling, and the database replication using the QKP schedule. We have chosen the number of processors: 4, 6, 10, and 14.

The biggest improvement of the database replication (28%) is on the `mushroom` database. It can be seen that the biggest improvement is at the relative support level 0.001. The improvements are much smaller, when the relative support is > 0.01 . The `mushroom` database is also one of the two databases where we have achieved a replication factor after reduction $\ll P - 1$ (for 14 processors). The lowest replication factor 2.7 on 14 processors was measured on the `mushroom` database. In most cases the replication factor is between $P - 1$ and P . The replication factor after reduction is also lower ($\ll P - 1$) on the `pumsb_star` database.

Overall, the improvement of the replication factor mostly ranges between $\approx 1\%$ and $\approx 13\%$. It sometimes happens that the replication factor is worse after reduction. The worsening is for the `pumsb` database -0.0464% , `pumsb_star` -2.2881% and -0.2538% . We consider these values as outliers.

Generally it holds that for two processors the database replication is very high, but mostly does not reach P for P processors. However, in most cases the replication factor is between $P - 1$ and P .

The most interesting case is the `mushroom` database. From the experiments it can be seen that the lower the support the better results. The best database replication factor is 10 on 14 processors.

To conclude, from the databases we can made hypothesis that the database replication factor is high for higher values of support and small for lower values of supports.

Improvement(in %):

$P/min_support^*$	0.0050	0.0040	0.0030
4	12.6096	11.6921	15.9684
6	13.6673	19.4744	20.7549
10	18.0931	18.0157	18.7086
14	17.6054	20.2953	21.7529

Database replication *without reduction*:

$P/min_support^*$	0.0050	0.0040	0.0030
4	1.76357	1.9325	1.86456
6	2.08358	2.14564	2.18368
10	2.36798	2.4311	2.49938
14	2.55512	2.55404	2.74345

Database replication *after reduction (using the DB-REPL-MIN algorithm)*:

$P/min_support^*$	0.0050	0.0040	0.0030
4	1.54119	1.70655	1.56682
6	1.79881	1.72779	1.73046
10	1.93954	1.99312	2.03178
14	2.10528	2.03569	2.14667

Table 11.15: Improvement of the database replication of the **kosarak** database.

Improvement(in %):

$P/min_support^*$	0.06	0.05	0.04	0.03	0.02	0.01
4	0.0365	0.0150	0.7585	1.4095	0.0960	0.0833
6	0.6032	0.6150	2.5080	2.8985	0.3852	4.9895
10	3.2480	2.6723	2.5703	3.7139	4.0293	3.8973
14	1.7851	4.0688	7.1765	6.3381	2.5573	4.3714

Database replication *without reduction*:

$P/min_support^*$	0.06	0.05	0.04	0.03	0.02	0.01
4	4	4	4	4	4	4
6	5.99995	5.99996	5.99909	5.99992	5.99999	5.99837
10	9.9964	9.99502	9.99673	9.99737	9.99766	9.99586
14	13.9603	13.9502	13.9414	13.9648	13.9715	13.9636

Database replication *after reduction (using the DB-REPL-MIN algorithm)*:

$P/min_support^*$	0.06	0.05	0.04	0.03	0.02	0.01
4	3.99854	3.9994	3.96966	3.94362	3.99616	3.99667
6	5.96376	5.96306	5.84863	5.82601	5.97688	5.69908
10	9.67172	9.72792	9.73978	9.62608	9.59482	9.60629
14	13.7111	13.3826	12.9409	13.0797	13.6142	13.3532

Table 11.16: Improvement of the database replication of the **accidents** database.

Improvement(in %):							
$P/min_support^*$	0.7	0.6	0.5	0.4	0.3	0.2	0.1
4	0.0000	0.0783	0.0548	0.0235	0.0235	0.6570	1.0717
6	0.0625	0.0313	0.1825	0.2347	0.4642	0.1512	1.9035
10	1.1765	0.1596	0.0688	0.2472	0.2941	0.3817	1.0889
14	0.2372	0.2257	0.2214	0.3886	1.0614	0.6143	4.2286

Database replication *without reduction*:

$P/min_support^*$	0.7	0.6	0.5	0.4	0.3	0.2	0.1
4	4	4	4	4	4	4	4
6	6	6	6	6	6	6	6
10	10	9.99875	10	10	10	10	10
14	13.9994	13.9987	14	14	14	14	14

Database replication *after reduction (using the DB-REPL-MIN algorithm)*:

$P/min_support^*$	0.7	0.6	0.5	0.4	0.3	0.2	0.1
4	4	3.99687	3.99781	3.99906	3.99906	3.97372	3.95713
6	5.99625	5.99812	5.98905	5.98592	5.97215	5.99093	5.88579
10	9.88235	9.98279	9.99312	9.97528	9.97059	9.96183	9.89111
14	13.9662	13.9671	13.969	13.9456	13.8514	13.914	13.408

Table 11.17: Improvement of the database replication of the **chess** database.

Improvement(in %):

$P/min_support^*$	0.3	0.2	0.1
4	0.0000	0.0000	0.0103
6	0.2218	0.0005	2.3822
10	0.3900	1.5442	1.2324
14	0.7933	1.3633	1.2881

Database replication *without reduction*:

$P/min_support^*$	0.3	0.2	0.1
4	4	4	4
6	6	6	6
10	9.96607	9.96607	10
14	13.9661	13.9661	13.9661

Database replication *after reduction (using the DB-REPL-MIN algorithm)*:

$P/min_support^*$	0.3	0.2	0.1
4	4	4	3.99959
6	5.98669	5.99997	5.85707
10	9.9272	9.81217	9.87676
14	13.8553	13.7757	13.7862

Table 11.18: Improvement of the database replication of the **connect** database.

Improvement(in %):

$P/min_support^*$	0.1	0.08	0.06	0.04	0.02	0.001
4	0.6155	0.8617	0.5663	1.0093	4.7883	11.0753
6	2.6015	3.9635	1.4903	1.8668	2.1500	14.6110
10	3.8776	3.2556	3.5445	9.6659	8.0022	22.7943
14	5.8516	5.9913	7.9623	7.7287	10.0239	28.9319

Database replication *without reduction*:

$P/min_support^*$	0.1	0.08	0.06	0.04	0.02	0.001
4	4	4	4	4	4	4
6	5.99951	6	5.99606	6	6	6
10	9.93599	9.98929	9.98769	9.99926	9.99852	9.96972
14	13.9791	13.8902	13.9357	13.979	13.9547	13.8765

Database replication *after reduction (using the DB-REPL-MIN algorithm)*:

$P/min_support^*$	0.1	0.08	0.06	0.04	0.02	0.001
4	3.97538	3.96553	3.97735	3.95963	3.80847	3.55699
6	5.84343	5.76219	5.9067	5.88799	5.871	5.12334
10	9.55071	9.66408	9.63368	9.03274	9.19842	7.69719
14	13.1611	13.058	12.8261	12.8986	12.5559	9.86177

Table 11.19: Improvement of the database replication of the `mushroom` database.

Improvement(in %):

$P/min_support^*$	0.25	0.3	0.32	0.35	0.42	0.49	0.56
4	7.3980	0.9556	6.3810	0.9308	1.9091	6.1343	6.5228
6	6.9107	8.4605	2.8155	4.2704	4.4023	4.6576	-2.2881
10	7.1429	4.4941	18.9267	9.2538	-0.2538	8.1635	7.4531
14	9.0592	5.8842	5.0286	22.2261	3.5505	13.2112	14.8919

Database replication *without reduction*:

$P/min_support^*$	0.25	0.3	0.32	0.35	0.42	0.49	0.56
4	4	3.72837	4	4	3.72837	3.99865	3.72833
6	5.72837	5.72837	5.72819	5.72801	5.72705	5.72502	5.51461
10	9.72616	9.70714	9.70202	9.47426	9.59337	9.71667	9.02567
14	13.6138	13.2065	13.3537	13.1593	13.2798	13.0995	12.124

Database replication *after reduction (using the DB-REPL-MIN algorithm)*:

$P/min_support^*$	0.25	0.3	0.32	0.35	0.42	0.49	0.56
4	3.70408	3.69274	3.74476	3.96277	3.65719	3.75336	3.48514
6	5.3325	5.24372	5.56691	5.4834	5.47493	5.45837	5.64079
10	9.03143	9.27089	7.86575	8.59753	9.61772	8.92345	8.35298
14	12.3805	12.4294	12.6822	10.2345	12.8083	11.3689	10.3185

Table 11.20: Improvement of the database replication of the `pumsb_star` database.

Improvement(in %):

$P/min_support^*$	0.9	0.85	0.8
4	0.0040	0.2263	0.0152
6	0.1925	0.2650	0.2111
10	-0.0464	0.8433	0.1771
14	0.6181	0.0179	0.9075

Database replication *without reduction*:

$P/min_support^*$	0.9	0.85	0.8
4	4	4	4
6	5.98371	5.9992	5.99839
10	9.97184	9.98096	9.98304
14	13.9612	13.9685	13.9393

Database replication *after reduction (using the DB-REPL-MIN algorithm)*:

$P/min_support^*$	0.9	0.85	0.8
4	3.99984	3.99095	3.99939
6	5.97219	5.9833	5.98573
10	9.97647	9.89679	9.96536
14	13.8749	13.966	13.8128

Table 11.21: Improvement of the database replication of the pumsb database.

12 Conclusion and future work

12.1 Conclusion

In our work, we have shown a method that parallelize an arbitrary algorithm for mining of FIs. We have proposed two methods for estimation of the size of a PBEC based on the MODIFIED-COVERAGE-ALGORITHM and explained why the sampling is just a heuristic. In order to make better estimation results, we have proposed estimation of the relative size of PBECs based on the VITTER-RESERVOIR-SAMPLING algorithm. We have shown how big error can be made by our “double sampling process”, see Theorem 6.4 and Corollary 6.5.

We have shown how to execute an arbitrary sequential algorithm for mining of all MFIs in parallel that mines a superset of all MFIs M in order to speedup the sampling process based on the MODIFIED-COVERAGE-ALGORITHM and proved that the size of M can be larger then \widetilde{M} , see Theorem 7.5 and Chapter 7.

Then in Chapter 8 we have proposed our three methods for parallel mining of MFIs, called PARALLEL-FIMI-SEQ, PARALLEL-FIMI-PAR, and PARALLEL-FIMI-RESERVOIR, on a distributed memory parallel computer. In Chapter 9 we have shown how to efficiently execute the ECLAT algorithm in Phase 4 of our new method. In Chapter 10, we have discussed the database replication factor and the possibilities of minimization of the database replication factor.

In Chapter 11 we have experimentally evaluated the performance of our new method and the errors of the estimates of the size of union of PBECs. Additionally, we have shown that minimizing the database replication factor based on the solution of the quadratic knapsack problem big improvement on all artificial databases and makes slight improvement on some real databases.

12.2 Future work

We would like to improve the PARALLEL-FIMI-RESERVOIR algorithm. The inefficiency in the algorithm comes from the fact that the reservoir sampling is embedded in a regular ECLAT algorithm, i.e., the support is computed for each frequent itemsets while sampling the FIs. This inefficiency could be removed by using smarter algorithm that would use

the same optimizations as for example the $fpmax^*$ algorithm. The $fpmax^*$ algorithm is an algorithm for mining of MFIs and uses a list of MFIs to check for support of newly generated frequent itemset.

The IBM database generator in some cases does not generate databases similar to the real databases. We have already developed some database characteristics, however their description is out of the scope of this thesis. Additionally, we would like to create a database generator that would generate more realistic and structured databases then the IBM generator.

13 Bibliography

- [1] Description of the Infiniband network.
Available at <http://www.intel.com/technology/infiniband/>.
- [2] Description of the Myrinet network.
Available at <http://www.myricom.com/myrinet/overview/>.
- [3] Description of the round robin tournament on Wikipedia.
Available at http://en.wikipedia.org/wiki/Round_robin_tournament.
- [4] Dictionary of Algorithms and Data Structures.
Available at <http://www.itl.nist.gov/div897/sqg/dads/>.
- [5] The source code of the solution of the quadratic knapsack problem.
Available at <http://www.diku.dk/hjemmesider/ansatte/pisinger/codes.html>.
- [6] R. Agrawal and J. C. Shafer. Parallel mining of association rules. *IEEE Transactions On Knowledge And Data Engineering*, 8(6):962–969, 1996.
- [7] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of 20th International Conference on Very Large Data Bases*, pages 487–499. Morgan Kaufmann, 1994.
- [8] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.
- [9] T. Calders and B. Goethals. Mining all non-derivable frequent itemsets. In *Proceedings of the 6th European Conference on Principles of Data Mining and Knowledge Discovery*, Lecture Notes in Computer Science, pages 74–85. Springer-Verlag, 2002.
- [10] A. Caprara, D. Pisinger, and P. Toth. Exact solution of the quadratic knapsack problem. *INFORMS Journal on Computing*, 11(6):125–137, 1999.
- [11] D. W.-L. Cheung, S. D. Lee, and Y. Xiao. Effect of data skewness and workload balance in parallel data mining. *Knowledge and Data Engineering*, 14(3):498–514, 2002.

- [12] D. W.-L. Cheung and Y. Xiao. Effect of data distribution in parallel mining of associations. *Data Mining and Knowledge Discovery*, 3(3):291–314, 1999.
- [13] V. Chvátal. The tail of the hypergeometric distribution. *Discrete Mathematics*, 25(3):285 – 287, 1979.
- [14] B. Goethals. <http://adrem.ua.ac.be/~goethals/software/>.
- [15] B. Goethals and M. J. Zaki. Frequent itemset mining repository. <http://fimi.cs.helsinki.fi>.
- [16] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics*, 17(2):416–429, 1969.
- [17] G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *FIMI '03, Frequent Itemset Mining Implementations, Proceedings of the ICDM 2003 Workshop on Frequent Itemset Mining Implementations*, volume 90 of *CEUR Workshop Proceedings*, 2003.
- [18] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *2000 ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM Press, 2000.
- [19] A. Javed and A. Khokhar. Frequent pattern mining on message passing multiprocessor systems. *Distributed and Parallel Databases*, 16(3):321–334, 2004.
- [20] J. L. Johnson. *Probability and Statistics for Computer Science*. Wiley-Interscience, New York, NY, USA, 2008.
- [21] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack problems*. Springer-Verlag, 2004.
- [22] R. Kessl and P. Tvrđík. Probabilistic load balancing method for parallel mining of all frequent itemsets. In *Proceedings of the 18th IASTED International Conference on Parallel and distributed computing systems 2006*, pages 578–586. ACTA Press, 2006.
- [23] R. Kessl and P. Tvrđík. Toward more parallel frequent itemset mining algorithms. In *Proceedings of the 19th IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 97–103. ACTA Press, 2007.

- [24] R. Motwani and P. Raghavan. *Randomized algorithms*. Cambridge university press, 1995.
- [25] Noga Alon and Joel H. Spencer. *The Probabilistic Method*. Wiley, New York, 2000.
- [26] S. Orlando, P. Palmerini, R. Perego, and F. Silvestri. A scalable multi-strategy algorithm for counting frequent sets. In *Proceedings of the 5th Workshop on High Performance Data Mining*, pages 421–435, Berlin, Heidelberg, 2002. Springer-Verlag.
- [27] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, and D. Yang. H-mine: hyper-structure mining of frequent patterns in large databases. In *Proceedings IEEE International Conference on Data Mining*, pages 441–448. IEEE Computer Society, 2001.
- [28] I. Pramudiono and M. Kitsuregawa. Parallel FP-Growth on PC Cluster. In *Proceedings of the 7th Pacific-Asia Conference of Knowledge Discovery and Data Mining*, pages 467–473. Springer, 2003.
- [29] M. Skala. Hypergeometric tail inequalities: ending the insanity. <http://ansuz.sooke.bc.ca/professional/hypergeometric.pdf>.
- [30] H. Toivonen. Sampling large databases for association rules. In *International Conference on Very Large Data Bases*, pages 134–145. Morgan Kaufman, 1996.
- [31] A. Veloso. New parallel algorithms for frequent itemset mining in large databases. In *Proceedings of the Symposium on Computer Architectures and High Performance Computing*, pages 158–166, 2003.
- [32] J. S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, 1985.
- [33] G. Yang. The complexity of mining maximal frequent itemsets and maximal frequent patterns. In *Proceedings of the tenth ACM SIGKDD International Conference on Knowledge Discovery and Data mining*, pages 344–353, 2004.
- [34] O. R. Zaiane, M. El-Hajj, and P. Lu. Fast parallel association rule mining without candidacy generation. *First IEEE International Conference on Data Mining*, pages 665–668, 2001.
- [35] M. Zaki and K. Gouda. Fast vertical mining using diffsets, Renssealer Polytechnical Institue technical report No. 01-1, 2001.

- [36] M. J. Zaki. *Scalable Data Mining for Rules*. PhD thesis, Department of Computer Science, Rensselaer Polytechnic Institute, 1998.
- [37] M. J. Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 12:372–390, 2000.
- [38] M. J. Zaki and C.-J. Hsiao. Charm: An efficient algorithm for closed itemset mining. In *SIAM International Conference on Data Mining*, pages 457–473. SIAM, 2002.
- [39] M. J. Zaki, M. Ogihara, S. Parthasarathy, and W. Li. Parallel data mining for association rules on shared-memory multiprocessors. Technical Report TR618, 1996.
- [40] M. J. Zaki, S. Parthasarathy, and W. Li. A localized algorithm for parallel association mining. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 321–330. ACM Press, 1997.
- [41] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In *3rd International Conference on Knowledge Discovery and Data Mining*, pages 283–286. AAAI Press, 1997.

14 Refereed publications of the author

- [42] Robert Kessl and Pavel Tvrdík. Probabilistic load balancing method for parallel mining of all frequent itemsets. In *PDCS '06: Proceedings of the 18th IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 578–586, Anaheim, CA, USA, 2006. ACTA Press.
- [43] Robert Kessl and Pavel Tvrdík. Toward more parallel frequent itemset mining algorithms. In *PDCS '07: Proceedings of the 19th IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 97–103, Anaheim, CA, USA, 2007. ACTA Press.

15 Unrefereed publications of the author

- [44] Robert Kessl. Parallel and sequential mining of frequent itemsets, 2004. Master Thesis.
- [45] Robert Kessl. Experimental evaluation of probabilistic load balancing for parallel mining of frequent itemsets, 2007. Poster.
- [46] Robert Kessl. Probabilistic load balancing for mining of frequent itemsets. In *Proceedings of Workshop 2008, Part A, Special issue*, pages 138–139. Czech Technical University in Prague, CTU Publishing House, 2008.
- [47] Robert Kessl. Parallel mining of frequent itemsets. In *Proceding of PhD. Conference 2009: Doktorandské dny '09*, pages 53–61. Institute of Computer Science/MatfyzPress, 2009.

A Discrete probability distributions and tails

A.1 Chernoff bounds

The Chernoff bound is used to bound the number of sucessfull independent Poisson experiments. Let X_1, \dots, X_n be n independent random variables such that $X_i \in \{0, 1\}$ and $P[X_i = 1] = p_i \in [0, 1]$. Let $X = \sum_i X_i$ and let μ be the expectation of X , then the Chernoff bounds, i.e., the probability $P[X \leq (1 - \delta)\mu]$ or $P[X \geq (1 + \delta)\mu]$ where $\delta \in [0, 1]$ is:

$$P[X \leq (1 - \delta)\mu] \leq e^{\frac{-\mu\delta^2}{4}}$$

$$P[X \geq (1 + \delta)\mu] \leq e^{\frac{-\mu\delta^2}{4}}$$

Another variant of the Chernoff bounds is provided in [25]. Let have the following assumptions: $p \in [0, 1]$, X_1, \dots, X_n mutually independent random variables with $P[X_i = 1 - p] = p$ $P[X_i = -p] = 1 - p$, and let $X = X_1 + \dots + X_n$. Then for $a > 0$:

$$P[|X| > a] < \exp^{-2a^2/n} \quad (\text{A.1})$$

The equation (A.1) is the actual equation used by Toivonen in [30] for proving Theorem 6.1.

A.2 Hypergeometric distribution and tails

The hypergeometric distribution describes the following problem: let us have an urn with N balls of which M are black and $N - M$ are white. A sample of n balls is drawn without replacement. The distribution of i , the number of black balls, is:

$$P[X = i] = \frac{\binom{M}{i} \binom{N-M}{n-i}}{\binom{N}{n}}.$$

The expectation of i is $E[i] = n \frac{M}{N}$. For any $\epsilon \geq 0$ the difference $E[i] - i$ is bound by:

$$P[i \geq E[i] + \epsilon \cdot n] \leq e^{-2\epsilon^2 n} \quad (\text{A.2})$$

and

$$P[i \leq E[i] - \epsilon \cdot n] \leq e^{-2\epsilon^2 n}. \quad (\text{A.3})$$

For more details, see [29].

A more precise bound can be computed using the Kullback-Leibler divergence of two Bernoulli distributed random variables, denoted by $D(\cdot || \cdot)$. Let $p = M/N$ and $\epsilon \geq 0$, then:

$$P[i \geq E[i] + \epsilon \cdot n] \leq \left(\left(\frac{p}{p+\epsilon} \right)^{p+\epsilon} \left(\frac{1-p}{1-p-\epsilon} \right)^{1-p-\epsilon} \right)^n = e^{-nD(p+\epsilon||p)} \quad (\text{A.4})$$

or

$$P[i \leq E[i] - \epsilon \cdot n] \leq \left(\left(\frac{p}{p-\epsilon} \right)^{p-\epsilon} \left(\frac{1-p}{1-p\epsilon} \right)^{1-p+\epsilon} \right)^n = e^{-nD(p-\epsilon||p)}. \quad (\text{A.5})$$

Therefore,

$$P[E[i] - \epsilon n \leq i \leq E[i] + \epsilon n] \leq 1 - (e^{-nD(p-\epsilon||p)} + e^{-nD(p+\epsilon||p)}). \quad (\text{A.6})$$

The *multivariate hypergeometric distribution* is the same as the hypergeometric distribution, except that the balls can have more colors, defined as follows: let the number of colors be C and the number of balls colored with color i is M_i and the total number of balls is $N = \sum_i M_i$. Let X_i , $1 \leq i \leq C$, be a random variable representing the number of balls colored by the i -th color. The sample of size n is drawn from balls and X_i balls, such that $n = \sum_{i=1}^C X_i$ are colored by the i th color. Then the probability mass function is:

$$P(X_1 = k_1, \dots, X_C = k_C) = \frac{\prod_{i=1}^C \binom{M_i}{k_i}}{\binom{N}{n}}.$$

where k_i are integers. The expectation is $E[X_i] = n \frac{M_i}{N}$. Obviously, the tail inequalities of the multivariate hypergeometric distribution are the same as for the hypergeometric distribution, i.e., the multivariate hypergeometric distribution with $C = 2$.

A.3 Multivariate binomial distribution

The multivariate binomial distribution, or so called multinomial distribution, is a distribution describing the outcome of n independent Bernoulli trials where each trial results in k possible outcomes. The i th outcome of each trial has the probability p_i , $\sum_{1 \leq i \leq k} p_i = 1$. The probability mass function of the multivariate binomial distribution is:

$$\begin{aligned} f(x_1, \dots, x_k; n, p_1, \dots, p_k) &= Pr(X_1 = x_1, \dots, X_k = x_k) \\ &= \begin{cases} \frac{n!}{x_1! \cdots x_k!} \cdot p_1^{x_1} \cdots p_k^{x_k}, & \text{when } \sum_{i=1}^k x_i = n \\ 0, & \text{otherwise} \end{cases} \end{aligned} \quad (\text{A.7})$$

for non-negative integers x_1, \dots, x_k .

B Selected sequential algorithms

This appendix describes selected sequential algorithms together with datastructures and optimizations, an edited version of our master thesis [44]. In this appendix, we describe some of the existing algorithms for mining of FIs and some of its optimizations. Namely: 1) the Apriori algorithm in Section B.1; 2) the FP-Growth algorithm in Section B.2; and 3) the Eclat algorithm in Section B.3. In Section B.4 we show optimizations of the Eclat algorithm and we finish the appendix with the algorithm that generates the association rules from FIs, see Section B.5.

B.1 The Apriori algorithm

The Apriori algorithm [7] is a BFS algorithm based solely on the monotonicity property, see Theorem 2.12. The Apriori algorithm uses the notion of *candidates itemsets*, see Definition 5.1. In the further text, we denote the set of all FIs of size k by F_k and the set of *candidates* on frequent itemsets by C_k . Obviously, $F_k \subseteq C_k$. The algorithm proceeds in steps. In step $k > 1$, it first generates a set C'_k of possibly frequent itemsets of size k , such that $C_k \subseteq C'_k$, from the set of frequent itemsets F_{k-1} of size $k-1$ computed in the previous step $k-1$. The set C'_k is generated in the following way: from F_{k-1} , the set of frequent itemsets of size $k-1$, we find all pairs of itemsets $U = (u_1, \dots, u_{k-1}), W = (w_1, \dots, w_{k-1}) \in F_{k-1}$ that are identical in the first $k-2$ items, i.e., $u_i = w_i, i \leq k-2$. From each such pair U, W a new candidate $V = \{u_1, \dots, u_{k-2}, u_{k-1}, w_{k-1}\}$ is constructed. The candidates C_k are generated from C'_k in the following way: for each $U \in C'_k$, we apply the monotonicity principle, i.e., we test whether each subset $W \subset V, k-1 = |W| = |V-1|$ is present in F_{k-1} . The reason is that all subsets of U must be frequent in order for U to be also frequent, see Corollary 2.13. Therefore, if some subset of U of size $k-1$ is not in F_{k-1} then U is deleted from C_k . The algorithm for generation of candidates follows:

Algorithm 24 The GENERATE-CANDIDATES functionGENERATE-CANDIDATES(**In:** Itemset F_k)

```

1:  $C \leftarrow \emptyset$ 
2: for all  $U = (u_1, \dots, u_k), W = (w_1, \dots, w_k) \in F_k$  do
3:   if  $u_k < w_k \wedge u_j = w_j, j < k$  then
4:      $C \leftarrow C \cup \{(u_1, \dots, u_{k-1}, u_k, w_k)\}$ 
5:   end if
6: end for
7: for  $U \in C$  do
8:   if TEST-SUBSET( $F_k, U$ ) = false then
9:     delete  $U$  from  $C$ 
10:   end if
11: end for
12: return  $C$ 
```

In the first step ($k = 1$), the Apriori algorithm starts with $C_1 = \{\{b_i\} : b_i \in \mathcal{B}\}$ and counts support of each $U \in C_1$ in a single scan of the database, creating F_1 . In steps $k > 1$, the algorithm must compute the support of each $U \in C_k$, i.e., we create the set $F_k = \{U | U \in C_k, \text{Supp}(U) \geq \text{min_support}\}$.

The algorithm ends if: 1) all candidates are deleted; 2) all candidates turn out not to be frequent. In both cases the resulting F_k is empty.

To make the explanation of the Apriori algorithm simple, we omit the details of the TEST-SUBSET and COMPUTE-SUPPORT algorithms. The TEST-SUBSET and COMPUTE-SUPPORT algorithms are described in Section B.1.2. However, it is not necessary to understand the two algorithms in order to understand the Apriori algorithm.

In the following text, we use the algorithm $\text{TEST-SUBSET}(F_k, U)$ that checks whether all subsets of size $|U| - 1 = k$ are present in the set F_k . Additionally, we use the algorithm $\text{COMPUTE-SUPPORT}(\mathcal{D}, C_k)$ that computes the support of each $U \in C_k$.

Since the evaluation of the support for each candidate is quite a time-consuming task, it has to be done as fast as possible on as few candidates as possible. Many candidates are generated uselessly, because they turn out not to be frequent.

An example of the execution of the Apriori algorithm on a small database is given in the Example B.1. The pseudocode of the Apriori algorithm can be found in Algorithm 25.

Example B.1: An example execution of the Apriori algorithm

Input: $D =$	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: center;">TID</th><th style="text-align: center;">Transaction</th></tr> </thead> <tbody> <tr> <td style="text-align: center;">1</td><td style="text-align: center;">$\{1, 2, 5\}$</td></tr> <tr> <td style="text-align: center;">2</td><td style="text-align: center;">$\{1, 3, 5\}$</td></tr> <tr> <td style="text-align: center;">3</td><td style="text-align: center;">$\{2, 4, 5\}$</td></tr> <tr> <td style="text-align: center;">4</td><td style="text-align: center;">$\{1, 2, 3, 5\}$</td></tr> </tbody> </table>	TID	Transaction	1	$\{1, 2, 5\}$	2	$\{1, 3, 5\}$	3	$\{2, 4, 5\}$	4	$\{1, 2, 3, 5\}$	$\mathcal{B} = \{1, 2, 3, 4, 5\}, min_support = 2$				
TID	Transaction															
1	$\{1, 2, 5\}$															
2	$\{1, 3, 5\}$															
3	$\{2, 4, 5\}$															
4	$\{1, 2, 3, 5\}$															
$k = 1$	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: center;">C_1</th><th style="text-align: center;">Support</th></tr> </thead> <tbody> <tr> <td style="text-align: center;">$\{1\}$</td><td style="text-align: center;">3</td></tr> <tr> <td style="text-align: center;">$\{2\}$</td><td style="text-align: center;">3</td></tr> <tr> <td style="text-align: center;">$\{3\}$</td><td style="text-align: center;">2</td></tr> <tr> <td style="text-align: center;">$\{4\}$</td><td style="text-align: center;">1</td></tr> <tr> <td style="text-align: center;">$\{5\}$</td><td style="text-align: center;">5</td></tr> </tbody> </table>	C_1	Support	$\{1\}$	3	$\{2\}$	3	$\{3\}$	2	$\{4\}$	1	$\{5\}$	5	<table border="1" style="border-collapse: collapse; width: 100px;"> <tr> <td style="text-align: center;">F_1</td></tr> <tr> <td style="text-align: center;">$\{\{1\}, \{2\}, \{3\}, \{5\}\}$</td></tr> </table>	F_1	$\{\{1\}, \{2\}, \{3\}, \{5\}\}$
C_1	Support															
$\{1\}$	3															
$\{2\}$	3															
$\{3\}$	2															
$\{4\}$	1															
$\{5\}$	5															
F_1																
$\{\{1\}, \{2\}, \{3\}, \{5\}\}$																
$k = 2$	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: center;">C_2</th><th style="text-align: center;">Support</th></tr> </thead> <tbody> <tr> <td style="text-align: center;">$\{1, 2\}$</td><td style="text-align: center;">2</td></tr> <tr> <td style="text-align: center;">$\{1, 3\}$</td><td style="text-align: center;">2</td></tr> <tr> <td style="text-align: center;">$\{1, 5\}$</td><td style="text-align: center;">3</td></tr> <tr> <td style="text-align: center;">$\{2, 3\}$</td><td style="text-align: center;">1</td></tr> <tr> <td style="text-align: center;">$\{2, 5\}$</td><td style="text-align: center;">3</td></tr> </tbody> </table>	C_2	Support	$\{1, 2\}$	2	$\{1, 3\}$	2	$\{1, 5\}$	3	$\{2, 3\}$	1	$\{2, 5\}$	3	<table border="1" style="border-collapse: collapse; width: 100px;"> <tr> <td style="text-align: center;">F_2</td></tr> <tr> <td style="text-align: center;">$\{\{1, 2\}, \{1, 3\}, \{1, 5\}, \{2, 5\}\}$</td></tr> </table>	F_2	$\{\{1, 2\}, \{1, 3\}, \{1, 5\}, \{2, 5\}\}$
C_2	Support															
$\{1, 2\}$	2															
$\{1, 3\}$	2															
$\{1, 5\}$	3															
$\{2, 3\}$	1															
$\{2, 5\}$	3															
F_2																
$\{\{1, 2\}, \{1, 3\}, \{1, 5\}, \{2, 5\}\}$																
$k = 3$	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: center;">C_3</th><th style="text-align: center;">Support</th></tr> </thead> <tbody> <tr> <td style="text-align: center;">$\{1, 2, 5\}$</td><td style="text-align: center;">2</td></tr> </tbody> </table>	C_3	Support	$\{1, 2, 5\}$	2	<table border="1" style="border-collapse: collapse; width: 100px;"> <tr> <td style="text-align: center;">F_3</td></tr> <tr> <td style="text-align: center;">$\{\{1, 2, 5\}\}$</td></tr> </table>	F_3	$\{\{1, 2, 5\}\}$								
C_3	Support															
$\{1, 2, 5\}$	2															
F_3																
$\{\{1, 2, 5\}\}$																

Algorithm 25 The APRIORI algorithm

APRIORI(**In:** Database \mathcal{D} , **In:** Integer $min_support$, **Out:** Set \mathcal{F})

- 1: $k \leftarrow 1$
 - 2: Compute all frequent items and store them into \mathcal{B}
 - 3: $C_k \leftarrow \{\{b\} : b \in \mathcal{B}\}$
 - 4: **while** C_k not empty **do**
 - 5: COMPUTE-SUPPORT(\mathcal{D}, C_k)
 - 6: **for all** $U \in C_k$ **do**
 - 7: **if** $Supp(U) < min_support$ **then**
 - 8: delete U from C_k
 - 9: **end if**
 - 10: **end for**
 - 11: $F_k \leftarrow C_k$
 - 12: $C_{k+1} \leftarrow \text{GENERATE-CANDIDATES}(F_k)$
 - 13: $k \leftarrow k + 1$
 - 14: **end while**
 - 15: $\mathcal{F} \leftarrow \bigcup_{i=1}^{k-1} F_i$
 - 16: **return**
-

B.1.1 Prefix trie

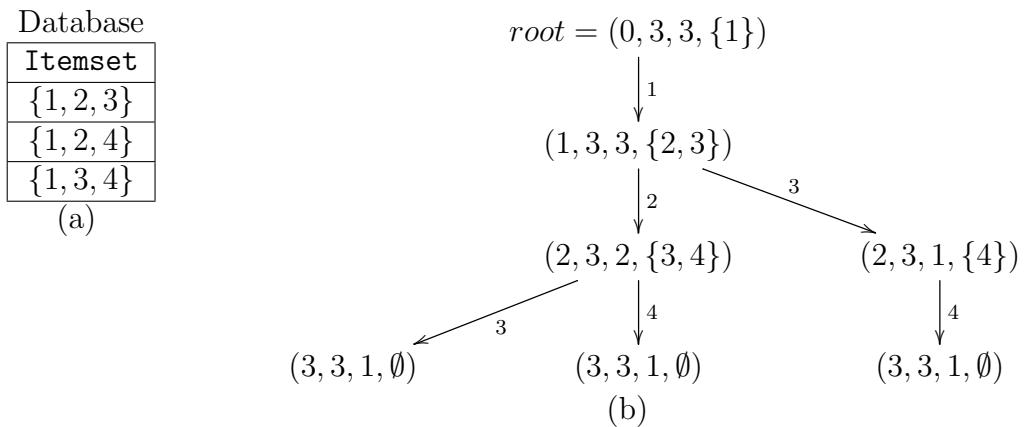
Definition B.1 (Prefix trie). Let $\{U|U \subseteq \mathcal{B}\}$ be a collection of itemsets. Each $U = (u_1, u_2, \dots, u_{|U|})$ is sorted according to some order $<$, i.e. $u_k < u_l, k < l$. Let $\mathcal{V} = \{v_i\}$ be a set of nodes and $E = \{e_i = (v_i, v_j)|v_i, v_j \in \mathcal{V}\}$ be a set of edges of an oriented acyclic graph $G = (\mathcal{V}, E)$. G is called a prefix trie iff: each node v_j corresponds to a prefix (u_1, u_2, \dots, u_l) of an itemset U and an edge (v_j, v_k) is present in E iff there exist prefix $(u_1, u_2, \dots, u_l, u_{l+1})$ of U . The node v_j has associated the item u_l and the node v_k has associated the item u_{l+1} . Each node v_j is represented by a tuple $(\text{depth}, \text{max_depth}, \text{support}, \text{children})$, where depth , max_depth , and support are integers. The field children is a set $\{(item, v_j)\}$.

Inserting the pair $(item, v)$ into the field children , we denote by $\text{children}[item] \leftarrow v$. Reading the node from the field children is denoted by $v \leftarrow \text{children}[item]$.

Note: the word *trie* comes from the noun reTRIEval and is pronounced as "tree", see [4].

All operations (e.g., subset test, support increment and GENERATE-CANDIDATES) in the Apriori algorithm are based on the prefix trie structure. Figure B.2 shows an example of a prefix trie. In the prefix trie, the Apriori algorithm stores the set of candidates of size k , C_k , or the FIs of size k , F_k . However, generally the trie can store itemsets of arbitrary sizes.

Example B.2: An example of a prefix trie data structure. The set of children is represented by arcs with labels. The root represents the empty itemset. The content of each node is $(\text{depth}, \text{max_depth}, \text{support}, \text{children})$ (the depth field is counted from 0). The prefix trie (b) for the database (a) is constructed by 3 calls: $\text{INSERT-PREFIXTRIE}(\{1, 2, 3\}, \text{root})$, $\text{INSERT-PREFIXTRIE}(\{1, 2, 4\}, \text{root})$, $\text{INSERT-PREFIXTRIE}(\{1, 3, 4\}, \text{root})$. The INSERT-PREFIXTRIE algorithm can be found in Algorithm 26



As an example of a data operation, we describe the INSERT-PREFIXTRIE procedure that

inserts an itemset into a prefix trie, see Algorithm 26.

Algorithm 26 The INSERT-PREFIXTRIE procedure (prefix trie)

INSERT-PREFIXTRIE(In:** Itemset U , **In/Out:** Node N)**

```

1: if  $|U| = N.\text{depth}$  then
2:   return
3: end if
4:  $i \leftarrow N.\text{depth}$ 
5: if  $U[i] \in N.\text{children}$  then
6:   INSERT-PREFIXTRIE( $U$ ,  $N.\text{children}[U[i]]$ )
7: else
8:    $N' \leftarrow \text{new Node}$ 
9:    $N'.\text{depth} \leftarrow N.\text{depth}$ 
10:   $N.\text{children}[U[i]] \leftarrow N'$ 
11:  INSERT-PREFIXTRIE( $U$ ,  $N'$ )
12: end if
```

The INSERT-PREFIXTRIE procedure is called: $\text{INSERT-PREFIXTRIE}(\{1, 2, 3\}, \text{root})$

B.1.2 Test-Subset function and the Compute-Support procedure using prefix trie

In the GENERATE-CANDIDATES function on code line 8 of Algorithm 24, we want to test if all subsets of size $k - 1$ of some itemset U of size k are contained in a set of itemsets of size $k - 1$, e.g., F_{k-1} . The set F_{k-1} is represented by a prefix trie with maximal height $\text{max_height} = k - 1$. The code line 8 shows that the algorithm is called by TEST-SUBSET(F_{k-1}, U). Since we represent the set F_{k-1} by a prefix trie, we show the TEST-SUBSET algorithm. The TEST-SUBSET algorithm has the first argument replaced by a prefix trie node and has an additional helper parameter (representing the depth of the recursion), i.e., let R_k be a root of a hash trie representing the set F_k the algorithm TEST-SUBSET($R_k, U, 0$) is then called by TEST-SUBSET(F_{k-1}, U), i.e., the TEST-SUBSET shown in Algorithm 24 could be implemented as:

Algorithm 27 The TEST-SUBSET function (using prefix trie)

TEST-SUBSET(In:** Set F_k , **In:** Itemset U)**

- 1: $R_k \leftarrow$ prefix trie representing the set F_k
 - 2: TEST-SUBSET($R_k, U, 0$)
-

The algorithm TEST-SUBSET, shown in Algorithm 27 works as follows: in the root, we get child for each item $b_i \in U$ and recursively test $U \setminus \{b_i\}$ for all subsets of size $k - 2$. Thus, in an interior node in which we get by following the item b_i , we will recursively test all children which we get by hashing items $b_j > b_i$. If the value returned from the recursive call is **true**, we continue with the recursive descent, otherwise **false** return. In a leaf node, we return **true**. If the return value from root is **true** then all subsets of t are in this prefix trie.

The COMPUTE-SUPPORT procedure works as follows: it iterates over the database transactions t incrementing the support of some candidates itemsets using the INCREMENT-SUPPORT procedure. The INCREMENT-SUPPORT procedure increments the support of all candidate itemsets that are subsets of the transaction t . The INCREMENT-SUPPORT procedure is almost the same as the TEST-SUBSET procedure except that the support of a leaf node is incremented and nothing returned.

The pseudocode of the TEST-SUBSET function and COMPUTE-SUPPORT procedure follows:

Algorithm 28 The TEST-SUBSET function (using prefix trie)

TEST-SUBSET(**In:** Node N, **In:** Itemset U, **In:** Integer index)

```

1: for all  $i, \text{index} \leq i < |U|$  do
2:   if N is internal then
3:     if  $U[\text{index}] \in N.\text{children}$  and  $|U| - i \geq \text{max\_depth} - \text{depth}$  then
4:        $\text{result} \leftarrow \text{TEST-SUBSET}(N.\text{children}[U[\text{index}]], U, \text{index} + 1)$ 
5:       if  $\text{result} = \text{false}$  then
6:         return false
7:       end if
8:     else
9:       return false
10:    end if
11:   else if N is leaf then
12:     return true
13:   end if
14: end for
15: return result

```

Algorithm 29 The COMPUTE-SUPPORT procedure

COMPUTE-SUPPORT(**In:** Database D, **In/Out:** Node N)

```

1: for all  $t \in \mathcal{D}$  do
2:   INCREMENT-SUPPORT(N, t, 0)
3: end for

```

Algorithm 30 The INCREMENT-SUPPORT procedure

INCREMENT-SUPPORT(**In/Out**: Node N , **In**: Itemset U , **In**: Integer index)

```

1: for all  $i, \text{index} \leq i < |U|$  do
2:   if  $N$  is internal then
3:     if  $U[\text{index}] \in N.\text{children}$  and  $|U| - i \geq \text{max\_depth} - \text{depth}$  then
4:       INCREMENT-SUPPORT( $N.\text{children}[U[\text{index}]]$ ,  $U$ ,  $\text{index} + 1$ )
5:     else
6:       return
7:     end if
8:     if  $N$  is leaf then
9:        $N.\text{support} \leftarrow N.\text{support} + 1$ 
10:    end if
11:   end if
12: end for
```

Since the number of subsets of size $k - 1$ of some itemset of size k is k , this algorithm needs at most $O(k^2)$ searches in the hash trie.

B.2 The FP-Growth algorithm

The FP-Growth algorithm [18] is a DFS algorithm that does not create candidates and thus does not count support for each candidate. It rather creates a frequent pattern tree (or FP-Tree in short) that represents the whole database. This algorithm needs only two scans of the database, first to compute frequent items and second to create an FP-Tree.

Definition B.2 (FP-Tree). *An FP-tree is a prefix trie that has associated the tuple (item, support, up-link, link, children) with each node. The support field is the support of the prefix of the item field. The up-link field is the link to the node at the previous level. Nodes with a particular item form a list linked by the link field. An FP-Tree also contains a header table in which pairs (item, head) are stored. This table contains heads of all linked lists.*

In the *FP-tree*, we store $U = (u_1, \dots, u_n)$ with $u_i \in \mathcal{B}$ and u_i is a frequent item. The items are sorted according to the support in descending order, i.e. $\text{Supp}(\{u_1\}) \leq \dots \leq \text{Supp}(\{u_n\})$.

Rationale: the following considerations explain briefly some details of the FP-Tree:

1. Only frequent items play a role in the mining process. Thus, we use only the frequent items for an FP-Tree construction.
2. Because we sort the items in each itemset stored in the tree, we maximize the sharing of the prefix and therefore reduce size of the tree and speed-up mining process. However, some recent publications states that the tree can be quite large.
3. The FP-Tree construction is the same as that of the prefix trie (used in the Apriori algorithm) with one exception: we have to update the tail of the linked list of item b when we add a new node with item b .
4. During the mining process, we need to find all nodes with a particular item. Thus each tree has a header table that has the form $(item, head)$ and each node has a link to another node, last node has *null* pointer as the link value.
5. The tree should be representation of the whole database.

An FP-tree construction consists of two phases. First, all frequent itemsets of size 1 are derived from the database (the first database scan). Second, all transactions with deleted infrequent items and items sorted by support (in descending order) are inserted into the FP-Tree (the second database scan). This leads to the following algorithm:

Algorithm 31 Function CONSTRUCT-FP-TREE

CONSTRUCT-FP-TREE(Database \mathcal{D} , Items \mathcal{B} , Integer $min_support$)

- 1: Count support for each $b_i \in \mathcal{B}$
 - 2: create empty tree T
 - 3: **for all** transaction $t \in \mathcal{D}$ **do**
 - 4: delete all $b_i \in t, Supp(\{b_i\}, \mathcal{D}) < min_support$
 - 5: sort items in each transaction by support in descending order
 - 6: insert the transaction t to the tree T and update header links
 - 7: **end for**
 - 8: **return** Constructed tree T
-

The insert procedure on line 6 works as the insert procedure for the prefix trie structure. The construction process implies the following properties of an FP-Tree:

Proposition B.3. *The FP-tree has the following properties:*

1. *An FP-Tree contains the complete information as the database from which it was constructed with the given min_support with respect to the data mining process.*
2. *An FP-Tree size is bounded by occurrences of all frequent itemsets in database, the height of an FP-Tree is bounded by size of the longest itemset in the database.*
3. *All frequent itemsets containing item b can be obtained by following an FP-Tree header links.*

To explain the FPGrowth algorithm, we need the following concepts:

Definition B.4 (Conditional pattern base of an item). *Let $b \in \mathcal{B}$ be an item and T an FP-Tree. Let N be the set of nodes reachable from the links of the header list of T for item b . The conditional pattern base of the item b is the set of all prefixes of the nodes N (i.e. the set of all prefixes of b in T). Each prefix of a node $n \in N$ is assigned the support of the node n .*

Definition B.5 (Conditional FP-Tree of an item). *Conditional FP-Tree of an item b 's is an FP-Tree that is constructed from b conditional pattern base.*

Definition B.6 (Conditional FP-Tree of an itemset). *Let $U, V \subseteq \mathcal{B}$ such that $V = U \setminus \{b\}$ and T be an FP-tree. Conditional FP-Tree of U is an FP-Tree T' that is constructed as follows:*

- (i) *Construct b conditional FP-Tree T_b from T .*
- (ii) *Repeat step (i) recursively on T_b for each itemset $b' \in V$.*

For deriving FIs, we use the property 3 of the Lemma B.3. First, we choose an item and create a conditional pattern base of this item. From the conditional pattern base we create conditional FP-Tree and output all frequent itemsets. This process is recursively repeated. An example FP-Tree is on Figure B.3.

The path from a root of an FP-Tree will be denoted as $(b_1 : s_1, b_2 : s_2, \dots, b_n : s_n)$, where b_j is an item at depth j and s_j is the support of the itemset (b_1, \dots, b_j) . We examine data mining process by example, beginning from item 6. First, we collect all frequent itemsets containing item 6 and derive frequent itemset (6) with support 2. And because there are

TID	Transaction
1	{1, 3, 4}
2	{5, 4, 6}
3	{1, 3, 5, 6}
4	{1, 3, 2}

Item	Support
1	3
2	3
3	2
4	2
5	1
6	1

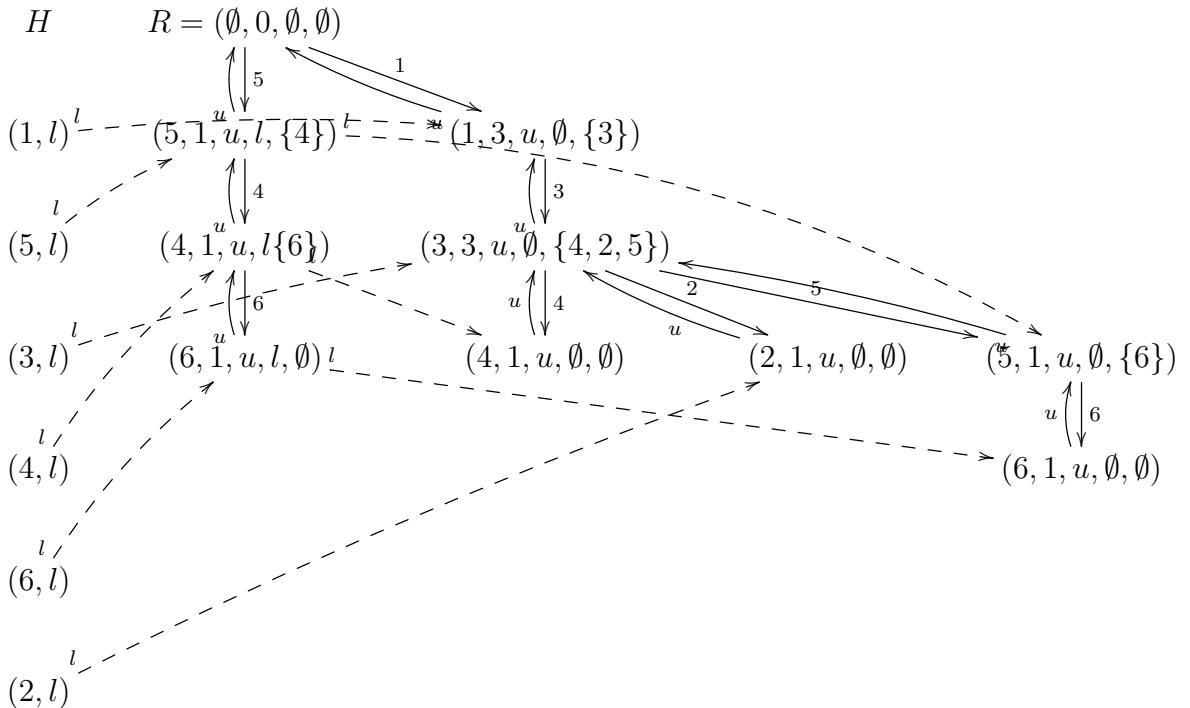
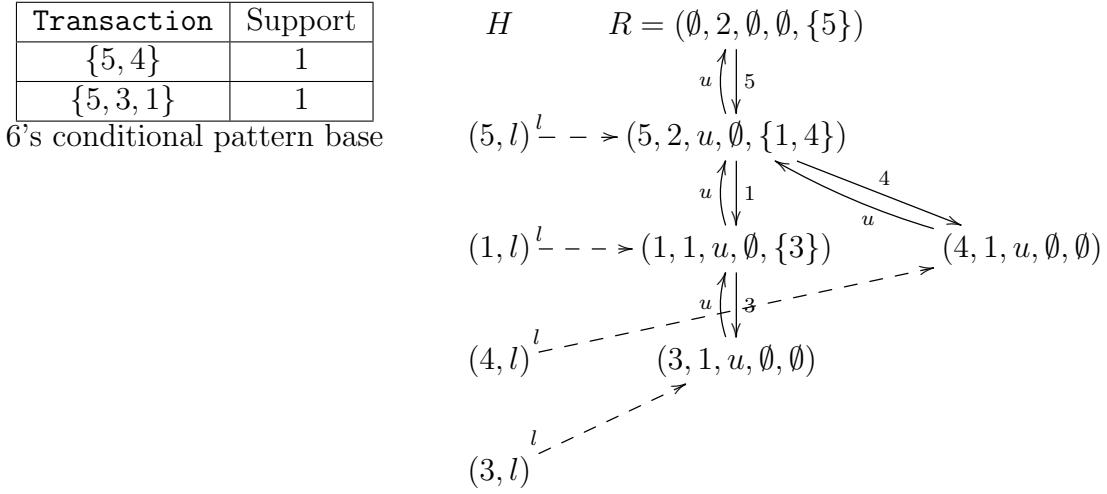


Figure B.3: An example of an FP-Tree. A link field is represented by edge with the label l and up-link fields by edge with the label u . A node contains (item, support, up-link, link, map). The map is represented by edge with a number as a label.

two paths $(1 : 3, 3 : 3, 5 : 1, 6 : 1)$ and $(5 : 1, 4 : 1, 6 : 1)$, we have 6's conditional pattern base $\{(1 : 1, 3 : 1, 5 : 1), (5 : 1, 4 : 1)\}$. Each itemset from 6's conditional pattern base occurs once in the database together with item 6. Construction of FP-Tree on this pattern base create 6's conditional FP-Tree (see Figure B.4). Continuing in the FI mining process only item 5 is frequent and it lead us to derive itemset $(5, 6)$ with support 2.

For item 5 the process is similar. One path is found $(1 : 3, 3 : 3, 5 : 2)$, thus we have the conditional pattern base $\{(1 : 2, 3 : 2)\}$ and we derive frequent itemset (5) with support 2. Creating conditional FP-Tree on this itemset creates FP-Tree with one leaf. 3's conditional pattern base is $\{(1 : 2)\}$ and $(3, 6)$ with support 2 is derived. 1's conditional pattern base

Example B.4: 6's conditional tree constructed from the tree on Figure B.3



is empty and (1, 3, 6) with support 2 is derived. Looking back on $\{(1 : 2, 3 : 2)\}$, and creating 1's conditional pattern base (which is empty) lead us to derive itemset (1, 6). This process leads to following observation: when the FP-Tree consists of single path then all combinations of items in this paths derives frequent itemset.

The pseudocode for the FPGrowth algorithm follows:

Algorithm 32 The FPGROWTH algorithm

FPGROWTH(**In:** Database \mathcal{D} , **In:** Integer $min_support$, **Out:** Set \mathcal{F})

- 1: Compute all frequent items and store them into \mathcal{B}
 - 2: $T \leftarrow \text{CONSTRUCT-FP-TREE}(\mathcal{D}, \mathcal{B}, min_support)$
 - 3: FPGROWTH-COMPUTATION($T, \emptyset, min_support$)
-

Algorithm 33 The FPGROWTH-COMPUTATION algorithm

FPGROWTH-COMPUTATION(**In:** FP-Tree T , **In:** Itemset U , **In:** Integer $min_support$, **Out:** \mathcal{F})

- 1: **if** T contains only single path P **then**
 - 2: **for all** combination W of nodes in the path P **do**
 - 3: $s \leftarrow min\{s : b \in W \wedge s = b.\text{support}\}$
 - 4: **if** $s \geq min_support$ **then**
 - 5: $\mathcal{F} \leftarrow \mathcal{F} \cup \{W \cup U\}$
 - 6: **end if**
 - 7: **end for**
 - 8: **else**
 - 9: **for all** items b in the header of T **do**
 - 10: **if** $Supp(\{b\}) \geq min_support$ **then**
 - 11: $\mathcal{F} \leftarrow \{\{b\} \cup U\}$
 - 12: **end if**
 - 13: Construct b 's conditional **FP-Tree** T_b from T , i.e., creating tree representing $U \cup \{b\}$
 - 14: **if** size of the tree $T_b \neq 0$ **then**
 - 15: FPGROWTH-COMPUTATION($T_b, U \cup \{b\}$)
 - 16: **end if**
 - 17: **end for**
 - 18: **end if**
-

B.3 The Eclat Algorithm

Papers [37, 36] use different approach than the Apriori algorithm. Eclat (which stands for **E**quivalence **C**lass **T**ransformation) uses lattice-based approach that utilizes vertical representation of a database. The Eclat algorithm is a DFS or BFS algorithm. Whereas all of the above algorithms use several scans of a database, this approach scans the database only once.

B.3.1 Support counting

Let $\mathcal{L} = (\mathcal{P}(\mathcal{B}); \subseteq)$ be a lattice, $b_i \in \mathcal{A}(\mathcal{L})$ be an atom and $\mathcal{T}(\{b_i\})$ be the tidlist of the atom b_i . Thus, the support of b_i can be computed as $|\mathcal{T}(\{b_i\})|$. We can get set of transaction ids containing itemset $\{b_i, b_j\}$, $i \neq j$, as $\mathcal{T}(\{b_i\}) \cap \mathcal{T}(\{b_j\})$ and $Supp(\{b_i, b_j\}) = |\mathcal{T}(\{b_i\}) \cap \mathcal{T}(\{b_j\})|$. In general, the support of a set $S \subseteq \mathcal{A}(\mathcal{L})$ can be computed as $|\bigcap_{b_i \in S} \mathcal{T}(\{b_i\})|$, see Section 2.4. In particular, we can use only two subsets of V to compute $Supp(V)$, because to create V we need two $U_1, U_2 \subseteq V, U_1 \cup U_2 = V$, i.e. $\mathcal{T}(V) = \mathcal{T}(U_1) \cap \mathcal{T}(U_2)$.

B.3.2 The depth-first search Eclat algorithm

In Section 2.4, we have discussed the PBECs and the hierarchy of PBECs. The hierarchy of PBECs forms a tree that can be used in a DFS algorithm. The Eclat algorithm is an algorithm that searches the tree of PBECs in a DFS fashion. This strategy utilizes the lattice decomposition of frequent itemsets induced into smaller classes. To compute the support of any itemset, we simply intersect list of transaction id's of any of its two subsets in lexicographic or reverse lexicographic order.

The depth-first search tree of the join semi-lattice of all FIs is depicted in Figure B.5. The algorithm proceeds recursively. Example of the tidlist constructed by the algorithm are in Example B.6. The algorithm ECLAT-DFS is summarized in Algorithm 34. The algorithm is called by $ECLAT\text{-}DFS(\mathcal{D}, min_support, \mathcal{F})$ and the output stored in \mathcal{F} .

Algorithm 34 The ECLAT-DFS algorithm

ECLAT-DFS(**In:** Database \mathcal{D} , **In:** Support $min_support$, **Out:** Set \mathcal{F})

- 1: Create vertical representation T of the database \mathcal{D}
 - 2: $\mathcal{A} \leftarrow$ all frequent items from \mathcal{D}
 - 3: ECLAT-DFS-COMPUTATION($\mathcal{A}, T, \emptyset, min_support, \mathcal{F}$)
-

Algorithm 35 The ECLAT-DFS-COMPUTATION algorithm

ECLAT-DFS-COMPUTATION(In: Atoms \mathcal{A} ,
In: Tidlists T ,
In: Itemset P ,
In: Support $min_support$,
Out: Set \mathcal{F})

Note: The tidlists of itemsets U , $\mathcal{T}(U)$, used in this algorithm are taken from T .

```

1: for all atom  $a_i \in \mathcal{A}$  do
2:    $\mathcal{A}_i \leftarrow \emptyset$ 
3:   for all atom  $a_j \in \mathcal{A}, a_i < a_j$  do
4:     if  $|\mathcal{T}(P \cup \{a_j\})| \geq min\_support$  then
5:        $\mathcal{A}_i \leftarrow \mathcal{A}_i \cup \{a_j\}$ 
6:        $f \leftarrow P \cup \{a_j\}$ 
7:        $\mathcal{F} \leftarrow \mathcal{F} \cup \{f\}$ 
8:     end if
9:   end for
10:  ECLAT-DFS-COMPUTATION( $\mathcal{A}_i, P \cup \{a_i\}, \mathcal{F}$ )
11: end for
```

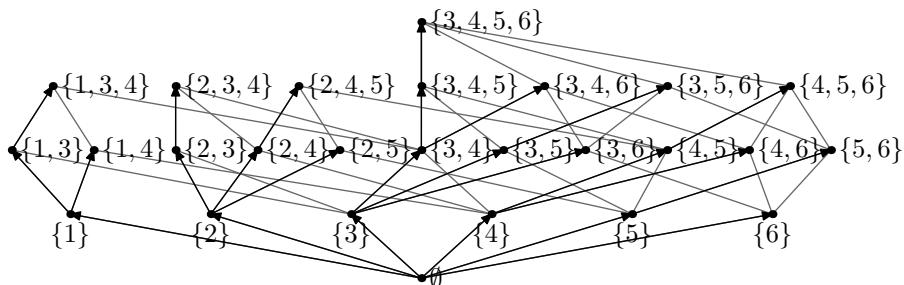


Figure B.5: The DFS tree of the execution of the Eclat algorithm using the order $1 < 2 < 3 < 4 < 5$ of the baseset $B = \{1, 2, 3, 4, 5\}$.

Example B.6: Bottom-up search strategy ($\text{min_support} = 2$)

Horizontal representation of \mathcal{D}		Vertical representation of \mathcal{D}				
TID	Transaction					
1	{1, 2, 3, 4}					
2	{3, 5}					
3	{1, 3, 4}					
4	{1, 2}					
5	{1, 3, 4, 5}					
6	{1, 2, 3, 4, 5}					
Frequent	×	×	×	×	×	×
itemset, $U =$	{1, 2}	{1, 3}	{1, 4}	{1, 5}	{2, 3}	{2, 4}
tidlist, $\mathcal{T}(U) =$	1 4 6	1 3 5 6	1 3 5 6	5 6	1 6	1 3 5 6
Frequent	×	×		×	×	×
itemset, $U =$	{1, 2, 3}	{1, 2, 4}	{1, 2, 5}	{1, 3, 4}	{1, 3, 5}	{1, 4, 5}
tidlist	1 6	1 6	6	1 3 5 6	5 6	5 6
Frequent	×		×			
itemset, $U =$	{2, 3, 4}	{2, 4, 5}	{3, 4, 5}			
tidlist,	1 6	6	5 6			
Frequent itemset	×	×				
itemset, $U =$	{1, 2, 3, 4}	{1, 3, 4, 5}				
tidlist,	1 6	5 6				

B.4 Possible optimizations of the DFS sequential algorithms

B.4.1 The “closed itemsets” optimalization

The concept of *closed itemsets*, see Definition 2.9 can be used for optimization of the DFS algorithms.

Let a DFS algorithm process prefix U and the possible branches (extensions) of U are denoted by Σ . The algorithm can extend U by all items $\Sigma' = \{b_i | b_i \in \Sigma, \text{Supp}(b_i) = \text{Supp}(U)\}$ without computation of the intermediate tidlists $\mathcal{T}(V \cup U), V \subseteq \Sigma'$, i.e., saving $O(2^{|V|})$ of intersections of tidlists.

B.4.2 Ordering of items in DFS algorithms

Consider a baseset \mathcal{B} and a database \mathcal{D} . Any DFS algorithm should expand every prefix U using the extensions Σ sorted by the support in ascending order. This allows for efficient computation of intermediate steps.

At a particular step of a sequential FIM algorithm, the prefix $\Pi = \{\pi_1, \dots, \pi_k\}, \pi_i \in \mathcal{B}$, and extensions $\Sigma = \{\sigma_1, \dots, \sigma_l\}, \sigma_i \in B$, and $\text{Supp}^*(\sigma_1) \leq \text{Supp}^*(\sigma_2) \leq \dots \leq \text{Supp}^*(\sigma_l)$. The algorithm can choose from many possible orders. Let choose two possible orders of σ_i for processing: 1) $\sigma_1, \dots, \sigma_l$ (smallest first); 2) $\sigma_l, \dots, \sigma_1$ (largest first).

1. A DFS algorithm processes every prefix Π in the following way: extend the prefix $\Pi \cup \{\sigma_1\}$ and consider $\sigma_2, \sigma_3, \dots, \sigma_l$ as extensions (in that order). Using this order it follows that the smallest partition of the database gets the largest partition of the search space.
2. A DFS algorithm processes every prefix in the following way: extend the prefix $\Pi \cup \{\sigma_l\}$ and consider $\sigma_l, \sigma_{l-1}, \dots, \sigma_2$ as extensions (in that order). Using this order it follows that the largest partition of the database gets the largest partition of the search space.

If we compare these two approaches, it is clear that the second case should be much slower than the first case. The reason is that it is more time-consuming to process an item that has large support than an item that has a small support. Other cases are somewhere in-between of these two cases. The optimal solution is to compute the support of each

extension $\Sigma = \{\sigma_1, \dots, \sigma_n\}$ for every prefix U and reorder the items, i.e. choose the order $\sigma_1 \leq \sigma_2 \leq \dots \leq \sigma_n$ such that $Supp(U \cup \{\sigma_1\}) \leq Supp(U \cup \{\sigma_2\}) \leq \dots \leq Supp(U \cup \{\sigma_n\})$. Such *dynamic* ordering of items is essential for the speed of a DFS FIM algorithm.

B.4.3 The concept of diffsets

If we start to mine large database with very large lists of transaction ids for an item (or atoms), the intersection time becomes too large. Furthermore, the size of a list of transaction ids of a frequent itemset also become very large and these lists of transaction id's cannot fit into the main memory. These problems are solved with so called *difference sets* (or *diffsets* in short) [35]. A diffset is the difference of two list of transaction ids.

Definition B.7 (Difference set). *Let $U \subset \mathcal{B}$ be an itemset and $b_i \in \mathcal{B} - U$ an item. $\mathcal{T}(U)$ denotes a set of transaction id's. Difference set (or diffset in short) is $\mathcal{D}(U \cup \{b_i\}) = \mathcal{T}(U) - \mathcal{T}(\{b_i\})$*

First we have to note that the size of a diffset of an itemset $U \cup \{b_i\}$ is no longer the support of the itemset. However, the support of $U \cup \{b_i\}$ can be computed as follows:

$$Supp(U \cup \{b_i\}) = Supp(U) - |\mathcal{D}(U \cup \{b_i\})|$$

Now let $U \subset \mathcal{B}$ be an itemset and $b_i, b_j \in \mathcal{B}, b_i < b_j$ and $b_i, b_j \notin U$ be two items. We use instead of transaction list $\mathcal{T}(U \cup \{b_i\})$ ($\mathcal{T}(U \cup \{b_j\})$), diffsets $\mathcal{D}(U \cup \{b_i\})$ ($\mathcal{D}(U \cup \{b_j\})$), respectively. We want to compute support of $U \cup \{b_i\} \cup \{b_j\}$ using only diffsets. From Definition B.7, we have $Supp(U \cup \{b_i\} \cup \{b_j\}) = Supp(U \cup \{b_i\}) - |\mathcal{D}(U \cup \{b_i\} \cup \{b_j\})|$. But we have only diffsets and not list of transaction id's. But it is easy to fix:

$$\begin{aligned} \mathcal{D}(U \cup \{b_i\} \cup \{b_j\}) &= \mathcal{T}(U \cup \{b_i\}) - \mathcal{T}(U \cup \{b_j\}) \\ &= \mathcal{T}(U \cup \{b_i\}) - \mathcal{T}(U \cup \{b_j\}) + \mathcal{T}(U) - \mathcal{T}(U) \\ &= (\mathcal{T}(U) - \mathcal{T}(U \cup \{b_j\})) - (\mathcal{T}(U) - \mathcal{T}(U \cup \{b_i\})) \\ &= \mathcal{D}(U \cup \{b_j\}) - \mathcal{D}(U \cup \{b_i\}) \end{aligned}$$

The concept of diffsets is used in the Eclat algorithm. Generally it is possible to use the diffsets in an arbitrary algorithm that uses the *vertical representation* of the database.

B.5 Discovering rules

To complete the overview of the sequential algorithms, we show how to create association rules from the FIs.

When we have discovered all frequent itemsets, we have to create all rules $X \Rightarrow Y$ with given confidence. Generation of all such rules is based on the following observation:

If we have frequent itemset L and its subset A , and $\text{Conf}(A, (L - A)) > \text{min_confidence}$, then if $A \Rightarrow (L - A)$ does not have enough confidence, then for all subsets $a \subseteq A$ the rule $a \Rightarrow (L - a)$ does not have enough confidence. For example, if the association rule $123 \Rightarrow 4$ does not have enough confidence, we need not check whether $12 \Rightarrow 34$ holds. Following algorithm uses this observation:

Algorithm 36 The GENERATE-ALL-RULES algorithm

GENERATE-ALL-RULES(In:** Set \mathcal{F})**

- 1: **for all** frequent itemset $U, |U| \geq 2$ **do**
 - 2: GENERATE-RULES(U, U)
 - 3: **end for**
-

Algorithm 37 The GENERATE-RULES algorithm

GENERATE-RULES(In:** Itemset U , **In:** Itemset W)**

Require: $|U| = k, |W| = m$

- 1: $A = \{V : V \subset W \wedge |V| = m - 1\}$
 - 2: **for all** $V \in A$ **do**
 - 3: compute confidence $\text{Conf}(V, U), c \leftarrow \text{Supp}(U)/\text{Supp}(V)$
 - 4: **if** $c \geq \text{min_confidence}$ **then**
 - 5: output $(V \Rightarrow U \setminus V)$, with confidence c and support $\text{Supp}(U)$
 - 6: **if** $|W| - 1 > 1$ **then**
 - 7: Generate-Rules(U, W)
 - 8: **end if**
 - 9: **end if**
 - 10: **end for**
-

C Lists of abbreviations

BFS	Breadth-First Search
cc-NUMA	cache-coherent Non-Uniform Memory Access
CI	closed itemsets
DM	Distributed Memory
DFS	Depth-First Search
diffset	difference set
Eclat	Equivalence class transformation
FI	Frequent itemset
FIMI	Frequent Itemset MIning
FP-Growth	Frequent Pattern Growth
FP-Tree	Frequent Pattern Tree
FPM	Fast parallel mining
FPGrowth	Frequent Pattern Growth
LPT	least processing time
MFI	Maximal frequent itemset
MLFPT	Multiple Local Frequent Pattern Tree
NUMA	Non-Uniform Memory Access
PBEC	Prefix-based equivalence class
TID	transaction id
tidlist	Transaction Id list
trie	a prefix trie, from the word retrieval
QKP	quadratic knapsack

D Used symbols

\mathcal{B}	Base itemset $\mathcal{B} = \{b_1, b_2, \dots, b_{ \mathcal{B} }\}, b_1 < b_2 < \dots < b_{ \mathcal{B} }$
$\mathcal{P}(S)$	The powerset of the set S , i.e., the set $\{s s \subseteq S\}$
U, V, W	Itemsets or sets
$U \Rightarrow V$	Association rule
\mathcal{D}	A database
D_i	A database partition, usually $D_i \cap D_j = \emptyset, i \neq j$ and $\mathcal{D} = \bigcup_i D_i$
$t = (U, id)$	A transaction from the database with unique identifier id
$\mathcal{T}(U)$	The transaction id list (or tidlist in short) of the transactions containing the itemset U as a subset
$\tilde{\mathcal{D}}$	A sample of the database \mathcal{D}
P	The number of processors
p_i	i -th processor, $1 \leq i \leq P$
$Supp(U, \mathcal{D})$	The <i>support</i> of an itemset U in database \mathcal{D} (or $Supp(U)$ if \mathcal{D} is clear from context)
$Supp^*(U, \mathcal{D})$	The <i>relative support</i> of an itemset, i.e., $Supp^*(U, \mathcal{D}) = Supp(U, \mathcal{D})/ \mathcal{D} $
$min_support, min_support^*$	The absolute minimal support and the relative minimal support
$Conf(U, W, \mathcal{D})$	Confidence of association rule $U \Rightarrow W$ (or $Conf(U, W)$ if \mathcal{D} is clear from context)
$min_confidence$	Minimal confidence
$Cover_{\mathcal{B}}(U, \mathcal{D})$	Cover of the itemset U in database \mathcal{D} , i.e., the set of transaction containing U as a subset
$Cover_{\mathcal{T}}(T, \mathcal{D})$	Cover of the transactions $T \subseteq \mathcal{D}$, i.e., an itemsets containing all items that are contained in <i>all</i> transactions T
\mathcal{F}	The set of all frequent itemsets
\mathcal{M}	The set of all maximal frequent itemsets
$\widetilde{\mathcal{M}}$	The approximation of the MFIs in the database \mathcal{D} , computed using $\tilde{\mathcal{D}}$
$F_i, 1 \leq i \leq P$	disjoint partitions of all FIs $\mathcal{F} = \bigcup_{i=1}^P F_i$
$F_k, k > 0$	The set of frequent itemsets of size k
$\widetilde{\mathcal{F}}$	The approximation of the FIs in the database \mathcal{D} , i.e., $\widetilde{\mathcal{F}} = \{U \exists W \in \widetilde{\mathcal{M}}, U \subseteq W\}$

$\tilde{\mathcal{F}}_s$	A sample of $\tilde{\mathcal{F}}$, i.e., $\tilde{\mathcal{F}}_s \subseteq \tilde{\mathcal{F}}$
C_k	Set of candidate itemsets on FIs of size k
\top	The top element of a lattice
\perp	The bottom element of a lattice
$\mathcal{A}(\mathcal{L})$	The set of all atoms of a lattice \mathcal{L}
$\mathcal{T}(U)$	List of transaction ids of an itemset U
$\mathcal{D}(U)$	Difference set (or diffset in short) of transaction ids of an itemset U
$[U \Sigma]$	Prefix based equivalence class with prefix U , i.e., $[U \Sigma] = \{W W = U \cup V, V \subseteq \Sigma \text{ and } \forall b_\Sigma \in \Sigma, b_U \in U : b_U < b_\Sigma\}$. Σ can be omitted if clear from context.
$\epsilon_{\tilde{\mathcal{D}}}$	Error of an approximation of the support of an itemset U in database \mathcal{D} computed from a database sample $\tilde{\mathcal{D}}$
$\delta_{\tilde{\mathcal{D}}}$	Probability of the error $\epsilon_{\tilde{\mathcal{D}}}$
$\epsilon_{\tilde{\mathcal{F}}_s}$	Error of an approximation of the relative size of a set $F \subseteq \tilde{\mathcal{F}}$, i.e., an error of the size $ F /\tilde{\mathcal{F}}$
$\delta_{\tilde{\mathcal{F}}_s}$	Probability of the error $\epsilon_{\tilde{\mathcal{F}}_s}$