Static Load Balancing of Parallel Mining of Frequent Itemsets Using Reservoir Sampling

Kessl, Robert
2010

Static Load Balancing of Parallel Mining of Frequent Itemsets Using Reservoir Sampling

Post-Graduate Student:
ING. ROBERT KESSL

Institute of Computer Science of the ASCR, v. v. i.
Pod Vodárenskou věží 2
182 07 Prague 8, CZ [1]

kesslr@cs.cas.cz

Supervisor:
PROF. ING. PAVEL TVRDÍK, CSC.

Department of Computer Systems
CTU FIT
Kolejní 550/2
160 00 Prague 6, CZ

tvrdik@fit.cvut.cz

Field of Study:
Computer Science

Abstract

One of the important data mining tasks is the search for co-occurences among the items in the databases, the so called frequent itemset mining. Let us consider a retail store. We track the contents of baskets of customers. The content of the baskets is stored in the database as transactions. In this database, we search for sets of items (itemsets in short) that occurs in at least $min\_support$ transactions.

The automated collection of data in companies allows to store huge amount of data. This creates the need for parallel mining of frequent itemsets. In this paper, we present a new method for parallel mining of frequent itemsets based on the reservoir sampling that statically load-balance the workload.

1. Introduction

The automated data collection causes extreme growth of the database sizes. Processing of databases of such sizes is almost impossible with a single processor. Therefore, new parallel algorithms that are able to process such amount of data are needed.

Today, large shared-memory machines parallel are still quite expensive. Distributed-memory multiprocessors can easily be built from cheap computers connected with a special network. Therefore, we consider designing algorithms for distributed-memory parallel machines.

One of the important data mining tasks is the search for co-occurences among the data. Let us consider a retail store. We track the contents of baskets of customers. The content of the baskets is stored in the database as transactions. In this database, we search for sets of items (itemsets in short) that occurs in at least $min\_support$ transactions. These itemsets are so called *frequent itemsets* (or FIs in short). From FIs, we create rules of type $X \Rightarrow Y$, where X, Y are two FIs. For example $\{$butter, bread$\} \Rightarrow \{$milk$\}$. The search of these co-occurences is divided into two parts: 1) find all *frequent itemsets*; 2) create *association rules* from the FIs.

The task of mining of FIs is computationally and memory demanding. It seems that the finding of all FIs is the most time-consuming part of the whole process. With the growth of retail-store databases it is important to design parallel algorithms for mining of FIs.

In [1] and [2] we have proposed new parallel methods for mining of FIs. We denote the set of all FIs by $\mathcal{F}$. The basic idea of the algorithms is to create a sample $\widetilde{\mathcal{F}}_s$ that is used to create disjoint partitions $F_i, F_j \subseteq \mathcal{F}$ such that $|F_i|/|\mathcal{F}| \approx 1/P$, where P is the number of processors. The relative size $|F_i|/|\mathcal{F}|$ is estimated using the sample $\widetilde{\mathcal{F}}_s$. The partitions F_i are then independently processed by each processor. In [1, 2], we have proposed a method of creation of $\widetilde{\mathcal{F}}_s$, based on a *modified* coverage algorithm. The problem with these two methods is that the sample $\widetilde{\mathcal{F}}_s$ is *non-uniform* and therefore, we do not have any guarantees on the load-balance. Additionally, the two methods needs an algorithm for mining of maximal frequent itemsets (MFIs in short). The MFIs are hold in main memory and used in the modified coverage algorithm. Since the number of MFIs can be quite large, the methods are in some cases very memory consuming.

In this paper, we show how to create a *uniform* sample $\widetilde{\mathcal{F}}_s$ using a different, faster and less memory consuming, method. The new method does not need an algorithm

[1] The author is also assistant professor at CTU FIT, Department of Theoretical Informatics and doctoral student of CTU FEE, Department of Computers.

for mining of MFIs and therefore is less memory consuming. Our new method needs only the algorithm for mining of FIs.

2. Notation

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 *itemsets*. Further, we need to view the baseset $\mathcal{B}$ as an ordered set. The items are therefore ordered using an arbitrary order $<$: $b_1 < b_2 < \ldots < b_n, n = |\mathcal{B}|$. Hence, we can view an itemset $U = \{b_{u_1}, b_{u_2}, \ldots, b_{u_{|U|}}\}$, $b_{u_1} < b_{u_2} < \ldots b_{u_{|U|}}$, as an ordered set denoted by $U = (b_{u_1}, b_{u_2}, \ldots, b_{u_{|U|}})$.

Let $U \subseteq \mathcal{B}$ be an itemset and `id` a unique identifier. We call the pair (id, U) a *transaction*. The `id` is called the *transaction id*. A database $\mathcal{D}$ is a set of transactions. In our algorithms, we need to sample the database $\mathcal{D}$. A *database sample* is denoted by $\widetilde{\mathcal{D}}$. We define the support as the number of transactions containing U, but in some literature, the relative support is defined by $Supp^*(U) = Supp(U)/|\mathcal{D}|$. We call U *frequent* in database $\mathcal{D}$ if $Supp(U, \mathcal{D}) \geq min\_support$. We can also define the frequent itemset using the relative support, denoted by $min\_support^*$, $0 \leq min\_support^* \leq 1$, i.e., an itemset is frequent iff $Supp^*(U, \mathcal{D}) \geq min\_support^*$.

We denote the set of all frequent itemsets computed from $\mathcal{D}$ by $\mathcal{F}$. The set of all frequent itemsets computed from $\widetilde{\mathcal{D}}$ is denoted by $\widetilde{\mathcal{F}}$. In our algorithms, we need to sample the set $\mathcal{F}$. A sample of frequent itemsets is denoted by $\widetilde{\mathcal{F}}_s$. In our case, the set $\widetilde{\mathcal{F}}_s$ is a sample of $\widetilde{\mathcal{F}}$, i.e., $\widetilde{\mathcal{F}}_s \subseteq \widetilde{\mathcal{F}}$.

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 3 (Monotonicity of support) *Let $U, V \subseteq \mathcal{B}$ be two itemsets such that $U \subsetneq V$ and $\mathcal{D}$ be a database. Then for the supports of U and V we have $Supp(U, \mathcal{D}) \geq Supp(V, \mathcal{D})$.*

The multivariate hypergeometric distribution describes the following problem: 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 ith 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 ith color. Then the probability mass function is:

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

We denote the number of processors by P and processor i by p_i.

3. The lattice of all itemsets

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

To decompose the $\mathcal{P}(\mathcal{B})$ into the prefix-based equivalence classes, 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 4 (prefix-based equivalence class) *Let $U \subseteq \mathcal{B}, |U| = n$ be an itemset. We impose some order on the set $\mathcal{B}$ and hence view $U = (u_1, u_2, \ldots, u_n), u_i \in \mathcal{B}$ as an ordered set. A* prefix-based equivalence class *(PBEC in short) of U, denoted by $[U]_l$, is a set of all itemsets that have the same prefix of length l, i.e., $[U]_l = \{W = (w_1, w_2, \ldots, w_m) | u_i = w_i, i \leq l, m = |W| \geq |U|, U, W \subseteq \mathcal{B}\}$*

To simplify the notation, we use $[W]$ for the PBEC $[W]_l$ iff $l = |W|$. Each $[W], W \subseteq \mathcal{B}$ is a meet sublattice of $(\mathcal{P}(\mathcal{B}), \subseteq)$. Additionally, we use the term prefix for both: (a) *ordered set*; (b) *unordered set*; if clear from context, e.g., let $\mathcal{B} = \{1, 2, 3, 4, 5\}$ with the order $1 < 2 < 3 < 4 < 5$ and $U = \{3, 1, 2\}$ then the term prefix means $U = (1, 2, 3)$. The *extensions* of the PBEC $[U], U \subseteq \mathcal{B}$ is an ordered set $\Sigma \subseteq \mathcal{B}$ such that $U \cap \Sigma = \emptyset$ and for each $W \in [U], W \backslash U \subseteq \Sigma$. We denote the PBEC together with the extensions Σ by $[U|\Sigma]$.

Let $\mathcal{B} = \{b_1, \ldots, b_n\}, b_1 < \ldots < b_n$. Let $U_i = \{b_i\}$ and $\Sigma_i = \{b_j | b_i < b_j\}$ then $[U_i|\Sigma_i]$ forms *disjoint* PBECs. Each PBEC $[U_i|\Sigma_i]$ can be recursively divided into disjoint PBECs in the following way: let $W_k = U_i \cup \{b_k\}, b_k \in \Sigma_i$ and $\Sigma'_k = \{b \in \Sigma_i | b_k < b\}$ then $[W_k|\Sigma'_k]$ forms disjoint PBECS. We omit the extensions from the notation if clear from context.

Further, we need to partition $\mathcal{F}$ into n disjoint sets, denoted $(F_1, \ldots, F_n)$, satisfying $F_i \cap F_j = \emptyset, i \neq j$, and $\bigcup_i F_i = \mathcal{F}$. This partitioning can be done using the PBECs. The PBECs can be collated into n partitions as follows: let have *disjoint* PBECs $[U_l]$, such that $\bigcup_l [U_l] = \mathcal{F}, 1 \leq l \leq m$ and sets of indexes of the PBECs $L_i \subseteq \{l | 1 \leq l \leq m\}, 1 \leq i \leq n$

such that $L_i \cap L_j = \emptyset$ and $\sum_i |L_i| = m$ then $F_i = \bigcup_{l \in L_i}([U_l] \cap \mathcal{F})$.

The basic idea of our algorithm is to create a database sample $\widetilde{\mathcal{D}}$ that allow us to estimate the support of an arbitrary itemset and using $\widetilde{\mathcal{D}}$, we create identically (but not independently distributed) sample of all FIs $\widetilde{\mathcal{F}}_s$. Let U_i be a prefix of the PBEC $[U_i|\Sigma_i]$. Using the sample, we can estimate the relative size of an arbitrary PBEC $[U_i]$ $\frac{|[U_i]\cap\mathcal{F}|}{|\mathcal{F}|} \approx \frac{|[U_i]\cap\widetilde{\mathcal{F}}_s|}{|\widetilde{\mathcal{F}}_s|}$. The knowledge of the relative size of PBECs allow us to create the partitions $F_i, 1 \leq i \leq P$ such that $|F_i|/|\mathcal{F}| \approx 1/P$.

4. 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 $\widetilde{\mathcal{D}}$, i.e., we can use $Supp^*(U, \widetilde{\mathcal{D}})$ instead of $Supp^*(U, \mathcal{D})$ – this significantly reduces the time complexity. The approach of estimating the relative support of U was described by Toivonen [3]. Further, we denote the set of all FIs computed from the database sample $\widetilde{\mathcal{D}}$ by $\widetilde{\mathcal{F}}_s$.

Toivonen uses a *database sample* $\widetilde{\mathcal{D}}$ 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 $\widetilde{\mathcal{D}}$ of $\mathcal{D}$; 2) compute all frequent itemsets in $\widetilde{\mathcal{D}}$; 3) check that all these FIs computed using $\widetilde{\mathcal{D}}$ are also FIs in $\mathcal{D}$ and correct the output. If an itemset is frequent in $\mathcal{D}$ and not in $\widetilde{\mathcal{D}}$, correct the output using $\mathcal{D}$. 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 $\widetilde{\mathcal{D}}$ by: $err_{supp}(U, \widetilde{\mathcal{D}}) = |Supp^*(U, \mathcal{D}) - Supp^*(U, \widetilde{\mathcal{D}})|$

The database sample $\widetilde{\mathcal{D}}$ 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 4 *[3] Given an itemset $U \subseteq \mathcal{B}$ and a random sample $\widetilde{\mathcal{D}}$ drawn from database $\mathcal{D}$ of size*

$$|\widetilde{\mathcal{D}}| \geq \frac{1}{2\epsilon^2_{\widetilde{\mathcal{D}}}} \ln \frac{2}{\delta_{\widetilde{\mathcal{D}}}},$$

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

Using a database sample $\widetilde{\mathcal{D}}$ with size given by the previous theorem, we can estimate $Supp^*(U, \mathcal{D})$ with error $\epsilon_{\widetilde{\mathcal{D}}}$ that occurs with probability at most $\delta_{\widetilde{\mathcal{D}}}$: It follows from Lemma 4 that if we compute the approximation $\widetilde{\mathcal{F}}$ of $\mathcal{F}$ from the database sample $\widetilde{\mathcal{D}}$ of size $|\widetilde{\mathcal{D}}| \geq \frac{1}{2\epsilon^2_{\widetilde{\mathcal{D}}}} \ln \frac{2}{\delta_{\widetilde{\mathcal{D}}}}$, we should get an estimate of the supports of itemsets $U \in \widetilde{\mathcal{F}}$, i.e., potentionally, we have a close approximation $\widetilde{\mathcal{F}}$ of $\mathcal{F}$.

5. The reservoir sampling algorithm

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

Vitter [4] formulates the problem of *reservoir sampling* as follows: given a stream of records; the task is to create 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 $\widetilde{\mathcal{F}}$, see Theorem 5. 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 [4] 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: RANDOM() which returns an uniformly distributed real number from the interval $[0, 1]$

The simplest reservoir sampling algorithm is summarized in Algorithm 2. It takes as an input an array R (reservoir/buffer) of size $n = |\widetilde{\mathcal{F}}_s|$, 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 $|\widetilde{\mathcal{F}}_s|$ FIs and stores them in memory into the buffer R.

The RESERVOIR-ALGORITHM follows:

Algorithm 2 The RESERVOIR-SAMPLING algorithm.

```
RESERVOIR-SAMPLING(In/Out: Array R of size n,
                   In: Integer n,
                   In: Function ReadNextFI,
                   In: Function SkipFIs)
 1: for j = 0 to n − 1 do
 2:    ReadNextFI(R[j])
 3: end for
 4: t = n
 5: while not eof do
 6:    t = t + 1
 7:    m = ⌊t × RANDOM()⌋ {pick uniformly a number
       from the set {1, ..., t − 1}}
 8:    if m < n then
 9:       ReadNextFI(R[m])
10:    else
11:       SkipFIs(1)
12:    end if
13: end while
```

The RESERVOIR-SAMPLING is quite slow, it is linear in the number of input records read by READNEXTFI(R), i.e., it is linear in $|\widetilde{\mathcal{F}}|$. Vitter [4] created a *faster* algorithm with the average running time $\mathcal{O}(|\widetilde{\mathcal{F}}_s|(1 + \log \frac{|\widetilde{\mathcal{F}}|}{|\widetilde{\mathcal{F}}_s|}))$, where $|\widetilde{\mathcal{F}}_s|$ is the size of the array R used by RESERVOIR-SAMPLING. The algorithm has the same parameters as the RESERVOIR-SAMPLING and we denote the Vitter's variant of the reservoir sampling algorithm by VITTER-RESERVOIR-SAMPLING.

Now, we analyse the relative size of a PBEC using the samples taken by the reservoir sampling algorithm. The reservoir sampling samples the set $\widetilde{\mathcal{F}}$ **without replacement**, resulting in $\widetilde{\mathcal{F}}_s$. The reservoir sampling process can be modeled using the *hypergeometric distribution*. 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 [5], i.e., estimation of the relative size of a PBEC but using an uniformly but not *independently* distributed sample. From these bounds follows the following theorem:

Theorem 5 (Estimation of the relative size of $F \subseteq \widetilde{\mathcal{F}}$) *Let $F \subseteq \widetilde{\mathcal{F}}$ be a set of itemsets. The relative size of F, $\frac{|F|}{|\widetilde{\mathcal{F}}|}$, is estimated with error $\epsilon_{\widetilde{\mathcal{F}}_s}$ with probability $\delta_{\widetilde{\mathcal{F}}_s}$ from a hypergeometrically distributed sample $\widetilde{\mathcal{F}}_s \subseteq \widetilde{\mathcal{F}}$ with parameters $N = |\widetilde{\mathcal{F}}|, M = |F|$ (see [5] for details) of size:*

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

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

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

Proof: Can be found in [5]. ∎

6. Summary of the previous two methods

In [1] we have proposed the PARALLEL-FIMI-SEQ method and in [2] we have proposed the PARALLEL-FIMI-PAR method. The idea of the two methods 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 whole method consists of four phases.

The four phases are designed in such a way that they statically load-balance the computation of all FIs. Phases 1–2 prepares the static load-balancing for Phase 4. In the Phase 3, we redistribute the database partitions so each processor can proceed independently with some PBECs. In the Phase 4, we execute an arbitrary algorithm for mining of FIs. To speedup Phases 1–2, we can execute each of Phase 1–2 in parallel.

The input and the parameters of the whole method are the following: 1) Minimal support $min\_support^*$; 2) The sampling parameters: real numbers $0 \leq \epsilon_{\widetilde{\mathcal{D}}}, \delta_{\widetilde{\mathcal{D}}}, \epsilon_{\widetilde{\mathcal{F}}_s}, \delta_{\widetilde{\mathcal{F}}_s} \leq 1$, see Sections 4 and 5; 3) The relative size of a PBEC: the parameter $\rho, 0 \leq \rho \leq 1$, see Sections 5; 4) Partition parameter: real number $\alpha, 0 \leq \alpha \leq 1$; 5) Database parts $D_i, 1 \leq i \leq P$. The database partitions are loaded by each processor at the beginning of the methods.

We assume that at the beginning of the computation, 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. Therefore, 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 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 $0 \leq \epsilon_{\widetilde{\mathcal{D}}}, \delta_{\widetilde{\mathcal{D}}}, \epsilon_{\widetilde{\mathcal{F}}_s}, \delta_{\widetilde{\mathcal{F}}_s} \leq 1$. Output of Phase 1 is a sample of frequent itemsets $\widetilde{\mathcal{F}}_s \subseteq \widetilde{\mathcal{F}}$. Generally, the only purpose of the first phase is to compute a sample $\widetilde{\mathcal{F}}_s$. First, each processor samples D_i (in parallel) and creates part $\mathcal{D}'_i$ and broadcasts them to other processors (all-to-all scatter[1]). Each processor p_i then creates $\widetilde{\mathcal{D}} = \bigcup_i \mathcal{D}'_i$. The the processors compute the set of all MFIs from $\widetilde{\mathcal{D}}$, denoted by $\widetilde{\mathcal{M}}$. The set $\widetilde{\mathcal{M}}$ is the upper boundary in the sense of set inclusion of the set $\widetilde{\mathcal{F}}$, i.e., for each $U \in \widetilde{\mathcal{F}}$ exists $m \in \widetilde{\mathcal{M}}$ such that $U \subseteq m$. We need to take a sample $\widetilde{\mathcal{F}}_s \subsetneq \widetilde{\mathcal{F}}$. This can be done using a *modified coverage algorithm*. The modified coverage algorithm randomly chooses $m \in \widetilde{\mathcal{M}}$ with probability $|\mathcal{P}(m)| / \sum_{m' \in \widetilde{\mathcal{M}}} |\mathcal{P}(m')|$. Then it picks uniformly a set $U \in \mathcal{P}(m)$. This approach does not generates an identically distributed sample $\widetilde{\mathcal{F}}_s$, but it gives reasonably good results and it is very quick. The difference between PARALLEL-FIMI-SEQ and PARALLEL-FIMI-PAR methods is that the second computes a set M, such that $\widetilde{\mathcal{M}} \subseteq M \subseteq \widetilde{\mathcal{F}}$, i.e., it computes a superset of $\widetilde{\mathcal{M}}$. Computation of $\widetilde{\mathcal{M}}$ in parallel makes the PARALLEL-FIMI-PAR faster then PARALLEL-FIMI-SEQ.

Phase 2 (lattice partitioning): the input of this phase is the sample $\widetilde{\mathcal{F}}_s$, the database sample $\widetilde{\mathcal{D}}$ (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 each PBEC $[U_i|\Sigma_i]$, and estimates the size of $[U_i|\Sigma] \cap \mathcal{F}$ using $\widetilde{\mathcal{F}}_s$. The PBECs $[U_i|\Sigma_i]$ are then assigned to the processors and the assignment is broadcast to the 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 $D_i, i = 1, \ldots, P$. 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 $Supp(W, \mathcal{D}) = \sum_{1 \leq i \leq P} Supp(W, D_i)$

7. Proposal of a new DM parallel method

Our new method is called *Parallel Frequent Itemset MIning – Reservoir* (Parallel-FIMI-Reservoir in short). This method works for any number of processors $P \ll |\mathcal{B}|$. The basic idea is the same as in PARALLEL-FIMI-SEQ and PARALLEL-FIMI-PAR methods. The main difference is the usage of the so called reservoir sampling algorithm instead of the modified coverage algorithm. This allow us to take an identically but *not independently* distributed sample $\widetilde{\mathcal{F}}_s$. We make the sample $\widetilde{\mathcal{F}}_s$ in parallel: in Phase 1, we execute an arbitrary algorithm for mining of FIs in parallel and the output of the FI mining algorithm is sampled using the reservoir sampling (in parallel). The input parameters are the same as in the PARALLEL-FIMI-SEQ and PARALLEL-FIMI-PAR methods

7.1. Detailed description of Phase 1

In this Section, we give a detailed description of the sampling process based on the *reservoir sampling* [4] that samples $\widetilde{\mathcal{F}}$ uniformly, i.e., it creates an identically distributed sample of $\widetilde{\mathcal{F}}$.

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 $\widetilde{\mathcal{D}}$ is distributed among the processors – each processor having a copy of the database sample $\widetilde{\mathcal{D}}$. 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 depth-first search (DFS in short) algorithm for mining of FIs, enumerating $[(b_j)] \cap \widetilde{\mathcal{F}}, b_j \in B_i$. The output, the itemsets $[(b_j)] \cap \widetilde{\mathcal{F}}$, 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. For terminating the parallel execution, we use the Dijkstra's token termination algorithm.

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

The task of the process is to take $|\widetilde{\mathcal{F}}_s| = -\frac{\log(\delta_{\widetilde{\mathcal{F}}_s}/2)}{D(\rho+\epsilon_{\widetilde{\mathcal{F}}_s}||\rho)}$ samples, see Theorem 5. Because the reservoir sampling 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 $|\widetilde{\mathcal{F}}_s|$ frequent itemsets using the reservoir sampling algorithm, producing $\widetilde{\mathcal{F}}_s$, and counts the number of FIs computed by the sequential algorithm. When the reservoir sampling finishes, processor p_i broadcasts f_i to all other processors. The processors then pick P random variables $X_i, 1 \leq i \leq P$ from multivariate hypergeometrical distribution with parameters: number of colors $C = P$, $M_i = f_i$ and choose X_i itemsets $U \in \widetilde{\mathcal{F}}_s$ at random out of the $N = |\widetilde{\mathcal{F}}_s|$ sampled frequent itemsets computed by p_i. The samples are then send to processor p_1. p_1 stores the received samples in $\widetilde{\mathcal{F}}_s$.

7.2. 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 $\widetilde{\mathcal{F}}_s$, the database $\widetilde{\mathcal{D}}$ (computed in Phase 1), the set $\mathcal{B}$, and a real number $\alpha, 0 < \alpha \leq 1$. For the purpose of this section, 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 $[U_k|\Sigma_k]$. 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 DFS sequential FI mining algorithm usually dynamically changes the order of items in $\mathcal{B}$ for each PBEC, i.e., the algorithm uses different order of items in the extensions. The PBECs are still disjoint and additionally the sequential algorithm is faster. Therefore, we need to prepare the PBECs in the same way as the sequential algorithm does. Let U be a prefix and $\Sigma = \{\epsilon_1, \ldots, \epsilon_n\} \subseteq \mathcal{B}$ the extensions. The sequential algorithm orders the items ϵ_i: $\epsilon_1 < \ldots < \epsilon_n$ such that $Supp(U \cup \{\epsilon_1\}) < \ldots < Supp(U \cup \{\epsilon_n\})$. We use the supports estimated using $\widetilde{\mathcal{D}}$ for making the order of the extensions.

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 $\frac{|[U_k] \cap \widetilde{\mathcal{F}}_s|}{|\widetilde{\mathcal{F}}_s|} \leq \alpha \cdot \frac{1}{P}$, where $0 < \alpha < 1$ is a parameter of the computation set by the user. The PBECs are created recursively, see Section 3. 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 $|F_i|/|\mathcal{F}| \approx 1/P$.

In the second step, 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. This 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 [6] for the proofs) is a best-fit algorithm, see Algorithm 3:

Algorithm 3 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_i. The indexes assigned to p_i, are stored in L_i.

Lemma 5 *[6]* LPT-SCHEDULE *is* $4/3$*-approximation algorithm.*

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

7.3. Detailed description of Phase 3

The input of Phase 3 for 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. 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. The database partition D'_i should contain all the information needed for computation of F_i. At the begginning of this phase, the processors has disjoint database partitions D_i such that $|D_i| \approx \frac{|\mathcal{D}|}{P}$. We expect that we have a distributed memory machine whose nodes are interconnected using a network such as Myrinet or Infiniband, 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 place[2]). That is: processor p_i send to processor p_j the set of transactions $\{t|t \in D_i, k \in L_j, \text{ and } U_k \subseteq t\}$, i.e., all transactions that contain at least one $U_k, k \in L_j$ as a subset: $D'_j = \bigcup_i \{t|t \in D_i, k \in L_j, \text{ and } U_k \subseteq t\} = \{t|t \in \mathcal{D}, \text{exists } k \in L_j, U_k \subseteq t\}$. The all-to-all scatter is done in $\lfloor \frac{P}{2} \rfloor$ communication rounds.

We can consider the scatter as a round-robin tournament of P players [7], which is the following procedure: if P is odd, a dummy processor can be added, whose scheduled opponent waits for the next round and the processors performs P communication rounds. For example let have 14 processors, in the first round the following processors exchange their database portions:

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:

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

7.4. 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), D'_q (computed in Phase 3), the set $\pi = \{(U_k, \Sigma_k)|U_k \subseteq \mathcal{B}, \Sigma_k \subseteq \mathcal{B}, U_k \cap \Sigma_k = \emptyset\}$ 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.

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 $(U_k, \Sigma_k) \in \pi, k \in L_q$ assigned to the processor, i.e., p_q enumerates all itemsets $W \in [U_k|\Sigma_k], (U_k, \Sigma_k) \in \pi$. 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 $[U_k|\Sigma_k]$.

7.5. The PARALLEL-FIMI-RESERVOIR algorithm

This algorithm samples $\widetilde{\mathcal{F}}$ using the *reservoir sampling*. The reservoir sampling samples $\widetilde{\mathcal{F}}$ uniformly. To make the algorithm faster, the reservoir sampling is executed in parallel. The method is summarized in the PARALLEL-FIMI-RESERVOIR method, see Algorithm 4.

8. Experimental evaluation

We have measured the speedup of our new method, the PARALLEL-FIMI-RESERVOIR method, on a cluster of workstations using three datasets.

The cluster of workstations was interconnected with the Infiniband network. Every node in the cluster has two dual-core 2.6GHz AMD Opteron processors and 8GB of main memory.

The datasets were generated using the IBM database generator. We have used datasets with 500k transactions and supports for each dataset such that the sequential run of the Eclat algorithm is between 100 and 12000 seconds ($\approx$ 3.3 hours) and two cases with running time 33764 (9.37 hours) and 132186 (36.71 hours) seconds. 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 experiments were performed with various values of the support parameter on 2, 4, 6, and 10 processors. The databases and supports used for evaluation of our algorithm is summarized in the Table 1.

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

In Phase 4 in our experiments, we use the ECLAT algorithm for mining of FIs. We have used the ECLAT in Phase 1 and 4 of the PARALLEL-FIMI-RESERVOIR method. The PARALLEL-FIMI-RESERVOIR method achieves speedup up to 8.6 on 10 processors.

There is an advantage of the PARALLEL-FIMI-RESERVOIR over the two previous methods [1, 2]: the need of computation of 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. The PARALLEL-FIMI-RESERVOIR does not suffer from this problem.

Figure 1 clearly demonstrate that for reasonably large and reasonably structured datasets, the speedup is linear with speedup ≈ 6 on 10 processors. The numeric values of the speedup are located in Table 2.

In [2], we have evaluated the PARALLEL-FIMI-PAR as faster then the PARALLEL-FIMI-SEQ method. We can compare the speedup of the PARALLEL-FIMI-RESERVOIR method with the PARALLEL-FIMI-PAR method. In the PARALLEL-FIMI-PAR method we have used the Eclat algorithm in Phase 4 and the Fpmax* [8] algorithm in Phase 1 as the algorithm for mining of MFIs. The speedup of PARALLEL-FIMI-PAR method is shown in Figure 1 the numerical average speedup values are located in Table 2. We can see that the speedup of the PARALLEL-FIMI-PAR is a bit smaller then the speedup of PARALLEL-FIMI-RESERVOIR. Additionally, in some cases, we were not able to finish the execution of the PARALLEL-FIMI-PAR due to large amount of used memory. In such cases the speedup is shown to be 0.

Algorithm 4 The PARALLEL-FIMI-RESERVOIR method.

PARALLEL-FIMI-RESERVOIR **In:** Double $min\_support^*$,
In: Doubles $\epsilon_{\widetilde{\mathcal{D}}}, \delta_{\widetilde{\mathcal{D}}}, \epsilon_{\widetilde{\mathcal{F}}_s}, \delta_{\widetilde{\mathcal{F}}_s}, \rho, \alpha$,
Out: Set $\mathcal{F}$)

1: **for** all p_i **do-in-parallel**
// Phase 1: sampling.
2: Read D_i and set $N_{\widetilde{\mathcal{D}}} \leftarrow \frac{1}{2\epsilon^2_{\widetilde{\mathcal{D}}}} \ln \frac{2}{\delta_{\widetilde{\mathcal{D}}}}$.
3: Creates a sample $D'_i \subseteq D_i$ and broadcast it to each other processor.
4: $\widetilde{\mathcal{D}} \leftarrow \bigcup_i D'_i$.
5: Execute in parallel an arbitrary algorithm for mining of FIs on database $\widetilde{\mathcal{D}}$ in parallel and create the sample $\widetilde{\mathcal{F}}_s$ using the VITTER-RESERVOIR-SAMPLING.
// Phase 2: partitioning.
6: p_1 creates PBECs $[U_k]$ such that $\frac{|[U_k] \cap \widetilde{\mathcal{F}}_s|}{|\widetilde{\mathcal{F}}_s|} < \alpha \cdot \frac{1}{P}$.
7: p_1 creates $L_j, 1 \leq j \leq P$ using the LPT-MAKESPAN algorithm.
8: *// Phase 3: data re-distribution.*
9: Redistribute the database partition D_i
10: *// Phase 4: parallel computation of FIs.*
11: compute support of $W \subseteq U_k$ in D_i and send the supports to p_1
12: p_1 outputs W
13: all p_i executes an arbitrary algorithm for mining of FIs in parallel that computes supports of $Supp(W, D'_q), W \in \bigcup_{k \in L_q} [U_k|\Sigma_k]$, $(U_k, \Sigma_k) \in \pi$.
14: **end for**

The database replication: we define the *database replication factor* as follows: let D'_i is the database partition used by the ith processor in Phase 4, i.e., the database part received in Phase 3. The database replication factor is defined as follows: $\frac{\sum_{i=1}^{P} |D'_i|}{|\mathcal{D}|}$. One would expect that the database replication factor will be small, e.g., for $P = 10$ a replication factor between $2 - 6$ would be expected. The oposite is the true. The replication factor is in all our experiments $\approx P$. The minimalization of the database replication factor is a hard task. The minimization of the database replication factor is an opened problem.

9. Acknowledgment

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

Dataset	Supports
T500I0.1P100PL20TL50	$0.11, 0.12, 0.13, 0.14, 0.15, 0.16, 0.17, 0.18$
T500I0.4P250PL10TL120	$0.2, 0.25, 0.26, 0.27, 0.3$
T500I1P100PL20TL50	$0.02, 0.03, 0.05, 0.07, 0.09$

Table 1: Databases used for measuring of the speedup and used supports values for each dataset.

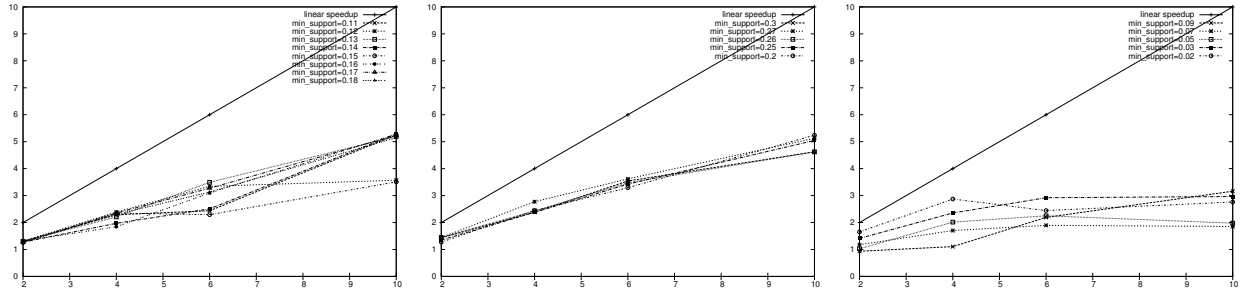

Figure 1: Speedup of the PARALLEL-FIMI-RESERVOIR method parametrized with the Eclat algorithm, measured on the `T500I0.1P100PL20TL50`, `T500I0.4P250PL10TL120`, `T500I1P100PL20TL50` (from left to right).

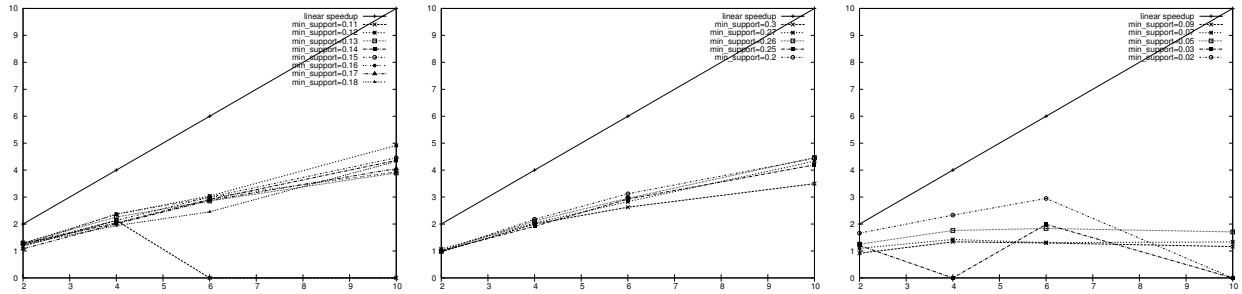

Figure 2: Speedup of the PARALLEL-FIMI-PARmethod parametrized with the Eclat algorithm, measured on the `T500I0.1P100PL20TL50`, `T500I0.4P250PL10TL120`, `T500I1P100PL20TL50` (from left to right).

datafile/PARALLEL-FIMI-RESERVOIR	2	4	6	10
T500I0.1P50PL10TL40	1.523	2.633	3.380	5.342
T500I0.4P250PL10TL120	1.389	2.481	3.470	4.932
T500I1P100PL20TL50	1.240	2.010	2.340	2.544
Total average	1.384	2.375	3.063	4.273

datafile/PARALLEL-FIMI-PAR	2	4	6	10
T500I0.1P50PL10TL40	1.596	2.668	3.438	5.135
T500I0.4P250PL10TL120	1.010	2.050	2.891	4.186
T500I1P100PL20TL50	1.227	1.714	1.876	1.401
Total average	1.277	2.144	2.735	3.574

Table 2: Numerical values of average speedup of the PARALLEL-FIMI-RESERVOIR and PARALLEL-FIMI-PAR methods for number of processors $P = 2, 4, 6, 10$.

References

[1] R. Kessl and P. 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.

[2] R. Kessl and P. 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.

[3] H. Toivonen, Sampling large databases for association rules. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, editors, *In Proc. 1996 Int. Conf. Very Large Data Bases*, pages 134–145. Morgan Kaufman, 09 1996.

[4] J.S. Vitter, Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, March 1985.

[5] M. Skala, Hypergeometric tail inequalities: ending the insanity. http://ansuz.sooke.bc.ca/professional/hypergeometric.pdf.

[6] R.L. Graham, Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics*, 17(2):416–429, 1969.

[7] http://en.wikipedia.org/wiki/Round_robin_tournament.

[8] 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, 19 December 2003, Melbourne, Florida, USA*, volume 90 of *CEUR Workshop Proceedings*, 2003.