Autonomously Improving Query Evaluations over Multidimensional Data in Distributed Hash Tables

Matthew Malensek, Sangmi Pallickara, and Shrideep Pallickara
Department of Computer Science
Colorado State University
Fort Collins, CO, USA
{malensek, sangmi, shrideep}@cs.colostate.edu

ABSTRACT

The proliferation of observational devices and sensors with networking capabilities has led to growth in both the rates and sources of data that ultimately contribute to extreme-scale data volumes. Datasets generated in such settings are often multidimensional, with each dimension accounting for a feature of interest. We posit that efficient evaluation of queries over such datasets must account for both the distribution of data values and the patterns in the queries themselves. Configuring query evaluation by hand is infeasible given the data volumes, dimensionality, and the rates at which new data and queries arrive. In this paper, we describe our algorithm to autonomously improve query evaluations over voluminous, distributed datasets. Our approach autonomously tunes for the most dominant query patterns and distribution of values across a dimension. We evaluate our algorithm in the context of our system, Galileo, which is a hierarchical distributed hash table used for managing geospatial, time-series data. Our system strikes a balance between memory utilization, fast evaluations, and search space reductions. Empirical evaluations reported here are performed on a dataset that is multidimensional and comprises a billion files. The schemes described in this work are broadly applicable to any system that leverages distributed hash tables as a storage mechanism.

Categories and Subject Descriptors

D.4.3 [**File Systems Management**]: Distributed file systems; I.2.8 [**Problem Solving, Control Methods, and Search**]: Graph and tree search strategies

General Terms

Algorithms, Design, Performance

Keywords

Autonomous query tuning, distributed hash tables, multidimensional data

CAC'13, August 5–9, 2013, Miami, FL, USA.

1. INTRODUCTION

Observational devices and sensors with networking capabilities have become more and more pervasive, leading to growth in both the rates and sources of data that ultimately contribute to data volumes. Datasets generated in such settings are often multidimensional; for example, in atmospheric datasets a given geospatial location has data corresponding to multiple features such as humidity, pressure, temperature, etc. These datasets are dynamic with continual addition of data and query evaluations. The increase in data volumes has been accompanied by a corresponding increase in the complexity and rates at which queries are performed on data. Given the data volumes involved, data must also be stored on multiple machines so that query evaluation can be performed concurrently. Furthermore, even on a given node it is infeasible to hold all data items in memory and then perform query evaluations.

Query evaluation frameworks in such settings must be decentralized and manage their data structures efficiently. Storage across multiple machines allows concurrent evaluations on portions of the datasets, requiring each machine to maintain a coarse-grained view of the entire dataset; this approximate global view assists in search-space reductions. There must also be support for fine-grained evaluation of queries over portions of the dataset stored at a given node. The system may also include optimization features such as reorientation of the data structures in response to data arrivals. The data structures may be on-disk or memory-resident and the information maintained in them have different granularities based on the scope of the data under its purview.

The problem we consider is the fast evaluation of queries over large, multi-dimensional datasets. One of these dimensions is time, so the queries can also be time-series based. This problem must be dealt with in the presence of high rates of arrivals for both new data and queries. Solutions must account for memory utilization, traversals of data structures, and search space reductions during query evaluations.

1.1 Research Challenges

Challenges stem from the characteristics of the data being stored and retrieved through query evaluations. The data is multidimensional and the range of values that the *types* associated with the data can take on can be unlimited and continuous. Across the stored data, there are differences in the density of both queries and values along each dimension. Furthermore, for a given dimension, the queries may span specific ranges of values. Human intervention to optimize

such query evaluations is infeasible because of the rates of data and query arrivals, the dimensionality and range of values, and data volumes.

1.2 Research Questions

Our goal is to minimize human intervention and autonomously tune data structures that assist in query evaluations in the aforementioned settings. To achieve this we must account not just for the data arrivals and the characteristics of the stored data, but also the queries that arrive. This work was driven by the following research questions.

1. How can we account for differences in the density of values across a specific dimension?
2. How we can exploit patterns in query arrivals to improve turnaround times?
3. Can we achieve (1) and (2) while being timely and autonomous? Specifically, can we get the system to adapt to changes in the datasets and query arrivals in a timely fashion?

1.3 Paper Contributions

By dynamically adapting to the density of values and queries expressed across ranges of a particular dimension, our reductions in the search space during query evaluations are greater than those for static, preconfigured settings. Our paper contributions include the following:

- We have devised a scheme to autonomously tune query evaluations over a large dataset dispersed over a collection of machines.
- Our approach can cope with data volumes and fast arrival rates. Our empirical evaluations were performed on a real dataset encompassing a billion files. The dataset was sourced from NOAA's NAM effort.
- We can sustain variability in the density of data values across different dimensions and also in how queries are expressed over different dimensions.
- Rather than optimize for corner cases, we focus on improving the most dominant use case. The more often we see a query pattern the greater the probability that the system will return similar queries faster. Similarly, the greater the density of values along a particular dimension, the greater the probability that query evaluations along that range will be faster.
- Our proposed learning schemes are broadly applicable to other multidimensional, voluminous datasets. We have contrasted our results with a well-known storage system. Our results should be broadly applicable to other DHT-based systems as well.

1.4 Approach

The algorithms described in this work were applied in the context of Galileo, our distributed file system for the storage and retrieval of multidimensional data. The *feature graph* in Galileo assists in the evaluation of queries by enabling fast, approximate results based on value ranges called *tick marks* that do not include false negatives. These results contain information about nodes that hold data blocks or metadata that match the specified query. The original query is then forwarded to the nodes in the approximation set to retrieve data blocks matching the query. The approximation set is a superset of the actual set of nodes that hold the matching data blocks. Our objective is to reduce the difference in the cardinality of the two sets while retaining the ability to suppress false negatives, i.e. the situation where a node holds particular data blocks matching the query but does not appear in approximation set.

Our approach to autonomous tuning incorporates two key strategies. The first one focuses on dynamic reorientation of graphs to account for data dimensions and also the order in which they are specified in a majority of the queries. This orientation is used to reduce number of edges that must be traversed during query evaluations and also to reduce the memory footprint. The second strategy targets improvements in tick marks corresponding to data values. Dynamic tick marks along a specific dimension allow accounting for variability in the distribution of values across a specific dimension. Accounting for such variability allows us to reduce the search space for queries. Both these strategies can be used in tandem.

While query turnaround time may not be of the utmost importance for all use cases, the adaptive reduction in search space provided by our algorithms also greatly increases scalability when dealing with large, heterogeneous clusters of commodity hardware. Furthermore, reducing the search space results in less wasted CPU cycles and power consumption, along with overall increased throughput as the system can handle more concurrent requests. This applies to any multidimensional dataset; Galileo is not limited to strictly atmospheric applications.

1.5 Experimental Data

In this study, we utilized real-world data from the North American Mesoscale Forecast System (NAM), provided by the National Oceanic and Atmospheric Administration. Our data consisted of samples from 2009-2013, creating a dataset of one billion (1,000,000,000) files. The metadata attributes we used for this study from the files included the spatial location of the sample, temporal range during which the data was recorded, percent maximum relative humidity, surface temperature (Kelvin), wind speed (meters per second) and snow depth (meters).

1.6 Paper Organization

The remainder of this paper is organized as follows: the next section provides an overview of Galileo and its architecture, and introduces the basis for the optimizations we made in this paper. Section 3 explains autonomous graph reorientation in the system, which strives to improve memory usage and query response times based on user activities. Section 4 details our dynamic indexing system, which responds to different workloads by reconfiguring the granularity of indexed data points. Section 5 contrasts the performance of Galileo with Apache HBase, followed by related work in Section 6. We bring the paper to a close with our conclusions and future work in Section 7.

2. GALILEO

The Galileo distributed file system [10, 11] is designed for high-throughput storage and retrieval of multidimensional data at the petabyte scale. It features a hybrid DHT-based network design and rich data structures for handling sci-

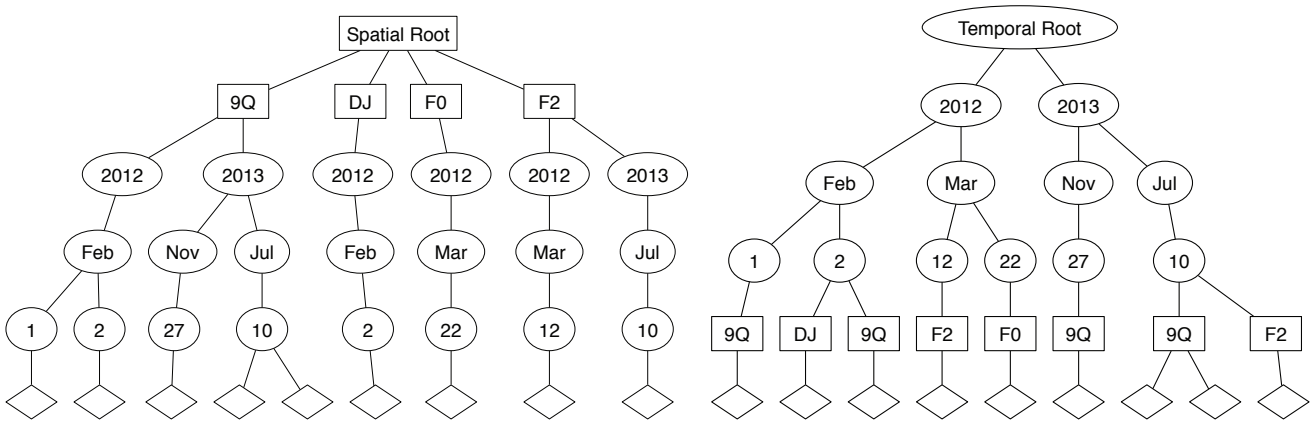

Figure 1: Two different metadata graph orientations of a dataset with spatio-temporal characteristics. Diamond-shaped leaf nodes point to locations of files on stored on disk.

entific storage demands, along with the ability to process and manage information through the Granules cloud runtime [14].

2.1 Network Structure

Galileo is implemented as a hierarchical distributed hash table (DHT). DHTs are a highly scalable, decentralized system for mapping key-value pairs to storage nodes. To do this, a hash space is divided up among all the nodes in the system. When performing storage or lookup operations, a *key* (which could be a file name or some other type of unique identifier) is passed through a hash function to determine which node in the system is responsible for storing the key's associated *value*. Unlike traditional DHTs such as Chord [20], CAN [17], or Pastry [19], Galileo is also a *zero-hop* DHT, similar to Apache Cassandra [9] or Amazon Dynamo [8]. This means that the system has enough information to route requests directly to their destination rather than traversing a network topology.

The hierarchical structure in Galileo is created by its two core network components: *Storage Nodes* (or simply *nodes*), which represent a computational resource, and *groups*, which contain an arbitrary number of nodes and subgroups. One key benefit from this structure is a reduction of the search space for queries, a functionality that generally is not supported in DHTs, but has been investigated in a number of systems, [6, 15] including Galileo [12]. For our specific dataset, we use a two-tiered hierarchy: when storing or retrieving data, group membership is determined by the sample's Geohash, [23] which maps spatial locations to character strings. To determine the specific node within the destination group that should be responsible for the data, a simple SHA-1 hash is used to help balance load. Depending on the particular dataset, any number of groups and hash functions can be used for partitioning data in Galileo.

2.2 File Blocks

Files in Galileo are generally stored as a number of *blocks*. A block is a multi-dimensional array of arbitrary data for supporting use cases similar to SciDB [3, 4] or NetCDF [18]. Galileo can also read and store files in a number of scientific formats, including NetCDF, BUFR, GRIB, and HDF5. When files are stored in the system, they are first inspected for relevant metadata attributes and indexed for fast and flexible retrieval. Galileo supports both exact-match (which could return multiple relevant files) and range-based queries. Queries return results to the user in the form of a *dataset*, which is a navigable data structure that describes the files and their metadata. If the user chooses to do so, the original blocks can then be retrieved from the system as well.

2.3 Metadata and Features

One primary function in scientific data storage is managing not only the files themselves, but also their associated metadata. During storage operations, Galileo inspects incoming files and extracts relevant *features* that describe the data. These features are used for indexing information in the system; each Storage Node maintains an in-memory instance of a *metadata graph*, which contains feature information for each file it is responsible for storing. The metadata graph can then be used to quickly locate files in the system.

An instance of the metadata graph consists of a hierarchy of features; an example is shown in Figure 1. The first tier of vertices, and entry point into the graph, is composed of multiple values of a single feature type. Each vertex contains a number of edges that lead to the next feature type, and so on. Collections of feature values, called *paths*, describe a particular traversal of the graph that leads to the file associated with the values; traversing the graph acts as a logical AND operator on the values specified. If a query path contains a wildcard, all descendants of the wildcard feature type are traversed.

Different graph orientations provide different traversal properties: some configurations of the metadata graph may reduce the overall number of vertices maintained, which also reduces memory usage, while others may provide faster lookup times when frequently-requested feature types are positioned at the top of the hierarchy. Figure 1 illustrates this behavior on a small scale, with the spatially-oriented graph containing 36 vertices while the temporally-oriented graph contains only 30.

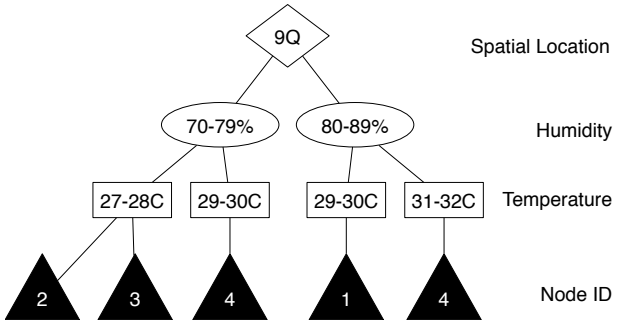

Figure 2: A simplified feature graph showing humidity and temperature ranges. A traversal of the graph leads to Storage Node identifiers that hold relevant information.

2.4 The Feature Graph

While the metadata graph provides efficient lookup functionality at the Storage Node level, it does not provide any assistance for clients that do not already know which nodes hold the data they are interested in. Our hierarchical partitioning scheme allows clients to locate data within a spatial region, but it still does not reduce the search space if the region of interest is not known. To cope with these issues, we introduced the *feature graph*, which is a lightweight global index of all the data stored in the system.

In order to make the feature graph a feasible addition to Galileo, incoming data is placed in ranges of reduced dimensionality called *tick marks*. Briefly, tick marks are a form of quantization that allows the system to index larger quantities of data quickly by decreasing its precision; for example, a temperature reading of 23 degrees Celsius may be placed in a tick mark range of 20-30 degrees. This 10-degree range represents a tick mark *granularity* of 10, and any samples falling within the range can be represented by a single vertex. Figure 2 provides an example of a simple feature graph. Querying the feature graph provides results in a form of a list of storage nodes that are likely to hold data relevant to the query. The feature graph is not bound to any particular storage system implementation, and has been used in a past study to augment the lookup capabilities of Apache HBase [12, 21].

Since the feature graph is a global data structure, storage nodes utilize a simple gossip protocol to publish graph updates in the form of paths to the other nodes. Galileo is an *eventually consistent* file system, so these updates occur asynchronously. To reduce communication, nodes communicate changes within their groups first, and then an elected *group leader* shares aggregate updates with the other groups in the system.

Our graph implementations provide flexible and fast lookup operations, but achieving optimal performance requires proper configuration. In this paper, we provide a framework for autonomously reconfiguring the system at runtime to achieve better performance by inspecting storage and query trends and then tuning graph parameters accordingly. This is accomplished from two angles: dynamic reorientation of the graphs, and autonomous partitioning of the feature graph tick marks.

3. GRAPH REORIENTATION

The order of features in our metadata and feature graph hierarchies determines how many vertices they will contain and how fast a query can be completed. To optimize memory consumption by reducing the number of vertices in the graph, features with a high variability in values should generally be placed near the top of the hierarchy; if placed near the end of a query path, a large number of leaf nodes would have to be created to accommodate the feature's variability. Unfortunately, optimizing search operations does not always align with reducing memory consumption: if users are generally interested only in the features stored as leaf nodes, then an exhaustive search of the graph during query evaluation becomes much more likely. With our four target features, Table 1 demonstrates the difference in vertices and edges between three notable orientations (out of a possible 24).

Table 1: Vertex and edge counts for different feature graph orientations.

Orientation	Vertices	Edges
Humidity Temperature Wind Speed Snow Depth	454,5692	2,267,984
Temperature Humidity Wind Speed Snow Depth	452,458	2,254,029
Snow Depth Wind Speed Temperature Humidity	250,531	1,240,178

While some orientations of the graph can result in a small percentage change in vertices, (less than 1% by substituting humidity for temperature in the path ordering) other configurations can result in much more drastic memory consumption changes.

3.1 Traversal Performance Monitoring

To evaluate traversal and memory costs of the feature and metadata graphs, we recorded traversal times and vertex counts for not only the entire graph, but each level of the feature hierarchy as well. This allows the system to strike a balance between memory-efficient and quickly-traversable orientations: if a particular level in the hierarchy accounts for a disproportionate amount of vertices or traversal time, it should be reconfigured. In cases where the memory available to the storage node is scarce, the graph will be oriented to optimize memory consumption first; otherwise, our focus is on query response times.

Queries submitted to nodes in the system can contain a number of feature values. For features where an exact value or range of values is not specified, a wildcard operator is inserted into the path. In general, when queries contain more wildcards, their traversal times through both the feature and metadata graphs will be higher. Therefore, Galileo also

monitors trends in query parameters to determine features that are often specified together, and may be related.

3.2 Dynamic Reorientation

Graph performance statistics collected at individual storage nodes provides enough data for the node to make informed decisions about how its graphs should be oriented. Due to the hierarchical partitioning schemes supported by Galileo, storage and retrieval trends may vary across an entire cluster, so each node must reorient its metadata and feature graphs to fit its individual access patterns. To accommodate this functionality, we modified our *path* data structure, which represents a graph traversal, to be reoriented as well. These general paths are represented as simple key-value pairs where feature types are mapped to feature values. This modification ensures that each node in the system can orient its feature graph optimally for its specific use cases and still transmit updates in an orientation-neutral manner.

To dynamically reorient its graph instances, a storage node first inspects mean traversal times for incoming queries. If a particular level in the hierarchy accounts for the majority of incoming queries or its traversal times are more than two standard deviations from the mean, then it is flagged for reconfiguration. For example, if users are primarily interested in temperature values, but they are placed at the bottom of the hierarchy, then their resulting traversal times will be much greater than those of other features. Additionally, if temperature values are the *only* type of query being submitted, they will account for the majority and also be flagged. For each iteration of the algorithm, only one feature or set of features is selected for optimization.

Once candidates for reconfiguration have been selected, their dependencies on other features are reviewed to determine if they can be moved closer to their dependent neighbors. Once a new hierarchy has been created, the amount of vertices that will have to be created or removed is calculated to evaluate its memory footprint. If all constraints are met, the graph is reoriented. Table 2 compares the performance of Galileo with and without dynamic reorientation; both systems started with a memory-efficient orientation of the feature graph and then were tested with a battery of queries that trended towards a randomly-selected feature.

Table 2: Benchmark of feature graph traversal times with and without dynamic reorientation.

Configuration	Traversal (ms)	SD (ms)
Standard	0.4	0.7
Dynamic	0.1	0.3

While graph traversals cost on the order of milliseconds, a heavily-loaded server greatly improve its throughput by locating information faster. This functionality provides optimization for both the feature and metadata graphs, so it benefits both nodes in the system that are routing requests and those that are servicing queries.

To cope with the rare case that usage patterns change during a graph reconfiguration, new configurations are given a trial period in which their performance is evaluated. If query throughput has decreased, the algorithm will be run again to create a new configuration that reflects current usage trends, or fall back to the previous configuration. This restriction also prevents toggling between configurations that have similar performance profiles.

Trends in the system may change due to the time of day or because of other external factors, so the data collected must be aged appropriately and discarded regularly to ensure a given configuration remains relevant. To do this, we provide an adjustable timeout threshold that gradually phases out old performance information.

4. DYNAMIC MODIFICATION OF INDEX DIMENSIONALITY

Feature values in Galileo can be bounded or unbounded, contain different primitive data types, or be expressed using a variety of units. For instance, humidity values can be measured by percentages ranging from 0-100, while snow depth may be expressed using centimeters or inches and will contain only positive readings. While it is possible for users of the system to provide details or domain knowledge about the data they are storing, the optimal granularity of tick marks for the feature graph is often unknown during system configuration, or may evolve over time. Additionally, due to the high dimensionality of the data being stored, relationships between features may not be initially obvious. To remedy these issues, the system inspects the information being stored or queried by users, and then autonomously reconfigures the feature graph with variable tick marks to optimize performance.

Figure 3 illustrates why a fixed tick granularity is inefficient for our billion-file subject dataset; with humidity increments of 5%, the first few vertices are relatively unused, while the ticks covering values from 80-100% account for the majority of the data and queries in the system. This can be remedied by reducing the tick granularity to 1% or less, at the cost of many more vertices being created. The figure also includes the kernel density estimation, an approximation of the probability density function for humidity values.

There are two key benefits from allowing a variable tick mark granularity and adjusting it dynamically: vertices reference fewer storage nodes, thereby reducing the search space of client queries, and fewer vertices must be created to achieve the same performance as a fixed tick granularity. To do this, Galileo must first detect storage and query imbalances in the tick marks, and then apply *proportional splits* to reduce the amount of data each vertex is responsible for.

4.1 Detecting Imbalances

Depending on the configuration of the feature graph, imbalances can form for both queries and storage operations. Queries may target a particular precision of values that the current feature graph doesn't have fine-grained vertices for, resulting in more storage nodes needing to be contacted for the information. Incoming data is almost always non-uniform in real-world situations as well. While detecting the existence of an imbalance in the feature graph is simple, ensuring that the system does not reconfigure the graph needlessly or excessively is a much more difficult problem. After all, the feature graph is a *global* data structure, and any changes made to it must be gossiped to the entire cluster.

To gain a consensus on vertices in the feature graph that

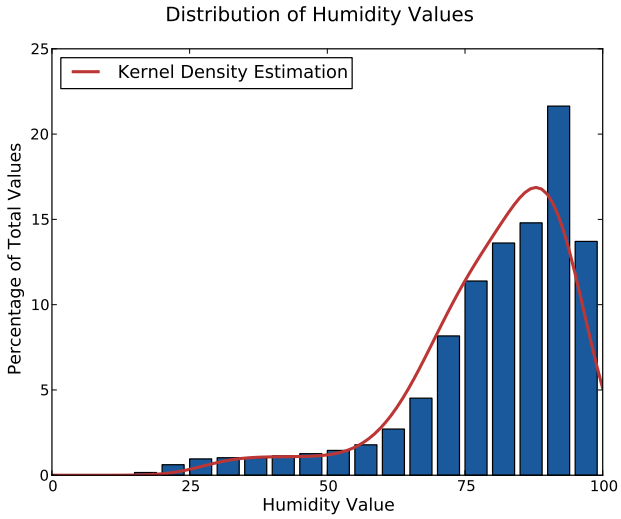

Figure 3: Storage distribution of humidity values with a uniform 5% tick mark granularity in our billion-file dataset.

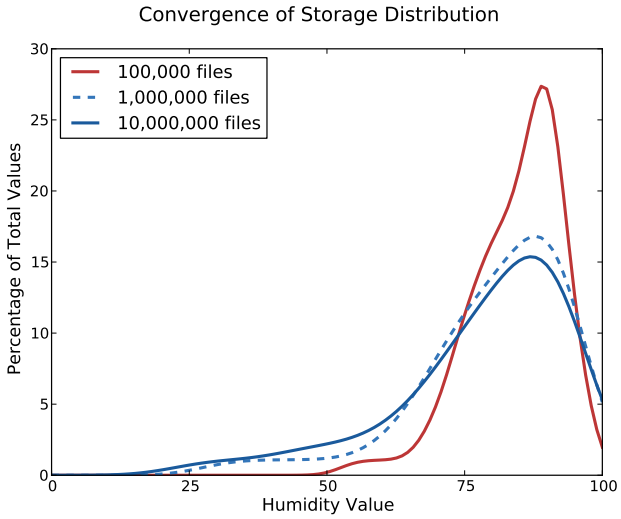

Figure 4: Probability density of humidity values as more files are stored in the system.

must be reconfigured, each storage node tracks the number of query or storage operations that involve each vertex in the graph. These "hit" counts are gossiped along with other state updates in the system. Additionally, nodes calculate the percentage of total hits occurring at each vertex to generate an approximation of the probability each node will be involved in a future query or storage request. Figure 4 shows the probability distributions of humidity values as more files are stored in the system; a clear trend towards files having values in the 80-100% range is evident as the dataset is subsampled. A small random sample of incoming files (about 1,000) exhibits a similar trend.

After each feature graph gossip update interval, probability distributions are inspected to determine if particular tick marks are exhibiting a larger percentage of overall queries and storage requests. If a tick is flagged in two consecutive graph updates, its feature type becomes a candidate for reconfiguration.

4.2 Network Coordination

Since the feature graph is shared globally, only one storage node needs to perform the reconfiguration. An eligible node must meet the following requirements:

1. It must host files stored under the tick in question, i.e., be returned in the results of a graph traversal involving the tick mark. This helps reduce the amount of communication required for reconfiguration.

2. Its thread pool for processing must have at least one free thread.

3. It may not be a group leader in the DHT hierarchy. Leaders are already responsible for additional communication operations.

In the case of ties, where multiple candidates meet the requirements, the node with the lowest identifier is selected to be a reconfiguration *coordinator*. Concurrent modification of the feature graph can be allowed as long as different nodes are modifying tick ranges that involve unrelated paths in the graph.

Reconfiguring the feature graph is not computationally expensive, but involves network IO latencies and coordination across the entire system. For this reason, we avoided using a locking mechanism on the graph during reconfiguration. Instead, modifications are gossiped throughout the system, and then graph paths for any new files that were stored during the reconfiguration are "replayed" from the system journal on the updated graph. To ensure the graph remains consistent, versioning information is also included in gossip messages.

4.3 Proportional Splits

Our system uses the *coefficient of variation*, (CV) a measure of how dispersed a probability distribution is, to detect hit imbalances in vertices. The coefficient of variation has been used in analyzing computational loads [5,24], and is expressed as the relationship between the standard deviation σ to the mean μ:

$$CV = \frac{\sigma}{\mu}$$

To reduce the coefficient of variation, there are two options: decreasing the standard deviation, or increasing the mean percentage of hits at each vertex. In general, this is achieved by performing a *proportional split* on overloaded vertices.

A proportional split involves identifying a vertex that is receiving a disproportionate amount of hits, querying storage nodes with relevant data, and then reducing the tick mark range by splitting it into two or more new tick ranges. Figure 5 provides a visual overview of the process. For a

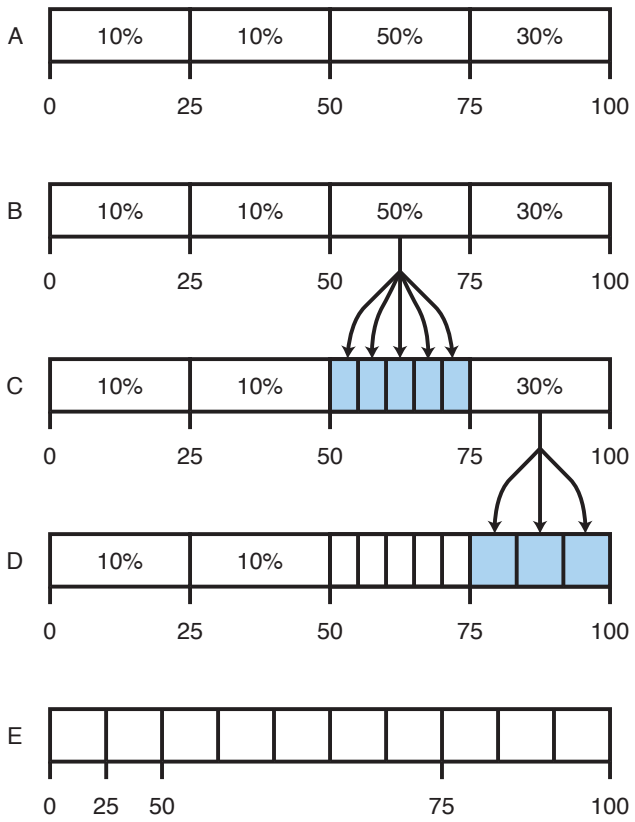

Figure 5: Visual demonstration of our proportional split algorithm. 4 ticks are present in phase A, of which two vertices are split. The resulting range consists of 10 ticks shown in phase E, with each responsible for 10% of the storage and query load.

feature type that has been flagged for reconfiguration, the process is repeated until $CV \leq 1.0$.

The proportional split algorithm proceeds as follows: the percentage of hits at each tick mark range, mean percentage of hits, and standard deviation is calculated. This information is used to compute the feature CV. If $CV > 1.0$, then the tick mark range with the largest percentage of hits is selected for splitting. Once a range has been selected, each storage node associated with the vertex is queried for all matching data. This operation only involves a metadata query, so the resulting files themselves do not have to be transfered to the coordinating node. The coordinator inspects the query results and determines the number of new tick ranges that should be created based on the relationship between storage nodes and data. After all, splitting a tick mark range into two vertices that point to the same set of nodes does not further reduce the search space. The coordinator attempts to split the tick mark range in such a way that data will be as evenly distributed as possible across the new vertices, and repeats the algorithm if necessary. Algorithm 1 contains the pseudocode for this process.

After the algorithm has completed, tick mark ranges are configured non-uniformly to match the hit trend their feature type is experiencing. Figure 6 illustrates the result of a splitting operation, with the vertices' (shown as bars) widths

Algorithm 1 Proportional Split: $split(feature)$

Require: $CV > 1.0$
$\mu \Leftarrow 1.0/numVertices$
$\sigma \Leftarrow \sqrt{variance(hitPercentages)}$
$CV \Leftarrow \sigma/\mu$

{Find vertex with highest percentage of hits}
$v \Leftarrow max(hitPercentages)$
$files \Leftarrow query(v)$
$balance\_ticks(files)$
$divide\_ticks(files)$

if $CV > 1.0$ **then**
split(feature) {Repeat the procedure until $CV \leq 1.0$}
end if

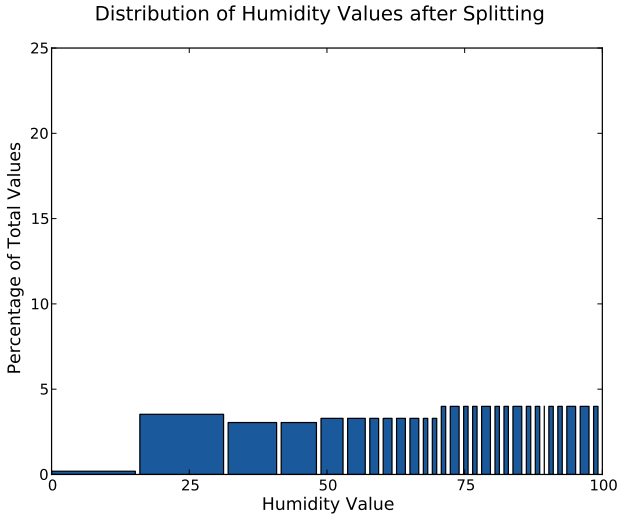

Figure 6: Distribution of tick mark ranges after a proportional split operation has taken place. Bar widths represent the range of values each vertex is responsible for.

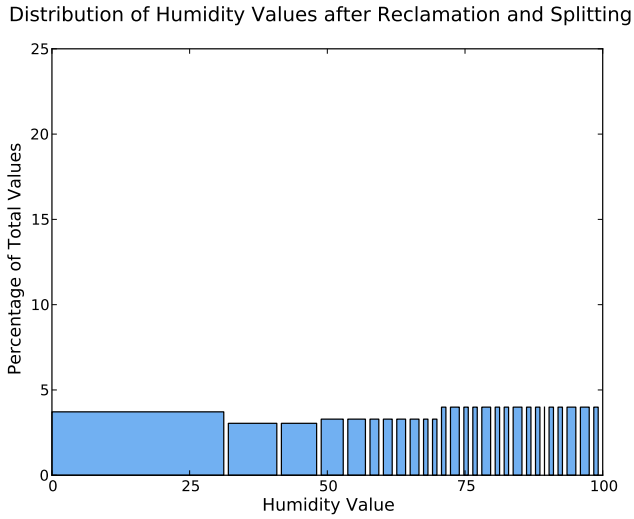

Figure 7: Distribution of tick mark ranges after a vertex reclamation followed by a proportional split operation. Contrasting with Figure 6, the first two tick ranges have been merged.

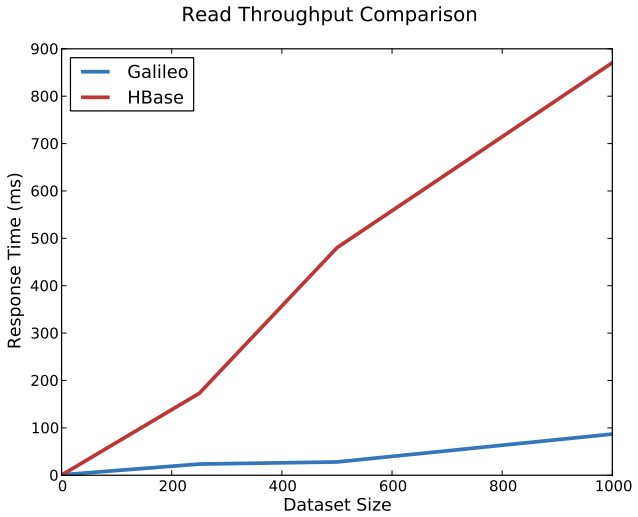

Figure 8: Read throughput in Galileo and HBase.

representing the range of humidity values each is responsible for. Since a storage request and a query both count as a hit, our algorithm can deal with changes in storage patterns *and* differences in user queries. In a new installation of the system, storage operations will generally drive how tick marks are configured, but our hit count strategy also naturally places an emphasis on servicing user requests by giving each query the same weight as a single storage operation. This leads to dynamic reconfiguration long after the system has been installed without needing to phase out old data on a regular basis.

4.4 Vertex Reclamation

While our proportional split algorithm helps even the balance of load across vertices in the feature graph, it does not account for under-utilized vertices. To rectify this limitation, the coordinating node attempts a *vertex reclamation* operation on the feature graph before proceeding with the proportional split process. Vertex reclamation works similarly to proportional splitting, but reconfigures vertices that have the lowest percentage of hits in the feature graph.

The reclamation algorithm begins by locating tick mark ranges receiving the lowest percentage of hits. It then inspects neighboring vertices to determine their differences in referenced storage nodes. If there are no differences, then the ranges can be safely merged. Otherwise, the neighboring vertex with the fewest differences is used as a candidate for merging. After performing the reclamation, the coefficient of variation is calculated to determine if the operation resulted in more or less dispersion in the probability distribution; if there is less dispersion, the changes are kept. Otherwise, the algorithm moves on to the tick range with next-lowest percentage of hits. Figure 7 contrasts with Figure 6 in that the first two tick ranges have been merged through reclamation.

Performing a reclamation is beneficial in two ways. First, reducing the number of vertices in the graph increases the mean percentage of hits spread across vertices of a given feature type, resulting in a lower coefficient of variation. Additionally, by allowing vertices to be reclaimed, previously-configured tick mark boundaries can be changed; a vertex could be reclaimed, and then re-split to provide a more efficient mapping of feature values to storage nodes.

5. HBASE COMPARISON

To ensure that the changes we made to Galileo performed as expected, we ran a series of throughput benchmarks on the system and revisited our performance comparison with Apache HBase [21] 0.92.1. The benchmarks were carried out on a 70-node cluster running OpenJDK 1.7.0. The cluster consisted of 47 HP DL160 servers (2.4 Ghz quad-core Intel Xeon processor, 12 GB of RAM, 15,000 RPM disk) and 28 SunFire X4100 servers (dual-core 2.8 Ghz Opteron 254 processors, 8 GB of RAM, 10,000 RPM disk).

To ensure files were stored using spatial locality in HBase, we adjusted our storage strategy by incorporating block UUIDs as our row keys, prefixed with a Geohash of each file's spatial location. We also utilized our standalone version of the feature graph and had its traversal results point to HBase block UUIDs so that specific items in the system could be referenced directly. Each record from our dataset was about 8 KB in size.

Figure 8 compares the read throughput of Galileo and Hbase. We submitted the same set of queries to both systems, and each test was run 100 times on different spatial regions. We also compared write throughput, shown in Figure 9, which exhibits similar trends in performance, although the throughput is reduced in both systems when compared to read operations.

This benchmark primarily underscores the fact that Galileo excels at the workloads it has been designed for; while HBase sees considerable usage in the geospatial community, it is a more general system, primarily designed for sparsely populated, semi-structured data.

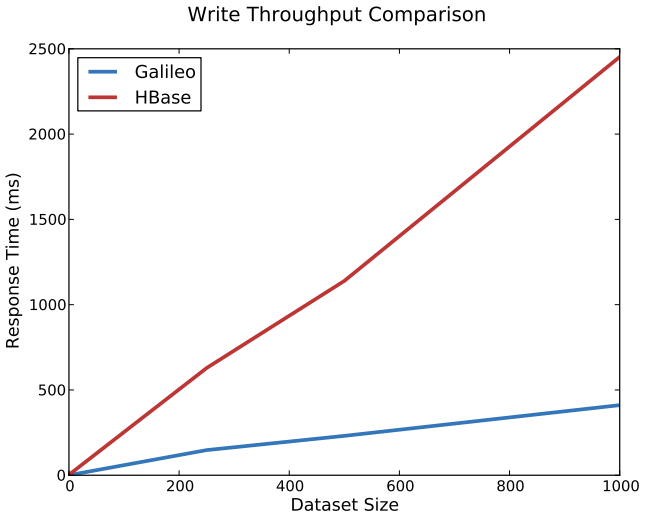

Figure 9: Write throughput in Galileo and HBase.

6. RELATED WORK

Cassandra [9] shares many attributes with Galileo in its network layout and storage system capabilities. It provides a number of different data partitioning approaches for users depending on their workloads, which can also be extended or reconfigured for different data types. Contrasting with Galileo, the partitioning algorithm used in Cassandra directly affects possible retrieval operations; using the random data partitioner backed by a simple hash algorithm does not allow for range queries or later reconfiguration of the partitioning scheme. Cassandra is also primarily concerned with write-heavy workloads on textual data rather than the multidimensional binary arrays that Galileo deals with.

Gavrila [7] has investigated R-tree index optimization for use in geographic information systems (GIS). Much like Galileo, the optimizations described involve preserving spatial relationships between data points through "packing." While the system targeted in the paper is not distributed across a number of computing resources, many of the challenges faced on a single machine are still relevant in a distributed setting. For instance, our optimizations in Galileo are primarily concerned with reducing network latencies, while the described system aims to reduce disk seek times. Ultimately, both objectives target latency reduction. The packing technique described in the paper can be run after creating a static database, or used to periodically adjust the system during idle times. It involves clustering data points and then evaluating cluster quality to determine how the system should be reconfigured.

SciDB [3, 4] deals with petabyte-scale datasets in a distributed environment much like Galileo. The system provides built-in computation and analysis tools, and stores metadata in a centralized *system catalog*. The system catalog is backed by a PostgreSQL relational database, meaning that queries are handled using a configurable *query optimizer* that generates a query plan based on the input SQL operators. This approach contrasts with Galileo in that the query facilitator is exhaustive and centralized, whereas the feature graph in Galileo is distributed across all nodes and is comparatively lightweight.

The Prefix Hash Tree (PHT) data structure [16] provides a binary trie index that runs on a traditional DHT. PHT uses the existing lookup interface provided by its host DHT to provide range and proximity queries. This indexing structure naturally leads to a hierarchy of nodes, much like Galileo, and query processing operations are spread across the hierarchy. The PHT design also allows it to provide its functionality while preserving the fault tolerance properties of traditional DHTs.

Replication, Load Balancing and Efficient Range Query Processing in DHTs by Pitoura et al. [15] aims to deal with query and storage load balancing through replication. The system described, HoTRoD, is built atop a locality-preserving DHT, much like Mercury [2] and OP-Chord [22]. Locality-preserving DHTs provide support for range queries by distributing data in an order-preserving way, but suffer from storage imbalances that arise from non-uniform data. To overcome this limitation and also deal with high-traffic nodes, HoTRod replicates *arcs* of nodes, which represent a sequence of neighboring peers in the hash space. These replicated arcs are "rotated" across overlapping virtual hash spaces and stored on different nodes, which can then be used to service requests.

As an enhancement to OP-Chord, Ntarmos et al. [13] add an additional hash space on top of the standard space called a *RangeGuard*. The RangeGuard helps support processing range queries across the underlying DHT and is composed of the most powerful nodes in the cluster. Nodes in the RangeGuard act as super nodes and are promoted through a specialized algorithm that accounts for heterogeneity. In this system, both the promoted nodes and standard nodes running in the traditional hash space can be used for query processing. The enhancements proposed in this particular work do not require any global knowledge to provide range query support, but requests must hop through the network rather than being processed immediately in parallel as done in Galileo.

Abdallah and Le [1] propose a different solution to the order-preserving distributed hash table: rather than relocating data in the system, nodes themselves are relocated in the hash space and data is migrated to provide a better balance of storage and query handling. In general, underutilized nodes are moved within the hash space to ensure that data migrations do not have a severe impact on performance. While migrating nodes is an interesting concept, it is likely quite expensive in a heavily-loaded system.

7. CONCLUSIONS AND FUTURE WORK

Query evaluations over voluminous datasets must account for the characteristics of the data and the queries that have been performed. In the case of multidimensional data this involves accounting for not just the distribution of values across a specific dimension, but also the order in which these dimensions are specified in the queries. Our scheme for feature graph orientations and dynamic tick marks allow us to optimize queries that occur commonly. This is done autonomously by the system. Graph reorientations allow us to reduce the number of paths that must be traversed within the feature graph for faster query evaluations. Dynamic tick marks allow us to reduce the search space for query evaluations and help mitigate the problem of queues building up at nodes with queries that will have no matching results.

Autonomous tuning allows us to account for the evolu-

tion of datasets and also for changes in how queries are performed. Our benchmarks demonstrate the feasibility of using our approach on a large multi-dimensional dataset. The approach described here is broadly applicable to any system that leverages distributed hash tables for storing data.

Currently, we rely on a windowing scheme to age out older queries and keep data structures in sync with queries that have been issued in the near past. A natural extension of this work is to account for query arrivals in addition to the type of queries. Such a time-series analysis of query arrival patterns will allow us to anticipate the types of queries and proactively orient data structures in support of them. As part of this future work we plan to either use statistical techniques such as ARIMA (Auto Regressive Integrated Moving Average) or machine learning techniques such as Hidden Markov Models to perform this time-series analysis.

To further enhance the performance of our dynamic index optimization algorithm, small movements of data between nodes could be implemented to reduce the amount of false positive results produced from a feature graph traversal. This data migration could be done in a lazy fashion, with direct data migration occurring on relatively idle nodes and on-demand transfers taking place on busy nodes. This optimization could also be incorporated into a fully-fledged load balancing scheme.

We may investigate the effects of utilizing other statistical methods to detect imbalances and facilitate proportional split operations. This could involve using the Gini coefficient and Lorenz curves for expressing overloaded vertices in the feature graph.

8. REFERENCES

[1] M. Abdallah and H. C. Le. Scalable range query processing for large-scale distributed database applications. In *IASTED PDCS*, pages 433–439, 2005.

[2] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: supporting scalable multi-attribute range queries. In *ACM SIGCOMM Computer Communication Review*, volume 34, pages 353–366. ACM, 2004.

[3] P. Brown. Overview of scidb: large scale array storage, processing and analysis. In *Proceedings of the 2010 international conference on Management of data*, pages 963–968. ACM, 2010.

[4] P. Cudré-Mauroux, H. Kimura, K. Lim, J. Rogers, R. Simakov, E. Soroush, P. Velikhov, D. Wang, M. Balazinska, J. Becla, et al. A demonstration of scidb: a science-oriented dbms. *Proceedings of the VLDB Endowment*, 2(2):1534–1537, 2009.

[5] P. Dinda. The statistical properties of host load. *Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 319–334, 1998.

[6] P. Ganesan, K. Gummadi, and H. Garcia-Molina. Canon in g major: designing dhts with hierarchical structure. In *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on*, pages 263–272. IEEE, 2004.

[7] D. Gavrila. *R-tree index optimization*. University of Maryland, Center for Automation Research, Computer Vision Laboratory, 1994.

[8] D. Hastorun et al. Dynamo: amazon's highly available key-value store. In *In Proc. SOSP*. Citeseer, 2007.

[9] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[10] M. Malensek, S. Pallickara, and S. Pallickara. Galileo: A framework for distributed storage of high-throughput data streams. In *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on*, pages 17 –24, dec. 2011.

[11] M. Malensek, S. Pallickara, and S. Pallickara. Exploiting geospatial and chronological characteristics in data streams to enable efficient storage and retrievals. *Future Generation Computer Systems*, 2012.

[12] M. Malensek, S. L. Pallickara, and S. Pallickara. Expressive query support for multidimensional data in distributed hash tables. In *Utility and Cloud Computing (UCC), 2012 Fifth IEEE International Conference on*, nov. 2012.

[13] N. Ntarmos, T. Pitoura, and P. Triantafillou. Range query optimization leveraging peer heterogeneity. In *In 3rd International Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P*. Citeseer, 2005.

[14] S. Pallickara, J. Ekanayake, and G. Fox. Granules: A lightweight, streaming runtime for cloud computing with support, for map-reduce. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–10. IEEE, 2009.

[15] T. Pitoura, N. Ntarmos, and P. Triantafillou. Replication, load balancing and efficient range query processing in dhts. *Advances in Database Technology-EDBT 2006*, pages 131–148, 2006.

[16] S. Ramabhadran, S. Ratnasamy, J. M. Hellerstein, and S. Shenker. Prefix hash tree: An indexing data structure over distributed hash tables. In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing*, 2004.

[17] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. *A scalable content-addressable network*, volume 31. ACM, 2001.

[18] R. Rew and G. Davis. Netcdf: an interface for scientific data access. *Computer Graphics and Applications, IEEE*, 10(4):76–82, 1990.

[19] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001*, pages 329–350. Springer, 2001.

[20] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[21] The Apache Software Foundation. Apache hbase, 2012.

[22] P. Triantafillou and T. Pitoura. Towards a unifying framework for complex query processing over structured peer-to-peer data networks. *Databases, Information Systems, and Peer-to-Peer Computing*, pages 169–183, 2004.

[23] Wikipedia Contributors. Geohash, 2012.

[24] Q. Zhang, A. Riska, W. Sun, E. Smirni, and G. Ciardo. Workload-aware load balancing for clustered web servers. *Parallel and Distributed Systems, IEEE Transactions on*, 16(3):219–233, 2005.