Incremental Elasticity For Array Databases

Jennie Duggan
MIT CSAIL
jennie@csail.mit.edu

Michael Stonebraker
MIT CSAIL
stonebraker@csail.mit.edu

ABSTRACT

Relational databases benefit significantly from elasticity, whereby they execute on a set of changing hardware resources provisioned to match their storage and processing requirements. Such flexibility is especially attractive for scientific databases because their users often have a no-overwrite storage model, in which they delete data only when their available space is exhausted. This results in a database that is regularly growing and expanding its hardware proportionally. Also, scientific databases frequently store their data as multidimensional arrays optimized for spatial querying. This brings about several novel challenges in clustered, skew-aware data placement on an elastic shared-nothing database.

In this work, we design and implement elasticity for an array database. We address this challenge on two fronts: determining when to expand a database cluster and how to partition the data within it. In both steps we propose *incremental* approaches, affecting a minimum set of data and nodes, while maintaining high performance. We introduce an algorithm for gradually augmenting an array database's hardware using a closed-loop control system. After the cluster adds nodes, we optimize data placement for n-dimensional arrays. Many of our elastic partitioners incrementally reorganize an array, redistributing data only to new nodes. By combining these two tools, the scientific database efficiently and seamlessly manages its monotonically increasing hardware resources.

1. INTRODUCTION

Scientists are fast becoming first-class users of data management tools. They routinely collect massive amounts of data from sensors and process it into results that are specific to their discipline. These users are poorly served by relational DBMSs; as a result many prefer to roll their own solutions. This ad-hoc approach, in which every project builds its own data management framework, is detrimental as it exacts considerable overhead for each new undertaking. It also inhibits the sharing of experimental results and their techniques. This problem is only worsening as science becomes increasingly data-centric.

Such users want to retain all of their data; they only selectively delete information when it is necessary [11, 30]. Their storage requirements are presently very large and growing exponentially. For example, the Sloan Digital Sky Survey produced 25 TB of data over 10 years. The Large Synoptic Survey Telescope, however, projects that it will record 20 TB of new data every *night* [16]. Also, the Large Hadron Collider is on track to generate 25 petabytes per year [1].

Scientists use sensors to record a physical process, and their raw data is frequently comprised of multidimensional arrays. For example, telescopes record 2D sets of pixels denoting light intensity throughout the sky. Likewise, seismologists use sensor arrays to record data about subsurface areas of the earth. Simulating arrays on top of a relational system may be an inefficient substitute for representing n-dimensional data natively [29].

Skewed data is a frequent quandary for scientific databases, which need to distribute large arrays over many hosts. Zipf's law observes that information frequently obeys a power distribution [33], and this axiom has been verified in the physical and social sciences [26]. For instance, Section 3.2 details a database of ship positions; in it 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.

There has been considerable research on elasticity for relational databases [14, 13, 18]. Skew was addressed for elastic, transactional workloads in [27], however this work supports neither arrays nor long-running, analytical queries. Our study examines a new set of challenges for monotonically growing databases, where the workload is sensitive to the spatial arrangement of array data.

We call this gradual expansion of database storage requirements and corresponding hardware *incremental elasticity*, and it is the focus of this work. Here, the number of nodes and data layout are carefully tuned to reduce workload duration for inserts, scale out operations, and query execution.

Incremental elasticity in array data stores presents several novel challenges not found in transactional studies. While past work has emphasized write-intensive queries, and managing skew in transactions, this work addresses storage skew, or ensuring that each database partition is of approximately the same size. Because the scientific database's queries are read-mostly, its limiting factor is often I/O and network bandwidth, prioritizing thoughtful planning of data placement. Evenly partitioned arrays may enjoy increased parallelism in their query processing. In addition, such databases benefit significantly from collocating contiguous array data, owing to spatial querying [29]. Our study targets a broad class of elastic platforms; it is agnostic to whether the data is stored in a public cloud, a private collection of servers that grow organically with load, or points in between.

We research an algorithm that determines when to scale out an elastic array database using a control loop. It evaluates the recent resource demand from a workload, and compensates for increased storage requirements. The algorithm is tuned to each workload such that it minimizes cluster cost in node hours provisioned.

The main contributions of this work are:

- Extend partitioning schemes for skew-awareness, n-dimensional clustering, and incremental reorganization.
- Introduce an algorithm for efficiently scaling out scientific databases.
- Evaluate this system on real data using a mix of science-specific analytics and conventional queries.

This paper is organized as follows. The study begins with an introduction to the array data model and how it affects incremental elasticity. Next, we explore two motivating use cases for array database elasticity, and present a workload model abstracted from them. Elastic partitioning schemes immediately follow in Section 4. Section 5 describes our approach for incrementally expanding a database cluster. Our evaluation of the partitioners and node provisioner is in Section 6. Lastly, we survey the related work and conclude.

2. ARRAY DATA MODEL

In this section, we briefly describe the architecture on which we implement and evaluate our elasticity. Our prototype is designed for the context of SciDB [2]; its features are detailed in [10]. It executes distributed array-centric querying on a shared-nothing architecture.

An array database is optimized for complex analytics. In addition to select-project-join queries, the system caters to scientific workloads, using both well known math-intensive operations and user-defined functions. Singular value decomposition, matrix multiplication, and fast fourier transforms are examples of such queries. Their data access patterns are iterative and spatial, rather than index lookups and sequential scans.

SciDB has a shared-nothing architecture, which supports scalability for distributed data stores. This loosely coupled construction is crucial for high availability and flexible database administration. The approach is a natural choice for elasticity - such systems can simply scale out to meet workload demand with limited overhead.

Scientists prefer a no-overwrite storage model [30]. They want to make their experiments easily reproducible. Hence, they retain all of their prior work, even when it is erroneous, resulting in a monotonically growing data store.

Storage and query processing in SciDB is built from the ground up on an *array* data model. Each array has dimensions and attributes, and together they define the logical layout of this data structure. For example, a user may define a 2D array of (x, y) points to describe an image.

An array has one or more named *dimensions*. Dimensions specify a contiguous array space, either with a declared range, or unbounded, as in a time series. A combination of dimension values designates the location of a *cell*, which contains any number of scalar attributes. The array's declaration lists a set of named *attributes* and their types, as in a relational table declaration.

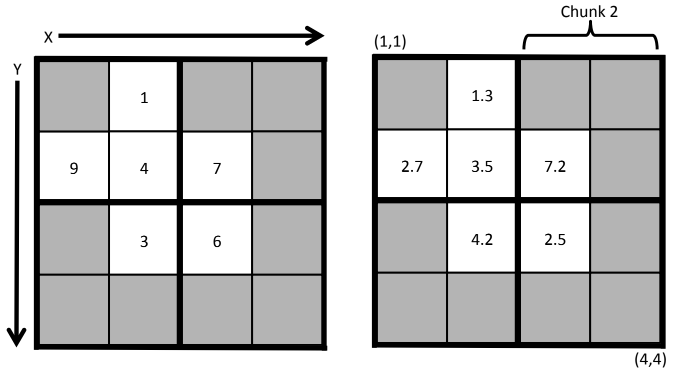

Figure 1: Array example in a scientific database with four 2x2 chunks.

Individual array cells may be empty or occupied, and only non-empty cells are stored in the database. Hence the array's on disk footprint is a function of the number of cells it contains, rather than the declared array size.

Scientists frequently query their data spatially, exploiting relationships between cells based on their position in array space. Therefore, *chunks*, or n-dimensional subarrays, are the unit of I/O and memory allocation for this database. In each array dimension, the user defines a chunk interval or stride. This specifies the length of the chunk in logical cells. Physical chunk size is variable, and corresponds to the number of non-empty cells that it stores. A single chunk is on the order of tens of megabytes in size [10].

SciDB vertically partitions its arrays on disk; each physical chunk stores exactly one attribute. This organization has been demonstrated to achieve one to two orders of magnitude speedup for relational analytics [31]. Array databases exploit the same principles; their queries each typically access a small subset of an array's attributes.

The data distribution within an array is either skewed or unskewed. The latter have approximately the same number of cells in each of its chunks. In contrast, skewed arrays have some chunks that are densely packed and others with little or no data.

Figure 1 shows an example of an array in this database. Its SciDB schema is:

```
A<i:int32, j:float>[x=1:4,2, y=1:4,2]
```

Array A is a 4x4 array with a chunk interval of 2 in each dimension. The array has two dimensions: x and y, each of which ranges from 1..4, inclusive. This schema has two attributes: an integer, i, and a float named j. It stores 6 non-empty cells, with the first containing (1, 1.3). The attributes are stored separately, owing to vertical partitioning.

This array is skewed - the data primarily resides in the center, and its edges are sparse. To balance this array among two nodes, the database might assign the first chunk to one host and store the remaining three on the other. Therefore, each partition would serve exactly three cells.

In summary, the array data model is designed to serve multidimensional data structures. Its vertically partitioned, n-dimensional chunking is optimized for complex analytics, which often query the data spatially. Its shared-nothing architecture makes SciDB scalable for distributed array storage and parallel query processing. In the next section, we will discuss how this database serves two real scientific workloads.

3. USE CASES AND WORKLOAD MODEL

In this section, we examine two motivating use cases for elastic scientific databases and present a benchmark for each. The first consists of satellite imagery for modeling the earth's surface, and the second uses marine vessel tracks for oceanography. We also introduce our cyclic workload model, which is derived from the use cases. The model enables us to evaluate our elastic array partitioners and assess the goodness of competing provisioning plans.

3.1 Remote Sensing

MODIS (Moderate Resolution Imaging Spectroradiometer) is an instrument aboard NASA's satellites that records images of the entire earth's surface every 1-2 days [3]. It measures 36 spectral bands, which scientists use to model the atmosphere, land, and sea.

We focus on the first two bands of MODIS data because queries that model visible light reference them frequently, such as calculating the land's vegetation index. Each array (or "band") has the following schema:

```
Band<si_value:int, radiance:double, reflectance:double,
 uncertainty_idx:int, uncertainty_pct:float,
 platform_id:int, resolution_id:int>[time=0,*,1440,
 longitude=-180,180,12, latitude=-90,90,12]
```

Both of our bands have three dimensions: latitude, longitude and time. The time is in minutes, and it is chunked in one day intervals. Latitude and longitude each have a stride of 12°. We elected this schema because its chunks have an average disk footprint of 50 MB; this chunk size is optimized for high I/O performance as found in [29]. This array consists of measurements (si_value, radiance, reflectance), provenance (platform_id, resolution), and uncertainty information from the sensors. Taken together, they provide all of the raw information necessary to conduct experiments that model the earth's surface. Inserts arrive once per day in our experiments, this is patterned after the rate at which MODIS collects data.

MODIS has a uniform data distribution. If we divide lat/long space into 8 equally-sized subarrays, the average collection of chunks is 80 GB in size with a standard deviation of 8 GB. The data is quite sparse, having less than 1% of its cells occupied. This sparsity comes about because scientists choose a high fixed precision for the array's dimensions, which are indexed by integers.

3.2 Marine Science

Our second case study concerns ship tracks using the Automatic Identification System [23] from the U.S. Coast Guard. We studied the years 2009-2012 from the National Oceanic and Atmospheric Administration's Marine Cadastre repository [5]; the U.S. government uses this data for coastal science and management.

The International Maritime Organization requires all large ships to be outfitted with an AIS transponder, which emits broadcasts at a set frequency. The messages are received by nearby ships, listening stations, and satellites. The Coast Guard monitors AIS from posts on coasts, lakes, and rivers around the US. The ship tracks array is defined as:

```
Broadcast<speed:int, course:int, heading:int,
 ROT:int, status:int, voyageId:int, ship_id:int,
 receiverType:char, receiverId:string,
 provenance:string>[time=0,*,43200,
 longitude=-180,-66,4, latitude=0,90,4]
```

Broadcasts are stored in a 3D array of latitude, longitude and time, where time is divided into 30-day intervals, and recorded in minutes. The latitude and longitude dimensions encompass the US and surrounding waters. Each broadcast publishes the vessel's position, speed, course, rate of turn, status (e.g., in port, underway), ship id, and voyage id. The broadcast array is the majority of AIS's data, however it also has a supporting vessel array. This data structure has a single dimension: vessel id. It identifies the ship type, its dimensions (i.e., length and width), and whether it is carrying hazardous materials. The vessel array is small (25 MB), and replicated over all cluster nodes.

AIS data is chunked into 4°x4°x30 day subarrays; their stored size is extremely skewed. The chunks have a median size of 924 bytes, with a standard deviation of 232 megabytes. Nearly 85% of the data resides in just 5% of the chunks. This skew is a product of ships congregating near major ports. In contrast, MODIS has only slight skew; the top 5% of chunks constitute only 10% of the data. AIS injects new tracks into the database once every 30 days, reflecting the monthly rate at which NOAA reports this data.

3.3 Workload Benchmarks

Our workloads have two benchmarks: one tests conventional analytics and the other science-oriented operations. The first, Select-Project-Join, evaluates the database's performance on cell-centric queries, which is representative of subsetting the data. The science benchmark measures database performance on domain-specific processing for each workload, which often accesses the data spatially. Both benchmarks refer to the newest data more frequently, "cooking" the measurements into data products. The individual benchmark queries are available at [4].

The MODIS benchmark extends the MODBASE study [28], which is based on interviews with environmental scientists. Its workload first interrogates the array space and the contents of its cells generating derived arrays such as the normalized difference vegetation index (NDVI). It then constructs several earth science specific projections, including a model for deforestation and a regridding the sparse data into a coarser, dense image.

AIS is used by the Bureau of Ocean Management to study the environment. Their studies include coastline erosion modeling and estimating the noise levels in whale migration paths. We evaluate their use case by studying the density of ships grouped by geographic region, generating maps of the types of ships emitting messages, and providing up-to-date lists of distinct ships.

3.3.1 Select-Project-Join

Selection: The MODIS selection test returns $1/16^{th}$ of its lat/long space, at the lower left-hand corner of the Band 1 array. This experiment tests the database's efficiency when executing a highly parallelizable operator. AIS draws from the broadcast array, filtering it to a densely trafficked area around the port of Houston, to assess the database's ability to cope with skew.

Sort: For MODIS, our benchmark calculates the quantile of Band 1's radiance attribute based on a uniform, random sample; this is a parallelized sort and it summarizes the distribution of the array's light measurements. AIS calculates a sorted log of the distinct ship identifiers from the broadcast array. Both of these operations test how the cluster reacts to non-trivial aggregation.

Join: MODIS computes a vegetation index by joining its two bands where they have cells at the same position in array space. It is executed over the most recent day of data. AIS generates a 3D map of recent ship ids and their their type (e.g., commercial, military). It joins Broadcast with the Vessel array by the ship id attribute.

3.3.2 Science Analytics

Statistics: MODIS's benchmark takes a rolling average of the light levels on Band 1 at the polar ice caps over the past several days for environmental monitoring. AIS computes a coarse-grained map of track counts where the ships are in motion for identifying locations that are vulnerable to coast erosion. Both are designed to evaluate group-by aggregation over dimension space.

Modeling: The remote sensing use case executes k-means clustering over the lat/long and NDVI of the Amazon rainforest to identify regions of deforestation. The ship tracking workload models vessel voyage patterns using non-parametric density estimation; it identifies the k-nearest neighbors for a sample of ships selected uniformly at random, flagging high-traffic areas for further exploration.

Complex Projection: The satellite imagery workload executes a windowed aggregate of the most recent day's vegetation index, a MODBASE query. The aggregate yields an image-ready projection by evaluating a window around each pixel, where its sample space is partially overlapping with that of other pixels, generating a smooth picture. The marine traffic workload predicts vessel collisions by plotting each ship's position several minutes into the future based on their most recent trajectory and speed.

3.4 Cyclic Workload Model

As we have seen, elastic array databases grow monotonically over time. Scientists regularly collect new measurements, insert them into the database, and execute queries to continue their experiments. To capture this process, our workload model consists of three phases: data ingest, reorganization, and processing. We call each iteration of this model a *workload cycle*.

Each workload starts with an empty data store, which is gradually filled with new chunks. Both workloads insert data regularly, prompting the database to scale out periodically. Although the system is routinely receiving new chunks, it never updates preexisting ones, owing to SciDB's no-overwrite storage model.

In the first phase, the data is received in batches, and each insert is of variable size. For example, shipping traffic has seasonal patterns; it peaks around the holidays. Bulk data loads are common in scientific data management [12], as scientists often collect measurements from sensors at fixed time intervals, such as when a satellite passes its base station. Inserts are submitted to a coordinator node, and it distributes the incoming chunks over the entire cluster.

During this phase, the database first determines whether the it is under-provisioned for the incoming insert, i.e., its load exceeds its capacity. If so, the provisioner calculates how many nodes to add, using the algorithm in Section 5. It then redistributes the preexisting chunks, and finally inserts the new ones using an algorithm in Section 4.

In determining when and how to scale out, the system uses storage as a surrogate for load. Disk size approximates the I/O and network bandwidth used for each workload's queries, since both are a function of their input size. Also, storage skew strongly influences the level of parallelism a database executes, making it a reliable indicator for both workload resource demand and performance.

After new data has been ingested, the database executes a query workload. In this step, the scientists continue their experiments, querying both the new data and prior results, and they may store their findings for future reference, further expanding the database's storage.

The elasticity planner in Section 5 both allocates sufficient storage capacity for the database and seeks to minimize the overhead associated with elasticity. We assess the cost of a provisioning plan in node hours. Consider a database that has executed ϕ workload cycles, where iteration i it has N_i nodes provisioned, an insert time of I_i, r_i for its reorganization duration, and a query workload latency of w_i. Hence, this configuration's cost is:

$$cost = \sum_{i=1}^{\phi} N_i(I_i + r_i + w_i) \qquad (1)$$

This metric sums the time for each iteration, and multiplies it by the number of cluster nodes, computing the effective time used by this workload. By estimating this cost, the provisioner approximates the hardware requirements of an ongoing workload.

4. ELASTIC PARTITIONERS FOR SCIENTIFIC ARRAYS

Well-designed data placement is essential for efficiently managing an elastic array 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 this work, we propose and 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 logical layout of their inputs. Hash partitioning is well-suited for fine-grained storage planning. 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.

4.1 Features of Elastic Array Partitioners

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. It also makes efficient use of network bandwidth, because data moves between a small subset of nodes in the cluster.

Table 1 identifies four features of elastic data placement for multidimensional arrays. Each of these characteristics

Partitioner	Incremental Scale Out	Fine-Grained Partitioning	Skew-Aware	n-Dimensional Clustering
Append	✓		✓	
Consistent Hash	✓	✓		
Extendible Hash	✓	✓	✓	
Hilbert Curve	✓		✓	✓
Incr. Quadtree	✓		✓	✓
K-d Tree	✓		✓	✓
Uniform Range				✓

Table 1: Taxonomy of array partitioners.

speeds up the database's cluster scale out, query execution, or both. The partitioning schemes in Section 4.2 implement a subset of these traits.

Partitioners having *incremental scale out* execute a localized reorganization of their array when the database expands. Here, data is only transferred from preexisting nodes to new ones, and not rebalanced globally. All of our algorithms, except Uniform Range, bear this trait.

Fine-grained partitioning, in which the partitioner assigns one chunk at a time to each host, is used to scatter storage skew, which often spans adjacent chunks. Hash algorithms subdivide their partitioning space with chunk-level granularity for better load balancing. Such approaches come at a cost, however, because they do not preserve array space on each node for spatial querying.

Many of the elastic partitioners are also *skew-aware*, meaning they use the present data distribution to guide their repartitioning plans at each scale out. Skewed arrays have chunks with great variance in their individual 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 table based on the storage footprint on each host.

Schemes that have *n-dimensional partitioning* subdivide the array based on its logical space. Storing contiguous chunks on the same host reduces the query execution time of spatial operations [29]. This efficiency, however, often comes at a load balancing cost, because the partitioners divide array space by broad ranges.

4.2 Elastic Partitioning Algorithms

We now introduce several partitioners for elastic arrays, each bearing one or more of the features in Table 1. These partitioners are optimized for a wide variety of workloads, from simple point-by-point access on uniformly distributed data to spatial querying on skewed arrays.

Append: One approach for range partitioning an array with efficient load balancing is an append-only strategy. During inserts, the partitioner sends each new chunk to the first node that is not at capacity. The coordinator maintains a sum of the storage allocated on the receiving host, spilling over to the next one when its current target is full. Append is analogous to traditional range partitioning because it stores chunks sorted by their insert order. This partitioner works equally well for skewed and uniform data distributions, because it adjusts its partitioning table based on storage size, rather than logical chunk count.

Append's partitioning table consists of a list of ranges, one per node. When new data is inserted, the database generates chunks at the end of the preexisting array. Adding a new node is a constant time operation for Append; it creates a new range entry in the partitioning table on its first write.

This approach is attractive because it has minimal overhead for data reorganization. When a new node is added, it picks up where its predecessor left off, making this an efficient option for a frequently expanding cluster. On the other hand, Append has poor performance if the cluster adds several nodes at once, because it only gradually makes use of the new hosts. Also, it is grouped by just one possible dimension: time, using insert order, hence any other dimensions are unlikely to be collocated. In addition, the append partitioner may have poor query performance if new data is accessed at more frequently, as in "cooking" operations which convert raw measurements into data products. The vegetation index in Section 3.3 is one such example.

Consistent Hash: For data that is evenly distributed throughout an array, Consistent Hash [24] is a beneficial partitioning strategy. This hash map is distributed around the circumference of a circle. Nodes and chunks are hashed to an integer, which designates their position on the circle's edge. The partitioner determines a chunk c_i's destination node by tracing the edge, starting at c_i's hashed position. The chunk is assigned to the first node it encounters.

When a new node is inserted, it hashes itself on the circular hash map. The partitioner then traces its map, reassigning chunks from several preexisting nodes to the new addition. This produces a partitioning layout with an approximately equal number of chunks per node. It executes lookup and insert operations in constant time, proportional to the duration of a hash operation.

Consistent Hash strives to send an equal number of chunks to each node. It does not, however, address storage skew. Its chunk-to-node assignments are made independent of the array's data distribution. It also does not cater to spatial querying, because its partitions using a hash function, rather than the data's position in array space.

Extendible Hash: Extendible Hash [19] is designed for distributing skewed data for point querying. The algorithm begins with a set of hash buckets, one per node. When the cluster scales out, the partitioner splits the hash buckets of the most heavily burdened nodes, partially redistributing their contents to the new hosts.

The algorithm determines a chunk's host assignment by evaluating the bits of its hashed value, from least to most significant. If the partitioner has two servers, the first host serves all chunks having hashes with the last bit equal to zero, whereas the remaining chunks are stored on the other node. Subsequent repartitionings slice the hash space by increasingly significant bits.

This approach uses the present data distribution to plan its repartitioning operations, therefore it is skew-aware. Because it refers to a flat partitioning table, Incremental Hash does not take into account the array's multidimensional structure. This makes for a more even data distribution at the expense of spatial query performance.

Hilbert Curve: For point skew, where small fragments of array space have most of the data, we propose a partitioner based on the Hilbert space-filling curve. This continuous curve serializes an array's chunk order such that the Euclidean distance between neighboring chunks on the curve is minimized. The algorithm assigns ranges of chunks based on this order to each node, hence it partitions at a finer granularity than approaches that slice on dimension ranges.

In its most basic form, the Hilbert Curve is only applicable to 2D squares; our partitioner uses a generalized implementation of it for rectangles [32]. Like Extendible Hash, when the cluster reaches capacity, it splits the most heavily burdened node(s) at the median of their range. This scheme does so at the granularity of a chunk, which may result in better load balancing than dimension ranges.

This approach is optimized for skew, because it splits the hottest partition one at a time, and it does so incrementally by only reorganizing the most in-demand nodes. It facilitates spatial querying by serializing the chunks along the space-filling curve.

Incremental Quadtree: The Quadtree is a tree data structure that recursively subdivides a 2D space into n-dimensional ranges. It is so named because at each repartitioning, it quarters the overloaded partition(s). This scheme is represented as a tree where each non-leaf node has exactly four children [20]. The Quadtree is an example of the more general binary space partitioners, which accommodate an arbitrary number of dimensions. If a Quadtree partitioner assigned one host to each of its leaf nodes, the algorithm would not be capable of incremental scale out. Each time a node reached capacity, the database would need to redistribute its contents over four nodes, three of which are new.

To address this challenge, we propose an Incremental Quadtree. It has the flexibility to subdivide an array on arbitrary dimensions, while maintaining the logical array space. The Incremental Quadtree selects the most loaded nodes for splitting for scalable repartitioning decisions. When a node is split, the partitioner decides how to most evenly redistribute the data based on its four quarters.

If the splitting host is a single leaf node in the Quadtree, the partition is quartered. The algorithm then identifies the quarter or pair of adjacent quarters having their summed size closest to half of the host's storage and marks this subarray as a new partition. If the host has already been quartered, the adjacent pair that is closest to halving the storage becomes a new partition. This approach makes each node's partition reside at exactly one level of the Quadtree, making contiguous chunks more likely to be stored together. It also reacts directly to areas of skew, splitting on the dimension(s) that are most heavily burdened.

The Incremental Quadtree has efficient scale out operations because in each repartitioning it only sends data to newly provisioned nodes based upon its splits. It is optimized for spatial querying because it moves chunks according to their position in array space. This scheme is skew-aware because at each scale out operation it subdivides the hosts with the largest storage load.

K-d Tree: A K-d Tree [9] is also an efficient strategy for range partitioning over skewed, multidimensional data. When a new machine is added to the cluster, the algorithm splits the most heavily burdened host. In the beginning, the splitting server finds the median point of its storage along the array's first dimension, denoting a plane where there are an equal number of cells on either side of it. The splitter cuts its range at the median, reassigning half of its contents to the new host. On subsequent splits, the partitioner cycles through the array's dimensions, such that each plane is split an approximately equal number of times.

The K-d Tree stores its partitioning table as a binary tree. Each host is represented by a leaf node, and the non-leaf nodes are partitioning points in the array's space. To locate a chunk, the algorithm traverses the tree from the root node. If the root is not a leaf, the partitioner compares the chunk's

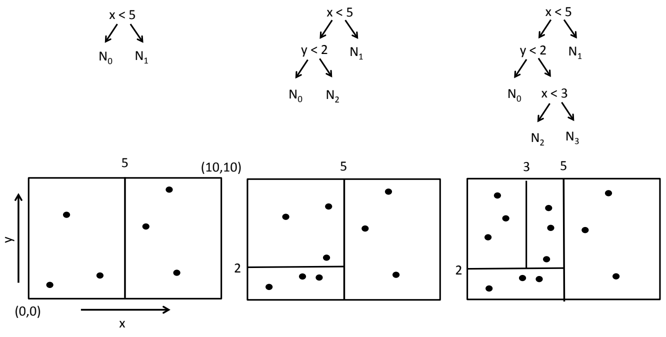

Figure 2: An example of K-d Tree range partitioning for skewed data.

position to its split point, progressing to the child node on the chunk's side of the divide. The lookup continues until it reaches a leaf node. This operation completes in logarithmic time relative to the cluster's node count.

Figure 2 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 N_3 joins the cluster.

This skew-aware scheme limits data movement for nodes that are not affected by recent inserts. It splits only the fullest nodes, making reorganization simple and limiting network congestion. This approach, and all of the other "splitting" algorithms, does have a limitation for unskewed data. If a cluster has a node count that is not a power of two, the partitioner suffers from storage skew because some of its partitions will be the result of fewer splits. For example, a database with 6 nodes distributing 200 GB of data uniformly would have two nodes hosting 50 GB and four nodes with 25 GB. The next algorithm, Uniform Range, addresses this shortcoming.

Uniform Range: We submit an alternative formulation of n-dimensional range partitioning to serve unskewed arrays. This algorithm starts by constructing a tall, balanced binary tree to describe the array's dimension space. The data structure is the same as the one in Figure 2, but with the requirement that each leaf node be an equal depth from the tree's root. 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. Of course, h can be increased as necessary, and the partitioner provides better load balancing with higher h values.

For a cluster comprised of n hosts, Uniform Range assigns its l leaf nodes in blocks of size l/n, where the leafs are sorted by their traversal order in the tree. This maintains multidimensional clustered access, without compromising on load balancing. When the cluster scales out, it rebalances, calculating a new l/n slice for each host.

When a node is added to the partitioner, it iterates over all of the tree leafs, and updates each's destination node based on the new l/n. This is a linear time operation, proportional to l.

This approach has a high reorganization cost compared to the incremental approaches because it executes a global reorganization at each scale out. A new node may create a cascade of moves, especially when several hosts are added to-

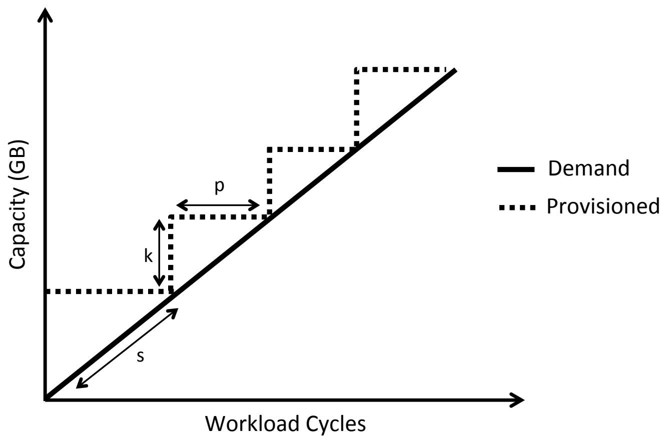

Figure 3: Leading staircase for provisioning array database nodes.

gether. Uniform Range maintains n-dimensional clustering for its arrays, although it is not skew-aware.

To recap, we have examined several partitioning algorithms that we extend for elastic array databases. Each is optimized for either spatial or point queries. Almost all incremental; hence they localize their redistributions by only sending chunks to newly added hosts.

5. ELASTIC PROVISIONING FOR ARRAY DATABASES

In this section we study an approach for provisioning machines for an expanding array database. The elastic planner is designed to provide users with a consistent level of service as they query a growing body of data. We first introduce a control loop for determining when to scale out, and by how many nodes. The tuning of this algorithm to a specific workload is addressed subsequently.

5.1 The Leading Staircase

Identifying the right hardware resources for a array database and its queries is an important challenge. The cluster needs to be large enough to maintain high performance, while not needlessly over-provisioned. If a distributed database expands eagerly, adding several nodes at each scale out, the system will save reorganization time and have better throughput at a higher provisioning cost. In contrast, if the planner injects one node at a time, it will rebalance the data more frequently, albeit using fewer servers. The provisioner quantifies this trade-off by minimizing its operating costs as defined by Equation 1 in Section 3.

To address this challenge, we propose a leading staircase algorithm, as depicted in Figure 3. Its name refers to the observation that an elastic array database expands its cluster size in a series of discrete steps, much like a flight of stairs increases in height as they are climbed. When the demand and provisioned curve first intersect, the cluster has reached capacity. The provisioner now models its next step. The planner projects future demand via a Proportional-Derivative (PD) Control Loop [7] based on the last s workload cycles. Using this projected demand, it adds k nodes, raising the database's capacity to serve the next p workload iterations.

This approach models the rate to expand the database to meet present demand as well as forecasted load increases. By steadily augmenting the cluster, the database enjoys a stable quality of service. The system never coalesces nodes because its resource demands grow monotonically. This is a departure from the management of elasticity for transactional workloads, which vacillate between peaks and troughs of hardware demand.

The provisioner models demand using an extension of the PD controller which it evaluates at each batch of inserts. This control loop is a generic feedback mechanism used for regulating systems such as home thermostats and automotive cruise control. The PD loop revolves around a *set point*, which defines its desired system state. For the elastic database this is a provisioning level that has sufficient resources for the next p workload cycles. Its *process variable* gives the regulator feedback, and the workload's present storage demand provides this guidance.

The PD loop calculates the next step height, k, using two components, the *proportional* and *derivative*. Its proportional term, π, denotes the present provisioning error, as quantified by taking the difference between the storage demand after the incoming insert less the cluster's present capacity. In this model the database consists of N homogeneous nodes, and each having a capacity of c GB. This approach, however, easily generalizes to a heterogeneous cluster by assigning individual capacities to the nodes. For the present load of l_i, it ha

$$\pi = l_i - N \times c \tag{2}$$

If the proportional term is negative, and the system is not over capacity, and the provisioner is done. Otherwise, it evaluates the derivative, Δ, which determines the rate of change of demand over the past s workload iterations

$$\Delta = (l_i - l_{i-s})/s \tag{3}$$

It then calculates the number of new nodes to provision, k, using the following equation:

$$k = \lceil(\pi + p\Delta)/c\rceil \tag{4}$$

The control loop first compensates for the demand that exceeds present capacity in its proportional term. It then forecasts the system's storage requirements for p future inserts based on the demand's derivative. Lastly, the controller converts GBs to nodes provisioned by dividing by the node capacity. It takes the ceiling of this quantity because the provisioner cannot allocate partial nodes.

This algorithm has several appealing properties. First, it is reactive to changing storage requirements at each workload cycle. By modeling current demand, it adjusts the scale out quantity to the most recent workload iterations. Second, it is efficient; the algorithm ensures that the cluster capacity closely matches database load. It is also simple, and as we will demonstrate, effective for elastic array databases.

5.2 Tuning Database Elasticity

The leading staircase has two workload-specific parameters: s, the number of samples from which to project demand, and p, the workload iterations to plan in the next provisioning step. Each captures a facet of how the system reacts to increasing load. The sample count controls algorithm's sensitivity to changes in storage demand, and p

designates how aggressively the provisioner expands as demand rises. In this section we put forth a series of techniques to fit the planner to a given workload.

Sampling What-If Analysis: Our first workload-specific parameter, s, determines how many samples to use for the controller's derivative term. If s is too large, the cluster will be slow to respond to changes in demand; too small, and its scale out will be volatile and inefficient. By determining the sample size that best predicts demand, the planner extrapolates the variability of a workload's long-term demand.

The leading staircase determines the cluster's reactivity by doing a what-if analysis on the observed workload cycles. The tuning starts when the provisioner accepts an insert which surpasses the initial cluster capacity, at the dth workload cycle, and may be refined as the workload progresses.

```
for $s = 1$ to $\psi$ do
  error[s] = 0
  for $i = s + 1$ to $d$ do
    $\Delta_{est} = (l_i - l_{i-s})/s$
    $\Delta_i = l_{i+1} - l_i$
    error[s] += $|\Delta_i - \Delta_{est}|$
  end for
  error[s] = error[s] / (d - s)
end for
return $s$ with minimum errors
```

Algorithm 1: What-if tuning of sampling parameter, s, based on d workload iterations.

The tuner fits s to the workload via a simulation over the previous workload iterations using Algorithm 1. It evaluates $s = 1, .., \psi$, where ψ is the extent of exploration requested by the user. For each s, the tuner evaluates a sliding window over the prior workload performance, learning the derivative over the last s points and predicting the change in demand from the next insert.

For each estimated derivative, the analysis takes the absolute difference between its prediction and the corresponding observed demand to quantify model error. It then averages the errors, electing the one with the lowest mean.

Scale Out Cost Model Next, the tuner calibrates how rapidly to expand the cluster in response to an ever-increasing demand for storage, the p in Figure 3. Recall that this term denotes how many workload iterations into the future the provisioner plans in its next scale out. A lazy configuration has a low p, looking ahead perhaps one insert at every expansion. This arrangement calls for sizable data movement, because at each expansion the database redistributes preexisting chunks to the new nodes. On the other hand, a high p setting prompts the database to expand eagerly, and it can better parallelize the rebalancing with larger stair steps. Over-provisioning is more likely for an eager configuration, reducing the benefits of elastic infrastructure.

The tuning algorithm uses an analytical cost model to compare competing values of p. The model approximates the cost of a p configuration in node hours, simulating Equation 1. For each p, the tuner simulates m workload iterations in the future, where m is a user-supplied parameter.

The tuner models the three phases of each workload iteration, its insert, potential reorganization, and query workload. For each insert and reorganization, it approximates the cost of I/O, at δ per GB, and network transfer, where the t denotes the bandwidth cost. Both of these constants are derived empirically in our experiments. After that, the model estimates the iteration's query workload latency using the last measurement of benchmark time, and scaling it by the projected increases in storage and parallelism. The model multiplies the projected duration of each workload cycle by its node count, estimating its summed node hours.

The cluster begins with a configuration of N_0 nodes. Recall that the modeling begins after d iterations, when the cluster first reaches capacity, hence $l_d \geq N_0 \times c$. The cost model works from a constant insert rate of μ, derived from the increase in storage over the last s workload cycles. The simulation starts at the cluster's present state, therefore l_0 is assigned to l_d. Workload cycle i has a projected load of:

$$l_i = l_0 + \mu \times i \tag{5}$$

Next, the model calculates the node count at iteration i as:

$$N_{i,p} = \begin{cases} N_{i-1,p} & \text{If } l_i \leq N_{i-1,p} \times c \\ \lceil (l_0 + \mu \times (i + p))/c \rceil & \text{otherwise} \end{cases}$$

If the l_i is less than the provisioned capacity, then the cluster size remains the same, otherwise it recalculates the capacity for p upcoming workload iterations. Now that the model has estimated its load and node count for iteration i, it predicts the workload's insert time:

$$I_{i,p} = \mu \frac{1}{N_{i,p}} \delta + \mu \frac{N_{i,p} - 1}{N_{i,p}} t \tag{6}$$

The coordinator node ingests the data, distributing it evenly among the cluster's $N_{i,p}$ hosts. The cost model estimates that the database inserts $1/N_{i,p}^{th}$ of the data locally, at an I/O cost of δ. The remaining fraction of the data is evenly distributed over $N_{i,p} - 1$ nodes, at a rate of t.

We next estimate the cost of rebalancing the database in the event of a cluster expansion:

$$r_{i,p} = \frac{l_i}{N_{i,p}} (N_{i,p} - N_{i-1,p}) t \tag{7}$$

The estimator determines the average load per node in the new configuration, $l_i/N_{i,p}$, and multiplies it by the number of new nodes, $N_{i,p} - N_{i-1,p}$. It then converts this estimate from data transferred to network time.

The analysis finally evaluates how the proposed configuration will affect query execution time. The last observed workload cycle had an elapsed query time of w_0, and was distributed among N_0 nodes. A future iteration i has a latency estimate of:

$$w_{i,p} = w_0 \frac{l_i}{l_0} \frac{N_0}{N_{i,p}} \tag{8}$$

The first term denotes the base workload time over l_0 GB of data. The subsequent fraction scales the base work by the projected increase in load over the next i workload cycles. The cost is multiplied by N_0 to capture the work completed during w_0 in node hours, and divided by $N_{i,p}$ to estimate its present level of parallelism.

We estimate the overall cost of a configuration of p as:

$$cost_p = \sum_{i=1}^{m} N_{i,p} (I_{i,p} + r_{i,p} + w_{i,p}) \tag{9}$$

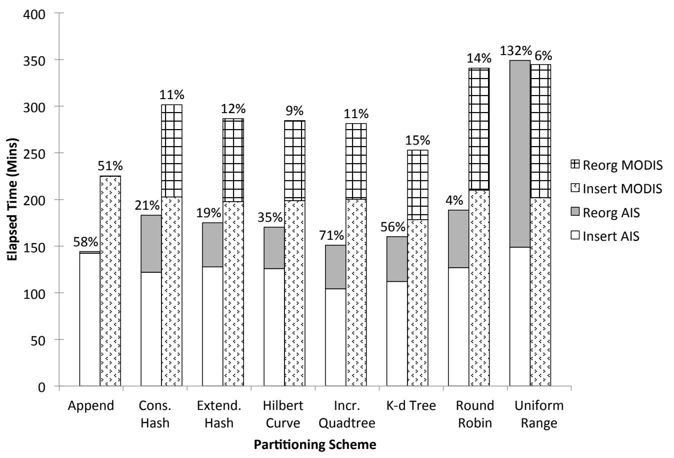

Figure 4: Elastic partitioner insert and reorganization durations. Labels denote load balancing performance in relative standard deviation.

This equation estimates the duration of m workload cycles, and multiplies each iteration time by its projected node count. This cost is in node hours, placing the analytical model in the same terms as the workload cycle cost in Equation 1. It is our goal to minimize the node hours consumed by a workload, hence the tuner selects the p with the lowest overall cost from Equation 9.

6. EXPERIMENTAL EVALUATION

Throughout this paper we have introduced a framework for managing elasticity for array databases. In this section, we first test our partitioning algorithms for both conventional queries and complex, science-oriented analytics, using the benchmarks in Section 3.3. Then, we evaluate the leading staircase provisioner and the tuning of its parameters. Our results will show that incremental handling of data placement and node provisioning produces significant efficiencies for expanding array databases.

6.1 Experimental Setup

We conduct our experiments on a dedicated cluster with 8 nodes, each of which has 20 GB of RAM and 2.1 GHz processors. The AIS ship tracking dataset is 400 GB in size, and its workload cycles are of length 120 days, for quarterly modeling of its 3 years. The MODIS workload contains 630 GB of data, drawn from 14 days, and has one workload cycle per day. We run each of our experiments three times and report the average. Each node has a capacity of 100 GB.

We use *Round Robin* as a baseline for our partitioners. It assigns chunks to nodes in circular order; hence to find chunk i in one of k nodes, Round Robin calculates i modulus k, tasking each host with an equal number of the chunks. The baseline is not designed for incremental elasticity - with each scale out, many of the chunks will shift their location, and it is not skew-aware in its management of storage.

6.2 Elastic Partitioners

Elastic partitioners efficiently reorganize a database as its underlying cluster expands. For this set of experiments, we configure the cluster to start with a pair of nodes, and add two servers each time it reaches capacity, ending with the database occupying all eight hosts in our testbed. We chose

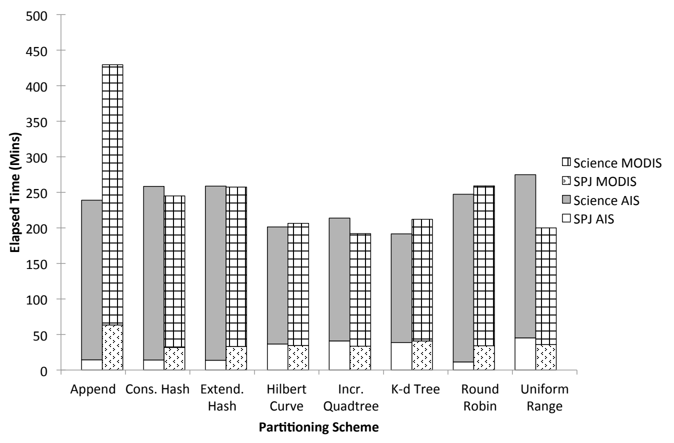

Figure 5: Benchmark times for elastic partitioners.

this configuration to evaluate how each partitioner performs over several moderately-sized cluster expansions, and examine all three phases of our workload model.

6.2.1 Data Loading and Redistribution

In Figure 4, we present the performance of our elastic partitioners for each workload when inserting and rebalancing its database. We assess the evenness of a scheme's storage distribution by the relative standard deviation (RSD) of each host's load. After each insert, this metric analyzes the database size on each node, taking its standard deviation and dividing by the mean. We average these measurements over all inserts to convey the storage variation among nodes as a percent of the average host load, and a lower value indicates a more balanced partitioning.

Insert time is near constant for each workload, independent of the partitioner. This is to be expected, as all algorithms benefit from evenly distributing the new data over their cluster. Append takes slightly longer than the others because it is almost always inserting data over the more costly network link.

Reorganization tells a more complicated story. Append has a fast redistribution for both workloads, as it requires no data movement, although this savings in time produces a poor load balancing owing to its limited use of the most recently added nodes, and its total reorganization and insert time is 20% less than the competition on average.

The global partitioning schemes, Round Robin and Uniform Range, have a considerably higher redistribution time. There is simply more overhead in reshuffling the chunks among all of the nodes. Interestingly, this cost is not as pronounced on the baseline for AIS data. Round Robin's circular addressing scheme parallelizes the transfer of large chunks because they are contiguous. The two global algorithms highlight the importance of incremental approaches; they have a reorganization time that is 2.5X longer on average, and their mean RSD is almost 10% worse.

The remaining algorithms complete in comparable times, with K-d Tree having a slight advantage owing to its precise skew management by splitting nodes at their median. It is interesting to see that Extendible Hash's performance is similar to that of Consistent Hash; the hash space is well-distributed in both cases.

These results clearly demonstrate that the fine-grained partitioners have an advantage in balancing load evenly. Our

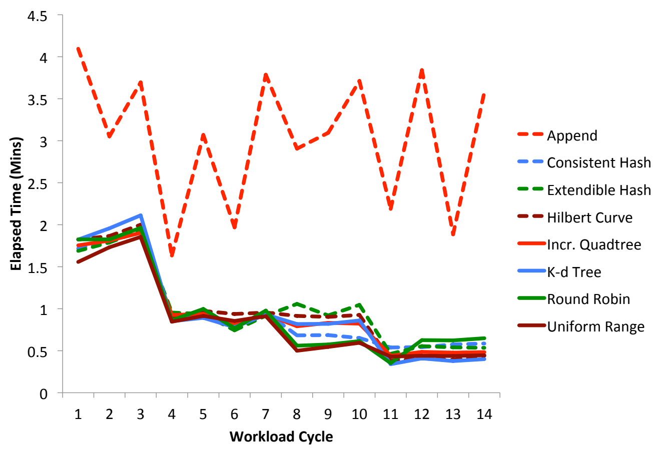

Figure 6: Join duration for unskewed data.

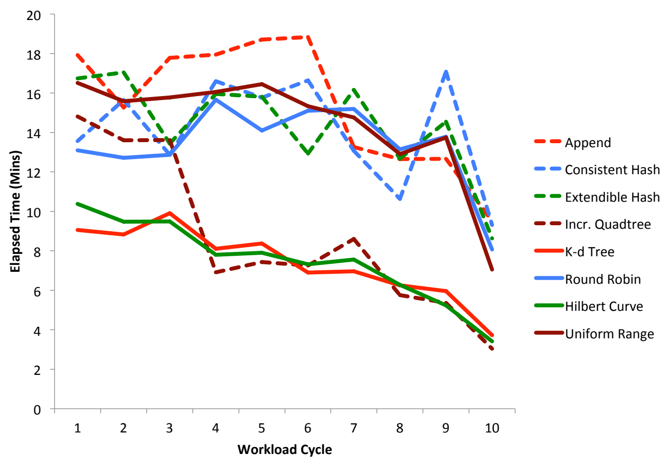

Figure 7: k-nearest neighbors on skewed data.

three fine-grained partitioners - Round Robin, Extendible Hash, and Consistent Hash - have an average RSD of 13%. In contrast, the remaining partitioners have a mean figure of 44%, over three times as much imbalance. If the primary objective of the partitioner is to evenly distribute array chunks, then these approaches are well-suited for this task.

6.2.2 *Benchmark Performance*

Figure 5 shows each partitioner's performance as they execute their workload's benchmark. In this section, we first evaluate each elastic partitioner's performance on conventional query processing and then explore their handling of scientific workloads.

Select-Project-Join: The simpler conventional queries are generally shorter running, stemming from their lack of complex math operations and having scant synchronization among nodes. Each scheme executes the SPJ benchmarks with performance proportional to the evenness of its data distribution. The range partitioners perform more slowly on AIS, as they slice the array's space at a coarser granularity than the hash-based approaches. Uniform Range and Incremental Quadtree on AIS are both particularly susceptible to slowdown due to poor load balancing.

In Figure 6, we examine a join in which the MODIS database computes vegetation indexes over the most recent day of measurements. Append's performance is erratic, as the chunks being joined are stored on only one or two hosts. This behavior is repeated in other experiments, because this partitioning has poor load balancing and does not exploit n dimensional space. All of the other partitioning schemes have similar runtimes as the queried chunks are evenly distributed across all nodes. The non-splitting partitioners, Consistent Hash and Uniform Range, have a slight reduction in latency for cycles 7-10 where the host count is 6. This is attributable to their assignment of an equal number of chunks per node regardless of the cluster size.

Science Analytics: For this complex, math-intensive set of queries, we see that both MODIS and AIS are quite sensitive to n-dimensional clustering. Each does best with a skew-aware, n-dimensional range partitioner, with K-d Tree for AIS and Incremental QuadTree on MODIS. These approaches group chunks based on their position in array space and address skew. Uniform Range slightly outperforms K-d Tree and Hilbert Curve for the MODIS study; the global partitioner's expensive data redistribution at each scale out has marginally better load balancing. AIS shows that Uniform Range is brittle to skew, and this approach has the poorest performance for the ship tracker use case.

AIS's k-nearest neighbor query durations are shown in Figure 7. Recall that this query estimates the volume of marine traffic based on the distance between a randomly selected ship and the vessels closest to it. This calculation profits from preserving the spatial arrangement of an array, as K-d Tree and Hilbert Curve clearly have the fastest execution time, completing this query in half the duration of the baseline. Incremental Quadtree starts with a high-level split, putting it on par with Uniform Range, however, once it has executes a skew-aware redistribution to four nodes, the algorithm readily catches up with the other high performers. The skew-aware range partitioners gradually reduce their latency as more nodes are added because they evenly distribute the time dimension. Append continues to present unstable execution times for this query, due to limited parallelism, and the hash algorithms have a similar variance in their duration, as their partitioning is unclustered.

In total benchmark time, the skew-aware, incremental, multidimensionally clustered strategies are fastest. Incremental QuadTree, Hilbert Curve, and K-d Tree complete their workloads 25% more rapidly than the baseline. Although these partitioners have poorer load balancing, they make up for it in more efficient spatial querying.

6.2.3 *Workload Cost*

In overall workload duration, and by proxy the cost in Equation 1, the top benchmarking strategies prevail. Incremental QuadTree, Hilbert Curve, and K-d Tree boast a greater than 20% mean improvement over the baseline, and similar gains over the hash-based schemes. This confirms that multidimensional clustering trumps pure load balancing for efficient end-to-end workload execution. Intuitively, this is because the database pays just once per scale out operation to reorganize the data, but repeatedly compensates for the scattered distribution of contiguous chunks with every query that accesses the data spatially.

6.3 Database Provisioning

We now evaluate the performance of the leading staircase algorithm for elastically scaling out an array database. This

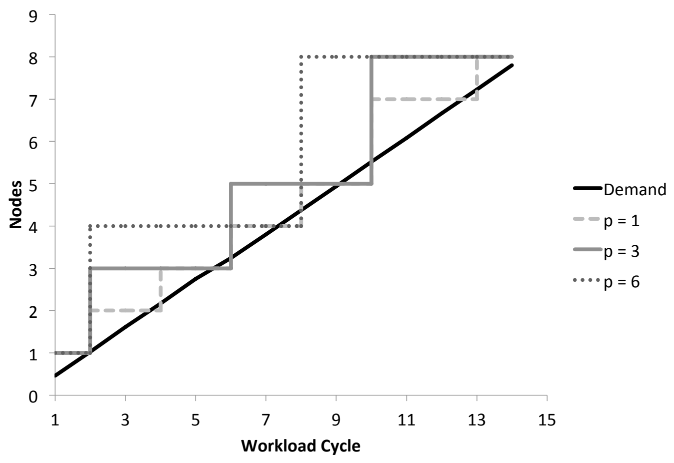

Figure 8: MODIS staircase with varying provisioner configurations.

study begins by demonstrating the leading staircase's behavior on real data, and then verifies the control loop's tuning via what-if analysis and analytical cost modeling.

Staircase Provisioner: We first experiment with controller having set points of 1, 3, and 6 workload cycles on the MODIS workload in Figure 8. Our arrays are partitioned with Consistent Hash, since it has even load balancing and a simple redistribution technique, permitting us to focus on provisioner performance. Each host continues to have a 100 GB capacity, prompting several scale out operations for all configurations, and we use four samples for our control loop, based on the findings in Table 2.

The most conservative approach, in which the provisioner scales out for just one workload cycle into the future, follows the demand curve very closely. Its inflection points are often slightly below demand; it is this state that prompts the scale out. This setting is rarely significantly over-provisioned, however, it executes six reorganizations, reshuffling the data with high frequency.

The moderately responsive configuration, with a scale out parameter of 3, consistently adds two nodes at a time. This approach reorganizes half as often as the prior configuration. This reduced volatility comes at a cost; the cluster has five workload cycles where this setting uses more nodes than the lazy set point.

The last setting, in which $p = 6$, produces an eager cluster expansion. It scales out in large steps, when the database is smaller, requiring less data movement in its redistribution. This configuration executes the query workload very quickly, using increased parallelism.

Staircase Tuning: Our provisioner tuning starts by evaluating s, using the what-if analysis in Algorithm 1. Recall that this parameter denotes the number of samples to evaluate when the control loop derives its next step. We evaluate this tuner on the first third of our iterations for each use case, and test on the remaining ones. The training phase shows what the tuner recommends based on a limited sample before the first scale out, and the test data verifies this parameter against subsequent workload cycles.

Table 2 shows the outcome of this analysis for $s = 1...4$. We test the accuracy of each demand projection by taking the absolute difference between the predicted and observed storage load, and report the average. The strong correlation

Samples (s)	1	2	3	4
AIS Train	1.6	1.8	2.0	2.2
AIS Test	0.6	0.6	0.7	0.7
MODIS Train	2.7	1.8	2	1.6
MODIS Test	4.2	3.2	3.2	3.1

Table 2: Demand prediction error rates (in GB) for various sampling levels in elastic partitioner tuning.

	Cost Estimate	Measured Cost
p = 1	61	13
p = 3	51	12
p = 6	86	16

Table 3: Analytical cost modeling of MODIS controller set points. Costs in node hours.

between the training and testing errors verifies that this parameter is well-modeled.

Each of our datasets is best served by a very different sample count. For AIS, it has the best estimate of future demand when $s = 1$, because the ship tracks have a noticeable variance in their monthly demand. This is unsurprising considering the commercial shipping is subject seasonal shifts due to holiday shopping patterns. In contrast, the MODIS workload realizes the most accurate storage demand estimates with more samples, as it has a steadier long-term patterns in growth.

After configuring s for each workload, we identify p, the most efficient number planning cycles for the provisioner's scale out. Table 3 shows the cost model's qualitative estimates in node hours, over workload cycles 5-8, the first several iterations, using $s = 4$ samples, per Table 2. We use the contents of Figure 8 to validate the cost model.

The modeled costs closely correlate with the observed resources used by this workload. A planning length of 6 expands too aggressively, and is not worthwhile for MODIS. The tuning also accurately anticipates that a lazy scale out is wasteful, and the observed costs validate this claim. The cost model correctly identifies that a set point of 3 will have the lowest cost, as this moderate approach correctly balances cluster expansion costs with efficient workload execution.

In closing, we find that the staircase provisioner provides a flexible approach to scaling out an elastic array database, and that by tuning its parameters this algorithm is easily fitted to a new workload. The control loop, paired with the analytical cost estimator, enables us to both capture the elasticity of real workloads and respond it efficiently.

7. RELATED WORK

There has been much interest in research at the intersection of database elasticity and array data management. In this section we review the current state of the art.

Scientific Databases: The requirements for scientific data management were explored in [6, 22, 30]. An implementation was reviewed in [10] and demonstrated in [11]. We discuss how these challenges shape our approach in Section 2.

RasDaMan [8] is an array database that simulates multidimensional data on top of relational infrastructure. In contrast, we work from a scientific database built entirely around n-dimensional arrays and chunks.

Pegasus [17] maps scientific workloads to a high performance grid. Our work is similar in that we balance load, however our contribution revolves around evenly distributing storage rather than workflows, and has an incrementally expanding hardware platform.

In [29], the authors studied the organization of scientific data. We leverage the same array data model and build upon their n-dimensional chunking scheme, but this work generalizes SciDB to elasticity by addressing how data is assigned to nodes, rather than how it is written to disk.

Elasticity: There has been considerable interest in bringing elasticity to relational databases. [15] studied elastic storage for transactional shared infrastructure, and [13] addressed transactional workloads for multitenancy. Zephyr [18] extended this line of inquiry, solving live migration for transactional databases. Schism [14] offers workload-driven partitioning strategies for OLTP in the cloud. In contrast, this study focuses on incrementally expanding scientific databases, which may or may not be in the cloud.

Data Partitioning: Shared-nothing database partitioning was studied in [25]. They optimally partitioned data over a static hardware configuration for transactional workloads. Our research is orthogonal, addressing elasticity for large, read-mostly analytics.

We build upon several well-known data partitioning algorithms, extending them for multidimensional use, and evaluate how well they serve elastic scientific databases. We consider Extendible [19] and Consistent [24] Hash, as well as K-d Trees [9], Hilbert Curves [32], and Quadtrees [20]. Similar to [21], we study incremental partitioning, however our approach is for n-dimensional arrays. Our results demonstrate significant improvements in performance from explicitly taking array dimensionality and skew into account for data partitioning.

8. CONCLUSIONS AND FUTURE WORK

In this work, we explore the challenge of elasticity for array databases. Our approach addresses the competing goals of balancing load and minimizing overhead incurred by an incrementally expanding shared-nothing database. We first survey two motivating use cases, using them to derive a workload model for this platform.

We extend a series of data placement algorithms for incremental reorganization. The schemes are classified according to four features: incremental scale out, fine-grained partitioning, skew-awareness, and multidimensional clustering. We found that fine-grained, chunk-at-a-time partitioning results in the best load balancing. The remaining three traits yielded faster query execution, owing to spatial querying over clustered data. The latter category performed best over all, with the K-d Tree having the lowest summed workload time for both use cases.

We then introduced the leading staircase algorithm to scale out the database elastically, and a set of techniques for tuning it. Our experiments confirm that the provisioning tuner captures the scale out behavior of two real datasets.

One future direction is to more tightly integrate workloads with data placement. This entails modeling CPU, I/O, and network bandwidth to better understand how the database spends time, and the individual chunks that stand to benefit most directly from residing on the same server.

9. ACKNOWLEDGMENTS

The authors thank Ashraf Aboulnaga for his feedback on earlier versions of this work. This research was funded by the Intel Science and Technology Center for Big Data.

10. REFERENCES

[1] wlcg.web.cern.ch.

[2] www.scidb.org.

[3] modis.gsfc.nasa.gov/data/.

[4] people.csail.mit.edu/jennie/elasticity_benchmarks.html.

[5] National Oceanic and Atmospheric. Administration. *Marine Cadastre*. http://marinecadastre.gov/AIS/.

[6] A. Ailamaki, V. Kantere, and D. Dash. Managing scientific data. *Comm. of the ACM*, 53(6):68–78, June 2010.

[7] K. H. Ang, G. Chong, and Y. Li. Pid control system analysis, design, and technology. *IEEE Trans. on CST*, 2005.

[8] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. The multidimensional database system rasdaman. *SIGMOD Record*, 27(2), 1998.

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

[10] P. G. Brown. Overview of scidb: large scale array storage, processing and analysis. In *SIGMOD '10*.

[11] P. Cudré-Mauroux et al. A demonstration of scidb: A science-oriented dbms. *PVLDB*, 2(2):1534–1537, 2009.

[12] P. Cudre-Mauroux et al. Ss-db: A standard science dbms benchmark. *XLDB*, 2011.

[13] C. Curino, E. Jones, S. Madden, and H. Balakrishnan. Workload-aware database monitoring and consolidation. In *SIGMOD '11*.

[14] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *PVLDB*, 3(1-2):48–57, 2010.

[15] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi. Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration. *PVLDB*, 4(8):494–505, May 2011.

[16] S. de Witt, R. Sinclair, A. Sansum, and M. Wilson. Managing large data volumes from scientific facilities. *ERCIM News*, (89):15, April 2012.

[17] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi, and M. Livny. Pegasus: Mapping scientific workflows onto the grid. In *Grid Computing*, 2004.

[18] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *SIGMOD '11*.

[19] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing - a fast access method for dynamic files. *ACM Trans. on Database Syst.*, 1979.

[20] R. A. Finkel and J. L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, 1974.

[21] P. Ganesan, M. Bawa, and H. Garcia-Molina. Online balancing of range-partitioned data with applications to peer-to-peer systems. In *VLDB '04*.

[22] J. Gray, D. T. Liu, M. A. Nieto-Santisteban, A. S. Szalay, D. J. DeWitt, and G. Heber. Scientific data management in the coming decade. *SIGMOD Record*, 34(4):34–41, 2005.

[23] United States Coast. Guard. *National Automatic Identification System*. http://www.uscg.mil/acquisition/nais/.

[24] 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. In *STOC '97*, pages 654–663, 1997.

[25] M. Mehta and D. J. DeWitt. Data placement in shared-nothing parallel database systems. *VLDB Journal*, 6(1):53–72, 1997.

[26] M. E. Newman. Power laws, pareto distributions and zipf's law. *Contemporary Physics*, 46(5):323–351, 2005.

[27] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *SIGMOD '12*.

[28] G. Planthaber, M. Stonebraker, and J. Frew. Earthdb: scalable analysis of modis data using scidb. In *BigSpatial '12*.

[29] E. Soroush, M. Balazinska, and D. L. Wang. Arraystore: a storage manager for complex parallel array processing. In *SIGMOD '11*.

[30] M. Stonebraker, J. Becla, D. J. DeWitt, K.-T. Lim, D. Maier, O. Ratzesberger, and S. B. Zdonik. Requirements for science data bases and scidb. In *CIDR '09*.

[31] M. Stonebraker et al. C-store: a column-oriented dbms. In *VLDB '05*.

[32] J. Zhang, S.-i. Kamata, and Y. Ueshige. A pseudo-hilbert scan algorithm for arbitrarily-sized rectangle region. In *Adv. in M. Vision, Img Processing, and Pattern Analysis*. 2006.

[33] G. K. Zipf. Human behavior and the principle of least effort. 1949.