

Out-of-Core Frequent Pattern Mining on a Commodity PC

Gregory Buehrer, Srinivasan Parthasarathy*, and Amol Ghoting
Department of Computer Science and Engineering
The Ohio State University, Columbus, OH 43210, USA

ABSTRACT

In this work we focus on the problem of frequent itemset mining on large, out-of-core data sets. After presenting a characterization of existing out-of-core frequent itemset mining algorithms and their drawbacks, we introduce our efficient, highly scalable solution. Presented in the context of the *FPGrowth* algorithm, our technique involves several novel I/O-conscious optimizations, such as approximate hash-based sorting and blocking, and leverages recent architectural advancements in commodity computers, such as 64-bit processing. We evaluate the proposed optimizations on truly large data sets, up to 75GB, and show they yield greater than a 400-fold execution time improvement. Finally, we discuss the impact of this research in the context of other pattern mining challenges, such as sequence mining and graph mining.

Categories and Subject Descriptors: H.2.8 [Database Management]: Database Applications - Data Mining;

General Terms: Algorithms, Performance.

Keywords: out of core, itemsets, data mining, pattern mining, secondary memory.

1. INTRODUCTION

The field of knowledge discovery and data mining is devoted to the challenge of extracting useful information from raw data. Over the past decade, technological advances have allowed us to gather an increasing amount of data in the areas of science, engineering, and business. Unfortunately, our ability to collect this data far outstrips our ability to analyze it *efficiently*. There are two main reasons for this problem. First, many data mining algorithms are computationally complex and scale non-linearly with the size of the data set. Second, when the data sets are extremely large, many data mining algorithms demand a memory footprint

*This work is supported in part by NSF grants #CAREER-IIS-0347662, #RI-CNS-0403342, and #NGS-CNS-0406386; Contact email – srini@cse.ohio-state.edu

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

KDD'06, August 20–23, 2006, Philadelphia, Pennsylvania, USA.

Copyright 2006 ACM 1-59593-339-5/06/0008 ...\$5.00.

that exceeds the size of main memory. This can lead to extreme slow down due to thrashing effects. While both these problems need to be addressed for ensuring the scalability of data mining algorithms, much of the work to date has focused on the first issue. The second issue, while not completely ignored, has not been satisfactorily addressed.

In this work, we focus on the relatively mature problem of frequent itemset mining [1]. Since its inception, the community has witnessed a proliferation of efficient frequent itemset mining algorithms in the literature [5, 9, 10, 13, 19]. Specifically, we concern ourselves with finding frequent itemsets in large, out-of-core data sets.

There are several feasible approaches to mine out-of-core data sets, which fall into two methodologies. The first methodology is to use an existing in-core algorithm, and leverage disk space to allow the meta-structures to exceed main memory. The disk-resident memory can either be managed by the virtual memory system, or explicitly by the programmer with the file system. Our performance study reveals that when these algorithms rely on the virtual memory system, they are extremely inefficient on out-of-core data sets. These algorithms incur a significant number of page faults, resulting in poor CPU utilization. Attempts to avoid the virtual memory system with explicit file handling are often difficult to program, and become quite daunting when the meta structures are pointer-based. The second methodology is to partition the data set such that each partition fits in main memory, use an in-core algorithm to mine each of these partitions, and aggregate and post-process the frequent itemsets discovered in each partition to obtain the global results [19]. While this approach is useful in some situations, typically it results in significant post-processing overhead and redundant computation. In essence, the former approach leverages the benefits of search space pruning by projection at the cost of decreased memory system performance, while the latter approach maintains good memory performance at the cost of a poor search space traversal.

To address the aforementioned shortcomings, we propose to take one of the most efficient in-core frequent itemset mining algorithms, *FPGrowth* [13], and identify strategies by which it can be made I/O-conscious. The proposed I/O-conscious optimizations ensure that the resulting algorithm exhibits excellent temporal and spatial locality. Leveraging recent architectural innovations such as 64-bit microprocessors, the algorithm affords excellent scaling on data sets close to available disk capacity, while maintaining good CPU utilization. This methodology results in a 400-fold improvement over any of the strategies we evaluated. While our

optimizations focus on the problem of frequent itemset mining, we believe that the key findings of this work will hold for many data mining tasks, especially those in the frequent pattern domain, such as graph and sequence mining.

Specifically, we make the following contributions:

- We characterize the performance of existing out-of-core frequent itemset mining solutions.
- We present several novel I/O-conscious optimizations, namely approximate hash-based sorting, and blocking, in the context of the *FPGrowth* algorithm.
- We empirically evaluate the effectiveness of our techniques on several data sets with sizes well beyond those that have been previously considered in the literature.

2. BACKGROUND

Frequent itemset mining plays an important role in a range of data mining tasks. Examples include mining associations [1], correlations [3], causality [20], episodes [15], and emerging patterns [7].

The frequent pattern mining problem was first formulated by Agrawal *et al.* [1] for association rule mining. Briefly, the problem description is as follows: Let $I = \{i_1, i_2, \dots, i_n\}$ be a set of n items, and let $D = \{T_1, T_2, \dots, T_m\}$ be a set of m transactions, where each transaction T_i is a subset of I . An itemset $i \subseteq I$ of size k is known as a k -itemset. The *support* of i is $\sum_{j=1}^m (1 : i \subseteq T_j)$, or informally speaking, the number of transactions in D that have i as a subset. The frequent pattern mining problem is to find all $i \in D$ with *support* greater than a minimum value, *minsupp*.

Agrawal and Srikant [2] presented *Apriori*, the first efficient algorithm to solve this problem. *Apriori* traverses the itemset search space in breadth-first order. Its efficiency stems from its use of the *anti-monotone* property: *If a size k -itemset is not frequent, then any size $(k + 1)$ -itemset containing it will not be frequent.* The algorithm first finds all frequent 1-items in the data set, and then iteratively finds all frequent l -itemsets using the frequent $(l - 1)$ -itemsets discovered previously.

This general level-wise algorithm has been extended in several different forms leading to improvements such as *DHP* [16] and *DIC* [3]. *DHP* uses hashing to reduce the number of candidate itemsets that must be considered during each data set scan. Furthermore, it progressively prunes the transaction data set as it discovers items that will not be useful during the next data set scan. *DIC* [3] processes the data set in chunks and considers new candidate itemsets as they are discovered in the chunk. Unlike *Apriori*, *DIC* does not require that the entire data set be scanned for each new candidate. Such an approach affords fewer passes.

We proposed *Eclat* [21] and several other algorithms that use equivalence classes to partition the problem into independent subtasks. The use of the vertical data format in *Eclat* allows for fast support counting by set intersection. The independent nature of subtasks, coupled with the use of the vertical data format, results in improved I/O efficiency because each subtask is able to reuse data in main memory. We also developed schemes for parallelizing and improving the performance of apriori-style algorithms on SMP systems [17, 18]. This work is the first to illustrate the benefits of improving memory performance in data mining algorithms in both the sequential as well as the parallel setting.

No.	Transaction	Sorted Transaction with Frequent Items
1	f, a, c, d, g, i, m, p	a, c, f, m, p
2	a, b, c, f, l, m, o	a, c, f, b, m
3	b, f, h, j, o	f, b
4	b, c, k, s, p	c, b, p
5	a, f, c, e, l, p, m, n	a, c, f, m, p
6	a, k	a

Table 1: A transaction data set with *minsupp* = 3

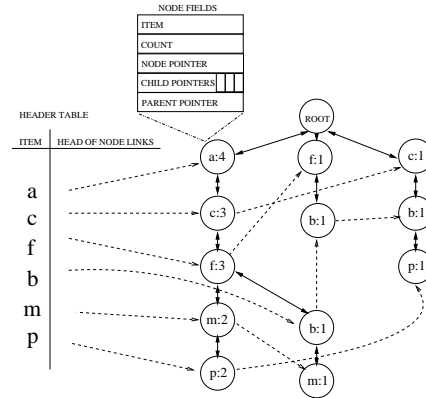


Figure 1: An FP-tree/prefix tree

Han *et al.* presented *FPGrowth* [13], an algorithm that significantly reduces the number of data set scans required. *FPGrowth* summarizes the data set into a succinct prefix tree or *FP-tree*. An additional benefit is that it does not have an explicit candidate generation phase. Rather, it generates frequent itemsets using tree projections in main memory. The payoff is an improved search space traversal. However, the pointer-based nature of the *FP-tree* requires costly dereferences. In previous work, we have shown this algorithm can be made cache-conscious [18, 8].

As our optimizations are presented in the context of the *FPGrowth* algorithm, we will next describe the *FP-tree* data structure and the *FPGrowth* algorithm in more detail. A prefix tree (or an *FP-tree* [13]) is a data structure that provides a potentially compact representation of transaction data set. Each node of the tree stores an item label and a count, where the count represents the number of transactions which contain all the items in the path from the root node to the current node. By ordering items in a transaction based on their frequency in the data set, a high degree of overlap is established.

A prefix tree is constructed as follows. First, we scan the data set to produce a list of frequent 1-items. Second, we sort the items in frequency descending order. Third, we sort the transactions based on the order from the second step. Fourth, we prune away infrequent 1-items. Finally, for each transaction, we insert each of its items into a tree, in sequential order, generating new nodes when a node with the appropriate label is not found, and incrementing the count of existing nodes otherwise.

Table 1 shows a sample transaction data set, and Figure 1 shows the corresponding prefix tree. Each node in the prefix tree consists of an *item*, *count*, *nodelink ptr*, (which points to the next item in the prefix tree with the same item-id) and *child ptrs* (a list of pointers to all its children). Pointers to the first occurrence of each item in the tree are stored in

Algorithm: FPGrowth

Input: A prefix tree D , minimum support min_s

Output: Set of all frequent itemsets

Mine the FP-Tree by calling $FP\text{-Growth}(FP\text{-tree}, null, min_s)$

$FP\text{-Growth}(tree, suffix, min_s)$

- (1) If tree has only one path
- (2) Output $2^{path} \cup suffix$ as frequent
- (3) Else
- (4) For each frequent 1-item β in the header table
- (5) Output the item $\cup suffix$ as frequent
- (6) Use the header list for β to find all frequent items in
- (-) conditional pattern base C for β
- (7) If we find at least one frequent item in the conditional
- (-) pattern base, use the header list for β , and C
- (-) to generate conditional prefix tree τ
- (8) If $\tau \neq \emptyset$ then
- (9) $FP\text{-Growth}(\tau, suffix \cup \beta)$
- (10) End For

Note: 2^{path} denotes the power set of the elements in path

Figure 2: FPGrowth Algorithm

a header table.

To compute the frequency count for an itemset, say ca , we proceed as follows. First, we find each occurrence of item c in the tree using the node link pointers. Next, for each occurrence of c , we traverse the tree in a bottom up fashion in search of an occurrence of a . The count for itemset ca is then the sum of counts for each node c in the tree that has a as an ancestor.

The *FPGrowth* algorithm is presented in Figure 2. First, it builds a prefix tree from the transaction database, removing all infrequent items, using the procedure outlined earlier. Next, using this prefix tree, the algorithm iterates through each item β in the tree, and performs two operations. In operation one, it uses the prefix tree to find all frequent items in the conditional pattern base for the item β . This involves scanning the tree bottom up beginning at all node locations for item β . The header table provides a starting point for this search. The remaining locations for item β are derived using the node link pointers. In operation two, assuming there has been at least one frequent item discovered in the conditional pattern base of item β , a projected database (represented as a prefix tree) is constructed for item β . The projected data set for an itemset is the subset of the transactions in the data set that contains the itemset. This operation also involves scanning the conditional pattern base of item β , in search for items to be included in the projected database. For each projected database or conditional FP-tree that is built, the algorithm proceeds recursively.

3. DRAWBACKS OF EXISTING SOLUTIONS

There are several approaches to find frequent itemsets in out-of-core data sets. In this section, we will briefly describe these approaches and identify their strengths and weaknesses. All empirical evaluations presented in this Section were performed on a PentiumD desktop PC with 256MB of RAM.

3.1 Scaling using Virtual Memory (VM)

For the past several decades, operating systems have provided programmers with a virtual address space that is sig-

Support	30%	25%	20%	15%	10%
CPU util. of FPGrowth	97%	98%	1%	<< 0.1%	<<0.1%
CPU util. of CC FPGrowth	99%	99%	4%	1%	<<0.1%

Table 2: CPU utilization for FPGrowth and Cache-conscious (CC) FPGrowth on Webdocs at varying support.

nificantly larger than the physical address space. Paging mechanisms have been responsible for moving instructions and data in and out of memory, as and when needed. Traditionally, commodity PCs have been 32-bit processors, which have a limit of 4GB on the amount of available virtual memory. Today, desktop PCs are afforded 64-bit memory addressing via technological innovations in CPU design. Examples of these new CPUs include the Intel PentiumD, the IBM PowerPC, and the AMD Athlon 64. This translates to a nearly unlimited amount of virtual memory. An in-core solution can then use this large amount of virtual memory to process out-of-core data sets, while relying on the OS's paging mechanism to automatically handle the movement of data between main memory and the disk sub-system. The key benefit of such an approach is its implementation simplicity.

To evaluate the efficacy of such an approach, we consider *FPGrowth* [13], which is one of the most efficient frequent itemset mining algorithms to date [9], and our *cache-conscious FPGrowth* algorithm, as presented in a previous effort [8]. We measure CPU utilization of both these algorithms on large, out-of-core data sets. The unmodified *FPGrowth* implementation is by Grahne and Zhu [11], as it has been shown to be amongst the most efficient [9].

Table 2 presents the CPU utilization of *FPGrowth* for mining the Webdocs data set (see Table 5 for details). It is clear that the CPU is largely idle, rendering a VM-based solution ineffective. Even our cache-conscious version, although an improvement, fairs quite poorly. In *FPGrowth*, about 28% of the execution time is spent in *Build First Tree()*, the routine that builds the very first prefix tree. 10% of the execution time is spent in the *Count FPGrowth()* routine. This routine finds the set of all viable items in the *FP-tree* (projected data set) that will be used to extend the frequent itemset at that point in the search space. 62% of the execution time is spent in the *Project FPGrowth()* routine, which scans the *FP-tree* to build a new projected *FP-tree* for the next recursive call.

Based on the provided performance characterization, together with an understanding of the algorithm, we attribute the poor CPU utilization to the following reasons. First, when the prefix tree is created, the memory address of a node in the tree is relatively independent of the memory address of its child and parent nodes. This is because transactions in the input data set can appear in any order. Consequently, prefix tree accesses exhibit poor spatial locality, both during the tree construction phase and the subsequent traversal phase. Furthermore, the algorithm does not benefit from page prefetching, because its memory access pattern lacks structure. These issues were addressed in the cache-conscious version, which provides an improvement to the CPU utilization over the original *FPGrowth*. Second, when the prefix tree is larger than the size of main memory, we

Support	25%	20%	15%	10%	7.5%
False Positives	1276	4376	40669	3530690	49498692
True Positives	585	1625	10668	219915	2303383

Table 3: Overheads from using a partitioning-based approach on Webdocs at varying supports.

have a negligible amount of temporal locality during execution. This prominent issue greatly affects every in-core frequent itemset mining algorithm we evaluated (only two shown due to space constraints). The above mentioned reasons cause the processor to wait on the completion of a page fault for the overwhelming majority of the execution time. *Even our cache-conscious algorithm falters considerably due to page faults while building the first tree.* In our case, the penalty is doubled because the cache-conscious algorithm builds a second, spatially improved tree.

3.2 Scaling using Partitioning

Savasere *et al.* presented *Partition* [19], an approach for out-of-core itemset mining. To mine large out-of-core data sets, they propose to first subdivide the original data set into smaller data sets that can be processed in main memory. Next, each of these partitions is mined in main memory using an in-core algorithm. This is followed by a union operation on all locally frequent itemsets discovered in each partition. Finally, to find the exact set of frequent itemsets, exact counts of all itemsets in the union are determined using a data set scan.

While this approach works well in some situations, it suffers from the following drawback. The technique is efficient only when the union set of discovered locally frequent itemsets (frequent in any one partition) is close to the actual set of actual frequent itemsets (in the data set). If this is not the case, one must calculate the exact support count of a much larger set of itemsets, often resulting in significant computational overheads. In a case study, we measure the number of false positives (number of itemsets in the union) and the number of true positives (number of globally frequent itemsets) as we varied the minimum support. We find that the number of false positives increases exponentially with decreasing support, as illustrated in Table 3. This finding renders a partitioning-based approach infeasible on large, out-of-core data sets.

3.3 Scaling using Recursive Projection

Grahne and Zhu presented *Diskmine* [12], an approach for out-of-core itemset mining that uses projections to partition the data. The approach recursively projects the data set until it fits in main memory and then uses an in-core algorithm to mine the projected data set. This technique affords improved memory system usage at the expense of a larger search space and increased computation. The premise is that during execution, the first prefix tree may not fit in main memory, at which time the tree is deleted and all frequent 2-itemset projections are created. If any of the resulting data structures for each frequent 2-itemset projection do not fit in memory, the algorithm recurses to frequent 3-itemset projections. This proceeds until the data structure does fit in main memory, at which point the mining process can proceed efficiently. The frequent itemsets can be determined by simply taking a union of the itemsets mined from each projection.

Support	20%	15%	10%	7.5%
Additional Scans	1711	6903	33930	75466
Time Required (hours)	18.5	74.7	367	817

Table 4: Number of additional data set scans needed when using recursive projection on the Webdocs data set at various supports.

We find there are two significant drawbacks to this approach. First, when projecting the data set, if the data structure does not fit in memory, the algorithm must project by including an additional item, and deleting the current data structure from memory. The new projection requires an additional full scan of the data set. Second, explicit candidate generation is then required, which can be very expensive [13]. In other words, when the frequent 1-item tree does not fit in memory, then all combinations of frequent 2-itemsets must be projected, even if a large subset of these itemsets are not actually frequent. In effect, one of the main advantages of the *FPGrowth* algorithm (elimination of candidate generation) is negated.

We implemented this technique to evaluate its potential. In Table 4, we show the number of additional scans as a function of support for the Webdocs data set, and its associated execution time cost¹. If the frequent 2-itemset trees do not fit in memory, this constraint is heightened. Equation 1 evaluates the additional number of data set scans, allowing n to be the number of frequent 1-items and r to be the depth of the recursion up to the point the projected trees fit in main memory.

$$scans = \frac{n!}{r!(n-r)!} \quad (1)$$

Current techniques to mitigate this liability, such as combining multiple frequent 2-itemset projections with a single scan, break down at low supports [12]. Intuitively, this technique assumes frequent 2-itemset projections to be many times smaller than main memory (so as to afford combining), which is often not the case.

3.4 Scaling by Solving a Related Problem

One last method we briefly describe involves adjusting the problem definition to avoid exceeding main memory. Examples include mining for closed itemsets, maximal itemsets, or non-derivable itemsets. These related problem definitions are quite useful in their own right, and attack two concerns simultaneously. First, they can reduce the amount of state and computation required. Strategies such as partitioning the data set can be more attractive when less information is required to process the data set. Second, they often result in a reduced result set, which eases human analysis. However, a drawback to this strategy is that the problem of mining large data sets is merely pushed to a slightly larger data set; an algorithm that could previously handle a 4GB data set can now perhaps process a 6GB data set. Such approaches do not permit scaling on truly large data sets and the underlying concern continues to persist. In this work we maintain the problem definition to be that of mining frequent itemsets because we feel that this information has uses beyond the values of the previous alternatives. Furthermore, from an algorithmic view point, frequent itemset mining shares

¹We require about 39 seconds to touch each item in the Webdocs data set.

Input: Transaction T, int totalFiles
Output: int X_t

CalculateFileNo(Transaction T,int totalFiles)

- (1) min = 0
- (2) max = totalFiles
- (3) check = 0
- (4) index = 0
- (5) While (min < max and index < |T|)
- (6) If (T[index] == check)
- (7) max = (min+max)/2
- (8) index++
- (9) Else
- (10) min = (min+max+1)/2
- (11) check++
- (12)End While
- (13)Return min

Figure 3: Geometric Partitioning Algorithm.

Input: int N, partition X
Output: int X_t

CalculateFileNo(Transaction T,int totalFiles, int index, int check)

- (1) If (|X| > 2N)
- (2) If (T[index] < check+N
- (3) $X_t = T[index] - check$
- (4) Else
- (5) div = (totalFreqItems - check)/(|X| - N)
- (6) $X_t = (T[index] - check)/div$
- (7) Else
- (8) div = (totalFreqItems - check)/(|X| - N)
- (9) $X_t = (T[index] - check)/div$
- (10)Return X_t

Figure 4: Arithmetic Partitioning Algorithm.

common structure with other itemset mining tasks, such a closed itemset mining and maximal itemset mining, and we believe our optimizations can be applied to these tasks with comparable benefits.

4. I/O-CONSCIOUS OPTIMIZATIONS

Of the methods described in the previous section, scaling using virtual memory results in the least amount of computational overhead. However, when executions spill onto disk, such algorithms exhibit severe performance degradation. This is understandable; typical main memory access times are about 15 nanoseconds, while typical disk access times are 5 milliseconds, constituting a 333,000-fold gap in performance. This gap is likely to widen in the future, because memory access times are improving faster than disk access times. Our strategy to achieve an out-of-core solution is to minimize the performance degradation due to this gap through data and computation restructuring, to improve locality. In this Section, we present several novel techniques for improving the I/O performance of frequent itemset mining algorithms. We choose to present these techniques via *FPGrowth* because it has been shown to be the most efficient frequent pattern mining algorithm to date [9].

4.1 Approximate Hash Sorting

As discussed earlier, the initial step in *FPGrowth* is to construct a global prefix tree. This first tree can be quite large; at low supports it can approach or even exceed the size of the data set. For in-core data sets, this construction time is typically a small percentage (< 5 %) of the total mining time. For out-of-core data sets, however, construction of the

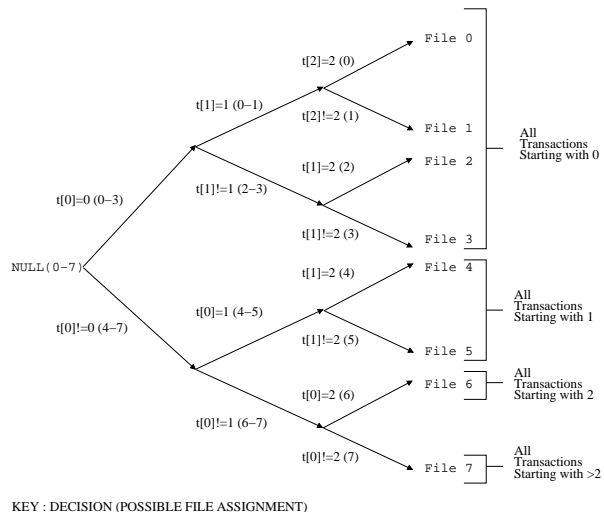


Figure 5: Geometric Partitioning Decision Diagram, for 8 Files.

first tree results in severe performance degradation. During our empirical study (presented in Section 3), we showed that our cache-conscious algorithm was a significant performance improvement over *FPGrowth* for out-of-core data sets. However, it spent over 90% of the execution time building the first tree, due to an excessive number of page faults. The reason is that transactions within the data set appear randomly, which results in random writes to the tree nodes in virtual memory during tree construction. Even if the initial data set had its transactions ordered, the problem would persist since the transactions are relabeled prior to tree construction (for improved overlap).

Our solution to this problem is to redistribute and approximately sort the transactions after the first scan of the database. Naturally, sorting on disk is quite slow. Traditional methods for external sorting (such as B-tree insertion and disk-based merge sort) do not provide an overall performance improvement. Exact sorting requires too much time. Instead, we leverage domain knowledge and the frequency information collected in the first scan to approximately sort the frequent transactions into a partition of blocks. Each block is implemented as a separate file on disk. The algorithm guarantees that each transaction in $block_i$ sorts before all transactions in $block_{i+1}$, and the maximum size of a block is no larger than a preset threshold. By blocking the frequent data set, we can build the tree on disk in fixed memory chunks. A block as well as the portion of the tree being updated by the block will fit in main memory during tree construction, reducing page faults considerably.

We use frequency distributions to choose one of the two partitioning algorithms listed in Figures 3 and 4 by building a simple model of the distribution. We build this model using the top 10% most frequent items. Essentially, if the item frequencies follow a geometric series (in descending order), we partition based on the algorithm in Figure 3, otherwise we partition based on the algorithm in Figure 4.

We first describe the algorithm in Figure 3. Let $X = |partition|$, or the total number of files. Let S represent the maximum file size. We define a function such that transactions with the most frequent item receive the top $X/2$ of

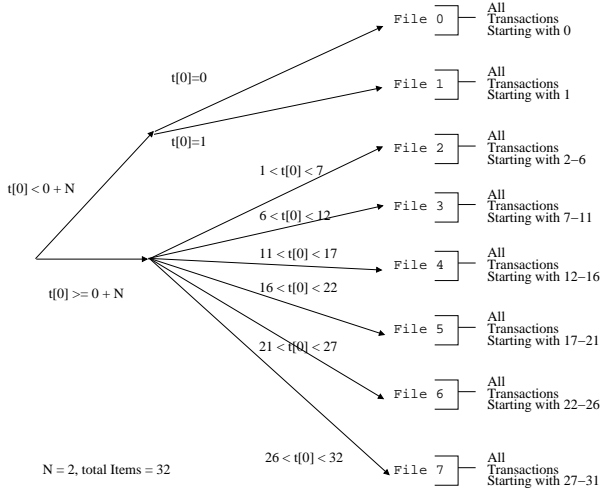


Figure 6: Arithmetic Partitioning Decision Diagram, for 8 Files.

Input: int N, partition X
Output: int index

```

CalculateFileNo(int fileNo)
(1) index=0
(2) x = log2|X|
(3) check = x
(4) For (i = 1; i < x; i++)
(5)   If (fileNo mod 2i < 2i-1)
(6)     index++
(7)   Else
(8)     Break
(7) Return index

```

Figure 7: Recursive Index Calculation.

the blocks, transactions with the second most frequent item receive the next $X/4$ of the blocks, and so on. Of these top $X/2$ partitions, the top $X/4$ are dedicated to the subset which also contain the second most frequent item. The bottom $X/4$ blocks of this subset are split into two equal sections. The top $X/8$ is dedicated to transactions containing the third most frequent item exists, and the lower $X/8$ for those which do not. This pattern recurses until the exact block number is known. Therefore, *in one scan*, each transaction is inserted into one of X blocks (typically 256), based on its contents. In the case that a partition has a size above our threshold, we evaluate its local distribution and process it recursively.

Let us illustrate this algorithm with an example. Suppose transaction $T = \{33, 11208, 11, 678, 14, 91, 278\}$. After scanning the data set, calculating frequencies (removing infrequent items) and relabeling, the transaction $T' = \{1, 2, 4, 6, 10\}$. Let the number of files $|X| = 8$, as in Figure 5. Our task is to determine X_t , the file assigned to T' . We examine the first element in the transaction, and if it is the smallest (most frequent) element possible, we assign it to the upper half of possible files. In our example, $1 = 1$ (first item in transaction=most frequent item), so we reduce the potential file assignment to 0-3. The second element is also its minimum ($2=2$, second item=second most frequent item), and therefore the file list is reduced to 0-1. The third element is not its minimum ($4 \neq 3$, third item in transaction is not the

third most frequent item) so the block assigned is 1.

If the item frequency model more closely resembles a linear distribution, we partition using the algorithm in Figure 4. Effectively, we assign the first N blocks to the most frequent items, and assign the remaining items to the remaining files equally. Lines assign the transaction to either a *dedicated* file if the item in the index is highly frequent, or a *shared* file if the item is not highly frequent. High frequency is relative; we allow for a parameter N to distinguish the threshold for the top items which receive dedicated files. In practice we set N to 5% of the total number of frequent items. A decision diagram for this algorithm is presented in Figure 6.

As stated earlier, it may be the case that a resulting file in the partition exceeds our threshold S . If so, we simply recurse on the file with the same procedure. However, we must calculate the new start index (in the transaction) to continue the partitioning. The index can be determined solely based on the file number, using the algorithm provided in Figure 7. Note that this results in $n * |partition| - (n - 1)$ files in total, where n is the number of file splitting calls. For sub-splitting files which surpass the maximum file size, we may neglect to build a model of the distribution for that subfile. Note that odd file numbers contain transactions whose last element was not the minimum (most frequent) value possible, and even valued file numbers contain transactions whose last element was a minimum. In practice we have found file parity provides sufficient information to evaluate which algorithm to use when sub-splitting; even numbered files are partitioned geometrically and odd numbered files are partitioned arithmetically. We provide additional details in a companion technical report [4].

With the knowledge that consecutive files are in relative order, tree building can proceed by processing the files in order with a minimum number of page faults. As will be illustrated in Section 5, this partitioning technique dramatically reduces the cost of building the main tree on disk, and provides significant total execution time improvement.

4.2 Improving Spatial Locality

Through the characterization presented in Section 3, we conclude that approximately 10% of the execution time is spent on finding frequent items in the conditional pattern base of an item, and an additional 62% of the execution time is spent on using the results of this step to create a new projected prefix tree. Both these procedures have very poor main memory utilization and suffer from a high page fault rate due to poor data locality.

We present the *I/O-conscious prefix tree*, a data structure designed to significantly improve main memory utilization through spatial locality. It is a modified prefix tree which accommodates fast bottom-up traversals, and shares many of the features we presented in the context of cache performance optimizations for in-core pattern mining [8]. As has been shown earlier, the cache-conscious algorithm is not effective on out-of-core data sets. An I/O-conscious solution must be mindful of the various OS mechanisms as well as the unit of transfer between disk and main memory. However, the principles of alleviating the performance gap between the cache and main memory also apply to the case involving main memory and disk.

Given a prefix tree, our solution to improve spatial locality is to reallocate the tree in virtual memory, such that the

Core loop: FPGrowth
Input: A prefix tree D , minimum support min_s

- (1) For each frequent item i in D
- (2) Find number of items j in conditional pattern base of i with support min_s
- (3) If $j > 0$
- (4) Build conditional prefix tree P for item i
- (5) FPGrowth(P, min_s)
- (6) End For

Figure 8: Core Loop for FPGrowth

new tree allocation is in depth-first order. We *malloc()* fixed sized blocks of memory, whose sum is equal to the total size of the prefix tree. Empirically, we found that a 4 MB block size is most effective. Next, we traverse the tree in depth-first order, and (in one pass) copy each node to the next location (in sequential order) in the newly allocated blocks of virtual memory. This simple reallocation strategy provides significant improvements, because *FPGrowth* accesses the prefix tree several times in a bottom up fashion, which is largely aligned with a depth-first order of the tree. Second, our node size is much smaller than the original node size, because we do not include child pointers, node pointers, and counts. The node fields *node link pointer* and *count* are required at the start of each bottom up traversal. Therefore, these fields are stored in a separate structure, without any performance degradation, as *node link pointer* and *count* accesses are not along the critical path.

4.3 Improving Temporal Locality

Temporal locality states that recently accessed memory locations are likely to be accessed again in the near future. Designers of OS paging mechanisms work under the assumption that programs will exhibit good temporal locality, and store recently accessed data in main memory accordingly. Therefore, it is imperative that we find any existing temporal locality and restructure computation to exploit it.

4.3.1 Page Blocking

The goal of restructuring the algorithm is to maximize reuse of the prefix tree once it is fetched into main memory. We accomplish this by reorganizing computation, and thus accesses to the prefix tree. Our approach relies on *page blocking*, and is analogous to our tiling techniques [8] for improving temporal locality in in-core pattern mining algorithms. The core loops for *FPGrowth* and *Blocked FPGrowth* are presented in Figures 8 and 9, respectively. It operates as follows:

First, we break down the tree into relatively fixed sized blocks of memory (page blocks) along paths of the tree from leaf nodes to the root. This is possible because our tree is allocated in depth first order. The blocks are identified using a starting and ending memory address. We would also like to point out that these blocks can partially overlap. Next, we iteratively fetch each block into main memory [8]. Then for each frequent item i , we traverse the part of its conditional pattern base that has leaf nodes located within the block’s starting and ending address. Thus, once a block is brought into the main memory, conditional pattern base accesses for all items that hit the block are managed in main memory. We would also like to point out that by controlling the size of the block, one can tune the implementation to

Core loop after blocking: FPGrowth
Input: A prefix tree D , minimum support min_s

- (1) For each *block* b in the I/O-conscious prefix tree
- (2) For each frequent item i in D
- (3) Find counts c_i for each item in the conditional pattern base of i with node locations in b
- (4) End For
- (5) Aggregate conditional pattern base counts c_i collected across all blocks for all items in D
- (6) For each *block* b in the I/O-conscious prefix tree
- (7) For each frequent item i in D
- (8) $j = c_i =$ number of items in conditional pattern base of i
- (9) If $j > 0$
- (10) Build conditional prefix tree P_i for item i with node locations in b
- (11) End For
- (12) End For
- (13) For each frequent item i in D
- (14) If conditional prefix tree P_i for item i exists
- (15) FPGrowth(P_i, min_s)
- (16) End For

Figure 9: Core Loops for Blocked FPGrowth

better utilize the available memory budget. A similar restructuring technique is used for generating projected prefix trees. For further details, please refer to [8].

5. EXPERIMENTAL EVALUATION

To evaluate our I/O optimizations, we employ a typical current-day desktop machine; an Intel PentiumD with a P-ATA 320GB hard drive, running 64-bit SUSE Linux 10. Of this 320GB of disk space, we allocate 170GB for swap purposes (VM). The PentiumD has two cores, but in this work, we only use 1 core throughout all experiments. We use two RAM configurations; 256 MB of DDR2 RAM and 1 GB of DDR2 RAM, and will specify the configuration in use in each experiment to follow. We believe this machine represents a commodity machine. In fact, we built it for about \$500.

5.1 Data Sets

We use several data sets to perform our evaluation. Web-docs is a 1.48GB data set from the Frequent Itemset Mining Implementations Repository (FIMI) [9] containing about 1.6 million transactions. This is the largest data set in the FIMI Repository, and at sufficiently low support the memory footprint for the tested algorithms is several times the size of main memory (using 256MB RAM). In addition, we generate several synthetic data sets using the IBM Quest Data Set Generator. D1 is a small 1GB data set with 4.7 million transactions. Transactions average 40 items in length, with 10,000 distinct items. D60 is a relatively sparse 60GB data set. It also averages 40 items per transaction but contains 100,000 distinct items. D75 is a much denser data set than D60, due to its increased transaction length (100) and decreased number of items (20,000). D60 and D75 are designed to test the end scalability of our solution. Details on these data sets can be found in Table 5.

5.2 Impact of Locality Improvements

We first evaluate the impact of spatial and temporal locality improvements due to our prefix tree modifications and our page blocking scheme. For this experiment, we use 256MB of RAM. We compare our algorithm (without approximate sorting) against *FPGrowth* by Grahne and Zhu

[11]. As can be seen in Figure 10, the algorithms behave similarly while in core, but we are able to improve execution time by 10- to 15-fold when the execution exceeds main memory. This is a significant improvement, and is afforded by reducing the time required during the tree mining phase. In fact, our timing results show that the tree mining time is reduced by two orders of magnitude. However, the overall execution time is only reduced by 10- to 15-fold because we incur twice the tree building time, since we must first construct a prefix tree, and then construct an I/O-conscious tree. This tree construction now becomes the dominant bottleneck, representing approximately 90% of the total execution time.

5.3 Impact of Approximate Sorting

We now evaluate how approximate sorting can help alleviate this bottleneck. For this experiment, we use 1GB of RAM. We set the filesize threshold to be 30% of main memory; we automate this via the `/proc/meminfo` system file. As can be seen in Figure 11, by using approximate sorting (OoC), we are able to reduce execution time on D60 and D75 by up to 25-fold. This improvement is on top of that afforded by the spatial and temporal locality improvements. For D60 at 0.2% support, 2041 files were generated, and 53.5 GB of virtual memory was consumed. The main prefix tree was 29.4GB. The number of files generated did not have an impact on performance, so long as the file split size was large enough to accommodate a low number of recursive splits.

In summary, *effective approximate sorting of the frequent transactions after the first data set scan reduces mining times from a few days to a few hours (on very large data sets), as the tree building phase is no longer the bottleneck.*

5.4 Scalability on Large Data Sets

The goal of this effort is to design an algorithm which can mine truly large data sets on a relatively inexpensive machine. To evaluate our progress towards achieving this goal, we investigate the performance of our algorithm on the D60 and D75 data sets (presented in Table 5). Returning to Figure 11, we examine how execution time varies with decreasing support. Our algorithm maintains a reasonable execution time throughout. We would like to point out that the overall time includes the initial two disk scans. For small data sets, these times are inconsequential. However, for large data sets, these scans are costly. On D60 and D75, we require approximately 1619 and 2008 seconds to touch the entire data set, respectively. This suggests a lower bound on execution time. When using our algorithm, we also touch the frequent portion of the data set at least one more time, due to our approximate hash sorting phase. Still, this cost is small in comparison to the excessive number of page faults incurred when using a naive VM-based solution, or the large number of additional scans required with the recursive projection-based techniques.

We claim that by improving data locality (and reducing page fault rate), we can sustain high CPU efficiency on a 64-bit processor. This will allow us to scale to data sets of any size, independent of the available main memory (given sufficient disk space). To evaluate this claim, we performed the following experiment using 1GB of main memory. We increased the portion of D75 that we mine, while maintaining a constant relative support of 1.5%. We recorded the execution time, the virtual memory consumption, and the

	Size	# Trans	# Items	Avg. Len.
Webdocs	1.48 GB	1,692,082	5,267,656	177
D1	1 GB	4,700,000	10,000	40
D60	60.5 GB	250,000,000	100,000	40
D75	74.4 GB	130,000,000	20,000	100

Table 5: Data sets used for experimentation.

total CPU utilization with increasing data set size. We calculated the CPU utilization by dividing the total user time of the execution by the total wall time. As seen in Figure 12, CPU utilization is relatively stable with increasing data set size. More importantly, we can see that overall system performance does not degrade as the working data structures in the algorithm move farther out of main memory. As illustrated, the total virtual memory required increased from 1.5GB to 34GB, but system utilization remained at about 20 % and execution time scaled linearly. We believe that with increased disk space, we can mine very large data sets (>500GB) at relatively low support values.

5.5 Comparison with Existing Algorithms

To compare our algorithm with existing approaches, we return to the Webdocs data set. Larger data sets are not suitable for existing algorithms. Here we use 256MB of RAM. The goal of this experiment is to evaluate how well existing algorithms cope as they exceed available main memory, and contrast them with our performance. We include *FPGrowth* here because it is the algorithm that we improve upon. We include *AFOPT* because its design intrinsically accommodates large data sets [14] through reduced memory consumption. However, as shown in Figure 13, when the virtual memory footprint of *AFOPT* exceeds the size of main memory, its performance degrades significantly. At a support of 20%, all three algorithms must use disk resident memory. The slowdowns in *AFOPT* and *FPGrowth* are attributed to the fact that they were not designed to exhibit high spatial and temporal locality, and thus do not utilize the memory hierarchy efficiently. *FPGrowth* fairs worse between the two, as it has a larger memory footprint. In fact, we have tested all the implementations from the FIMI repository, including algorithms such as *Apriori* and *Eclat*, and all exhibited the behavior seen from *AFOPT* and *FPGrowth* when their virtual memory footprint exceeds the size of main memory. However, our optimized out-of-core algorithm maintains its efficiency even as it spills into disk resident memory, resulting in over a 400-fold performance improvement.

It can be seen that our algorithm uses more virtual memory than the other two algorithms (Figure 13). This can be attributed to the space overheads of our page blocking technique. However, even with increased memory consumption, we do not exhibit a performance degradation as our memory accesses are localized, unlike the other algorithms.

6. DISCUSSION

Traditionally, in-core data mining solutions have not been capable of processing large, out-of-core data sets. There are two main reasons for this problem. First, 32-bit computing platforms afford a virtual memory of size at most 4GB. Second, even in the presence of a 64-bit address space, in-core solutions are not able to utilize the memory hierarchy effectively, resulting in poor CPU utilization. Finally, dealing

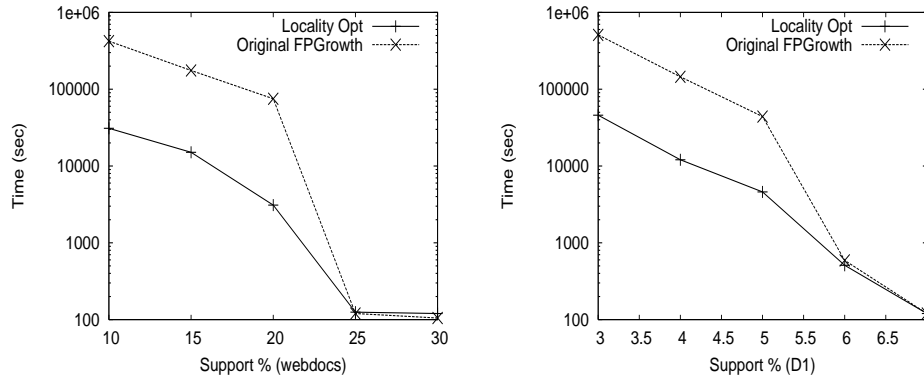


Figure 10: Evaluation of spatial and temporal locality improvements on Webdocs (left) and D1 (right).

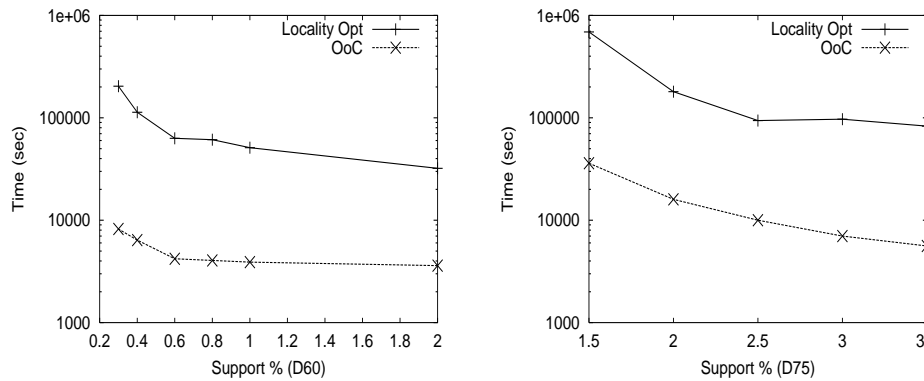


Figure 11: Scalability on D60 (left) and D75 (right) as a function of support.

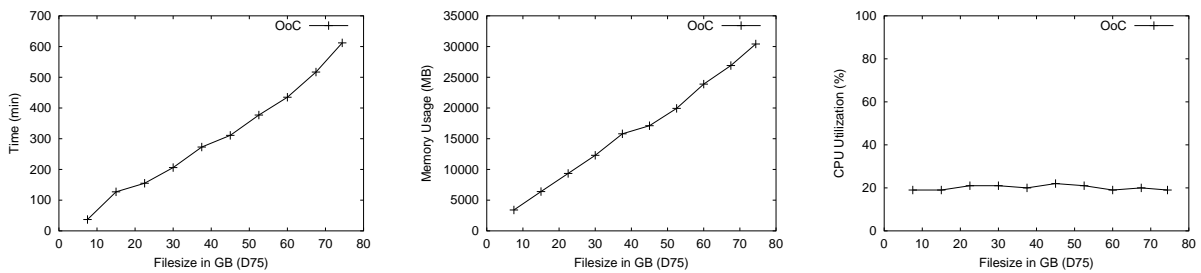


Figure 12: Execution time, memory usage, and CPU utilization as a function of data set size.

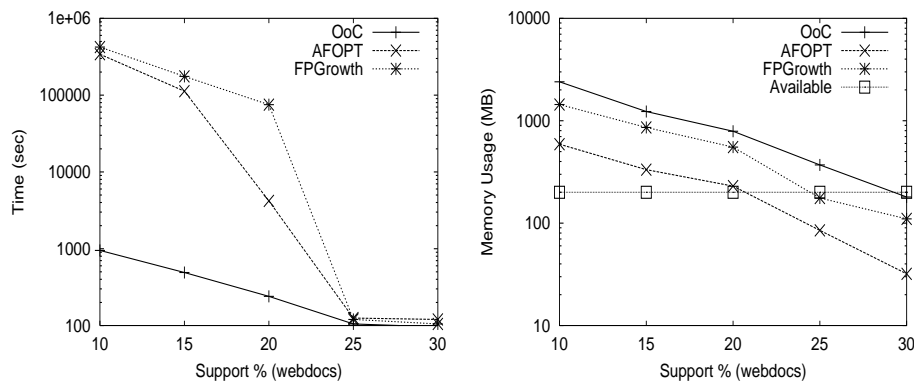


Figure 13: A comparison of execution time (left) and memory consumption (right) for several algorithms on the Webdocs data set.

with explicit disk operations for custom paging can be quite difficult, particularly with pointer-based meta structures.

In this paper, we propose a very different methodology for designing out-of-core data mining algorithms. Specifically, we propose to leverage the 64-bit address space made available on today's commodity processor. Furthermore, for situations in which we are unable to achieve good CPU utilization due to poor data locality, we propose to use relatively simple data and computation restructuring techniques to improve spatial and temporal locality, and thus improve CPU utilization. The proposed design methodology has two key benefits. First, the solutions are significantly easier to implement; existing solutions simply need to be reconsidered from the point-of-view of data locality. Second, an efficient solution can be devised using an off-the-shelf PC, making it extremely affordable.

It is our contention that a large percentage of data mining algorithms will be able to glean the benefits of such a design methodology. Algorithms in the areas of tree mining, sequence mining, and graph mining are particularly amenable to this design methodology. Solutions to these problems spend a considerable amount of time estimating the support for patterns, often re-reading the same blocks of data in a streaming fashion. In all likelihood, the working set for these algorithms will not fit main memory when mining out-of-core data sets. These algorithms do not necessarily use prefix trees, however, as such the community could benefit from an investigation into how their data structures can be made I/O-conscious. Through localized data placement, I/O-conscious data structures also have the potential of reducing the average disk seek time. This in turn can reduce the energy consumption of out-of-core executions, which is especially important for large scale data centers [6].

Trends indicate that emerging architectures will likely incorporate Chip Multiprocessing (CMP). CMPs, or *multi-cores* incorporate more than one processing element on the same die, providing true multiprocessing capabilities. With the availability of additional processing elements, orthogonal to improvements afforded through improved data locality, one can improve the I/O performance of an algorithm by employing a helper thread to prefetch pages from disk to main memory [22]. We believe that such an approach can significantly improve performance in many situations, and are currently investigating approaches by which an algorithm can efficiently relay information to a helper thread for prefetching data.

7. CONCLUSION

In this paper, we show that existing out-of-core frequent itemset mining solutions do not scale to truly large data sets. To address this limitation, we present a highly scalable solution. Presented in the context of the *FPGrowth* algorithm, we leverage 64-bit computing capabilities available on today's commodity processor. We demonstrate that I/O-conscious optimizations such as approximate hash sorting and blocking afford improved spatial and temporal locality, permitting the scaling of *FPGrowth* to out-of-core data sets. Empirically, we illustrate that our approach scales to truly large data sets, with sizes beyond those that have been previously considered in the literature. We believe that the proposed methodology is directly applicable to other data mining tasks as well.

8. REFERENCES

- [1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 1993.
- [2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 1994.
- [3] S. Brin, R. Motwani, and C. Silverstein. Beyond market basket: Generalizing association rules to correlations. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 1997.
- [4] G. Buehrer, S. Parthasarathy, and A. Ghoting. Out of core frequent pattern mining on a commodity pc. In *OSU Technical Report*, volume OSU-CISRC-4/06-TR36, 2006.
- [5] D. Burdick, M. Calimlim, and J. Gehrke. MAFIA: A maximal frequent itemset mining algorithm for transactional databases. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2001.
- [6] J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle. Managing energy and server resources in hosting centers. *SIGOPS Oper. Syst. Rev.*, 35(5):103–116, 2001.
- [7] G. Dong and J. Li. Efficient mining of emerging patterns: Discovering trends and differences. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 1999.
- [8] A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, A. Nguyen, Y. Chen, and P. Dubey. Cache-conscious frequent pattern mining on a modern processor. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 577–588, 2005.
- [9] B. Goethals and M. Zaki. Advances in frequent itemset mining implementations. In *Proceedings of the ICDM workshop on frequent itemset mining implementations*, 2003.
- [10] K. Gouda and M. Zaki. Efficiently mining maximal frequent itemsets. In *Proceedings of the International Conference on Data Mining (ICDM)*, 2001.
- [11] G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *Proceedings of the ICDM Workshop on Frequent Itemset Mining Implementations*, 2003.
- [12] G. Grahne and J. Zhu. Mining frequent itemsets from secondary memory. In *Proceedings of the International Conference on Data Mining (ICDM)*, 2004.
- [13] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2000.
- [14] G. Liu, H. Lu, J. X. Yu, W. Wei, and X. Xiao. Afopt: An efficient implementation of pattern growth approach. In *Proceedings of the ICDM workshop on frequent itemset mining implementations*, 2003.
- [15] H. Mannila, H. Toivonen, and A. Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1997.
- [16] J. Park, M. Chen, and P. Yu. An effective hash-based algorithm for mining association rules. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 1995.
- [17] S. Parthasarathy, M. Zaki, M. Ogihara, and W. Li. Memory placement techniques for parallel association mining. *International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 1998.
- [18] S. Parthasarathy, M. Zaki, M. Ogihara, and W. Li. Parallel data mining for association rules on shared-memory systems. *Knowledge and Information Systems Journal*, 2001.
- [19] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 1995.
- [20] C. Silverstein, S. Brin, R. Motwani, and J. Ullman. Scalable techniques for mining causal structures. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 1998.
- [21] M. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 1995.
- [22] J. Zhou, J. Cieslewicz, K. Ross, and M. Shah. Improving database performance on simultaneous multithreading processors. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 2005.