Design and Analysis of a Multi-dimensional Data Sampling Service for Large Scale Data Analysis Applications*

Xi Zhang[1,2], Tahsin Kurc[1], Joel Saltz[1,2], Srinivasan Parthasarathy[1,2]
Department of Biomedical Informatics[1]
Department of Computer Science and Engineering[2]
The Ohio State University, Columbus, OH 43210, USA
{xizhang, kurc, jsaltz}@bmi.osu.edu, srini@cse.ohio-state.edu

Abstract

Sampling is a widely used technique to increase efficiency in database and data mining applications operating on large dataset. In this paper we present a scalable sampling implementation that supports efficient, multi-dimensional spatio-temporal sample generation on dynamic, large scale datasets stored on a storage cluster. The proposed algorithm leverages Hilbert space-filling curves in order to provide an approximate linear order of multidimensional data while maintaining spatial locality. This new implementation is then bootstrapped on top of our previous implementation, which efficiently samples large datasets along a single dimension (e.g., time), thereby realizing a service for spatio-temporal sampling. We evaluate the performance of our approach comparing it to the popular R-tree based technique. The experimental results show that our approach achieves up to an order of magnitude higher efficiency and scalability.

1 Introduction

Large-scale, multidimensional, dynamically growing datasets have become a major consumer of resources in many scientific applications, thanks to the development of new technologies such as advanced sensors that can rapidly capture data at high-resolutions and Grid technologies that enable simulation of complex numerical models. Low-cost, large scale, disk-based storage clusters can be used to host vast volumes of dynamic datasets. However, given disk access and network overheads, managing and retrieving this data efficiently is a challenging task.

This work is particularly motivated by emerging data-driven scientific analysis applications in which large, multi-dimensional, dynamic datasets that are generated by advanced equipments and sensors, need to be mined and analyzed. Data analysis, or mining, processes tend to be iterative, exploratory, and interactive in nature. They may require multiple passes over the data, which may be prohibitively expensive for large datasets. 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, there is an immediate need for a *scalable framework for efficient storage and processing of dynamic, multidimensional data*.

Consider the LEAD [18] application as an example. LEAD is a large-scale infrastructure for atmospheric science research. It allows researchers to dynamically and adaptively respond to weather changes in order to generate real time predictions of tornadoes and other potentially devastating weather events. When the system is fully functional, the datasets will be collected by small scale regional Doppler radars that can be mounted on cell phone towers. These small scale radars have a 30 km radius and can collect data at high resolution and frequency. The raw data captured by the sensors on these radars will be periodically sent to data processing and archiving centers for storage and analysis.

A sample query against these datasets could be stated as: *Retrieve a* **multidimensional sample** *from the CAPS radar, ACARS, and NEXRAD Doppler level II data and limit the results to data obtained or relevant to an 80 mile radius around New Orleans (spatial) and limit results to those obtained over the past two hours (temporal).* Such ad-hoc sampling queries that project the data set along a multidimensional bounding box (MBR) can be used to effectively

*This work is primarily supported by NSF grant NGS-CNS-0406386. The authors would also like to acknowledge support from NSF grants CAREER-IIS-0347662, #RI-CNS-0403342, #CCF-0342615, #ACI-9619020 (UC Subcontract #10152408), #EIA-0121177, #ACI-0203846, #ACI-0130437, #ANI-0330612, #ACI-9982087, #CCF-0342615, #CNS-0406386, #CNS-0426241, Lawrence Livermore National Laboratory under Grant #B517095 (UC Subcontract #10184497), NIH NIBIB BISTI #P20EB000591, Ohio Board of Regents BRTTC #BRTT02-0003.

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

summarize data for tasks such as scientific visualization and weather prediction. Given the sheer size of the data gathered continuously and real-time analysis requirements, it may be too time consuming to retrieve the entire dataset and process it, even on a parallel machine, especially if a disastrous weather pattern is being observed and must be acted upon.

Successful deployment of efficient support for data querying and retrieval in such an application would require efficient strategies to sample large, multi-dimensional, dynamic datasets. The key intuition is when you operate on less data, data analysis can be carried out more quickly, allowing one to react changing weather patterns rapidly . The focus of our work is to develop support that will enable efficient execution of sampling queries and that will take advantage of parallel processing and high-performance networking platforms. The multidimensional nature of many scientific datasets complicates the data retrieval and sample generation process on both single node machines and distributed platforms. There has been extensive literature on index structures for multidimensional data access. However, not much work has been done on how to efficiently generate multidimensional samples from dynamic and out-of-core datasets on a parallel machine. I/O cost is one of the primary bottlenecks in the sample generation process. A possible approach to reduce I/O costs is to use parallel machines and distribute the work associated with data maintenance and sample retrieval across multiple nodes. This paper proposes a parallel multidimensional sampling algorithm and evaluate its efficiency and scalability.

In an earlier work [22], we developed support for creating samples from an array of data elements. We have shown that the algorithm is very efficient and scales well. However, our previous work assumed either the dataset was single dimensional or the sampling request is limited to a pre-determined dimension of the dataset (e.g., time dimension). Our approach in this paper is to produce a linear ordering and a mapping to a one dimensional space from the multi-dimensional space of the data using Hilbert space filling curves. We distribute the ordered data elements across multiple storage nodes to achieve parallel I/O when retrieving samples. Mapping to single dimensional space also makes it possible to leverage the infrastructure developed in the previous work [22]. One advantage of using a Hilbert curve based mapping is that it maintains multi-dimensional locality. That is, points close to each other in the multi-dimensional space are mapped to close indices on the Hilbert curve. Our algorithm then builds a multi-level index structure around this ordering. This index structure facilitates fast retrieval of multi-dimensional samples that encompass constraints over time and space. We develop a query execution scheme that builds on the recursive nature of the Hilbert space filling curve in order to efficiently execute range sampling queries. We demonstrate that such an approach significantly outperforms the popular R-tree based approach up to an order of magnitude and has good scaling properties on data storage clusters.

2 Related Work

Until recently there has been little work done on how to improve the performance of generating a sample from out-of-core datasets. Researchers have looked at generating samples over in-memory databases [15]. The assumption is that the data set is static and samples are assumed to fit in main memory. Reservoir sampling [21] 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.* [9]. However, using this approach one cannot generate a variable sized sample over a variable time range. Also, this strategy cannot be trivially extended to parallel machines. Previous work on sampling from spatial database [16] assumes the dataset is static and already indexed, moreover, the paper focuses on sample selection criteria instead of sample generation efficiency.

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 [20], clustering [3] and several other machine learning algorithms. These algorithms do not know the desired sample size a priori. Progressive sampling [17, 19] has been proposed for these algorithms so they can efficiently 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.

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 [2]. Numerous declustering methods have been proposed in the literature. Grid-based methods [1, 4, 5] have been developed to decluster Cartesian product files, while graph-based methods [6, 12, 13] 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.

3 Problem Definition and Architecture Overview

Our objective is to support sampling range queries of the following form:

SELECT SAMPLE x%
FROM Dataset D
WHERE Tuple a **in [**$l_1..u_1, l_2..u_2, \ldots, l_n..u_n$**]**
AT TIME t_0 **(or BETWEEN TIME[**t_1, t_2**])**

Here, the dataset D is a multi-dimensional dataset. The list of tuples that can be used to answer the query is specified by a bounding box in the multi-dimensional space underlying the dataset ($[l_1..u_1, l_2..u_2, \ldots, l_n..u_n]$ in the *WHERE* statement in the query). Each element in D is associated with a tuple $(a_1, a_2, ..., a_n)$ as the coordinate vector, where each attribute corresponds to a different dimension of the dataset. This n-dimensional attribute vector is also interpreted as the coordinates of the element in the multi-dimensional space. We consider the coordinate space to be a discrete space. This limitation of our data model, however, can be mitigated by converting continuous space into discrete space with normalization and discretization. In this work, we will primarily use uniform sampling which basically assign each point equal probability of selection. Other type of sampling such as stratified sampling and biased sampling can be obtained as well by post-processing uniform samples. Our framework also supports progressive sampling that requests a series of increasing percentage of samples.

We target dynamically updated datasets. New data elements are added to the dataset overtime. Thus, each data element is also associated with a time stamp t_i. To facilitate temporal range queries (*AT TIME* or *BETWEEN TIME* in the query form), we regard the time stamp for each tuple as one of the coordinate elements.

Our previous work [22] developed an architecture, which consists of three main layers, to support pre-processing, management, and querying of large scale dynamic datasets. We implement the proposed framework within that architecture. Here, we briefly describe the components of that architecture.

The *data preprocessing layer* implements methods and runtime support for pre-processing incoming data elements so that they can be efficiently stored and queried. The preprocessing of data updates is done in groups of tuples, referred to as a *stream window*. A window D_i contains tuples that have arrived in a time interval of (T^i_{start}, T^i_{end}). The tuples in a stream window are partitioned into a user-defined number of *bins* and reorganized within each bin. Bins are the unit of storage on disk and data retrieval from disk.

The *storage management layer* is responsible for efficient management of storage space and placement of bins on disks in the system. If there are multiple nodes and multiple disks, the storage management layer employs data distribution strategies for assigning bins to nodes and disks. It also manages indexing structures so that the data of interest, defined by a range query, can be searched for quickly.

The *query processing layer* implements the support for receiving queries from clients, scheduling multiple queries for execution, and executing each query on the underlying hardware infrastructure, and returning the results to the client.

4 Multidimensional Sampling Service

4.1 Organizing Multidimensional Data for Sampling

Since data selections in our target queries are specified on ranges or points on some attributes, it is desirable that data points that are close together in the multidimensional attribute space be clustered close in the destination one-dimensional space. Such a mapping would reduce the number of disk blocks accessed and the number of seek operations. We propose a multidimensional sampling algorithm based on linear clustering of multidimensional dataset by Hilbert space filling curve. Several publications [14, 8] have shown that Hilbert space filling curve based linear clustering achieves good locality. Hilbert space filling curve is a one-dimensional curve which visits every point within a n-dimensional space in a specific order [11]. Figure 1 shows two examples of Hilbert space filling curve for 2-dimensional space (coordinates in each dimension are represented using 1 bit in Figure 1(a) and 2 bits in Figure 1(b)). In general, for a n-dimension attribute space where each dimension is represented by k bits, there will be 2^{nk} points in the curve. The useful property of the Hilbert space filling curve is that points that are close in the multi-dimensional space are mapped to indices which are close in the one dimensional space.

We should note that mapping from multidimensional attribute space into a linear attribute space is different from the common dimensionality reduction approaches such as principal component analysis (PCA) [10]. PCA usually identifies dominant sub-dimensions and transform the data points into these dimensions. Linear mapping approach does not discriminate against any dimension, and in practice, it could be applied to data points postprocessed by dimensionality reduction methods.

In our implementation, when a stream window is received, the data preprocessing layer maps the elements in

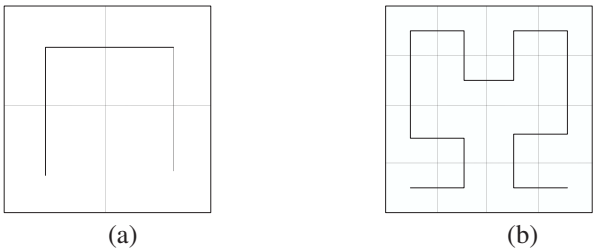


Figure 1. Two Examples of Hilbert Space Filling Curve

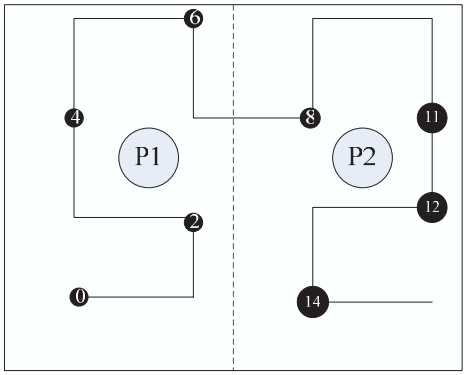


Figure 2. Partitioning Linearizing Dataset into Bins

this stream window into a linear array. The basic concept is to organize this linear array into a set of contiguous bins. Figure 2 shows one example where the bin capacity is 4 data elements.

To enable efficient sample generation, we apply a geometric binning scheme developed in [22] instead of equal-size bins. This scheme generates k bins, whose sizes follow a geometric progression (n/2, n/4, n/8, ...), where n is the number of elements in the linear array and k is a user-defined value. The elements in the linear array are assigned to bins using a random function. Each data element is assigned to one of the k bins randomly with a probability that is proportional to the size of the bin. Elements within a bin are ordered according to the linear ordering obtained by the Hilbert space filling curve so that the spatial locality is preserved. The randomization ensures that each bin can be viewed as containing mutually exclusive samples. An example of the geometric bin partition is shown in Figure 3. The key advantage of using geometric binning scheme is that it will minimize the data we have to retrieve from disk [22]. For example, if we have four bins, and if the sample size requested within a query is between $n/8$ and $n/8 + n/16$, then the query will be answered using

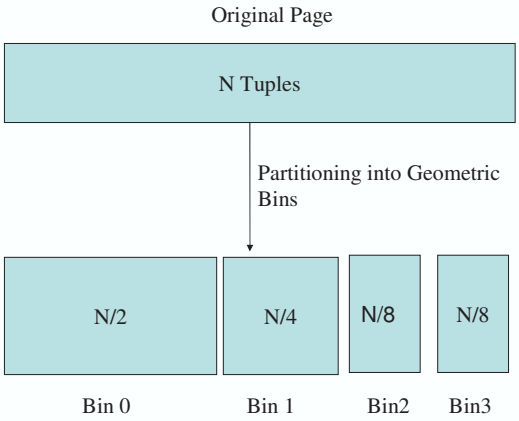


Figure 3. Geometric Bin

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.

4.2 Indexing Structure

Each bin has a one-dimensional bounding box, corresponding to the minimum and maximum values of the Hilbert curve indexes of the data elements stored in that bin. The indexing scheme is based on a binary search mechanism on top of a dynamic array structure. The entries of the array contain $(\{T^i_{start}, T^i_{end}\}, B^i_n, coord_{start}, coord_{end})$. The first element of the tuple is the time interval for the bin. The second element is the bin number identifier. The third and fourth elements define the linear coordinate range (obtained by the Hilbert space filling curve mapping) of the bin. Finding the set of bins that contains the data points intersecting with a range query is essentially a binary search on the range information stored in the dynamic array.

4.3 Data Distribution and Placement

When we have multiple disks we adopt the following round robin placement strategy. Assume we have m parallel disks and the streaming window instance D_i contains k bins. Bin distribution is an mapping f from the bin set $\{0, 1, ..., k-1\}$ to the parallel disk set $\{0, 1, ..., m-1\}$ as follows.

$$f(i) = i\%m, 0 \leq i \leq k$$

4.4 Range Query Execution Algorithm

The recursive nature of the Hilbert curve construction process [11] can be utilized to develop an efficient query execution algorithm.

Suppose we have a 2-dimensional attribute space with 3 bits in each dimension. For a point P with attribute [110, 011], the linear coordinate will be a 6-bit number. The linear coordinate is initially unknown and is represented by "??????" in this example. The recursive property of the Hilbert curve assures that the first two bits of the linear coordinate will only be determined by the first bit of two coordinates. In this case they are "11", therefore the linear coordinate would be "11????". This property can be extended into n-dimensional attribute space. Based on this property, for a range query (e.g., [101..111, 001..011]), after parsing the query range, we know the first bit of the first dimension is "1" and that of the second dimension is "0". Therefore, the point which intersects with the query range must have the first two bit as "11", and we convert the 2-d range into a linear range as [110000..111111]. However, this linear range would contain a set of these sequential points and we need to filter those extra data points. In practice, this process will continue recursively from the most significant bit to the least significant bit and generate a set of non-overlapping linear ranges. Whenever we find there is a bit shift in one of the dimensions, we effectively split the range in that dimension into two ranges with the same bit. This split process is essentially a partition along the middle of each dimension. If we have a range with the remaining $i+1$ bits in the form of $[\,0_i 0_{i-1} \ldots 0_0 .. 1_i 1_{i-1} \ldots 1_0\,]$ in every dimension, we regard this is as a *full subregion* and the resulting linear range will be exactly the query result set and no further filtering is required. The algorithm is detailed in Figure 4.

After receiving the range queue Q_R (Figure 4), the algorithm loads the indexing scheme presented in previous section and performs an index lookup to locate the corresponding bins. Since each bin has already been sorted on the linear coordinate, we can efficiently perform another binary search on each in-memory bin and retrieve all of the data points intersecting the query. The algorithm is shown in Figure 5. Both algorithms are easily parallelizable by placing bins into different disks and building localized indexes. As the linear coordinate in each node will be non-intersecting, each query can be executed on each node in parallel.

4.5 Sampling using R-trees

R-tree [7] is a spatial data structure analogous to a B^+ tree used for storing multi-dimensional data points and polygons. The data points and polygons are represented in the tree by their minimal bounding rectangles. The root of the R-tree is the minimal bounding rectangle which encloses all objects in the database. Each node in the tree corresponds to the minimal bounding rectangle for all of the objects in its subtree. R-tree is a popular data structure that has been used in a wide variety of areas including commercial databases and scientific applications. Here we will use the R-tree as the alternative data structure to compare against our proposed approach. The sampling procedure on an R-tree file is based on the query-first algorithm proposed in[16]. The input is a multi-dimensional query region and a sampling ratio. The R-tree index returns a queue that consists of points that fall inside the query region. For each point in the queue, we dynamically determine whether to select it or not according to the sampling rate.

Input: Query region R.
Output: Range queue Q_R, whose elements are linear ranges.
Set dimension = n, bits per dimension = k.
Q_0: Query queue for current level, each element corresponding to query region.
Q_1: Query queue for next level.
Insert R into Q_0.
Set level = k, maxsize = user determined value.
while (Q_0.size $\neq$ 0 && Q_0.size $\leq$ maxsize) **do**
  Range r;
  while ((r = Q_0.pop) $\neq$ NULL) **do**
    Intersect r with the center of each attribute at the level bit and generate a set of subregions S.
    for (each range k in **S**) **do**
      if (k is *full subregion*) **then**
        convert k into a linear range L and designate it as *full range*
        Add L to Q_R
      else
        Add k to Q_1
  Q_0 = Q_1
  level$--$;
if (Q_0.size $\neq$ 0) **then**
  Convert each remaining region k in Q_0 into linear range L and insert L into Q_R;
Return Q_R;

Figure 4. Range Query Algorithm.

5 Experimental Evaluation

In this section, we examine the performance of our proposed algorithm. The issues we evaluate here include: preprocessing cost, query retrieval performance, the benefits of binning, and scalability. We detail the experimental setup next.

5.1 Setup:

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

Input: Range queue, Q_R, and Index, Id.
Output: Data elements queue, Q_D
Load Id into memory
Range r;
while ((r = Q_R.pop) $\neq$ NULL) **do**
  Perform a range lookup on the Id and obtain corresponding bins set B
  for (every bin b_i in B) **do**
    Load b_i in memory, perform binary search
    If the range is not a *full range*, filter out every non-intersecting data point
    Add the data point d to Q_D
Return Q_D

Figure 5. Index Lookup Algorithm

- 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 around 25MBytes/s from each disk. The nodes in this cluster are connected using a 100Mbps switch.
- C2-Infiniband: This cluster consists of 16 compute nodes with two 2.4GHz Intel Pentium 4 Xeon processors, 4 GB of memory, and 62 GB of local scratch space. Unless specified otherwise, this cluster is used for all the experiments. We measure the application level I/O bandwidth to be 23MBytes/s from each disk. The nodes in this cluster are connected using an 8Gbps Infiniband interface.

Our implementation is written in C, and MPI is used for message passing. The experiments were performed using synthetic datasets. We conduct the experiments for sequential and parallel environments. In the experiments, we set the size of each tuple to be 1024 bytes. Each coordinate of a data element is represented by an integer, and thus is 4 bytes. The size of each stream window is $n = 1024$ elements and the number of bins is $k = 8$. With this setting, the smallest bin has 8 data points (i.e., the size of the smallest bin is $8KB$). 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 of a data element.

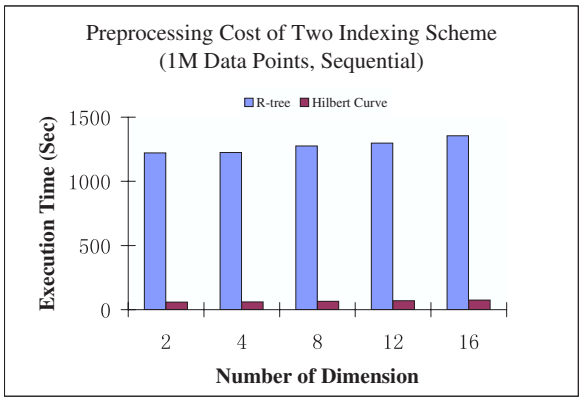
5.2 Preprocessing Cost

This set of experiments compares the preprocessing cost of our proposed linear clustering based index creation and R-tree based creation both on sequential and parallel setting. For both tests, we use a stream window with 1 Million (1M) data elements. Figure 6 shows that in both sequential and parallel setting, the preprocessing cost of the R-tree based scheme is 8-10 times higher than that of the proposed Hilbert curve based scheme. For our targeted application scenario, where there is constant influx of data, preprocessing efficiency should be considered as an important factor when choosing the appropriate middleware system to manage these datasets. The R-tree data structure is optimized for ad-hoc insertions and updates, whereas our scheme is designed for batch insertions. The high preprocessing cost with the R-tree implementation also is a result of large number of page splits. The Hilbert curve based scheme uses in-memory sorting to preorder the data points and avoid the costly I/O operations. Consequently, one drawback of our current implementation is its higher memory space requirement.

5.3 Query Retrieval Performance

These experiments compare the query retrieval performance of our proposed scheme and the R-tree based scheme in both sequential and parallel setting. We use a stream window with 1 Million (1M) data elements. Each coordinate of a data element is randomly drawn from a range of $[0...1023]$ units and is represented in 10 bits. The multidimensional query in this experiment is created randomly and has a range of 512 units in each dimension. In the experiments, we vary the number of dimensions of the query and of the dataset. Figure 7(a) shows the query response time and Figure 7(b) shows the effective disk bandwidth for a single node. The results show that our proposed scheme achieves 3-5 times better performance than R-tree based index and scales well with increasing dimensionality. Figure 8 shows the response time and effective bandwidth when we fix the number of dimensions to be 4 and vary the size of multidimensional range query (i.e., the range in number of units). The results show that both indexing schemes suffer from very low effective bandwidth when the query region is small. The main reason for the inefficiency is the software system overhead. However, for the proposed scheme, the sample retrieval performance increases significantly, as the query size increases, and reaches close to 70% of the raw disk bandwidth when the query size is around 512 units per dimension.

Figure 9 shows the speedup as we vary the number of nodes on the two clusters with different network configurations. We used both low-dimensional (number of dimensions = 4) and moderately high-dimensional (number of dimensions = 16) datasets. On the C1-FastEthernet cluster, as the size of the query box increases, the speed up decreases. The reason is that performance is limited by the bandwidth of the inter-processor network. In our experimental setting, the maximum network bandwidth was measured to be around 11.6MB/sec, which is much lower than the disk bandwidth achieved by a single node when the query size is



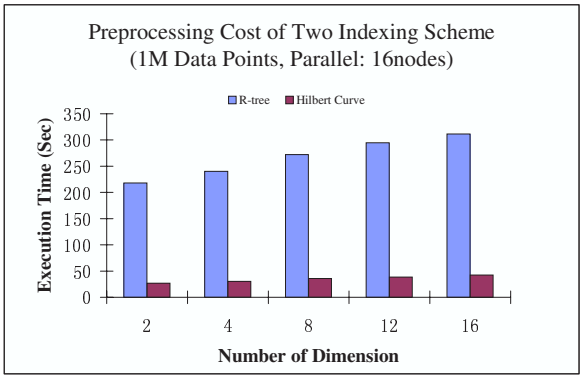
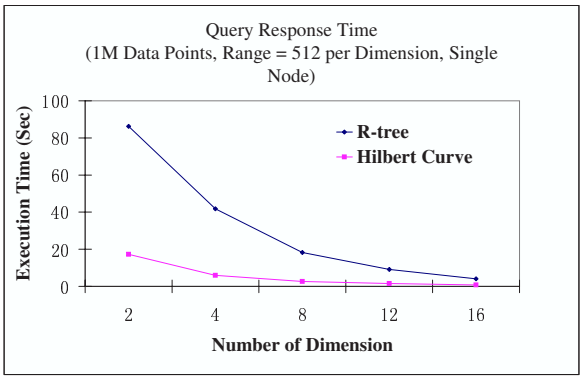


Figure 6. Preprocessing Cost for the R-tree based and the Hilbert Curve based Schemes.



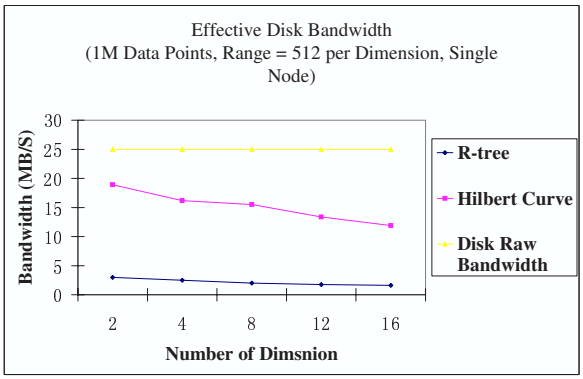


Figure 7. Query Response Time and Effective Bandwidth with Varying Dimensionality

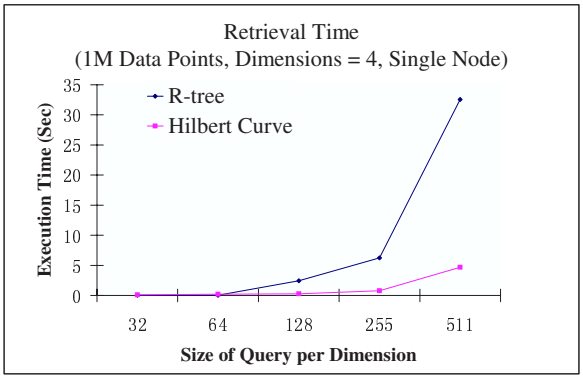
large. On the other hand, with a faster network such as Infiniband, where the maximum network bandwidth is much higher, the results show that the proposed architecture is scalable, especially for high dimensional datasets. Our results show that the proposed algorithm can take advantage of high-performance communication networks.

5.4 Benefits of Binning for Sample Generation

The next set of experiments compares the performance of sample generation of our proposed binning strategy with the non-binning strategy and the R-tree based strategy. When using the non-binning strategy, we still order the elements within a stream window using a Hilbert curve mapping, but no bins are generated. That is, the entire stream window is stored on one of the nodes in the system. In our implementation we assign the stream windows to nodes in round robin fashion. When a sampling query is executed, the algorithm tests each element for sample membership. 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 using the range query execution strategy described in Section 4.4. As is seen in Figure 10, our binning strategy achieves more than an order of magnitude speedup over the non-binning and R-tree based sampling strategies 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 the binning strategy marginally outperforms the non-binning and R-tree based strategies as our approach touches a fractionally smaller number of disk blocks compared to the other strategies.

5.5 Scalability with Dataset Size

The final set of experiments study the scalability of our proposed algorithm with a larger dataset. In these experiments, we used a query that covers the whole temporal range of the dataset and the overall size of the dataset is 40GB. Figure 11 shows that the proposed algorithm scales well as the size of the dataset is increased.



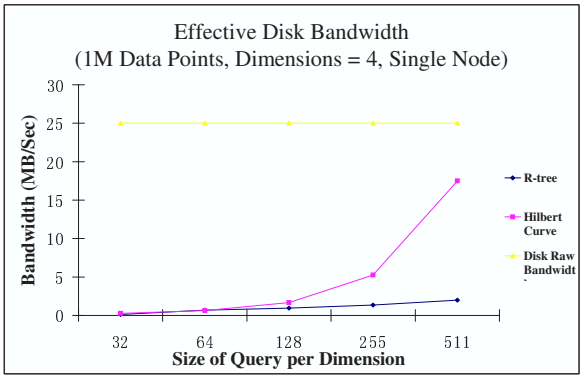


Figure 8. Query Responsive Time and Effective Bandwidth with Varying Query Size

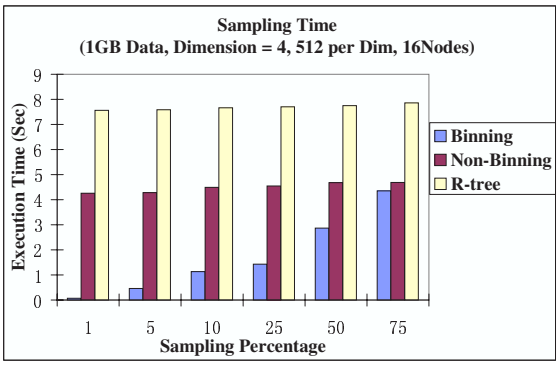


Figure 10. Query Response Time with Different Sampling Strategies

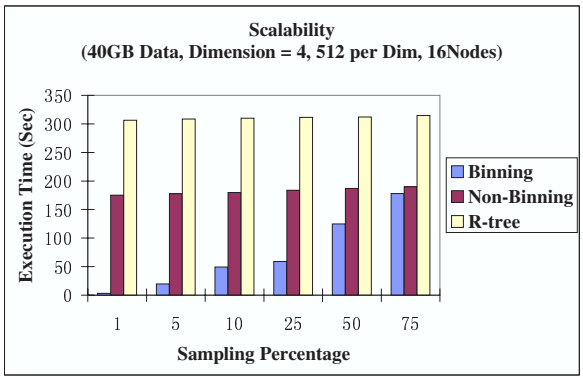


Figure 11. Scalability of Different Sampling Strategies

6 Conclusion and Future Work

We have presented the design and implementation of an algorithm to support sampling queries on large scale, multidimensional and dynamic data sets. The key contribution of our work is 1) a novel application of the Hilbert space filling curve to generate a locality-preserving linear mapping and ordering of multi-dimensional data and 2) the use of geometric size bins for placement of data on disk for efficient sample generation. This scheme allows for sample generation in time proportional to the size of the sample. We further demonstrate the system is scalable. Experimental results show that the proposed approach is applicable for ad-hoc temporal database queries. Our results demonstrate good load balancing and speedup.

Future work includes several directions. First, we plan to incorporate a sample quality evaluation function into our framework so that we could support user-defined sampling criteria. Second, we will obtain real query workloads and evaluate our algorithm with respect to the end-application performance improvement. Finally, we will investigate the effect of dynamic workloads and design adaptive strategies to accommodate such cases.

References

[1] L. T. Chen and D. Rotem. Declustering objects for visualization. pages 85–96, Dublin, Ireland, Aug. 1993.

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

[3] 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.

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

[5] C. Faloutsos and P. Bhagwat. Declustering using fractals. pages 18–25, San Diego, CA, Jan. 1993.

[6] M. Fang, R. Lee, and C. Chang. The idea of de-clustering and its applications. pages 181–188, Kyoto, Japan, 1986.

[7] A. Guttman. R-Trees: A dynamic index structure for spatial searching. pages 47–57, Boston, MA, June 1984.

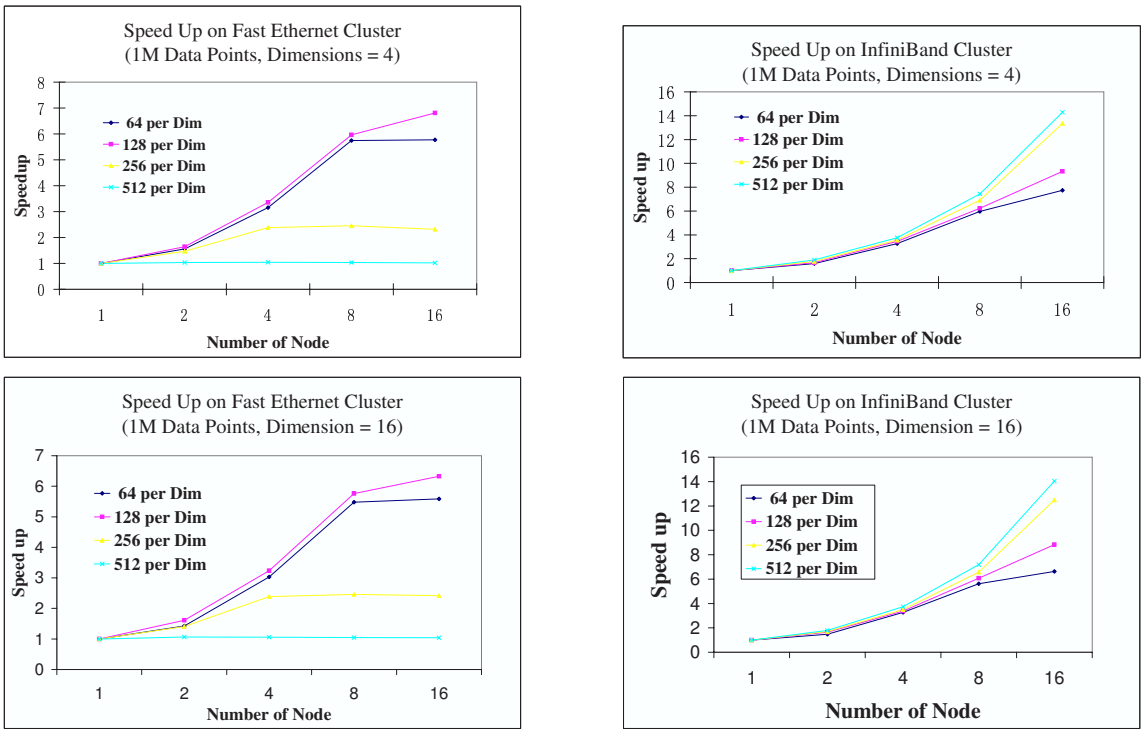


Figure 9. Speedup Values.

[8] H. V. Jagadish. Linear clustering of objects with multiple attributes. In *SIGMOD '90: Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 332–342, New York, NY, USA, 1990. ACM Press.

[9] 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.

[10] I. Jolliffe. *Principle Component Analysis*. Spring-Verlag, New York, NY, USA, 1986.

[11] J. K. Lawder and P. J. H. King. Querying multi-dimensional data indexed using the hilbert space-filling curve. *SIGMOD Rec.*, 30(1):19–24, 2001.

[12] D.-R. Liu and S. Shekhar. A similarity graph-based approach to declustering problems and its applications towards parallelizing grid files. pages 373–381, Taipei, Taiwan, Mar. 1995.

[13] B. Moon, A. Acharya, and J. Saltz. Study of scalable declustering algorithms for parallel grid files. pages 434–440, Honolulu, Hawaii, Apr. 1996. Extended version is available as CS-TR-3589 and UMIACS-TR-96-4.

[14] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *Knowledge and Data Engineering*, 13(1):124–141, 2001.

[15] 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.

[16] F. Olken and D. Rotem. Sampling from spatial databases. In *ICDE*, pages 199–208, 1993.

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

[18] B. Plale, D. Gannon, D. A. Reed, S. J. Graves, K. Droegemeier, B. Wilhelmson, and M. Ramamurthy. Towards dynamically adaptive weather analysis and forecasting in lead. In *International Conference on Computational Science (2)*, pages 624–631, 2005.

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

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

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

[22] 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 *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 91–100, New York, NY, USA, 2005. ACM Press.