SciDB DBMS Research at M.I.T.

Michael Stonebraker[1], Jennie Duggan[1], Leilani Battle[1], Olga Papaemmanouil[2]
[1] Computer Science and Artificial Intelligence Laboratory, MIT
[2] Department of Computer Science, Brandeis University
{stonebraker, jennie, leilani}@csail.mit.edu, olga@cs.brandeis.edu

Abstract

This paper presents a snapshot of some of our scientific DBMS research at M.I.T. as part of the Intel Science and Technology Center on Big Data. We focus our efforts primarily on SciDB, although some of our work can be used for any backend DBMS. We summarize our work on making SciDB elastic, providing skew-aware join strategies, and producing scalable visualizations of scientific data.

1 Introduction

In [19] we presented a description of SciDB, an array-based parallel DBMS oriented toward science applications. In that paper we described the tenets on which the system is constructed, the early use cases where it has found acceptance, and the state of the software at the time of publication. In this paper, we consider a collection of research topics that we are investigating at M.I.T. as part of the Intel Science and Technology Center on Big Data [20]. We begin in Section 2 with the salient characteristics of science data that guide our explorations. We then consider algorithms for making a science DBMS elastic, a topic we cover in Section 3. Then, we turn in Section 4 to query processing algorithms appropriate for science DBMS applications. Lastly, in Section 5 we discuss our work on producing a scalable visualization system for science applications.

2 Characteristics of Science DBMS Applications

In this section we detail some of the characteristics of science applications that guide our explorations, specifically an array data model, variable density of data, skew, and the need for visualization.

Array Data Model Science data often does not fit easily into a relational model of data. For example, Earth Science data [18] often comes from satellite imagery and is fundamentally array-oriented. Astronomy telescopes are effectively large digital cameras producing pixel arrays. Downstream, the data is processed into a 2-D or 3-D coordinate system, which is usually best modeled as an array. Moreover, it is often important to keep track of time, as objects are recorded over days or weeks of observations; time is simply another dimension in an array-based system, but must be glued on to a relational model of data.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Field observations are invariably spatially-oriented and queried in a 3-D space (e.g., latitude, longitude, and time). Often additional dimensions are present (e.g., elevation). Searching in a high dimensional space is natural and fast in a multi-dimensional array data model, but often slow and awkward in a relational data model.

Lastly, much of the analytics that scientists run are specified on arrays, for example k-nearest neighbors, spatial smoothing, fourier transforms, and eigenvectors. These are naturally executed in an array DBMS, but require relational data to be cast into arrays for execution. For these reasons, we believe that array DBMSs are a natural fit for science data, and our research has focused in this area.

Sparse or Dense Array Some arrays have a value for every cell in an array structure. For example, it is common practice to "cook" satellite imagery into a dense array, where each cell represents a tile on the earth surface and the cell value is some interpolation over a time period of the actual collected imagery data. For example, one common interpolation is to select the cell value from the possible ones, which has the least cloud cover. This interpolation produces a very large, dense array. On the other hand, the raw satellite imagery is recorded in a three-dimensional space, for example (latitude, longitude, time), or a four-dimensional space (latitude, longitude, altitude, time). Sometimes spherical coordinates are used. In any case, the recorded data is very sparse. Our experience is that an array DBMS must be prepared to cope with data that ranges from very sparse to very dense.

Skewed Data If we imagine a database consisting of the position of each resident of the United States, then the density of points in Manhattan is 10^5 greater than the density in Montana. This sort of skew, whereby some regions of array space have substantially more data than others, is very common in science applications. In astronomy, there are regions of the sky that are more interesting than others, and the telescope is pointed there more often. In a database of ship positions, the vessels congregate in and around major ports waiting to load or unload. In general, our experience is that moderate to high skew is present in most science applications.

Visualization Focus In business data processing, a form-based interface is exceedingly popular. For example, to look up pending airline reservations, one inputs one's frequent flyer number into a form. To find the balance in one's checking account, one enters the account number, and so forth. Scientists rarely want this sort of form-based user interface. Instead, they usually want a visualization system through which they can browse and inspect substantial amounts of data of interest. For example, one Earth Science group is interested in snow cover in the Sierra Nevada mountains. Hence, they want to "fly over" the study area, browsing satellite imagery and then zoom into areas of interest. Therefore, the front end issuing queries to a science database is often a visualization system.

3 Elasticity

Given that scientists never want to discard data, this leads to a "grow only" data store. Also, the amount of data that they want to process often increases with time, as they collect more data from experiments or sensors. Hence, a science DBMS should support both data elasticity and processing elasticity. Of course, elasticity should be accomplished without extensive down time; in the best of all worlds, it is accomplished in background without incurring any downtime. Often science data is loaded an experiment-at-a-time or a day-at-a-time, i.e., periodically. In between load events, it is queried by scientists.

We have constructed a model of this desired elasticity behavior. It consists of three phases, a loading phase where additional data in ingested, followed by a possible reorganization phase, followed by a query phase whereby users study the data. These phases repeat indefinitely, and the job of an elasticity system is three fold:

- predict when resources will be exhausted
- take corrective action to add another quanta of storage and processing
- reorganize the database onto the extra node(s) to optimize future processing of the query load

Our model specifies a cost function for these three activities that minimizes a combination of provisioned resources and time elapsed in querying, data insertion, and incremental reorganization. We use this model to study the impact of several data partitioners on array database performance in terms of workload latency. Our elastic partitioners are designed to efficiently balance storage load, while minimizing reorganization time during cluster expansion operations.

We have implemented this model for SciDB, and the details can be found in [14]. In addition, we have studied the behavior of the model on two different use cases. First, we have explored a MODIS satellite imagery database [1], with appropriate queries primarily from [18]. This data set is large, sparse and uniform, as the satellite covers each portion of the earth's surface at the same rate. In addition, we have explored a U.S. Coast Guard database of ship positions, AIS [7], which are highly skewed as noted above.

The query phase of our workload is derived from real use cases, and each has a mix of select-project-join (SPJ) and domain-specific science queries, many of which are spatial. The SPJ benchmarks have three components: selection, a distributed sort, and an equi-join, and these queries capture simple relational transformations on the data. Our science benchmarks are customized to each use case. AIS executes a k-nearest neighbor query, studying ship density patterns, a ship collision detector, and a regridding from detailed lat/long space to an aggregated n-dimensional projection. Our MODIS evaluation includes a k-means clustering, used to model rainforest deforestation, a windowed aggregate to generate a smoothed image of the satellite's recordings, and a rolling average of measurements for environmental monitoring. We detail our benchmark queries in [1].

3.1 Elastic Array Partitioning

Well-designed data placement is essential for efficiently managing an elastic, scientific database cluster. A good partitioner balances the storage load evenly among its nodes, while minimizing the cost of redistributing chunks as the cluster expands. In this section, we visit several algorithms to manage the distribution of a growing collection of data on a shared nothing cluster. In each case we assume an array is divided into storage "chunks", specified by a "stride" in a subset of the array dimensions. Moreover, every array is assumed to have a time dimension, where the insertion time of values is recorded. Obviously, this dimension increases monotonically.

Elastic array partitioners are designed to incrementally reorganize an array's storage, moving only the data necessary to rebalance storage load. This is in contrast to *global* approaches, such as hash partitioning. The classic hash partitioner applies a function to each chunk, producing an integer, and this hash value, modulus the number of cluster nodes, assigns chunks to nodes. Using this technique will move most or all of the data at each redistribution, a high cost for regularly expanding databases.

Elastic and global partitioners expose an interesting trade off between locally and globally optimal partitioning plans. Most global partitioners guarantee that an equal number of chunks will be assigned to each node, however they do so with a high reorganization cost, since they shift data among most or all of the cluster nodes. In addition, this class of approaches are not skew-aware; they only reason about logical chunks, rather than physical storage size. Elastic data placement dynamically revises how chunks are assigned to nodes in an expanding cluster, and also makes efficient use of network bandwidth, because data moves between a small subset of nodes in the cluster. Note that skewed data will have significant variance in the stored chunk sizes. Hence, when a reorganization is required, elastic partitioners identify the most heavily loaded nodes and split them, passing on approximately half of their contents to new cluster additions. This rebalancing is skew resistant, as it evaluates where to split the data's partitioning tables based on the storage footprint on each host.

In this work, we evaluate a variety of range and hash partitioners. Range partitioning stores arrays clustered in dimension space, which expedites group-by aggregate queries, and ones that access data contiguously, as is common in linear algebra. Also, many science workloads query data spatially, and benefit greatly from preserving the spatial ordering of their inputs. Hash partitioning is well-suited for fine-grained storage partitioning, because it places chunks one at a time, rather than having to subdivide planes in array space. Hence, equi-joins and most "embarrassingly parallel" operations are best served by hash partitioning.

3.2 Hash Partitioning

In this section, we discuss two elastic hash partitioners. The first, *Extendible Hash*, is optimized for skewed data, and the alternative, *Consistent Hash*, targets arrays with chunks of uniform size. Both algorithms assign chunks to nodes one at a time. Each chunk is numbered 1...k based on its position within the source array, and the engine hashes the chunk number to find its location.

Extendible Hash [15] is designed for distributing skewed data. The algorithm begins with a set of hash buckets, one per node. When the cluster increases in size, the partitioner splits the hash bucket of the most heavily loaded hosts, partially redistributing their contents to the new nodes. For data that is evenly distributed throughout an array, Consistent Hash [16] is a beneficial partitioning strategy. Think of the hash map distributed around the circumference of a circle, where both nodes and chunks are hashed to an integer, which designates their position on the circle's edge. The partitioner finds a chunk's destination node by tracing the circle's edge from the hashed position of the chunk in the clockwise direction, assigning it to the first node that it finds. When a new node is inserted, it accepts chunks from several pre-existing nodes, producing a partitioning layout with an approximately equal number of chunks per node. This algorithm assigns the logical chunks evenly over the cluster, however, it does not, address storage skew, because the chunk-to-node assignments are made independent of individual chunk sizes.

3.3 Range Partitioning

Range partitioning has the best performance for queries that have clustered data access, such as grouping by a dimension or finding the k-nearest neighbors for a cell. In this section, we examine three strategies for clustered data partitioning, n-dimensional (K-d Tree and Uniform Range), and time-based (Append).

A *K-d Tree* [12] is an efficient strategy for range partitioning skewed, multidimensional data. The K-d Tree stores its partitioning table as a binary tree. Each node is represented by a leaf, and all non-leaf nodes are partitioning points in the array's space. To locate a chunk, the algorithm traverses the tree, beginning at the root node. If the root is a non-leaf node, the partitioner compares the chunk's first logical coordinate to the node's split point, progressing to the child node on the chunk's side of the divide. The lookup continues until it reaches a leaf node, completing this operation in logarithmic time.

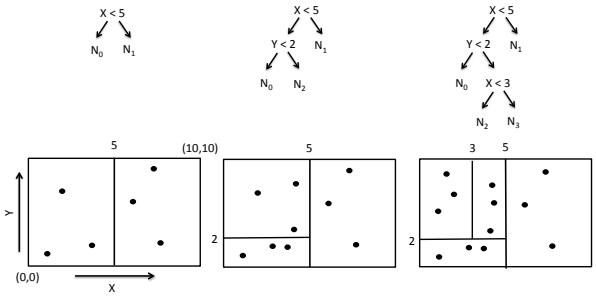


Figure 1: An example of K-d Tree array partitioning.

In this scheme, each host is responsible for an n-dimensional subarray, and partitioning points are defined as planes in array space. When a new cluster node is added, the algorithm first identifies the most heavily loaded host. If this is the first time that the host's range has been subdivided, the partitioner traverses the array's first dimension until it finds the point where there exists an equal number of cells on either side of it, the dimension's median. The splitter cuts the hot host's range at this point, reassigning half of its contents to the new addition. On

subsequent splits, the partitioner cycles through the array's dimensions, such that each is cut an approximately equal number of times.

Figure 1 demonstrates a K-d Tree that begins by partitioning an array over two nodes; it is divided on the x-axis at the dimension's midway point, 5. The left hand side accumulates cells at a faster rate than the right, prompting the partitioner to cut its y-axis for the second split, where this dimension equals 2. Next, the K-d Tree returns to cutting vertically as the third node joins the cluster.

A second variation, *Uniform Range*, optimizes for unskewed arrays. In this approach the array assigns an equal number of chunks to each node, and it executes a complicated global reorganization at every cluster expansion to maintain this balance. This algorithm starts by constructing a tall, balanced binary tree to describe the array's dimension space. If the partitioner has a height of h, then it has $l = 2^h$ leaf nodes, where l is much greater than the anticipated cluster size. Each non-leaf node in the tree specifies a split point, where a dimension is divided in half, and the tree rotates through all dimensions, subdividing each with an equal frequency. For a cluster comprised of n hosts, Uniform Range assigns its l leaf nodes in groups of size $\frac{l}{n}$, where the leaves are sorted by their traversal order in the tree. When the cluster scales out, this tree is rebalanced by calculating a new $\frac{l}{n}$ slice for each host; hence the partitioner maintains multidimensional clustered array storage, without compromising load balancing. This approach has a high reorganization cost compared to K-d Tree, because such operations move most or all of the array at each rebalancing.

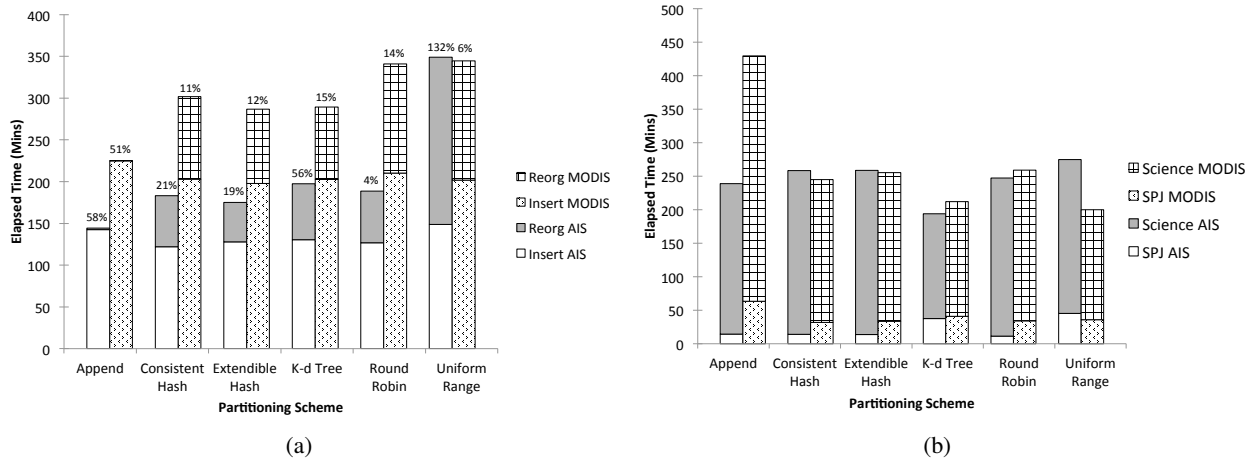
A third variant of range partitioning is an *Append* strategy. Append subdivides the no-overwrite array on its time dimension alone, by sending each newly inserted chunk to the first node that is not at capacity. A coordinator maintains a count of the storage allocated at each node, spilling over to the next one when the current one is full. This partitioner works equally well for skewed and uniform data distributions, as it adjusts its layout based on storage size, rather than logical chunk count. Append partitioning is attractive because it has minimal overhead for data reorganizations. When a node is added, it stores new chunks when its predecessor becomes full, making it an efficient option for a frequently expanding cluster. On the other hand, this partitioner has poor performance if the cluster adds many nodes at once, since it will use only one new node at a time.

To recap, we have studied a collection of elastic partitioning algorithms. All, except Append, are designed for either a skewed or uniform data distribution. The schemes (excluding Uniform Range) are all incremental; hence they move a subset of the chunks during a scale out operation, writing only to new nodes. In the next section we compare these schemes with a baseline strategy of *Round Robin* allocation. Round Robin assigns nodes to chunks circularly based on chunk number. Hence, if chunk n is being assigned to a cluster of k nodes, this approach will send it to node n modulus k. The baseline evenly distributes the logical chunk numbers over all nodes, however when the cluster expands, all hosts usually shift their data to add one or more hosts to the partitioning rotation.

3.4 Elastic Partitioner Results

We studied elastic partitioning on a cluster starting with two nodes, which expands in increments of two nodes. Our MODIS case study consists of adding 630 GB to an empty database over 14 days in 1 day increments. In addition, we experimented with adding 400 GB of AIS ship data spanning 3 years, inserted in 36 batches, each covering one month. In both cases, this is the rate at which we receive data from its source.

Figure 2(a) demonstrates the performance of the various partitioning schemes on our two benchmarks during the ingest and reorganization phases. For both of our use cases, the insert time is nearly constant cost because all of the schemes load the data and then spread it over the cluster according to the partitioning scheme under evaluation. The cost of redistribution during the three expansions is less uniform. Append is a clear winner in this space, as it does not rebalance the data; it only shifts future additions to the newly provisioned nodes. K-d Tree and hash partitioning both perform well, as they incrementally reorganize the data by writing only to the newly provisioned nodes. Round Robin and Uniform Range globally redistribute the data, and hence have a higher time requirement.



(a) (b)

Figure 2: (a) Elastic partitioner performance for data ingest and reorganization. Percentages denote relative standard deviation of storage distribution. (b) Benchmark performance of elastic partitioning schemes.

We assess the evenness of a scheme's storage distribution by the relative standard deviation (RSD) of each host's load, and the percentages are shown in Figure 2(a). After each insert, this metric analyzes the database size on each node, taking the standard deviation and dividing by the mean. We average these measurements over all inserts to indicate the storage variation among nodes as a percent of the average host load, and a lower value indicates a more balanced partitioning.

The randomized allocations, Consistent Hash, Extendible Hash, and Round Robin do best because they subdivide the data at its finest granularity, by its chunks. Append exhibits poor load balancing overall; this scheme only writes to one node at a time, no matter how many are added. Skew strongly influences the performance of our range partitioners. AIS has significant hotspots near major shipping ports, hence it has a very imbalanced storage partitioning for Uniform Range, although this scheme is the best for our uniformly distributed MODIS data. K-d Tree also has difficulty handling skew because it can only subdivide one dimension for each node addition. A more comprehensive approach (i.e., quad-trees) may work better and will be studied as future work.

Figure 2(b) shows the query performance of the two use cases on the benchmark queries in between each load cycle. Our benchmarks demonstrate that for SPJ queries, the partitioners perform proportionally to the evenness of their storage distribution. The science benchmarks show that clustered data access is important for array-centric workloads. K-d Tree has the best performance for both workloads, as it facilitates clustered reads, and is moderately skew resistant. Append performed poorly in the science benchmarks for two reasons. First, it balances query execution poorly because this strategy partitions the data by time; therefore when a pair of new nodes are added, one of the hosts will not be used immediately. Second, new data is "hotter" in our benchmarks, as some of the queries "cook" the new measurements into results, and compare them with prior findings.

In summary, we found that K-d Tree was the most effective partitioner for our array workloads, as it breaks up hotspots, and caters effectively to spatial querying. For data loading and reorganization, the append approach is fastest, but this speed comes at a cost when the database executes queries over imbalanced storage. When we consider end-to-end performance, summing up the findings in Figures 2(a) and 2(b), K-d Tree is the fastest solution for MODIS, whereas Append and the skewed range partitioner are on par for AIS.

4 Query Processing

At first blush, one could simply use a traditional cost-based relational optimizer, and then repurpose it for the operators found in an array DBMS. There are two reasons why this is not likely to succeed. First, the commutative join and filtering operations from relational systems are not prevalent in scientific workloads.

Instead, there are many more non-commutative operations that cannot be pushed up or down the query plan tree. Earth science [17], genomics [21], and radio astronomy [22] all exhibit this recipe of few joins paired with complex analytics.

Hence, the opportunities for rearranging the query execution plan are more limited. Second, it is not obvious that the relational join tactics (hash join, merge sort, iterative substitution) are the best choices for array data.

In the common case where skew is present, we have invented a novel *n-way shuffle join*, which is effective at both balancing the query load across nodes as well as minimizing network traffic to accomplish the join. In short, the algorithm entails locating sparse and dense areas of the arrays, and then sending sparse areas to the corresponding dense ones to perform the join. Hence, it minimizes the amount of network traffic to accomplish a distributed join. We have shown that this algorithm can be added to the merge-sort and iterative substitution join algorithms in SciDB and never results in significantly worse performance than non-shuffle approaches. When dense areas in one array line up with sparse areas of a second array, dramatic performance improvements result. We also propose a second approach, which we call a *load balancing shuffle join*. This approach assigns chunks to nodes such that each node executes the join on the same number of cells. Our implementation uses integer programming to find a chunk allocation that minimizes data movement subject to achieving an even distribution of join work across our cluster nodes.

In Figure 3, we evaluated merge join for a pair of 2D 100 GB synthetic arrays that share dimensions and logical chunk sizes. We varied the degree of skew for our input data; for the uniform case all of our chunks are of the same size. For each other case, the per-node partitioning follows a Zipfian distribution, where the parameter denotes the skewness of the input, and higher values denote greater imbalance in the data's distribution. As a baseline, we use the traditional *move-small* strategy of sending the smaller array to the nodes of the larger one for merge joins. Figure 3 shows the time used for data alignment (DA) and join execution (JE).

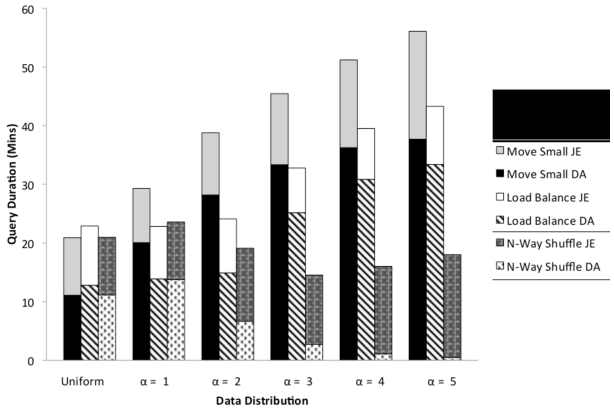
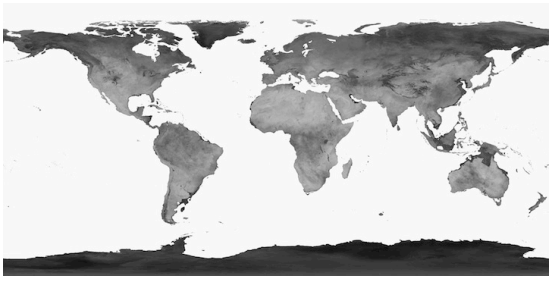


Figure 3: Join duration with varying skew and data shuffling strategies.

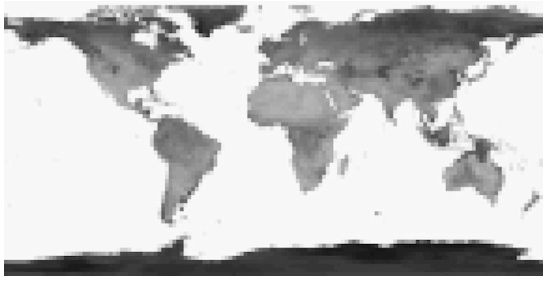
For the uniform case, all of the algorithms perform comparably. When we have slight skew ($\alpha = 1$), the load balancer transfers about the same amount of data as the n-way shuffle, but it has better parallelism in the join execution, producing a slight win. As the skew increases, the n-way shuffle significantly outperforms the other techniques, moving less and less data. For $\alpha \geq 3$, we consistently have a speedup of 3X. We are in the process of finishing a SciDB implementation for our algorithms, obtaining more complete performance numbers, and writing a paper for publication on this work. We then expect to continue this work by developing a complete optimizer that integrates this join processing with the other SciDB operators, and addresses the ordering of cascading joins.

5 Visualization

Modern DBMSs are designed to efficiently store, manage, and perform computations on massive amounts of data. As a result, more analytics systems are relying on databases for the management of big data. For example, many popular data analysis systems, such as Tableau [9], Spotfire [10], R and Matlab, are actively used in conjunction with database management systems. Furthermore, in [23] they show that distributed data management and analysis systems like Hadoop [3] have the potential to power scalable data visualization systems.

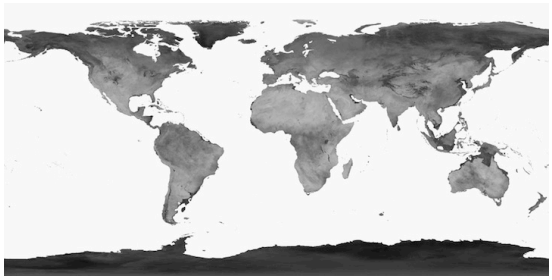


(a) Baseline heatmap visualization of NDSI data



(b) Aggregating NDSI data at 10,000 points resolution

(c) Aggregating NDSI data at 100,000 points resolution



(d) Aggregating NDSI data at 1,000,000 points resolution

Figure 4: Heatmap visualizations produced by ScalaR, using aggregation to reduce the NDSI dataset stored in `ndsi_array`. Dark areas represent high amounts of snow cover

Unfortunately, many information visualization systems do not scale seamlessly from small data sets to massive ones. A given visualization may work well on a small data set with a modest number of points, but will paint the screen black when presented with an order of magnitude more data. Having the individual visualization system deal with this scalability issue has two major flaws. First, code must be included in perhaps many individual modules to accomplish this task, an obvious duplication of effort. Second, visualizations run on the client side of a client-server interface, for which large amounts of data may have to be passed back and forth and computing resources are more limited than on the server side of the boundary. Obviously, a shared server-side system, running close to the DBMS, should prove attractive.

To address these issues, we have developed a flexible, three-tiered scalable interactive visualization system named ScalaR [11] that leverages the computational power of modern DBMSs for back-end analytics and execution. ScalaR decouples the visualization task from the analysis and management of the data by inserting a middle layer of software to mediate between the front-end visualizer and the back-end DBMS. ScalaR has been implemented for SciDB, but is back-end agnostic in design, as its only requirements are that the back-end must support a query API and provide access to metadata in the form of query plans. ScalaR relies on query plan estimates computed by the DBMS to perform resolution reduction, i.e., to summarize massive query result sets on the fly. ScalaR's resolution reduction model can be applied to a wide variety of domains, as the only requirement for using ScalaR is that the data is stored in a DBMS. For example, it has been used with SciDB to visualize NASA MODIS satellite imagery data [1], LSST astronomy data [4], and worldwide earthquake records [8].

We have run several experiments with ScalaR, assessing the runtime performance and resulting visual quality of various reduction queries over two SciDB arrays containing NASA MODIS satellite imagery data. Specifically, we visualized normalized difference snow index (NDSI) calculations over the entire world at various output sizes (data resolutions). The NDSI measures the amount of snow located at a particular latitude-longitude cell on the earth. As a baseline, we also ran the query on the raw data, with no resolution reduction.

Figure 4(a) shows a visualization of the baseline for one of these arrays, which we will refer to as ndsi_array. The baseline query completed in 5.68 seconds. The ndsi_array is a dense SciDB array, containing roughly 6 million data points. To perform aggregation reductions, we used SciDB's `regrid` operation, which divides an array into equal-sized sub-arrays, and returns summaries over these sub-arrays by averaging the array values. Figures 4(b) through 4(d) are the visualized results of these four reduction queries. The smallest reduction to 10 thousand data points was the fastest with 1.35 seconds. The other reductions were comparable to the baseline. Thus we can see that ScalaR produces small aggregate summaries of the NDSI data very quickly, but the resulting image is blurred due to the low number of data points in the result. However, at a resolution of 100,000 data points we produce a visualization that is is a very close approximation of the original with an order of magnitude fewer data points.

Our current focus is twofold. First we are conducting a substantial user study of ScalaR at UCSB on MODIS data and at the University of Washington on astronomy data. In each case scientists have agreed to test the system on real world problems. Besides obtaining feature and ease-of-use feedback, we will obtain traces of user activity. These traces will be used to train our middle tier prefetching system, whereby we plan to use all available server resources to predict (and prefetch) future user data. Our system contains two experts, one is a path expert, which predicts the direction and speed on user browsing and fetches forward along this path. Our second expert looks for patterns in the recent user activity and then looks for similar patterns further afield. Depending on their success, experts are given more or less space in the cache and more or less machine resources to prefetch items. Our prefetching system is nearly operational and we hope to test it in the near future.

We also plan to extend ScalaR's front-end visualizer, which currently requires users to specify all components of the final visualization, including what resolution reduction technique to use, the data limit to impose on the back- end, the x- and y-axes, the scaling factor, and coloring. Lack of experience with visualizing the underlying data can make it difficult for users to make these specific visualization choices in advance, and can result in many iterations of trial and error as users search for a suitable way to visualize the data.

To help users quickly make better visualization choices, we are designing a predictive model for identifying the most relevant visualization types for a given data set. We will use the model to produce sample visualizations in advance, reducing the number of choices a user must make and simplifying his visualization task.

To train our predictive model, we are creating a corpus of visualizations from the web. For each visualization we have access to the data set that was used to create it. We will use this underlying data to learn what features potentially correspond to specific visualization types. The visualizations are collected from a wide variety of web sources, including the Many Eyes website [5], various ggplot2 examples [2], and the D3 image gallery [13].

6 Conclusions

This paper has presented the flavor of on-going array DBMS research. It presented our research directions in the areas of elasticity, query processing and visualization. Other work in this area has been omitted because of space limitations including provenance [24, 25] and on using the DBMS for the MODIS processing pipeline [18].

References

[1] Benchmarks for Elastic Scientific Databases. `http://people.csail.mit.edu/jennie/elasticity_benchmarks.html`.

[2] ggplot2. `ggplot2.org`.

[3] Hadoop. `http://hadoop.apache.org/`.

[4] Large Synoptic Survey Telescope. `http://www.lsst.org/lsst/`.

[5] Many Eyes. http://www-958.ibm.com/software/data/cognos/manyeyes/.

[6] NASA MODIS. http://modis.gsfc.nasa.gov/.

[7] U.S. Coast Guard AIS Ship Tracking Dataset. http://marinecadastre.gov/AIS/default.aspx.

[8] U.S. Geological Survey Earthquake Hazards Program. http://earthquake.usgs.gov/.

[9] Tableau Software. http://www.tableausoftware.com/, 2013.

[10] TIBCO Spotfire. http://spotfire.tibco.com/, 2013.

[11] L. Battle, M. Stonebraker, and R. Chang. Dynamic Reduction of Query Result Sets for Interactive Visualization. In *IEEE BigDataVis Workshop*, 2013.

[12] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[13] M. Bostock, V. Ogievetsky, and J. Heer. D3: Data-driven documents. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 2011.

[14] J. Duggan and M. Stonebraker. Elasticity in Array DBMSs. (submitted for publication).

[15] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing–a fast access method for dynamic files. *ACM Trans. Database Syst.*, 4(3):315–344, 1979.

[16] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. STOC, 1997.

[17] G. Planthaber. Modbase: A scidb-powered system for large-scale distributed storage and analysis of modis earth remote sensing data. Master's thesis, MIT, 2012.

[18] G. Planthaber, M. Stonebraker, and J. Frew. EarthDB: scalable analysis of MODIS data using SciDB. In *BigSpatial*, 2012.

[19] M. Stonebraker, P. Brown, D. Zhang, and J. Becla. SciDB: A Database Management System for Applications with Complex Analytics. *Computing in Science Engineering*, 15(3):54–62, 2013.

[20] M. Stonebraker, S. Madden, and P. Dubey. Intel "big data" science and technology center vision and execution plan. *SIGMOD Record.*, 42(1):44–49, 2013.

[21] R. Taft, M. Vartek, N. R. Satish, N. Sundaram, S. Madden, and M. Stonebraker. Genbase: A complex analytics genomics benchmark. Technical report, MIT CSAIL, 2013.

[22] G. van Diepen. SciDB Radio Astronomy Science Case. http://scidb.org/UseCases/radioAstronomy_usec

[23] H. Vo, J. Bronson, B. Summa, J. Comba, J. Freire, B. Howe, V. Pascucci, and C. Silva. Parallel visualization on large clusters using MapReduce. In *Large Data Analysis and Visualization (LDAV)*, 2011.

[24] E. Wu and S. Madden. Scorpion: Explaining Away Outliers in Aggregate Queries. In *VLDB*, 2013.

[25] E. Wu, S. Madden, and M. Stonebraker. SubZero: A fine-grained lineage system for scientific databases. In *ICDE*, 2013.