I/O Conscious Algorithm Design and Systems Support for Data Analysis on Emerging Architectures*

G. Buehrer[1], A. Ghoting[1], Xi Zhang[1], S. Tatikonda[1], S. Parthasarathy[12], T. Kurc[2], and J. Saltz[12]
The Ohio State University, Columbus, OH, USA

Abstract

Advances in data collection and storage technologies have given rise to large dynamic data stores. In order to effectively manage and mine such stores on modern and emerging architectures, one must consider both designing effective middleware support and re-architecting algorithms, to derive performance that commensurates with technological advances. In this article, we present a top-down view of how one can achieve this goal for next generation data analysis centers. Specifically, we present a case study on frequent pattern algorithms, and show how such algorithms can be re-structured to be cache, memory and I/O conscious. Furthermore, motivated by such algorithms, we present a services oriented middleware framework for the derivation of high performance on next generation architectures.

1 Introduction

Over the past decade, processor speeds have increased *40-fold* according to Moore's law. However, DRAM and disk access times have not kept up. Consequently, programs that exhibit poor data locality tend to keep the processor stalled, waiting on the completion of a disk access and/or memory access, for a large fraction of the time. This is often referred to as the *memory wall* problem and results in poor CPU utilization. Given the memory intensive nature of data mining algorithms, and the widening performance gap between the processor and lower levels of the memory hierarchy, it is our conjecture that these algorithms are grossly inefficient in terms of CPU utilization. We verified this conjecture in recent work [13].

Advanced architectural designs, even those possessing intelligent mechanisms for hiding memory and disk access latency, do not necessarily translate to improved application performance. Architectural innovations such as *prefetching* and *simultaneous multi-threading*, designed to reduce the effects of the memory wall and poor ILP (instruction level parallelism), have largely been ignored by the data mining community. Improving execution time will require rethinking by the algorithm designer. Researchers will need to reconsider computation structure and data layout to leverage the aforementioned innovations to improve performance.

A cluster of nodes with high speed interconnects and high-capacity commodity disks can create *active storage nodes* that enhance the ability to store, preprocess, and manipulate large-scale scientific and business data. These characteristics of commodity clusters make them cost-effective and viable to be *end-nodes* in the Data Grid [9] and to form the building blocks for next generation data centers with mass storage systems, serving very large data sets. We expect that such data centers will be ubiquitous along with increasing cost-effectiveness of spinning storage and that such centers will be an essential cog of future Data-Grid and Knowledge-Grid architectures [9, 7]. In this kind of environment, data management and manipulation must be coordinated through a middleware framework that exists between data sources and applications. Such a middleware will facilitate program development in data mining and at the same time will allow the algorithm designer to glean the benefits of next generation cluster-based data centers.

We believe that in order to derive high performance on next generation computing infrastructures, one must consider both designing effective middleware support and re-architecture algorithms. From the viewpoint of re-architecting algorithms, designers must consider approaches to improve spatial locality, temporal locality and the degree of ILP. From the viewpoint of designing effective middleware support, one must investigate both, the types of services that are desired by a range of data mining algorithms and how they can be designed to maximally utilize the available infrastructure.

In this paper we present a case study in designing fast frequent pattern mining solutions based on the aforementioned top-down view. Specifically, we make the following contributions. First, we present algorithmic improve-

*This work is primarily supported by NSF grant #NGS-CNS-0406386. The authors would also like to acknowledge NSF grants #CAREER-IIS-0347662 and #RI-CNS-0403342. [1]Department of Computer Science and Engineering, [2] Department of Biomedical Informatics. Contact email: srini@cse.ohio-state.edu

1-4244-0054-6/06/$20.00

ments that allow for frequent pattern mining algorithms to be cache, memory, and I/O conscious. Second, we present a services framework consisting of a sampling service and a sorting service, which are key components of the interactive pattern mining process. Finally, we present a preliminary evaluation of the proposed optimizations and services.

The rest of this paper is organized as follows. We present cache, memory, and I/O conscious optimizations for frequent pattern mining in Section 2. In Section 3, we describe various services for data mining algorithms, together with their application in frequent pattern mining. A preliminary evaluation of the proposed algorithmic improvements and services is presented in Section 4. We present related work in Section 5. Finally, we conclude in Section 6.

2 Cache, Memory, and I/O Conscious Frequent Pattern Mining

To illustrate efficient platform utilization for process state management, we explore improving frequent pattern mining. We seek to improve the effectiveness of the algorithm in all three dimensions listed previously, namely the cache performance, the main memory performance, and the disk performance.

Frequent Pattern Mining Defined: The frequent pattern mining problem was first formulated by Agrawal *et al.* [1]. The goal is to find groups of items or values that co-occur frequently in a transactional data set. Briefly, the problem description is as follows. Let $I = \{i_1, i_2, \cdots, i_n\}$ be a set of n items, and let $D = \{T_1, T_2, \cdots, T_m\}$ 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 $i \in D$ that have *support* greater than a minimum support value, *minsupp*.

FPGrowth [17] is a frequent pattern mining algorithm that uses an annotated prefix tree known as the *FP-tree* as a data set representation. Most algorithms for itemset mining exploit the *anti-monotone* property: *If a size k-itemset is not frequent, then any size $(k+1)$-itemset containing it will not be frequent.* *FPGrowth* also exploits this principle. Furthermore, it uses the pattern-growth based search methodology. Before we detail our optimization techniques, we will first introduce the reader to the prefix tree and the *FPGrowth* algorithm.

A prefix tree (or an FP-tree [17]) is a data structure that provides a compact representation of a transaction data set. Each node of the tree stores an item label and a count. The count field 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, a high degree of overlap is established. The compressed nature of this representation allows in-memory frequent pattern mining, because in most practical scenarios, this structure fits in main memory.

In summary, the *FPGrowth* algorithm works as follows. Beginning with frequent 1-items in the data set, each k-itemset is extended with frequent items that occur in the projected data set for the k-itemset to create $(k+1)$-itemsets. The projected data set for an itemset is the subset of the transactions in the data set that contains the itemset. This process is carried out recursively in depth-first order of the search space. Each level in the recursion uses the *FP-tree* as a data set representation. Several independent evaluations suggest that *FPGrowth* is the most efficient frequent pattern mining algorithm [14] to date.

Obstacles: The primary access pattern in *FPGrowth* is a bottom up traversal of the prefix tree. When we scan the prefix tree, we are only concerned with the *itemlabel* and *parentpointer* fields associated with the tree node. In the prefix tree proposed by Han *et al.* [17], each node has a list of child pointers, a parent pointer, a nodelink pointer, a count, and an item label. Except for the item label and the parent pointer, all other fields in the prefix tree node are not required for the main access pattern. Consequently, once we fetch a prefix tree node, only two fields are actually used. This significantly degrades cache line utilization and results in wastage of memory bandwidth.

Due to the way a prefix tree is constructed, in all likelihood, a node and its child-node will not be present in adjacent locations in main memory. The prefix tree is constructed as the data set is scanned, and thus, successive tree accesses during a bottom-up traversal need not be contiguous in memory. Due to the lack of temporal locality, this node is not likely to be present in any other cache line. The result is commonly a cache miss.

Cache-conscious Improvements: Spatial locality states that items whose addresses are near one another tend to be referenced close together in time. We present the *memory conscious prefix tree* [12], a data structure designed to significantly improve cache performance through spatial locality. A memory conscious prefix tree is a modified prefix tree which accommodates fast bottom up traversals and improves cache line and page usage. First, given a prefix tree, our solution to improve spatial locality is to reallocate the tree in main memory, such that the new tree allocation is in depth-first order of the original tree. We *malloc*() one contiguous block of memory equal to the total size of the prefix tree. Next, we traverse the tree in depth-first order, and (in one pass) copy each node to the next block of memory (in sequential order). This simple reallocation strategy provides significant improvements, because all algorithms

access 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, next pointers, or counts. These data members are eliminated because in *FPGrowth*, child pointers, node links pointers, and counts are only used to create the tree. The dominant traversals of the tree are bottom up and do not use these fields. This removes unwanted data from the critical path of prefix tree traversal. Each node is less than half of its original size, which allows at least twice as many nodes to reside on one cache line (or page).

Temporal locality states that recently accessed memory locations are likely to be accessed again in the near future. Hardware designers assume temporal locality, and store recent accesses accordingly. It is imperative that we find any temporal locality in the algorithm and redesign it accordingly. We can restructure the algorithm to improve temporal locality. The goal of restructuring the algorithm is to maximize reuse of the prefix tree once it is fetched into cache. We accomplish this by reorganizing computation, and thus, accesses to the prefix tree, in the algorithm. Our approach is called path tiling. We break down the tree into relatively fixed sized blocks of memory (tiles) along paths of the tree from leaf nodes to the root. This is possible because our tree is allocated in depth first order. The tiles are identified using a starting and ending memory address. We would also like to point out that these tiles can partially overlap. Each of these tiles is fetched iteratively and operated upon to completion, improving temporal locality.

SMT Improvements: Simultaneous Multithreading [31] (SMT) is a processor design that combines hardware multithreading with superscalar processor technology to allow multiple threads to issue instructions each cycle. SMT permits all thread contexts to simultaneously compete for and share processor resources by maintaining several thread contexts on chip. Unlike conventional superscalar processors, which suffer from a lack of per-thread instruction-level parallelism, simultaneous multithreading enables multiple threads to compensate for low single-thread ILP. The performance consequence can be significantly higher instruction throughput and program speedups for database workloads, web and scientific applications. SMT has been incorporated into the Intel Pentium 4 processor in the form of HyperThreading technology [18] which supports two thread contexts on chip.

A natural candidate for a two-thread decomposition of a frequent pattern mining algorithm is to use an extant strategy like that proposed in [25]. Such a strategy would involve decomposing execution into two independent threads of computation. However, when we evaluate this strategy for *FPGrowth* on an SMT, we were not able to gain any benefit from simultaneous multithreading. A detailed study revealed that the first benefit from above is not materialized in frequent pattern mining implementations when using this extant strategy. This is because there is insufficient computation to overlap with the memory stalls. For now, however, we leverage the second benefit of SMT for improving ILP in frequent pattern mining. We devise a novel parallelization strategy in which the two threads follow each other through the same FPGrowth() calls. These threads are not independent, but rather, they operate on the same tile simultaneously. This is accomplished through fine grained parallel execution of the tiled loops. The workload for each tile is partitioned across the two threads. By co-scheduling the two threads, when one thread fetches a portion of the tile into the cache, it will be reused by the second thread. This results in significant cache reuse between the two threads.

I/O Improvements: While the above-mentioned algorithmic improvements facilitate efficient pattern mining, the initial tree construction can be quite costly for out of core and distributed data sets. Furthermore, since we must build a second tree (the memory conscious tree), the penalty is doubled. After the first scan, frequencies for each item are known, and thus relabeling and reordering of items within a transaction can proceed. However, the order of the transactions is random, which results in significant latency during the building process, as we need to deal with an excessive number of random disk seeks and page faults.

To alleviate these costs, we redistribute and sort the transactions after the first scan of the database. Naturally, sorting on disk is quite slow. Instead, we leverage domain knowledge of transaction databases to approximately sort it 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. In practice, we use a threshold of 50% of the size of main memory. The transactions within blocks are then sorted in main memory, which is very inexpensive relative to all other phases.

Our partitioning routine is defined as follows. Let X= $|partition|$. We define a recursive logarithmic function such that transactions with most frequent item receives the top X/2 of the segments, the second most frequent item receives the next X/4 of the segments, etc. Of these top X/2 partitions, the top X/4 are dedicated to this subset which also contains the second most frequent item. The bottom X/4 blocks of this subset are split into X/8 if the third most frequent item exists, or the lower X/8 if it does not exist. This pattern recurses until the exact segment number is known. Therefore, in one scan each transaction is inserted into one of X segments (typically 128) based on its itemset. Then each partition which is above our threshold is recursed upon.

The recursive partitioning mechanism is based on parity. Since even-numbered segments matched their last checked index, the same log-based algorithm is applied. However, odd-numbered segments contain transactions which did not have their last checked element in common. Accordingly, these partitions are distributed linearly based on the frequency of the first k items, starting with the next available unchecked index. The distribution can be approximated with some accuracy because at this point in the computation, counts for each frequent item are known.

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 4, this partitioning technique dramatically reduces the cost of redistributing the transaction data set, and provides significant total execution time improvement.

3 Services

We will now briefly describe several services that have been designed for data mining applications. In addition to frequent pattern mining, we believe these services will be useful for a wide range of data mining tasks such as sequence mining [2], graph mining [20], outlier detection [4], and decision tree induction [27].

Storage and I/O Services: The data storage service implements strategies for managing and accessing data in distributed disk space. It also supports basic update-propagation trigger services for dynamic data sets. Storage Services support data replication and de-clustering. In order to achieve high I/O rates, organization and distribution of data across the system is paramount. The objective of data distribution is to balance the storage and I/O load across the nodes so that the maximum aggregate disk bandwidth can be used, while minimizing communication overhead. The performance of a particular data distribution depends on data access patterns and the data set being accessed. With dynamic data sets, as new data is added to a data set, the data set may need to be redistributed. In some cases, query workloads may have a diverse set of query types (e.g., queries accessing the entire data set, queries focusing in on a particular set of attributes or range of attribute values), which may exhibit a variety of access patterns. In this case, portions of the data set can be replicated and redistributed based on the types of queries. In our framework, a *planner* datalet computes an I/O and communication schedule for the data based on the output of the data replication and distribution algorithm. The goal of the planning phase is to compute a schedule which minimizes data transfer time. The schedule is stored in a container termed a manifest. Each host node maintains an index of the location of all datalets. Finally, a *data mover* datalet is responsible for executing the schedule and transferring the data. Next, we briefly describe two services which make use of the storage service.

Sampling Service: Researchers in the database and data mining fields have increasingly turned to sampling as a means of trading quality for improved response time. Adhoc sampling queries that project the data set along space and time have been used to effectively summarize data for tasks such as network monitoring and anomaly detection. Our sampling service is designed with data mining applications in mind and is capable of supporting a variety of use cases. First, it is capable of handling very large streaming data sets that do not fit in main memory. Second, it is capable of efficiently generating parameterized samples in which both the time range and sample size are variable. Third, it is capable of maintaining and retrieving a sample in parallel, so as to leverage the benefits of parallel I/O on next generation clusters.

The sampling service [32] is composed of three abstract layers; a data pre-processing layer, a storage management layer, and a query processing layer. The data pre-processing layer prepares the data set for subsequent placement in the storage management layer by performing an in-memory randomization of the data set into bins. The storage management layer handles distributed placement and indexing of bins to satisfy future requests. The query processing layer answers sampling queries posed by the user by generating appropriate bin requests to the storage management layer. The sampling service exports an SQL-like interface that supports parameterized sampling queries and can be used by the user or within an application[1]

For situations in which one does not know the desired sample size apriori, progressive sampling has been proposed so as to efficiently converge to a sample size. The idea is to evaluate model accuracy over progressively increasing sample sizes until the gain in accuracy between models over consecutive samples is below a threshold. Our sampling service also supports progressive sampling queries.

Sorting Service: For many large data sets, efficient placement on disk is a critical step towards an effective solution. We build into our framework a sorting service [6] designed to partition large data sets across the cluster. This service is designed to accommodate dynamic updates to the data stores, and data input is assumed to be streaming. Each node can accept new data. Data sets are partitioned into datalets, which are data blocks bounded by the main memory of the host node. This bound accommodates efficient in-memory sorting when needed. When an incoming data object is processed, the receiving node checks its manifest for the mapping be-

[1]We have extended the sampling service to support multidimensional sampling queries [33]. The work will we presented at IPDPS 2006 and hence has not been included in this paper.

Name	Number of transactions
DS1 - T40I15D300K	300000
DS2 - T60I15D300K	300000
DS3 - T70I15D300K	300000
DS4 - T100I15D300K	300000
DS5 - Webdocs.dat	500000

Table 1. Data sets used for in-core itemset mining

	filesize	filesize/ Available main memory	No. of Transactions	Support
Webdocs 500K	0.45GB	2.3	451,000	5%
Webdocs Full	1.4GB	7.3	1,693,000	5%
T2500K	2.6GB	13.26	2,500,000	2%

Table 2. Data sets used for out of core itemset mining.

tween the node and the object. The object is then transferred to the appropriate host. In some cases, the datalet may reach its maximum size. At that time, the block is redistributed across the cluster, and a global communication proceeds to update all manifests.

The particular comparison function for sorting is datalet driven. For transactional data, we implement two datalet manifests. The first method sorts transactions based on its items, as defined in section 2. The second method requires the datalet at the incoming node to maintain a frequency for each unique item in the data set, and sorts the transaction based on frequencies, where the most frequent item is sorted first. This second method requires additional information to be propagated by nodes which receive the original transaction, but it allows for highly efficient mining. Both methods begin with a log distribution, and use frequency data to reform blocks to linear distributions based on parity. It is the latter method which is incorporated for the experiments in Section 4.

4 Preliminary Results

To analyze the proposed optimizations and services, we use a system with an Intel Xeon processor and 4GB of physical memory. The processor runs at 2GHz and has a 4-way set associative 8KB L1 data cache, an 8-way set associative 512KB unified L2 cache, and 2MB L3 cache. The cache line sizes are 64 bytes for the L1 and L2 caches, and 128 bytes for the L3 cache. The system bus runs at 100MHz and delivers a bandwidth of 3.2GB/s.

Cache and Memory Conscious Improvements: We now empirically evaluate the benefits of our cache and memory conscious optimizations. We use four synthetic data sets generated by the IBM Quest Dataset Generator and a real data set called Webdocs, as presented in Table 1. For the synthetic data sets, the naming parameters are the average transaction length T, the average maximal pattern length I, and the number of transactions D. Webdocs [14] was chosen, because most other FIMI data sets are too small. We do realize the limitations of using this synthetic data set generator [34], but truly large real data sets are not readily available. Although the synthetic data sets only have 300,000 transactions, these data sets are very dense, and the FIMI implementations we use are unable to handle a larger number of transactions. Throughout this section, we compare execution time with respect to the fastest known implementation of *FPGrowth* from the FIMI repository [15]. Execution times for this implementation are summarized in Table 3 and we present speedup numbers with respect to these times. Also note that speedup is based on overall execution time, including the time required to create the first tree. In the analysis to follow, we only consider DS3 and DS4. Please refer to [12] for analysis that includes all the data sets from Table 1.

Benefits of Improving Spatial Locality: From Figure 1, it is evident that we achieve a significant performance improvement due to improved spatial locality. Most trials provided between 30% and 60% improvement. When the hardware prefetcher is enabled, there is an additional 10 to 25% speedup. The Pentium 4 processor has a 64 byte cache line size. Therefore, when we traverse the cache-conscious prefix tree, we can fit up to 8 tree nodes in one cache line. In the baseline implementation, each node spans at least 20 bytes, and at most 3 nodes would fit in a cache line. The cache-conscious tree directly improves cache utilization and also facilitates hardware prefetching, because the prefetcher can easily predict simple serial access patterns. We also measured the performance improvements afforded by improving spatial locality and using hardware prefetching, without a reduction in node size. On average, we achieved only a 12% performance improvement. This indicates that the reduction in node size contributes significantly towards the overall performance improvement.

Benefits of Improving Temporal Locality: Path tiling provides a significant improvement over that provided by spatial locality and prefetching. Returning to Figure 1, we see cumulative speedups ranging from 1.9 to 3.2. Some of the benefit of increased spatial locality is tempered due to tiling, but overall, we see significant speedup. By reusing cached content, a large fraction of the misses are eliminated. It can be seen that as we lower support, the impact of tiling greatens. We attribute this to larger prefix tree sizes and greater benefits from temporal locality. [2]

Benefits of SMT: An extant parallelization strategy did not provide more than 3% improvement on an SMT-processor. Therefore, the benefits we see in Figure 1 are

[2]Note that we cannot evaluate path tiling without the improved spatial locality; it is the depth-first ordering of the cache-conscious prefix tree that provides the possibility of using path tiling.

	DS1 (0.20%)	DS2 (0.83%)	DS3 (0.83%)	DS4 (1.33%)	DS5 (10%)
Baseline	192 sec	269 sec	627 sec	3798 sec	949 sec
Cache-conscious	77 sec	80 sec	145 sec	773 sec	220 sec

Table 3. Execution Time Comparison

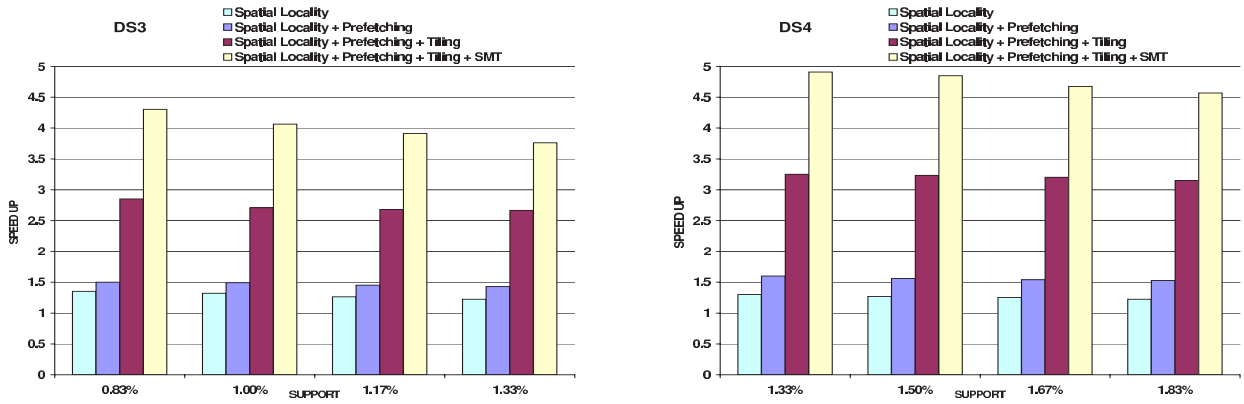

Figure 1. Speedup on DS3 and DS4

due to the reuse of cached data between threads, and thus, improved ILP. *The use of SMT gives us an overall speedup of up to 4.8. Cumulatively, our optimizations increase L1 hit rate to 94% (from 89%) and L2 hit rate to 98% (from 43%).*

I/O Conscious Improvements: We evaluate our algorithmic modification for frequent itemset mining on out of core data sets using a 900 MHz Pentium III with 256MB RAM and an 80 GB IDE hdd. In practice, this machine provides at most 200 MB of main memory to any one process. The data sets and supports are provided in Table 2. Dataset T2500K is a synthetic data set, created with the generator used earlier. The other two data sets can be found in the FIMI repository [14].

Benefits of I/O Conscious Transaction Preprocessing: We present the benefits of preprocessing the transaction data set. As mentioned, our technique is to partition the data set into sections where each transaction in block$_i$ sorts before all transactions in block$_{i+1}$. We then perform an in memory sort on the partitions during the tree building phase. In Figure 2, we display the processing time up until mining of the first tree begins. This is the total time to build the main prefix tree on disk. Base code is our code executing on the originally ordered data set, and Approx Sorting is the code which first executes our partitioning algorithm. Note that this graph *includes all partitioning and sorting times.* By preprocessing the transactions, the total time to build the first tree is improved by at least a factor of 10. This clearly shows that I/O conscious algorithms can vastly reduce the number of page faults in the virtual memory system. Furthermore, executing intelligent partitioning and then sorting these small partitions in main memory is significantly faster than full sorting the transaction data set. For example, our code requires 139 seconds to partition sort webdocs500k, whereas full sorting using a typical disk-based merge sort requires 25351 seconds. This is in part due to the fact that we presume the distribution of the data set to be log-based. We are investigating other disk-based sorting techniques. Also, if this assumption is incorrect, it will be corrected in a subscan because there will be a disproportionate amount of transactions in odd numbered partitions. We have used this algorithm on every database in the FIMI Repository, with very good results. The reader is directed to our Technical Report [5] for details.

Figure 2. Preprocessing times for three out of core data sets.

Execution Time Improvements due to the Sampling Service: The goal of our software sampling infrastructure is to improve application performance. To eval-

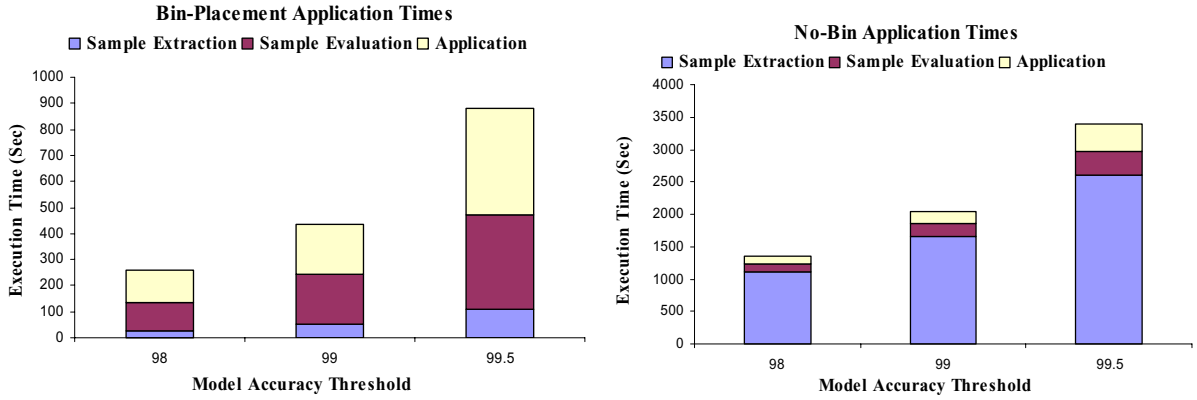

Figure 3. (a) Improvements in application runtime due to bin placement (b) Application runtime without bin placement

uate this requirement, we implement a parallel frequent pattern mining algorithm for network intrusion detection. The data set for this experiment is 20GB, and the support is 0.1%. Our algorithm uses progressive sampling to determine the optimal sample size required. In progressive sampling, successive samples are evaluated against the previous sample using a similarity metric. When the similarity between several consecutive samples is above a particular threshold, the accuracy of the sample is considered optimal. The application is then executed on that sample. For our experiment, we employ the accuracy model described in [24]. The similarity metric for two samples d_1, d_2 is defined as follows:

$$Sim(d_1, d_2) = \frac{\sum_{x \in A \cap B} max\{0, 1-\alpha|sup_{d_1}(x) - sup_{d_2}(x)|\}}{\|A \cup B\|}$$

A and B are frequent itemsets for d_1 and d_2, and $sup_{d_1}(x)$ is the frequency count of x in d_1. We set the scaling factor α to 1. In this experiment, sampling is initiated at 0.5% and proceeds in small increments until two successive samples are within a predetermined similarity threshold. Figure 3 depicts the results based on three thresholds: 98%, 99%, and 99.5%. In all three cases, it is clear that our bin placement strategy greatly improves I/O times. In fact, I/O is no longer a significant component when evaluating execution time. When using a similarity threshold of 98%, the mined sample is 3% of the total data set, or 600 MB. Accumulative application execution time is *over 5 times faster* using bin placement. At 99%, the speedup is 4.5, and at 99.5% the speedup is 3.8. We conclude that an improved sampling infrastructure can provide a dramatic improvement on application performance.

5 Related Work

Several run-time support libraries and parallel file systems have been developed to support efficient I/O in a parallel environment [10, 19, 23, 30, 22, 26, 28, 29, 30]. A simple strategy to improve the I/O bandwidth of sampling algorithms is to use a parallel file system such as PVFS [8] which transparently partitions a file across several storage nodes for fast parallel retrieval. Such systems mainly focus on supporting regular strided access to uniformly distributed data sets. Thus, a downside of using this strategy is that the file system may not be able to balance the load for a sample request optimally as it is not aware of the data distribution associated with the sample requests beforehand. For this reason, we do not use a parallel file system; rather we leverage parallel disks by explicit data placement and retrieval. Data declustering is the process of distributing data blocks among multiple disks (or files). On a parallel machine, data declustering can have a major impact on I/O performance for query evaluation [11]. Numerous declustering methods have been proposed in the literature. Recently, there has been considerable research activity pertaining to the design of data stream management systems. An overview is presented in a recent survey paper by Babcock *et al.* [3]. Existing stream management systems build upon the assumption that all the required data summaries can be maintained in main memory and do not deal with data placement on disks.

Several works address out of core itemset mining. AFOPT [21] is a top-down approach to itemset mining with a prefix tree, where the least frequent item sorts first. This method has the benefit that subtrees are used one at a time; it does not build a prefix tree for the whole data set. However, it makes the assumption that the frequent one item trees fit in main memory. We propose to mine large data sets, where this assumption will not hold. Grahne and Zhu developed a recursive projection technique [16] for maintaining structures in main memory when employing the FPGrowth algorithm. The technique makes a significant number of full database scans when the original data set is large and support is rela-

tively low. This is because the algorithm has no mechanism to support projected data sets larger than main memory; instead it must recurse into the search space until the projected data set fits in main memory. This also incurs the additional cost of projecting candidates which may not be frequent.

6 Conclusion

In this work, we present a top-down view of our systems framework for next generation data analysis centers which will accommodate the dynamic and large scale nature of future workloads. We show that by restructuring computation, we can improve cache, memory, and disk level performance. The restructured algorithm can then make use of the proposed services to provide a highly scalable solution. Finally, we illustrate these ideas with a case study in frequent pattern mining, and through empirical analysis, we show that significant speedups are achievable.

References

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 1994.

[2] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 1995.

[3] B. Babcock*et al.* Models and issues in data stream systems. In *ACM Symposium on Principles of Database Systems*, 2002.

[4] S. Bay and M. Schwabacher. Mining distance-based outliers in near linear time with randomization and a simple pruning rule. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining*, 2003.

[5] G. Buehrer, S. Parthasarathy, and A. Ghoting. Out of core itemset mining on emerging 64 bit architectures. Technical Report OSU-CISRC-1/06-TR16, The Ohio State University, January 2006.

[6] G. Buehrer, S. Parthasarathy, A. Ghoting, X. Zhang, S. Tatikonda, T. Kurc, and J. Saltz. A sorting service for next generation data analysis centers. Technical report, The Ohio State University, January 2006.

[7] M. Cannataro and D. Talia. Knowledge grid: An architecture for distributed knowledge discovery. *Communications of the ACM*, 2003.

[8] P. Carns, W. Ligon, R. Ross, and R. Thakur. Pvfs: A parallel file system for linux clusters. In *Proceedings of the Annual Linux Showcase and Conference*, 2000.

[9] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The Data Grid: Towards an Architecture For the Distributed Management and Analysis of Large Scientific Datasets. *Journal of Network and Computer Applications*, 2001.

[10] P. F. Corbett and D. G. Feitelson. The Vesta parallel file system. *ACM Trans. Comput. Syst.*, 14(3):225–264, Aug. 1996.

[11] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Commun. ACM*, 35(6):85–98, June 1992.

[12] 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.

[13] A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, A. Nguyen, Y. Chen, and P. Dubey. A characterization of data mining algorithms on a modern processor. In *Proceedings of the ACM SIGMOD Workshop on Data Management on New Hardware*, pages 1–5, 2005.

[14] B. Goethals and M. Zaki. Advances in frequent itemset mining implementations. In *Proceedings of the ICDM workshop on frequent itemset mining implementations*, 2003.

[15] 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.

[16] G. Grahne and J. Zhu. Mining frequent itemsets from secondary memory. In *Proceedings of the Fourth IEEE International Conference on Data Mining (ICDM'04)*, pages 91–98, 2004.

[17] 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.

[18] IntelCorp. *Intel Hyper-Threading Technology*, 2004.

[19] D. Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74, Nov. 1994.

[20] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proceedings of the International Conference on Data Mining (ICDM)*, 2001.

[21] 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.

[22] J. M. May. *Parallel I/O for High Performance Computing*. Morgan Kaufmann Publishers, 2000.

[23] N. Nieuwejaar and D. Kotz. The Galley parallel file system. pages 374–381. ACM Press, May 1996.

[24] S. Parthasarathy. Efficient progressive sampling for association rules. In *Proceedings of the International Conference on Data Mining*, 2002.

[25] 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.

[26] J.-P. Prost, R. Treumann, R. Hedges, B. Jia, and A. Koniges. MPI-IO/GPFS, an optimized implementation of MPI-IO on top of GPFS. Nov. 2001.

[27] J. Quinlan. *Programs for Machine Learning*. Morgan Kaufmann, 1993.

[28] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing '95*, Dec. 1995.

[29] X. Shen and A. Choudhary. A distributed multi-storage i/o system for high performance data intensive computing. In *International Symposium on Cluster Computing and the Grid (CCGrid 2002)*, May 2002.

[30] R. Thakur, A. Choudhary, R. Bordawekar, S. More, and S. Kuditipudi. Passion: Optimized I/O for parallel applications. 29(6):70–78, June 1996.

[31] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 1995.

[32] H. Wang, S. Parthasarathy, A. Ghoting, S. Tatikonda, G. Buehrer, T. Kurc, and J. Saltz. Design of a next generation sampling service for large scale data analysis applications. In *Proceedings of the 19th Annual International Conference on Supercomputing (ICS05)*, pages 91–100. ACM Press, 2005.

[33] X. Zhang, T. Kurc, J. Saltz, and S. Parthasarathy. Design and analysis of a multidimensional sampling service for large scale data analysis applications. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[34] Z. Zheng, R. Kohavi, and L. Mason. Real world performance of association rule algorithms. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2001.