Design of a Next Generation Sampling Service for Large Scale Data Analysis Applications

H. Wang[1], S. Tatikonda[1], A. Ghoting[1], G. Buehrer[1], S. Parthasarathy*[1,2], T. Kurc[2], and J. Saltz[1,2]
Department of Computer Science and Engineering[1]
Department of Biomedical Informatics[2]
The Ohio State University, Columbus, OH 43210, USA

Abstract

Advances in data collection and storage technologies have resulted in large and dynamically growing data sets at many organizations. Database and data mining researchers often use sampling with great effect to scale up performance on these large data sets at a small cost to accuracy. However, existing techniques often ignore the costs of computing a sample. This cost is often linear in the size of the dataset, not the sample, which is expensive. Furthermore, for data mining applications that leverage progressive sampling or bootstrapping based techniques, this cost can be absolutely prohibitive, since they require the generation of several samples.

To address this problem, we present a solution in the context of a state of the art data analysis center. Specifically, we propose a scalable sampling service that supports sample generation with cost linear in the size of the sample. We then present an extremely efficient parallelization of this service. Our solution leverages high speed interconnects (e.g. Myrinet, Infiniband) for parallel I/O operations with pipelined data transfers. We export an interface that supports both ad-hoc SQL-like querying for database applications as well as a stand-alone service for data mining applications. We then evaluate our work using queries abstracted from a network monitoring and analysis application, which uses both database and progressive sampling queries. We demonstrate that our implementation achieves good load balancing and expected speedup (up to an order of magnitude when compared to extant approaches).

Keywords*: Sampling, Parallel I/O, Data Centers, Data Mining*

1 Introduction

Over the last decade, research in science and engineering has become increasingly data-driven. This trend is fueled by advances in sensor and computing technologies coupled with inexpensive spinning storage. With the help of grid computing

*Email: srini@cse.ohio-state.edu

and advanced sensors, large-scale and dynamically growing datasets are becoming the norm rather than the exception in many application domains.

Storage platforms with large storage space can be relatively easily and inexpensively built to host vast volumes of streaming data. However, given disk access and network overheads, accessing and managing this data efficiently is difficult. Analyzing or mining the data is typically an iterative process and requires multiple passes over the data, which may be prohibitively expensive. These problems are exacerbated when data is streaming in at a high rate and needs to be processed and mined in close to real time. Consequently, the need of the hour is a *scalable framework for fast and efficient storage and processing of dynamic data*.

Researchers in the database and data mining fields have increasingly turned to sampling, as a means of trading off quality for improved response times [16, 28, 32]. Specifically, ad-hoc sampling queries that project the dataset along space and time have been used to effectively summarize data for tasks such as anomaly detection and network monitoring. Progressive sampling [27] and similar techniques can mitigate the issues related to the quality of the sample, but they make it imperative that sample generation be inexpensive. Despite the recognized importance of sampling in data mining and data analysis, very little work has been done in making the process of sample generation efficient. As noted by Provost and Kolluri [28], *"most discussions on sampling assume that producing random samples efficiently from large datasets is not difficult. This is simply not true."*. In fact most algorithms require O(N) time where N represents the size of dataset and not the sample size (S).

A simple example can illustrate this problem. Assume one has to compute a sample from a transactional database. Assume further that the average size of the transaction is roughly 40 bytes and a disk block is 4 kilobytes. There are roughly 100 records per block. Sampling at 1% or higher amounts to touching every disk block. Since disk I/O will dominate the overall costs, *each sampling pass is equivalent to one scan of the entire database*. An architecture to efficiently support such sampling queries for next generation data analysis applications is the focus of this work. The challenges are daunting.

First, in emerging application domains like networking intrusion detection, datasets are often very large and highly dynamic or streaming in nature. Samples of network transactions are often used to build a model for normal behavior [19]. Significant deviations from this model of normality are used to detect intrusions. Both the sampling and the detection must occur in real-time. Thus, it is imperative that one have the ability to produce samples in an online fashion. Most existing work in sampling assumes that the dataset is static and that the desired sample can be maintained in main memory.

Second, the iterative nature of many data analysis algorithms requires the ability to efficiently generate variable size samples. Certain applications demand sample size be progressively increased, until some quality criterion has been satisfied [25, 27]. Furthermore, the sample may not be a random sample over the entire stream, but rather a sample of a certain historic time range. Thus, we need the ability to generate samples in which both the size of the sample and the time range of the sample are variable, i.e., a parameterized sample. Existing work in this area has only looked at the maintenance of fixed size samples. New schemes to support interactive sampling queries with a variable size and time range are needed.

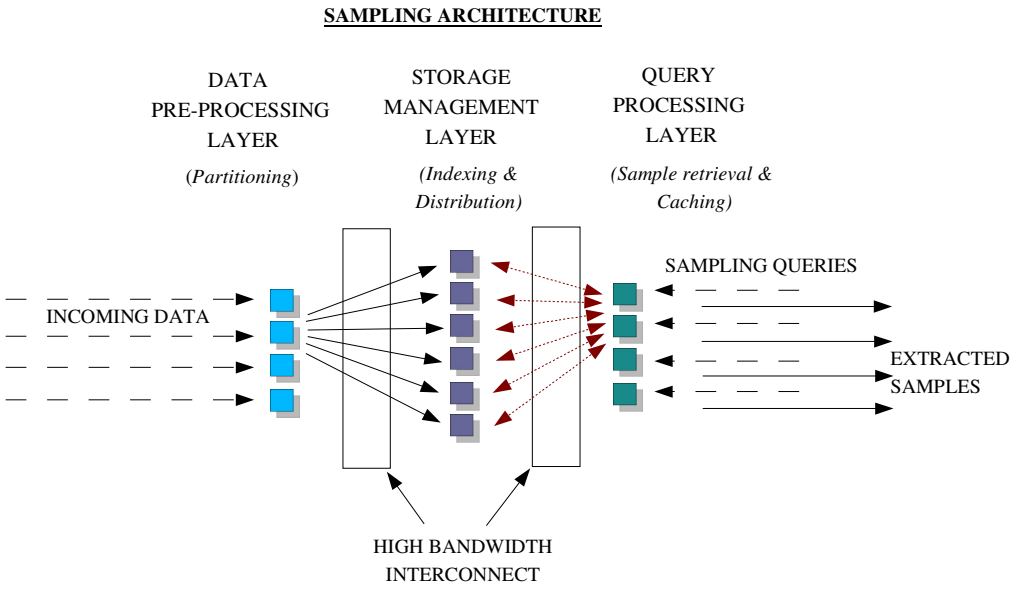

Figure 1. Sampling Architecture

Third, I/O is the primary performance bottleneck through the sample generation process. A natural candidate to improve I/O performance is to parallelize the sample maintenance and retrieval phases over a cluster of nodes. With advances in networking technology, high bandwidth interconnects such as those provided by Infiniband and Myrinet make it possible to transfer data over a network at rates exceeding disk I/O bandwidth. Consequently, this allows us to pipeline data transfer over a network as it is being read from a disk. Efficient solutions that leverage such technologies effectively for parallel sample maintenance and retrieval are needed.

In this paper, we present the architecture of a sampling service to address the above challenges. Our sampling architecture is composed of three abstract layers: a data preprocessing layer, a storage management layer, and a query processing layer (Figure 1). The data preprocessing layer prepares the dataset for subsequent placement in the storage management layer by performing an in-memory randomization of the dataset into bins. The storage management layer handles distributed placement and indexing of bins for future sample requests. The query processing layer answers sampling queries posed by the user by generating appropriate bin requests to the storage management layer. Such an abstraction can be mapped to a physical setting depending on organizational capabilities. For example, the physical architecture can consist of a compute cluster, a storage cluster, and a memory cluster, connected through a high bandwidth interconnect, and the data preprocessing layer, the storage management layer, and the query processing layer can be mapped on to these clusters respectively. Such a consolidation of resources permits the derivation of high performance for sampling and other data analysis tasks in which subtasks that are compute, memory, or disk I/O intensive can be identified. In the absence of such dedicated resources, the three layers can be mapped to nodes within a single cluster connected with a high bandwidth interconnect. The characteristics of this layered architecture make it viable to be mapped as end-nodes in the Data Grid [5] and to form the building blocks for

next generation data centers with mass storage systems (such as NPS[1]) serving very large datasets. We expect that such data centers will be increasingly ubiquitous along with increasing cost-effectiveness of spinning storage and that such centers will be an essential component of future Data-Grid and Knowledge-Grid architectures [5, 2].

Specifically, in this paper, we make the following contributions:

- We present the architecture of a sampling service to address the above challenges.
- We present an intelligent online data placement strategy to enable efficient querying and sample generation on dynamic datasets.
- We extend our placement strategies for efficient parallel sample maintenance and retrieval in a cluster setting with high bandwidth network substrates.
- We have designed and implemented a query service to support sampling queries with variable size and time ranges.
- We evaluate the effectiveness of our approach on PC clusters with a commodity network and advanced state-of-the-art Infiniband and Myrinet networks.

The following section presents related work. Section 3 describes our proposed sampling infrastructure. We describe the query interface in Section 4, followed by experiments in Section 5, and finally conclusions.

2 Related Work

Given the importance of sampling, it is surprising that little work has been done on how to improve the performance of generating a sample from out-of-core datasets. Researchers have looked at generating samples over an in-memory database and an excellent survey is provided by Olken and Rotem [23]. The assumption here is that the data set is static and samples are assumed to fit in main memory. Reservoir sampling [33] was proposed to maintain a true fixed size random sample of a data stream at any given instant. From the perspective of data analysis applications, the drawback here is that the algorithm assumes that the sample fits in main memory, and the sample request has a fixed size. The sample time range is also always fixed – from the beginning of the stream to the current point in time. A sampling scheme to maintain large samples on disk has been proposed by Chris Jermaine *et al.* [15]. However, using this approach one cannot generate a variable sized sample over a variable time range. This strategy cannot be trivially extended to parallel disks. Moreover, the approach does not generate a true random sample, but a biased one, as noted by the authors.

There has been a lot of work on the use of sampling for data analysis applications. Sampling has been successfully used for association rule mining [32], clustering [10] and several other machine learning algorithms. These algorithms do not know the desired sample size a priori. Progressive sampling [27] has been proposed for these algorithms so they can efficiently

[1] http://www.netezza.com

converge to the desired sample size. The idea is to evaluate model accuracy over progressively larger samples until gain in accuracy between consecutive samples falls below a certain threshold. More recently, progressive sampling for mining association rules [25] has been proposed, which shows that with efficient sample-generation schemes such approaches can be a viable approach for handling streaming datasets.

Several run-time support libraries and parallel file systems have been developed to support efficient I/O in a parallel environment [6, 17, 22, 31, 20, 26, 29, 30, 31]. A simple strategy to improve the I/O bandwidth of sampling algorithms is to use a parallel file system such as PVFS [3] 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 datasets. 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 [8]. Numerous declustering methods have been proposed in the literature. Grid-based methods [4, 11, 12] have been developed to decluster Cartesian product files, while graph-based methods [13, 18, 21] are aimed at declustering more general multi-dimensional datasets. These methods assume a static data set and are designed to improve I/O performance for data access patterns generated by multi-dimensional range queries. A range query specifies the requested subset of a dataset via a bounding box in the multi-dimensional attribute space of the dataset. All the data elements whose attribute coordinates fall into the bounding box are retrieved from disk. The approach proposed in this paper is targeted at dynamic data sets and queries that specify the desired data subset by a range query and a user-defined sampling amount.

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.* [1]. 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. If a situation arises in which the data summary does not fit in main memory, the summary undergoes lossy compression so as to fit in main memory. Such a lossy approach does not work for applications such as network intrusion detection. In these applications one needs the ability to retrieve both a summary of the dataset as well as a precise representation of dataset.

3 Sampling Architecture

Our sampling architecture targets both static and dynamic or streaming datasets. The issues involved in building a sampling architecture to handle dynamic and streaming datasets are far more challenging when compared to those involving static datasets. Therefore, we will detail our sampling architecture in the context of these datasets, although the schemes are equally applicable to static ones. Three components of our architecture are detailed next.

3.1 Component 1: Data Stream Preprocessing Layer

Current methods for sample extraction can easily touch all disk blocks, even for small sample requests over out-of-core datasets. Our solution to the problem relies on the notion of stream windows and bins. A stream window corresponds to a user defined number of transactions (n) on the incoming data stream. Each window of n transactions is further partitioned into a user defined number of bins (k) using a randomization function. This function ensures that each bin can be viewed as mutually exclusive sample of the stream window. Essentially, each incoming transaction is assigned to one of the k bins with probability proportional to the size of the bin. Placement within a bin is ordered according to arrival by default, preserving temporal ordering. However, a user-specified hash function may be specified to order transactions within a bin. Such a hash-based ordering scheme can be leveraged to support stratified or periodic sampling, which has been used to effectively model and mine network traffic [24]. This process is repeated for every n transactions. Each bin can be of a fixed size. By default, however, bin size varies, following a geometric progression (n/2, n/4, n/8, etc.). The benefits of using a geometric progression are detailed in 3.3.

As motivated in the introduction, data analysis sampling queries typically involve sample requests of variable sizes. Geometrically progressing bin sizes allow efficient and quick processing of such multi-resolution sampling query requests. Clearly the flexibility offered by variable sized bins can enable more efficient sample extraction for both queries with small sample sizes and queries with large sample sizes. Each bin is placed contiguously on disk. This ensures that when extracting a sample if we choose a bin of size roughly equal to the sample size, the total cost of generating the sample will be proportional to the sample size, or in the worst case proportional to the bin size. If a sample request spans multiple stream windows, then samples from each one can be appropriately combined to derive the requested sample.

3.2 Component 2: Storage Management Layer

3.2.1 Data Distribution and Placement

When we have multiple disks we adopt the following round robin placement strategy. Assume we have m parallel disks. Let bin(r, i) denote bin i within stream window r. Bin distribution is an onto mapping f from the bin set $\{0, 1, ..., k-1\}$ to the parallel disk set $\{0, 1, ..., m-1\}$ such that bin with id (r, i) will be assigned to node $f_r(i)$ defined as follows: Given a mapping f_{r-1} for window $(r-1)$, f_r will be determined by formula

$$f_r(x) = [f_{r-1}(x) + 1]\%m \tag{1}$$

Since we are defining f in an iterative way, f_0 is needed for the first bin placement. One can simply use:

$$f_0(i) = i\%m. \tag{2}$$

If bin $(r, i)(0 \leq i < k)$ is assigned to disk y, then bin $(r+1, i)$ will be assigned to the next disk in order, $(y+1)\%m$. This bin distribution strategy has the following two properties:

1. $f_i = f_{i+m}$; after every m windows, $m \times n$ elements will be equally distributed onto m disks.
2. Given m disks, any bin with id (r, i) will be on disk k if and only if $(r + i - k)\%m = 0$.

Property 2 follows from the fact that $f_r(i) = [f_{r-1}(i) + 1]\%m$.

$$f_r(i) = (f_{r-1}(i) + 1)\%m = (f_{r-2}(i) + 2)\%m = ... = (f_0(i) + r)\%m = (i\%m + r)\%m = (i + r)\%m = k \quad (3)$$

Thus, given a bin identifier we can determine in constant time on which disk it resides. An example data placement is shown in Figure 2(a).

3.2.2 Time-based Index Structure

The indexing scheme we implement is a simple binary search mechanism on top of a dynamic array structure. The nodes of the array contain $(r, t_{r\_start}, t_{r\_end})$. The first element of the tuple is the stream window identifier. The second and third elements define the time interval of the window. Since all bins are contiguous on disk and for the multiple disk case the partitioning algorithm is fixed and dependent on r and i, we do not need to store on which disk the data is located. It can be computed on the fly.

Operations associated with our index are efficient. Computing the stream window R that contains the X^{th} transaction is a constant time operation. We simply evaluate $R = ceiling(X/n)$. Computing the stream window that contains transactions starting at time T requires O(logC) time, where C is the number of stream windows stored to-date. This is accomplished via a binary search based on the time information stored in each node [7].

Using a more complex index structure, such as an R-tree or B+ tree, is not warranted. Updates and deletions are typically made in batch and are in FIFO order. Thus, insertions will take constant time; we simply update the state information within the dynamic array and add the node. Similarly, deleting entries requires constant time. Note that locating all entries within a stream window that is greater than or less than a particular X or T incurs an additional overhead, which is bounded by n in the worst case.

3.3 Component 3: Query Processing Layer

3.3.1 Sample Construction

We outline our approach to sample construction using a simple example. As in section 1, assume k is the number of bins. Under a geometric schedule, the bin sizes are $n/2, n/4, \ldots, n/2^{k-1}, n/2^{k-1}$, totaling n (the stream window size). Suppose

we wish to compute a $BetweenT$ sample query denoted as $(t_i, t_j, x\%)$, which requests a sample equal to x% of the data between time t_i and t_j. We initially assume that the range t_i to t_j exactly covers c stream windows, such that $c\%m = 0$ where m is the number of disks.

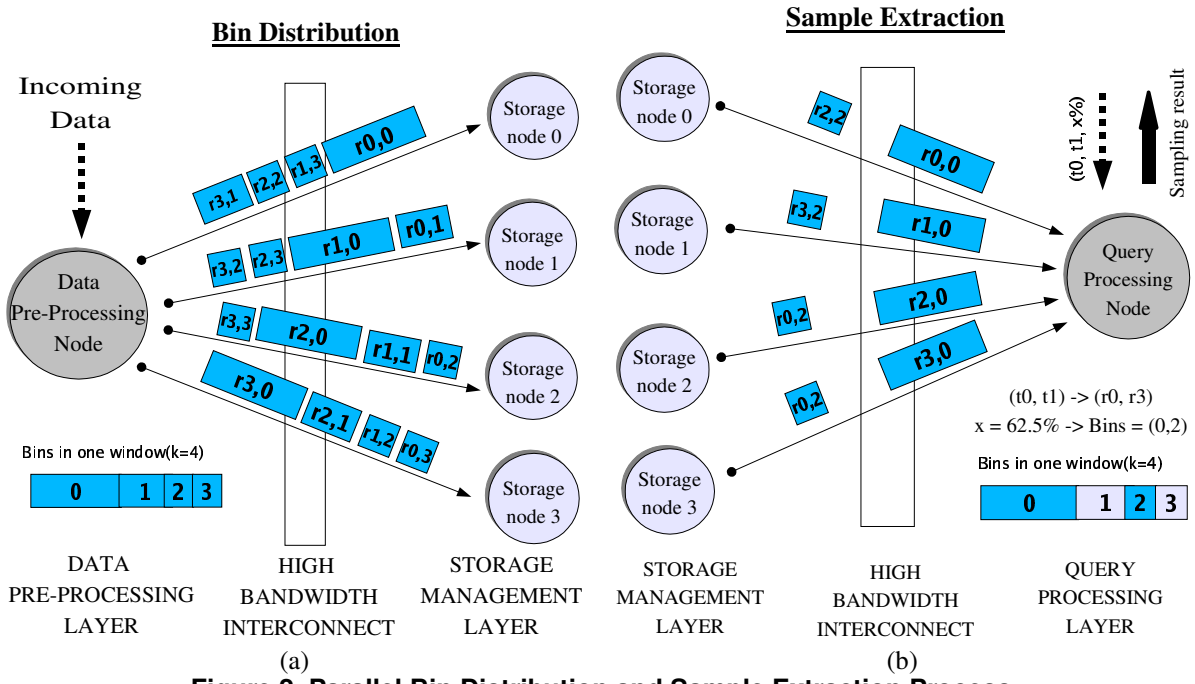

Figure 2. Parallel Bin Distribution and Sample Extraction Process

Our approach partitions this query into smaller, atomic subqueries, each of which can independently operate on a single window. A union of the results from these atomic queries corresponds to a sample satisfying the original query request. For each atomic query we choose the bins so that the minimum number of disk blocks have to be touched. For example, if we have four bins per stream window, and if the sample size requested within an atomic query is between $n/8$ and $n/8 + n/16$, then the query will be answered using the bins containing $n/8$ and $n/16$ elements, respectively. On the other hand, if the sample size requested is between $n/8 + n/16$ and $n/4$, then the sample is generated using the bin containing $n/4$ elements. An important point to note is that whatever the bin schedule within a stream window, under our simplifying assumption ($c\%m = 0$) and along with the placement strategy outlined in Figure 2(a), it is easy to show that the load among the m nodes will be perfectly balanced. An example of this is shown in Figure 2(b) for $m = k = 4$, $x = 62.5\% = 1/2 + 1/8$.

Further, it can be easily shown that the maximum overhead (extra disk blocks beyond the desired sample size) for an atomic query is $((n/2^{k-1})/b) - 1$ where b is the size of a disk block. In order to minimize this overhead, we must either reduce n or increase k. There is a tradeoff involved in varying n and k. If we reduce n or increase k beyond a certain value, we may potentially make the smallest bin smaller than one disk block, which will introduce an overhead in accessing the bins themselves. Another disadvantage of reducing n is that for the same stream we need to use more windows. Processing smaller windows will increase the placement and indexing overhead. It is therefore important to find the proper n and k for a specific system.

Now let us relax the assumption on our query, i.e. it can partially span windows on each end, and it need not span a multiple of the number of nodes (disks) in the system. From a workload balancing perspective this may introduce unbalance, however, this is bounded by n for each side of the time range. This indicates that at most $2n$ elements are not equally collected from m disks. One way to lessen this problem is by reducing n, as described earlier. For large sample sizes ($SampleSize >> n$) this unbalance will not be significant. Another minor aspect to consider is the necessity to purge entries at each end window that do not satisfy the time range requirements. Thus for end windows the overhead for the atomic query need not be bounded by $((n/2^{k-1})/b) - 1$ as it would depend on the distribution of the data.

The principal advantage of this sample extraction approach is its simplicity and concomitant efficiency. A potential limitation with this approach is that one may loose close temporal correlations within the stream window when we partition it into k bins. However, this is not a problem for many data mining and data retrieval operations. For example, for data mining tasks such as clustering, classification and association mining [14], temporal correlations in the data set do not matter. In fact even for mining sequential patterns or a time series [14], there is an efficient workaround. As long as the same time series or sequential pattern is always hashed to the same bin, our approach will work fine. Moreover, as the records in each disk block (or bin) will be ordered, one can still support efficient join operations (or intersection operations) [34], the mainstay of many frequent itemset algorithms.

3.3.2 Limiting Bias Across Related Queries

Our approach will produce unbiased samples for unrelated queries. This is guaranteed by our initial randomization strategy. However if a user is working with a certain subset of the data, he/she may be interested in evaluating performance over multiple independent samples (say X). In that case, our approach will result in biased samples (especially for large X) in the sense that two elements within the same bin are more likely to appear in a sample than two elements across bins within a stream window. This is especially true of smaller sized sample requests.

To limit this bias we introduce the notion of a bias threshold b which can take a value from 0 to $1 - \frac{1}{k}$. $1 - b$ corresponds to the probability of selecting a particular bin for inclusion in the sample. Essentially the bias limiting algorithm randomly picks bins that span the stream windows in the desired time range with probability $1 - b$. Next, atomic subqueries for each bin based on the desired sample size and bin size are created. For example, if the desired sample size is $S/8$ and the randomly picked bins have sizes $S/2$ and $S/4$ respectively, then we will select $2S/24$ elements from the bin containing $S/2$ elements and $S/24$ elements from the other bin. The union of the atomic answers to these atomic queries constitutes the desired bias limited sample.

If we set $b = 0$, all the bins will be selected. This provides an unbiased sample, but no performance improvement over traditional sampling methods. For $b = 1 - \frac{1}{k}$, we will select only one bin in the stream window; our method will be biased, but the performance gain is significant. The selection of b will depends on X and the requirements of the user.

3.3.3 Benefits from Caching

To improve the performance of our method one can leverage data caching. The caching strategy keeps track of all bins located in memory (collective memory of multiple nodes). A simple modification to our index structure (a flag bit) allows us to maintain state of cached bins. The runtime system starts processing the query using the bins that are in data cache; at the same time, it initiates asynchronous I/O operations to retrieve the remaining bins from disk.

We do not provide more details on the caching as it is beyond the scope of this paper. Currently, we are in the process of developing various cache replacement policies. Variable bin size complicates cache replacement strategies. Note that in terms of replacement policies, random replacement is likely to be the method of choice (as opposed to LRU which is the default OS mechanism) given that we are looking at statistically independent samples. We are also developing a strategy where bins are partially stored in memory and disk (for large bins).

4 Implementation to Support Sampling Queries in Network Intrusion Detection Applications

Our sampling service consists of a set of methods for stand-alone applications to interact with it, an SQL-like interface for expressing ad-hoc sampling queries, and runtime support for online data placement and query execution on a parallel machine.

We have identified several types of sampling queries in the context of network intrusion detection applications and data analysis tasks that use sampling as a preprocessing step (e.g. progressive sampling for classification or association rules).

1. **LastT** (**LastX**) Queries: Queries on the network transaction stream which return a sample of the data flowing into the system over the last T seconds (X transactions). Such a query could be useful for detecting Denial-Of-Service and Probe attacks.

2. **BetweenT** (**BetweenX**) Queries: Queries on the historical network transaction data that return a sample of the data between a given time interval (transaction interval or logical time interval) in the past. Such queries can be useful for modeling normal behavior in the system. The rationale being that samples are unlikely to contain intrusions and therefore are good models for what is normal. Note that LastT or LastX queries may also be used for this purpose.

3. **Percent vs. Bounded Sample Size** Queries: When computing a sample query using any of the above methods one may desire to specify fixed sample size (e.g. 2,000 transactions) or a percentage of the data to be sampled (e.g. 10% of all transactions in the domain of interest). Such a query can be used in conjunction with the above queries.

4. **Progressive** Queries: Progressive sampling is very useful for empirically determining which sample size works best for a given problem. Such queries may be used in conjunction with the queries listed above. Typically it is applicable only to historical data (BetweenT) using percent sampling.

SELECT **SAMPLE 5%** X.A, X.M
FROM X
BETWEEN TIME T1 and T2
(a) BetweenT Query

SELECT **SAMPLE 5000** *
FROM X
LAST 100000 **TRANSACTIONS**
(b) LastX Query

SELECT **PSAMPLE**(5,(1%,5%,10%,20%,40%)) *
FROM X
LAST 1000000 **TRANSACTIONS**
(c) Progressive Sample Query

Figure 3. Sample Queries

Based on these query types, our current implementation supports the queries of the form:

SELECT SAMPLE x% | **PSAMPLE**(n,(x_1%,...,x_n%))

$< attributes >$

FROM D

BETWEEN TIME [RANGE] | **LAST** k **TRANSACTIONS**

The syntax structure for some of the queries listed above are detailed in Figure 3. The first query will retrieve a 5% sample of the data in X inserted between time T1 and T2. It will only retrieve data for the attributes A and M. The second query will retrieve a fixed sample of size 5000 from X over the last 100,000 transactions. The last query will retrieve a progressive sample of X of varying sizes. The samples returned will be 1%, 5%, 10%, 20% and 40% over the last 1,000,000 transactions.

5 Experiments

5.1 Setup:

The experiments were conducted on three different clusters to measure performance across three different interconnects.

- C1-FastEthernet: This cluster consists of 16 Intel Pentium III 900MHz single-CPU nodes. Each node in this cluster has 512 MB memory and one 100GB disk. We measure the application level I/O bandwidth to be 25MBytes/s from each disk. The nodes in this cluster are connected using a 100Mbps switch.

- C2-Infiniband: This cluster consists of 32 2.4GHz Intel Pentium 4 Xeon processors. Each node in this cluster has 4 GB memory and one 80GB disk. We measure the application level I/O bandwidth to be 23MBytes/s from each disk. The nodes in this cluster are connected using a 10Gbps Infiniband interface.

- C3-Myrinet: This cluster consists of 32 900MHz Intel Itanium 2 processors. Each node is this cluster has 4GB memory and one 80GB disk. We measure the application level I/O bandwidth to be 22MBytes/s from each disk. The nodes in this cluster are connected using a Myrinet 2000 interface.

The experiments were performed using datasets that model a real network transaction monitoring and analysis application. Each dataset has 1000 distinct items with 10 items per transaction. Each item is represented by an integer, and thus each

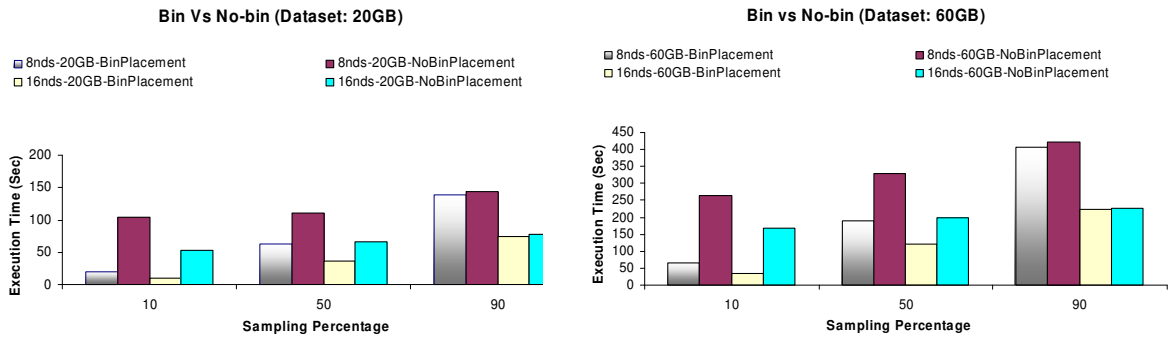

Figure 4. Execution time for sample extraction when using the BinPlacement strategy vs NoBinPlacement strategy

transaction spans 40 bytes. Furthermore, each transaction is time stamped with a 4 byte value. This time stamp indicates the time at which a particular data element enters the system and is generated randomly using an exponential distribution. *Through extensive performance analysis we determined optimum values for* n *and* k *to be* 32 *and* 16 *respectively. Unless otherwise noted, we will use these values for* n *and* k *in all our subsequent experiments.* Although the experimental results are obtained using synthetic datasets, the results will hold for real datasets, as our sampling strategies do not depend on the values in each transaction. Our implementation is written in C, and MPI is used for message passing.

5.2 Benefits of data placement through binning:

These set of experiments compare the performance of our strategy (labeled "BinPlacement" in the graphs) with that of an extant strategy that does not use the concept of bins (labeled "NoBinPlacement" in the graphs). When using the NoBinPlacement strategy (the default state-of-the-art sample generation strategy), a stream window is partitioned into P blocks, where P is the number of processors, and the contents of each block are stored on consecutive disk locations. When a sampling query is executed, the system tests each element for sample membership, based on the desired random distribution of the sample. In order to improve performance, rather than reading one element at a time during the inclusion test, we read a block of the data into a dedicated buffer and test for inclusion in this buffer. As seen in Figure 4, our BinPlacement strategy achieves significant speedup (up to factor of 5) compared to the NoBinPlacement sampling strategy for small sample sizes. This is attributed to the fact that the number of disk blocks that need to be touched in the former is proportional to the size of the sample. For larger sample requests (sampling percentage 90) the BinPlacement strategy marginally outperforms the NoBinPlacement strategy as the former touches a fractionally smaller number of disk blocks compared to the latter that needs to touch all the disk blocks. In Figure 4 we also note that execution time improvements scale with dataset size as we move from a 20GB dataset to a 60GB dataset

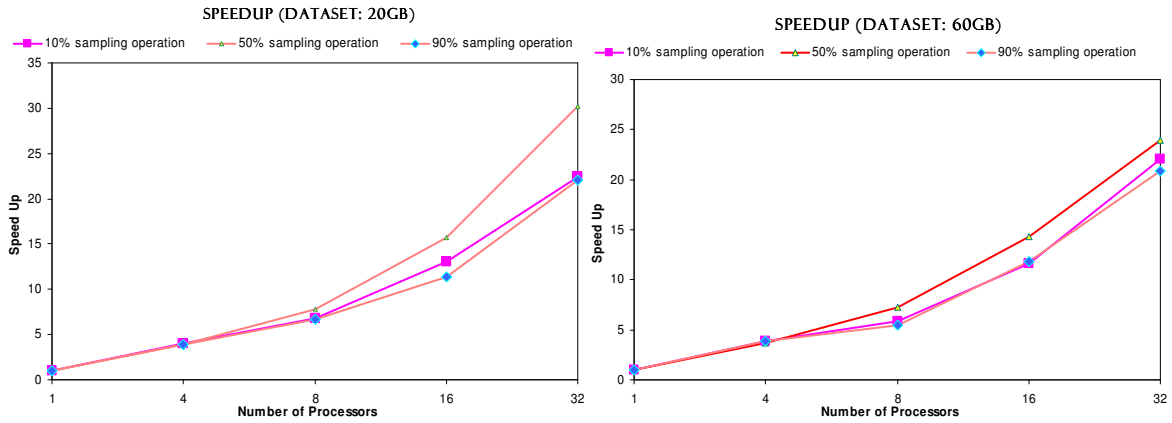

Figure 5. Speedup for sample extraction using the BinPlacement strategy with increasing number of processors

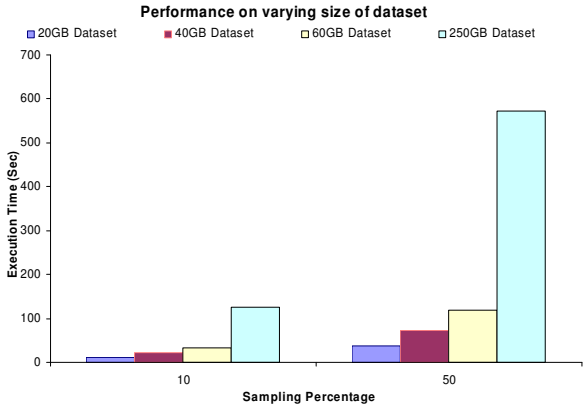

Figure 6. Execution time as the dataset size is varied for different sample percentages

5.3 Benefits of Parallel Sample Extraction:

We test the efficiency of our parallel algorithm by measuring execution time as we vary the number of nodes (1, 4, 8, 16, 32) used at the storage layer in the sampling extraction process. Figure 5 shows that, as expected, parallel sample extraction scales very well with execution time. An important point to note here is that the bin distribution strategy developed in this paper achieves good load balance across the nodes, in terms of both execution time and the amount of data retrieved.

5.4 Scalability with dataset size:

To validate our hypothesis that the BinPlacement strategy scales well with dataset size, we measure execution time on 16 nodes as we vary the size of the dataset up to a quarter of a terabyte. Figure 6 shows that execution time scales well with the data set size and no additional overheads are introduced with larger datasets.

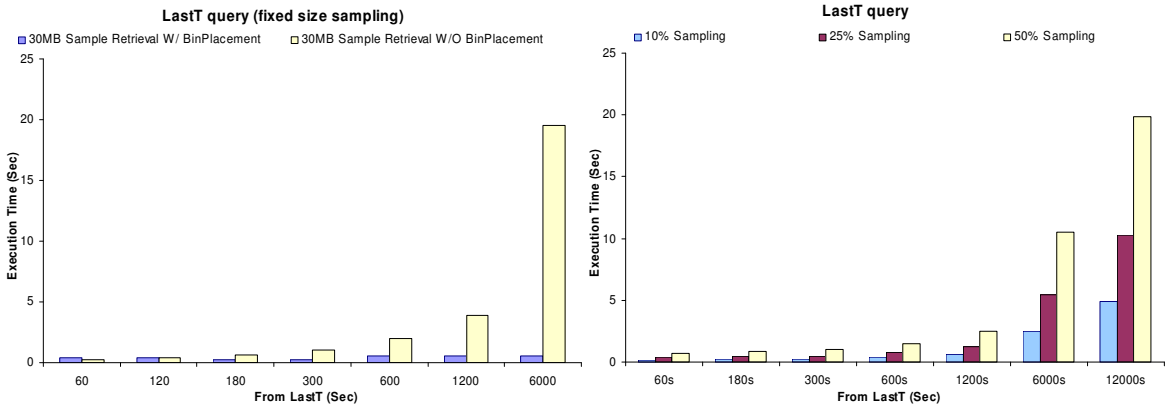

Figure 7. Execution time for lastT queries (a) fixed sample size and (b) variable sample size

5.5 LastT Query Performance:

In these set of experiments, we evaluate the performance of $LastT$ queries in which the query requests a sample set from the data received within the last T seconds. Note that for these experiments dataset size = 20GB. Figure 7(a) shows the execution time for fixed sized sample of size 30MB as the time range T is varied. Figure 7(b) shows the execution time for different percentage samples as the time range T is varied. From both these figures, we can observe that when using BinPlacement, execution time increases only with increasing sample size and is independent of the query range. This is a desirable feature, especially for applications analyzing very large datasets and when the time range specified in the query is very large (as observed from these results). Note that when using the NoBinPlacement strategy, execution time increases as the query range increases, and thus execution time is not independent of the query range.

5.6 Execution time comparing performance on different interconnects:

We compare execution time when using our BinPlacement strategy on three different interconnects. Due to the unavailability of all three interconnects on the same cluster, we used three different clusters. For these experiments, the dataset size is 10GB. We do not expect execution time to vary significantly with different processor speeds as the sample extraction process is disk I/O intensive, and the disks on different clusters provide similar bandwidths. This warrants comparing execution time numbers across different clusters. When using clusters connected using Myrinet and Infiniband, for sample extraction using eight and sixteen nodes, network bandwidth is not a bottleneck. Consequently, disk I/O is the bottleneck, and performance on these two clusters is nearly identical as they have very similar disk bandwidths (Figure 8). However, when using a cluster with FastEthernet, disk I/O bandwidth outperforms network I/O bandwidth, making network I/O bandwidth the bottleneck. This effect is also illustrated in Figure 8 with significantly higher execution when using FastEthernet as compared to using the Myrinet and Infiniband interconnects.

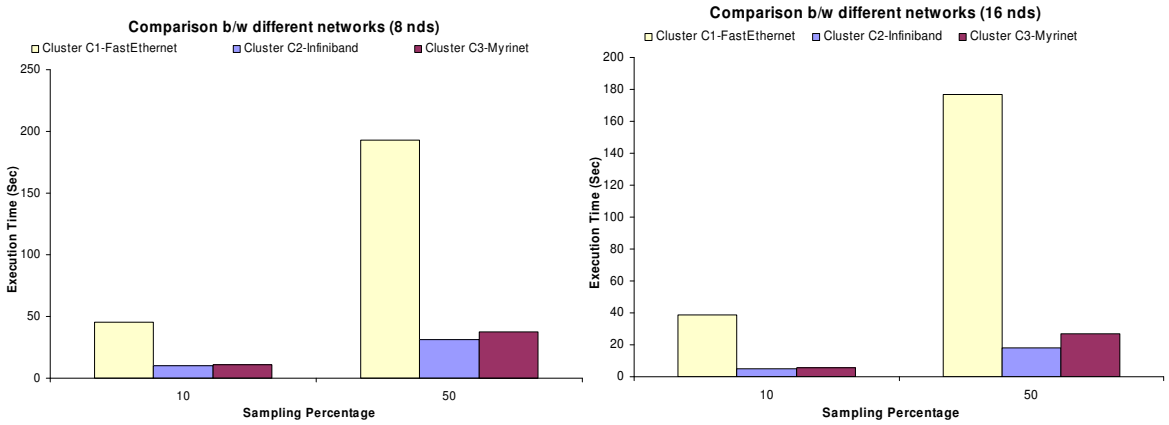

Figure 8. Execution time when using different interconnects (a) 8 nodes and (b) 16 nodes

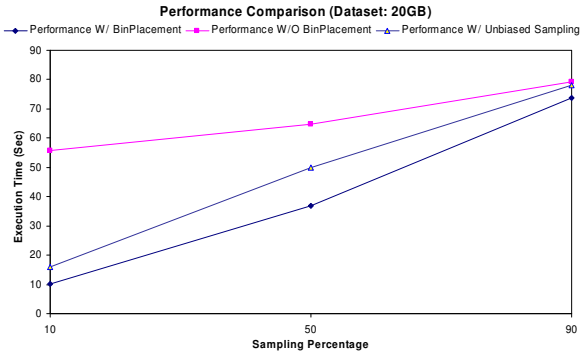

Figure 9. Performance Evaluation of Unbiased Sampling

5.7 Unbiased sampling:

Our default sample extraction strategy is deterministic. This creates a bias between successive queries with the same parameters. For many applications, however, it is required that multiple samples be independent (e.g. ensemble classification [9]). To facilitate this, a flag can be set in the query to request the sample be generated nondeterministically. Random bins from the appropriate time window are chosen until a) the sum of the size of the bins exceeds the sample request, and b) the total number of bins exceeds a randomly determined threshold. A proportional amount of each bin is then selected. The technique compromises performance for independence. For example, suppose a user requests two successive 10% samples from a particular window. If the independence flag is not set, the results of the two queries will have a significant number of elements in common. However, if the flag is set, the bins used for each sample will be chosen randomly. This minimizes the number of common elements. As seen on figure 9, the unbiased sampling technique clearly outperforms the NoBinPlacement strategy, while maintaining a good degree of independence. The performance gain over NoBinPlacement can be attributed to the geometrically progressing bin sizes, as in most cases not every bin will need to be loaded from disk.

5.8 Execution time improvements for end applications:

The end goal of our software sampling infrastructure is to improve application performance. To evaluate this requirement, we implemented a parallel frequent pattern mining algorithm for network intrusion detection. The dataset for this experiment is 20GB, and our 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 results for several consecutive samples are above a particular similarity threshold, the accuracy of the sample is considered optimal. The application is then executed on that sample. For our experiment, we employed the accuracy model described in [25]. 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. This can be seen in figure 10. Figures 11 and 12 depicts results based on three thresholds, namely 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 effects are no longer a significant component when evaluating execution time. When we are satisfied with 98% similarity, the mined sample is 3% of the total dataset, 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 architecture has a dramatic effect on application performance.

6 Conclusions

We have presented an infrastructure to support sampling queries on large scale and dynamic datasets. The key contribution of our work is a randomized placement driven scheme based on the notion of bins for storing and declustering data across disks as and when it arrives. This scheme allows the system to generate samples in time linear to the size of the sample. We further demonstrate how the system can be scaled up on a cluster of workstations interconnected with a state-of-the-art network. An application can interface with our system either using an SQL-like interface that supports ad-hoc querying or through a stand-alone services-oriented interface.

Experimental performance results show that the proposed architecture is viable for ad-hoc temporal database queries. Our results demonstrate good load balancing and expected speedup (up to an order of magnitude when compared to other approaches) on queries abstracted from a network monitoring and analysis application. Experiments carried out on as many as 32 nodes, and the use of datasets as large as 250GB, bear this out. We evaluated our infrastructure on a real network intrusion detection workload, which employs several steps of progressive sampling followed by full frequent pattern mining. By employing our bin placement strategy, we show that disk I/O is no longer the bottleneck. Consequently we see an

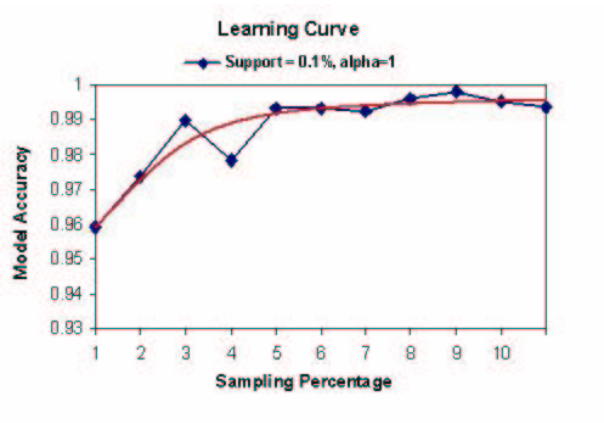

Figure 10. Learning Curve

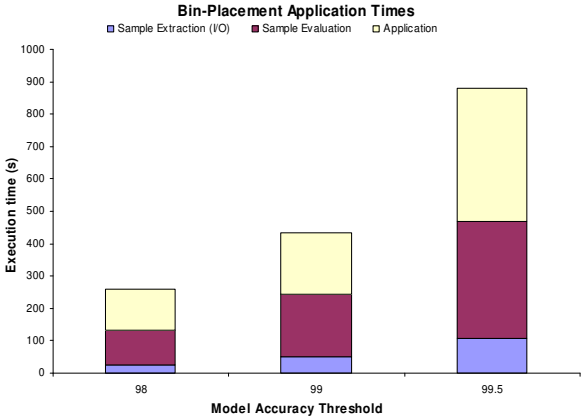

Figure 11. Improvements in application runtime due to bin placement

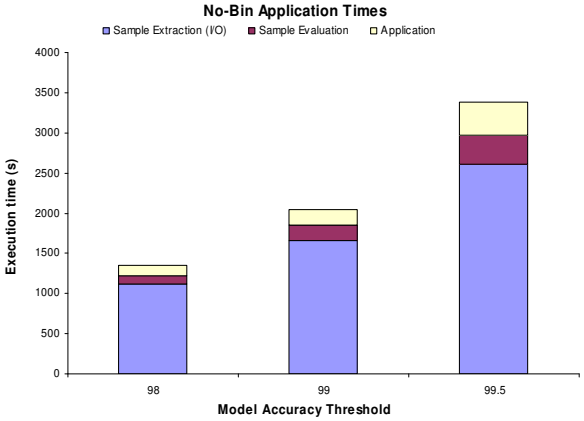

Figure 12. Application runtime without bin placement

execution time improvement of more than 500% for this real workload.

References

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

[2] M. Cannataro and D. Talia. Knowledge grid an architecture for distributed knowledge discovery. In *CACM, Vol. 46, No. 1, pp. 89-93*, 2003.

[3] 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.

[4] L. T. Chen and D. Rotem. Declustering objects for visualization. In *Proceedings of the 19th International Conference on Very Large Data Bases*, pages 85–96, Dublin, Ireland, Aug. 1993.

[5] 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, 2001.

[6] P. F. Corbett and D. G. Feitelson. The Vesta parallel file system. *ACM Transactions on Computer Systems*, 14(3):225–264, Aug. 1996.

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw Hill, 1990.

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

[9] T. G. Dietterich. An experimental comparison of three methods for constructing ensembles of decision trees: Bagging, boosting, and randomization. *Machine Learning*, 40(2):139–157, 2000.

[10] P. Domingos and G. Hulten. A general method for scaling up machine learning algorithms and its applications to clustering. In *Proceedings of the International Conference on Machine Learning*, 2001.

[11] H. C. Du and J. S. Sobolewski. Disk allocation for Cartesian product files on multiple-disk systems. *ACM Transactions on Database Systems*, 7(1):82–101, Mar. 1982.

[12] C. Faloutsos and P. Bhagwat. Declustering using fractals. In *the 2nd International Conference on Parallel and Distributed Information Systems*, pages 18–25, San Diego, CA, Jan. 1993.

[13] M. T. Fang, R. C. T. Lee, and C. C. Chang. The idea of de-clustering and its applications. In *Proceedings of the 12th VLDB Conference*, pages 181–188, 1986.

[14] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kauffman, 2001.

[15] C. Jermaine, A. Pol, and S. Arumugam. Online maintenance of very large random samples. In *Proceedings of the International Conference on Management of Data*, 2004.

[16] G. John and P. Langley. Static versus dynamic sampling for data mining. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining*, 1996.

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

[18] D.-R. Liu and S. Shekhar. A similarity graph-based approach to declustering problems and its applications towards parallelizing grid files. In *Proceedings of the International Conference on Data Engineering*, pages 373–381, Taipei, Taiwan, Mar. 1995. IEEE Computer Society Press.

[19] M. Mahoney and P. Chan. Learning rules for anomaly detection of hostile network traffic. In *Proceedings of the International Conference on Data Mining*, 2003.

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

[21] B. Moon, A. Acharya, and J. Saltz. Study of scalable declustering algorithms for parallel grid files. In *Proceedings of the Tenth International Parallel Processing Symposium*. IEEE Computer Society Press, Apr. 1996.

[22] N. Nieuwejaar and D. Kotz. The Galley parallel file system. In *Proceedings of the 1996 International Conference on Supercomputing*, pages 374–381. ACM Press, May 1996.

[23] F. Olken and D. Rotem. Random sampling from database files: A survey. In *Proceedings of the International Conference on Scientific and Statistical Database Management*, 1990.

[24] J. Pan, C. Faloutsos, and S. Seshan. Fastcars: Fast, correlation-aware sampling for network data mining. In *Proceeding of the IEEE GlobeCom Global Internet Symposium*, 2002.

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

[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. In *Proceedings of the 2001 ACM/IEEE SC01 Conference*. ACM Press, Nov. 2001.

[27] F. Provost, D. Jensen, and T. Oates. Efficient progressive sampling. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining*, 1999.

[28] F. Provost and V. Kolluri. A survey of methods for scaling up inductive algorithms. *Data Mining and Knowledge Discovery*, 1999.

[29] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing '95*, San Diego, CA, Dec. 1995. IEEE Computer Society Press.

[30] 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.

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

[32] H. Toivonen. Sampling large databases for associations. In *Proceedings of the International Conference on Very Large Databases*, 1996.

[33] J. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 1985.

[34] 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*, 1997.