Expressive Query Support for Multidimensional Data in Distributed Hash Tables

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

***Abstract*—The quantity and precision of geospatial and time series observational data being collected has increased in tandem with the steady expansion of processing and storage capabilities in modern computing hardware. The storage requirements for this information are vastly greater than the capabilities of a single computer, and are primarily met in a distributed manner. However, distributed solutions often impose strict constraints on retrieval semantics. In this paper, we investigate the factors that influence storage and retrieval operations on large datasets in a cloud setting, and propose a lightweight data partitioning and indexing scheme to facilitate these operations. Our solution provides expressive retrieval support through range-based and exact-match queries and can be applied over massive quantities of multidimensional data. We provide benchmarks to illustrate the relative advantage of using our solution over an established cloud storage engine in a distributed network of heterogeneous computing resources.**

***Index Terms*—Distributed File Systems, Cloud Infrastructure, Distributed Hash Tables, Data Partitioning, Query Evaluation**

I. Introduction

Designing and implementing distributed storage systems always involves trade-offs. Distributed hash tables (DHTs) generally have a number of desirable features in a distributed environment: they are decentralized, extremely scalable, and provide excellent load-balancing capabilities. However, these benefits do not come without a cost. Storage and retrieval semantics in a DHT are generally composed solely of *get* and *put* operations or variants thereof. While these operations are sufficient for information retrieval when the hash key is known, they are not amenable to situations where only a subset of the information required to retrieve a file is available or when a range of values is requested.

The research focus of this paper is support for exact match and range queries over multidimensional data managed by a DHT. The *dimensions* we consider include a geospatial component for location and elevation, a chronological component used for time series, and additional dimensions that could be numeric or string-based. In this case, *range queries* spanning multiple dimensions involve: (1) contracting or expanding geospatial regions, (2) constraining chronological components to a portion of the available time series data, or (3) specifying upper or lower bounds for numeric attributes of an element.

In this paper, we have adapted the DHT storage paradigm by using a hierarchical hashing scheme to create logical groupings of data with common attributes. These commonalities are often problem-specific; for instance, readings from multiple sensor arrays in particular regions could be grouped together to facilitate future analysis. Other properties used to correlate data points might include the time the data was generated or the specific device type that created the information. To create these logical groups, we control the dispersion of data items over a set of nodes such that small changes in the data values do not result in wide fluctuations in the set of nodes responsible for the corresponding hash space. An important aspect of this strategy is that it is executed without introducing large storage imbalances across the nodes in our system. We call this feature that manages the collocation of similar data items over a subset of nodes ***controlled dispersion***, which is one of the centerpieces of our solution.

To exploit our data partitioning strategy, we also introduce a lightweight, graph-based index that is shared among computing resources through a simple *gossip* protocol. The index is used to significantly reduce the search space of distributed queries, eliminating any nodes that do not have relevant data from the search. This optimization means faster response times, less network congestion, and lower CPU load, resulting in lower power and resource consumption. We have implemented this design in our cloud storage framework, Galileo [1], [2].

A. Usage Scenarios

Galileo is primarily tailored for scientific use with multidimensional data streams. Information handled by the system often has geospatial and time series properties and must be stored and retrieved quickly. Therefore, the space and time complexity of the partitioning and indexing algorithms used in our solution should not preclude fast and timely evaluation of range-based and exact-match queries. For example, atmospheric data often must be processed within a given time frame or it will not be useful in forecasts of environmental conditions. Other uses include data visualization, which can require nearly real-time streaming responses to queries, and data mining to discover correlations or trends.

Nodes in Galileo are processes running on commodity

hardware that can be assimilated into the cluster one node at a time in a scale-out manner. To help manage more of the overall system load and deal with heterogeneity, more powerful nodes can advertise as multiple *virtual nodes*.

B. Research Challenges

Supporting efficient search and storage capabilities for multidimensional data in a DHT provided a number of challenges that we address in this paper:

1) *Query Flexibility*. Users should be able to specify exact-match queries or use ranges and wildcards to retrieve their data.
2) *Correctness of Query Results*. Reductions in the search space to optimize query response times should produce results that are identical to those generated by doing an exhaustive search, i.e., reductions in the size of the search space should not come at the expense of correctness.
3) *Storage with Spatial Locality*. Information with common properties should be placed using our controlled dispersion paradigm to facilitate range queries and efficient disk access patterns.
4) *Adaptable Placement and Partitioning*. To maintain spatial locality, the storage algorithm employed by the system should be adaptable as storage needs evolve.
5) *Load Balancing*. To effectively utilize available computing resources, storage and processing requests should be balanced across the resources in system.
6) *Scalability*. Adding more computational resources should not have a significant impact on storage or query performance. Large indexing schemes or strategies that require excessive global state to be exchanged should be avoided.

C. Paper Contributions

This paper demonstrates the viability of using a hierarchical partitioning and indexing strategy to effectively reduce the search space of queries in a distributed setting. Our solution involves using controlled dispersion to place data items within the system, which in turn reduces the amount of information that must be stored in the global index. Using the techniques described in this paper, it is possible to provide query support that is much more expressive and flexible than that of a standard distributed hash table while still maintaining many of the core benefits of DHTs. We also contrast our approach with the well-known HBase cloud storage system, illustrating the effectiveness of our strategy in a distributed environment.

D. Paper Organization

The rest of this paper is organized as follows. Section II describes our distributed storage system and provides an overview of its intended use cases and architecture. Section III investigates storage and partitioning schemes using a hierarchical hashing structure. Section IV explains our distributed index and illustrates how our chosen storage scheme facilitates efficient retrieval of information, followed by a comparison of our system to HBase in Section V. We bring the paper to a close with a survey of related work in Section VI and our conclusions and future work in Section VII.

II. System Overview

Data storage in the scientific domain is the primary use case Galileo is designed to handle. Specifically, Galileo provides support for multidimensional data that has both spatial and temporal components, though any number of dimensions (called *features*) can be indexed and queried by the system. Galileo can read data stored in scientific formats such as Network Common Data Format (NetCDF) [3] or Hierarchical Data Format 5 (HDF5) [4] and also provides its own native multidimensional storage unit called *blocks*. This work provides support for retrieval of blocks with exact-match or range-based queries.

A. Granules

Galileo is based on the Granules [5] open-source distributed stream processing system. Granules provides support for computations that can be expressed using the MapReduce paradigm or as directed, cyclic graphs. These computations are orchestrated by Granules across a number of computing resources with a flexible scheduling strategy. Galileo provides an API that allows users to exploit the distributed computation features of Granules to process scientific data that has been stored in the system, or even pre-process incoming data streams before storage.

B. Network Topology

Galileo is organized as a Distributed Hash Table (DHT). A DHT is a type of *overlay network* that is created by partitioning a hash space among a number of computing resources. DHTs are generally decentralized and highly scalable; examples include Chord [6], Pastry [7], and Symphony [8]. Much like Apache Cassandra [9] and Amazon Dynamo [10], Galileo is a *zero-hop* (or *one-hop*) DHT, meaning requests are routed directly to their destination instead of taking intermediate hops through the network.

Contrasting with traditional flat DHTs, Galileo has a hierarchical structure. Individual *nodes* that represent computational resources in the system are placed into *groups*, which further subdivide the network. Groups can also contain any number of subgroups, and the number of groups is a user-configurable parameter. Research has shown that there are several performance and reliability advantages in using hierarchical DHTs [11], [12], and in particular they can provide benefits for storage and retrieval operations [13]. The logical groupings that arise from a hierarchical structure allow Galileo to place similar data items closer to each other in the network, which greatly increases the efficiency of range-based queries.

C. Metadata

The authors of SciDB [14] identified multiple differences between scientific and business-oriented data when designing their storage system. In particular, they found that scientific

data often has a much higher quantity and dimensionality, requiring a storage paradigm that can deal with data in the petabyte scale. Additionally, these data items have large sets of associated metadata that must be managed and stored.

To cope with these storage needs, Galileo maintains a hierarchical *metadata graph* for data management capabilities at individual nodes in the system. This graph stays resident in main memory, making it possible to quickly evaluate queries and then respond with results in the form of subgraphs called *datasets*. Datasets can be traversed, modified, and then used retrieve files from the system.

D. Experimental Data

For the purposes of this study, we sourced real-world data from the North American Mesoscale Forecast System (NAM) [15], which is maintained by the National Oceanic and Atmospheric Administration (NOAA). The NAM is run four times daily, and we sampled data recorded from 2009-2012 using our NetCDF input plugin to generate a dataset containing one billion (1,000,000,000) Galileo blocks, each of which is 8 KB. The data attributes we indexed and queried against included the spatial location for 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).

III. Hashing and Storage Methodology

Galileo supports streaming data that incrementally enters the system from a variety of sources. These data items are constantly evolving over time and can share a number of common attributes. Therefore, simply applying a standard hash function on the incoming data results in an approximately even distribution of files across all the nodes in the system, but does not account for similarity in the data being stored.

Inspecting the dimensions present in incoming data streams facilitates our controlled dispersion strategy by creating logical groupings of data in the hash space, but also increases the likelihood of storage imbalances across nodes in the system; a large amount of similar data could be stored in the same location, essentially eliminating the benefits of distributed storage. Using a hierarchical approach allows a balance to be struck between these two storage extremes: placing logically similar data items in the same groups and then using a second hash function to place data on specific nodes within the groups ensures that similar information is relatively balanced across a subset of the nodes in the system.

For our implementation, we used a two-tiered hashing hierarchy to determine where incoming data streams would be stored in a system consisting of 48 nodes. Since our experimental data was distributed across North America, we grouped the data items based on their spatial location and then applied a second hash function on the remaining dimensions of the data stored in each group.

A. Geohash: Spatial Hashing Algorithm

To obtain a location-based hash for spatial groupings, we applied the *Geohash* Algorithm [16] on incoming data. A Geohash is a string-based representation of a bounding box around a location created by interleaving bits obtained from latitude and longitude pairs. For example, the latitude and longitude coordinates of N 39.54, W 107.32 fall within the Geohash bounding box of *9x58vy4*. Longer Geohash strings represent more precise spatial regions, a characteristic which can be exploited during the hashing process to obtain a specific granularity for positioning data in the system.

Since Galileo deals with range queries in addition to exact-match semantics, it is beneficial to limit the precision of the Geohashes for incoming data to provide coarser-grained groupings. For instance, using the first two characters (10 bits) of a Geohash results in spatial "hash buckets" of approximately 600 by 1000 kilometers. Increasing the precision to four characters (20 bits) results in a bucket size of 20 by 30 kilometers. In the case of a 20-bit hash, users of the system can predict that data samples taken within about 20 kilometers of each other will be placed in the same or similar spatial group. Galileo's retrieval system also exploits this property of Geohashes to expand or contract the desired search space in queries.

The Geohash precision used for positioning files in the system can be tuned by users depending on their storage needs; if a user plans to store data belonging to a small geographic region, a more precise hash should be used. Conversely, a less precise Geohash would be optimal for data spread across the entire Earth. The precision can also be modified over time as storage needs evolve, or change depending on the application that is streaming data into Galileo; unlike a traditional DHT our retrieval system, described in Section IV, does not require the hashing algorithm to stay consistent in order to locate data.

B. Feature Hashing

Once a group has been chosen for a data item based on its spatial characteristics, an additional level of hashing is required to select a destination node within the group. At this stage in the hashing process, any number of the remaining available data dimensions can be used as input for the hash function. Our particular dataset has a temporal range associated with each data item, so we used the initial recording time as input to the SHA-1 hash algorithm and then divided the hash space among the nodes in each group.

C. Data Distribution and Load Balancing Evaluation

To determine the impact of our hierarchical hashing scheme on how files are distributed in the system, we compared the distribution results of our controlled dispersion strategy against the same data inserted using a flat SHA-1 hash of all metadata values. Figure 1 illustrates the distribution of files in the system using the flat hash; each of the 48 nodes represented in the figure contains approximately 2% of the data in the system. This distribution mechanism provides excellent load balancing capabilities, but does not assist the system's retrieval engine because similar data items are spread across all nodes in the cluster.

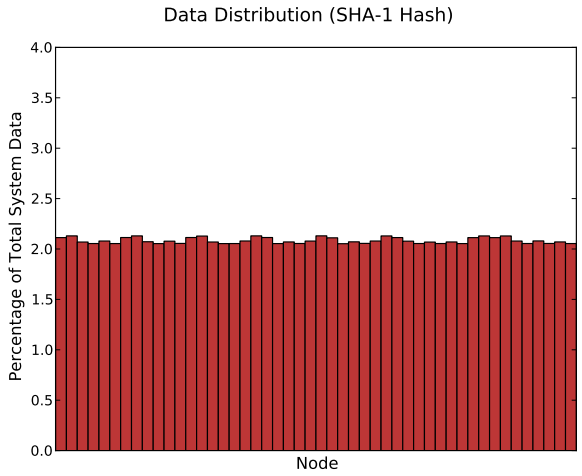

Fig. 1. Distribution of files in the system using a flat SHA-1 hash across all data dimensions.

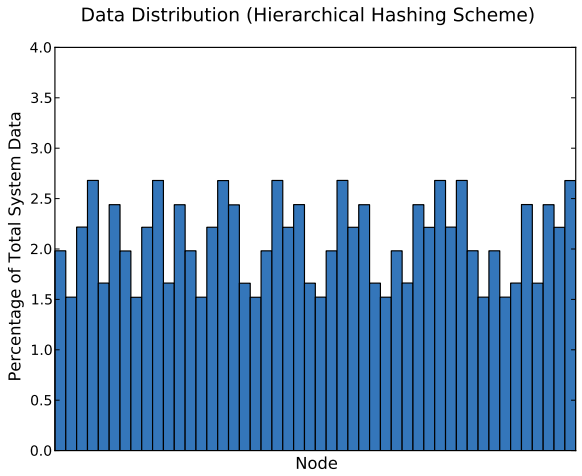

Fig. 2. Distribution of files in the system using our two-tiered hierarchical hashing scheme.

The distribution results for our hierarchical hashing scheme are shown in Figure 2. Unlike the flat SHA-1 hash, we have imposed a greater storage imbalance in the system, but similar data items are now placed logically closer to each other. Table I contains a summary of the differences between the two storage schemes. While our hierarchical solution provides less balance in load, there are no nodes in the system with an extreme shortage of data; the lightest-loaded node in a 48-node system still contains 15.2 million blocks (1.5%) of the total system data.

TABLE I
PERCENTAGE OF TOTAL DATA STORED AT EACH NODE

	Flat SHA-1 Hash	**Hierarchical Hash**
Average (%)	2.08	2.08
Min (%)	2.05	1.52
Max (%)	2.13	2.68
SD (%)	0.03	0.41

IV. INDEXING AND RETRIEVAL

Once data has been stored across the nodes in the system, an efficient means for retrieval is necessary. In a traditional DHT a hash function is used for both storage and retrieval operations, but our support for range queries across any number of data dimensions makes the retrieval process much more challenging. Additionally, data storage needs often evolve over time and may require changes to be made to the hashing hierarchy, meaning that all previously-stored data would not be reachable using a new set of hash functions.

To alleviate these issues, we have developed a lightweight global indexing scheme called the *feature graph*. This graph is similar in design to the per-node local metadata graph, but is a completely separate entity in the system. In general, maintaining a global index of all information being stored in a distributed system is costly both in terms of memory and network IO, so we focused on ensuring that the addition of this index would not have a noticeable impact on the system's performance.

A. Feature Graph Implementation and Structure

Any node in the system can be contacted to perform a storage operation, which will then route the request directly to its destination node. Upon arrival, the data is fully inspected to determine its attributes, which could include spatial location, temporal information, features, and details about the device that generated the data. These individual pieces of metadata become vertices that will be inserted into the feature graph. A collection of vertices for a given data item is pieced together to form a *path*, which has a specific ordering of features. If a feature is not present in the incoming data stream, a null vertex is inserted in its place.

To decrease the amount of data points being stored in the global index, the resolution of each feature is reduced. For example, the precision of a Geohash can be reduced by applying a simple bit shift. Smaller units of time, such as minutes or even hours can be ignored from temporal values, and numeric readings can be placed within coarser-grained ranges of values. This operation is especially crucial in storing collections of floating-point values since exact matches are not possible. These reductions in resolution place data points in common groups, called *tick marks* in our implementation. Tick marks can be derived from domain knowledge about the information being stored or from the data source itself; many datasets stored in formats like NetCDF describe expected ranges and the attributes of each dimension. Each tick mark is assigned to a vertex in the feature graph.

The granularity of tick marks used impacts system performance in a number of ways. For instance, increasing granularity also increases the number of vertices stored in the feature graph, which in turn decreases the number of storage

nodes associated with each vertex. As the number of nodes associated with each vertex drops, fewer nodes will need to be contacted during a query operation, leading to faster response times. On the other hand, increasing granularity does not come without a cost: more vertices require more memory and more state to be exchanged between nodes in the system.

Table II provides an overview of three tick mark configurations we used to achieve a smaller or larger granularity in the feature graph. For example, the "coarse" granularity allocates temperature readings within ranges of 10 Kelvin and humidity levels within ranges of 5%. This would effectively place all temperatures from 300-309 K into one group, 310-319 K into another group, and so on. The wind speed and snow depth readings in our dataset already contained a small range of values, so were not constrained further in our tests. It is also important to consider that each dimension in our dataset is represented as a floating-point number, so even an integer value of 1 will still reduce the resolution of the data being indexed. These configurations help illustrate the impact of tick marks on the performance and resource consumption of the feature graph.

TABLE II
FEATURE GRAPH GRANULARITY CONFIGURATIONS: VALUES REPRESENT TICK MARK RANGES

Index Granularity	Temperature (K)	Humidity (%)
Coarse	10	5
Medium	5	2
Fine	1	1

Figure 3 demonstrates how increasing or decreasing the granularity of the tick marks impacts query performance. For this benchmark, we assumed that the query does not contain any geospatial information and therefore the destination group cannot be ascertained from the query parameters. In this worst-case scenario, the figure shows the average reduction in search space as additional features are specified in the query. Traversing a path is similar to performing a logical AND operation across available dimensions, so the sharp reduction in search space as more dimensions are specified is fairly intuitive. If an approximate geospatial location is provided with enough precision, no more than 17% of the nodes in our 6-group system will be contacted. The search space within a group can also be reduced similarly as more dimensions are added. Table III contains statistics on how the number of vertices and edges in the graph increases as granularity increases, and the resulting memory consumption.

Vertices in the feature graph have a number of components: an adjacency list, an associated tick mark that data is placed within, and a list of nodes and replicas that contain data within the range. Once a path of vertices has been constructed it can be added to the feature graph. Starting with the first vertex, the system determines if the graph already contains a vertex associated with the same tick mark. If such a vertex exists, its list of storage nodes is updated (if necessary) and the next

TABLE III
GRAPH STATISTICS WITH DIFFERING INDEX GRANULARITY

Index Granularity	Memory (MB)	Vertices	Edges
Coarse	1.8	13,927	193,348
Medium	6.3	56,967	614,627
Fine	38.0	454,569	2,267,984

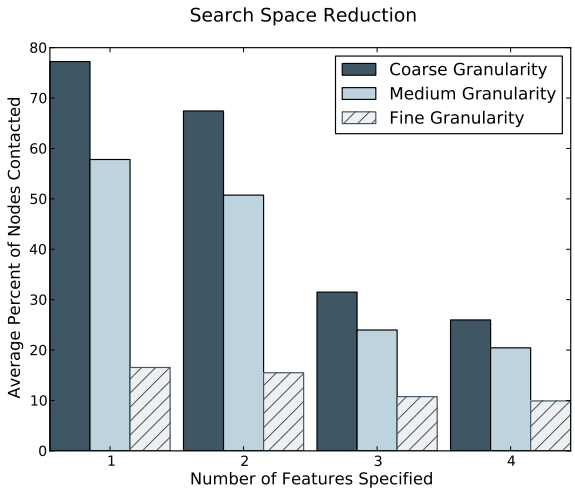

Fig. 3. Reduction of the search space with different tick mark configurations as more dimensions are added.

vertex in the path is added as an adjacent vertex. Otherwise, a new vertex is created. This process continues until the end of the path is reached. The resulting graph generated by this process has a large number of edges pointing to destination computing resources in the system, but it also means that the entire remaining search space is known at any vertex in the graph.

B. Graph Optimization

Before inserting any new data into the system, a lookup is performed to determine if an identical path already exists in the feature graph. If such a path exists and contains references to nodes that are candidates for storing the new data, one of the eligible nodes is selected at random to complete the storage operation. This optimization helps ensure that logically similar data items stay grouped within the system. In addition, if new data matches a preexisting path and does not require any graph updates, then no state will need to be exchanged between nodes as a result of the new data insertion.

Over time, the feature graph reaches a state where only a small number of new vertices and edges must be created as data is stored in the system. It is possible to pre-allocate a large portion of the graph using the configured tick mark ranges during the initial setup of the system, which decreases the amount of vertices and edges that must be created as new data is inserted.

Our approach for inserting paths in the feature graph imposes a hierarchical structure on the search space; search terms

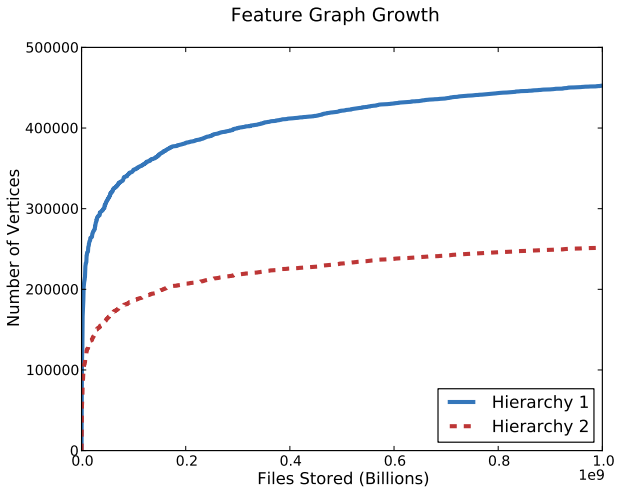

Fig. 4. Vertex growth for two different path hierarchies as more data is added to the system using the "fine" granularity configuration.

near the top of the hierarchy can be found without traversing as far through the graph. This also has an impact on the number of vertices inserted in the graph depending on the possible range of values for each dimension. A visualization of the growth of vertices in the feature graph is provided in Figure 4. "Hierarchy 1" refers to a path layout that inserts features in the following order: temperature, humidity, wind speed, and then snow depth. "Hierarchy 2" reverses the order of the features; this results in a sharp reduction of vertices in the graph but places the features with a lower range of values at the top of the graph. In this situation, there is a clear trade-off between the size of the graph and the speed at which search terms can be located. We have leveraged functionality in Galileo to permit users to reorient the feature graph at runtime depending on their memory and query needs.

C. Gossip Protocol

Galileo employs an *eventually consistent* model, meaning that changes to the system are not visible to all nodes immediately. To disseminate these state changes, we developed a simple gossip protocol that involves sharing collections of new paths that have been added at each node. Within a group, nodes monitor the status of their neighbors with small, frequent *heartbeat* messages. These heartbeats are sent through the entire group at a regular interval, set to one second in our current implementation. As new information enters a node, "dirty" paths through its feature graph that have been updated are maintained in a separate data structure and then included in the heartbeat messages. A monotonically increasing graph state identifier is incremented each time an update is received, and the group eventually converges on a consistent feature graph.

Table IV provides an overview of the size of heartbeat messages sent by a node while storing approximately 500 Galileo blocks per second. While the updates are larger than a proportionately-sized part of the overall feature graph, they still provide some inherent "compression" when incoming data is similar due to duplicate vertices in the paths. As the heartbeat interval increases, the size of the messages decreases.

TABLE IV
HEARTBEAT MESSAGE SIZE, INSERTING 500 BLOCKS PER SECOND

Index Granularity	Message Size (KB)	Vertices	Edges
Coarse	8.7	90	528
Medium	28.0	365	996
Fine	45.0	631	1,208

To deal with the rest of the nodes in the system, groups elect a *leader* that is responsible for informing other group leaders of updates to the feature graph. Leaders communicate in the same fashion as they do within their group, but instead of communicating with adjacent nodes the leaders simply publish updates on a corresponding group stream within the Granules framework. This means that leaders only need to know of the existence of other groups, but not their assigned leaders. Continuing down through the network hierarchy, subgroups communicate in the same way. Subgroups lowest in the hierarchy will be most consistent as updates trickle back up to the higher groups.

D. Consistency and Fault Tolerance

While our feature graph and partitioning system help provide fast responses to queries, we do not guarantee that the index state is consistent across all nodes in the system. However, using the feature graph is not required for retrieval operations; requests can be broadcast through the entire system or to specific groups if desired, but latencies will increase as more nodes are involved in a query. In fact, Galileo supports running in a fully-stateless mode to facilitate usages where group sizes are small and can be located reliably. As mentioned previously, the feature graph can also be used to reduce the search space within a group rather than across the entire system.

Galileo primarily expects transient failures to occur within the network, so if a node is started with an older version of the feature graph it can request a full update from one of its neighbors. Detailed graph information including the time of the last update and counts for vertices and edges is included in heartbeat messages, so a node can quickly determine that it is not synchronized with its group. The system also ensures that a set replication level is maintained for all data, so if a node is not responding then one of the replicas can be requested instead.

V. PERFORMANCE EVALUATION

To benchmark the effectiveness of our new partitioning strategy and index, we compared the read and write throughput of Galileo to Apache HBase [17] version 0.92.1, an open-source implementation of Google BigTable [18]. We ran HBase on Hadoop [19] and HDFS version 1.0.3. Our test environment was a heterogeneous 75-node cluster composed

of 47 HP DL160 servers (Xeon E5620, 12 GB RAM, 15000 RPM Disk) and 28 Sun Microsystems SunFire X4100 servers (Opteron 254, 8 GB RAM, 10000 RPM Disk). We reimplemented our storage strategy to fit the BigTable data model by using block UUIDs as our row key, prefixed with a Geohash of the block's spatial location. This ensures that rows are sorted using spatial locality. Since HBase does not support range queries without scanning across records, we also modified our indexing strategies to operate on top of HBase, which provided our test program with block UUIDs that could be retrieved directly from the system. Each record from our dataset was approximately 8 KB in size. We also used the official Java API provided with HBase to communicate with the system rather than the interactive shell or a third-party interface.

Table V compares the read throughput of Galileo and HBase. Queries were constrained to a 20-kilometer geographic region and then sent to both systems, with the HBase request submitted as a single batch operation. Each test was performed 100 times on different spatial areas to balance requests across the cluster. In the case of Galileo, no more than two groups were contacted per query. Read throughput was the primary area we aimed to improve with the feature graph; by contacting fewer nodes, Galileo can provide extremely fast retrievals.

TABLE V
READ THROUGHPUT: GALILEO VS HBASE (100 RUNS)

Blocks	**Galileo**		**HBase**	
	Read (ms)	SD (ms)	Read (ms)	SD (ms)
1	0.89	0.33	0.97	0.07
250	22.59	3.72	172.65	48.43
500	28.31	4.02	479.12	456.01
1000	85.07	11.23	865.68	158.86

Write operations, outlined in Table VI, take longer but scale up similarly to the read results. Data from a different spatial region was submitted to both systems for each of the 100 iterations of this test. These tests dealt with data from North America starting roughly in California, United States, and moved further toward the east coast with each iteration.

TABLE VI
WRITE THROUGHPUT: GALILEO VS HBASE (100 RUNS)

Blocks	**Galileo**		**HBase**	
	Write (ms)	SD (ms)	Write (ms)	SD (ms)
1	0.94	0.42	5.18	0.03
250	146.33	13.62	627.71	120.03
500	232.80	14.20	1,138.09	208.61
1000	409.79	14.91	2,442.61	414.26

This benchmark illustrates the stark difference in intended use cases between Galileo and HBase. Galileo is mainly concerned with retrieving large amounts of small files based on their metadata and transferring them to client applications, whereas HBase deals primarily with sparse, semi-structured or unstructured data for processing operations. The two systems could be used in similar areas, but they have much different data models that may fit some problems better than others. Ultimately, this benchmark shows that there is a niche in current state-of-the-art distributed storage systems than can be filled by Galileo.

VI. RELATED WORK

Cassandra [9] shares many attributes with Galileo in its network layout and storage system. It provides a number of different data partitioning approaches for users depending on 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.

SciDB [14] is a scalable scientific storage system that supports multidimensional data. Although its name implies a link with relational databases, SciDB is not concerned with providing ACID guarantees or strong transaction support. Instead, SciDB focuses on incremental scalability and petabyte scale datasets. The system also provides built-in computation and analysis tools, whereas Galileo is only concerned with storage; analysis can be performed outside the system within the Granules framework or some other distributed computation engine. Metadata is stored in a centralized *system catalog* implemented as a PostgreSQL database, contrasting with the combination of feature graph and metadata graphs used in Galileo.

Apache HBase [17] is an open-source implementation of Google BigTable [18] designed to handle massive amounts of tabular data. HBase runs on HDFS [20], the distributed file system included with Hadoop [19]. HBase is not specifically designed for Geospatial storage, but its tabular data model is well-suited for multidimensional data. Contrasting with Galileo's eventual consistency, HBase provides strictly consistent reads and writes. In addition, HBase does not support queries across all of a file's dimensions as Galileo does.

Citing the use of hierarchies in traditional distributed applications such as multicast and DNS, Ganesan, Gummadi, and Garcia-Molina [13] propose a paradigm called *Canon*, which provides a hierarchy on top of existing flat DHTs. Canon subdivides system computational nodes into *domains*, which provide logical groupings of resources. Domains can contain any number of subdomains, and a domain that contains system nodes is referred to as a *leaf domain*. Leaf domains are structured in the same way as a traditional flat DHT.

Crescendo, a hierarchical version of Chord [6] implemented using Canon, provides a link structure in which each domain is allocated a discrete Chord ring. The Chord ring for each domain is derived by *merging* subdomains recursively, with

the top-level domain's ring containing the entire DHT. Once the ring hierarchy has been created, routing requests through the network can be done in a similar fashion to a standard Chord implementation.

2T-DHT [12] implements a two-tier DHT hierarchy for publish/subscribe systems. In 2T-DHT, the hierarchy is used to organize nodes based on their uptime and available resources. All nodes begin in a lower tier and then migrate to the higher tier as they demonstrate their stability. The 2T-DHT network is implemented as multiple Chord rings, which reduces the amount of communication required to publish messages to all nodes. This communication pattern is similar to that used in Galileo's gossip protocol.

VII. Conclusions and Future Work

A. Conclusions

The partitioning and indexing scheme we have implemented in Galileo allows clients to make efficient exact-match and range queries across a number of dimensions, a feature not supported by traditional DHT-based storage systems. This functionality is made possible by (1) ensuring that data items with some similarity, e.g., spatial locality, time series, or another attribute are stored using our controlled dispersion strategy, (2) indexing the location of these data items, and (3) reasoning about the data stored in the system at a lower resolution, thus providing a higher-level or general view of the information to reduce network traffic and memory consumption.

We have shown that our modifications to the DHT paradigm are effective in providing advanced query support through our extensive benchmarks, including a favorable comparison against the HBase cloud storage system. In addition, we have shown that our indexing scheme scales up even when storing massive datasets and still provides significant performance improvements. These features were provided without sacrificing efficient load balancing of storage resources in the system, a fundamental aspect of DHTs.

B. Future Work

Clustering algorithms such as STREAM [21] and CluSTREAM [22] could provide more accurate groupings of similar data within our storage system. These algorithms focus on using a small amount of CPU time and memory to generate their clusters, and therefore would complement our lightweight gossip protocol. In addition, many stream-based clustering algorithms can effectively handle data evolution over time, meaning the storage algorithm would not require modification or reconfiguration as incoming data changes. Additionally, Artificial Neural Networks could be used to predict and react to changing query workloads or new resource constraints and provide information that could be used to reorient our feature graph dynamically.

References

[1] 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*, dec. 2011, pp. 17 –24.

[2] ——, "Exploiting geospatial and chronological characteristics in data streams to enable efficient storage and retrievals," *Future Generation Computer Systems*, 2012.

[3] R. Rew and G. Davis, "Netcdf: an interface for scientific data access," *Computer Graphics and Applications, IEEE*, vol. 10, no. 4, pp. 76–82, 1990.

[4] Q. Koziol and R. Matzke, "Hdf5–a new generation of hdf: Reference manual and user guide," *National Center for Supercomputing Applications, Champaign, Illinois, USA, http://hdf. ncsa. uiuc. edu/nra/HDF5*, 1998.

[5] 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*. IEEE, 2009, pp. 1–10.

[6] 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*, vol. 31, no. 4, pp. 149–160, 2001.

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

[8] G. Manku, M. Bawa, P. Raghavan *et al.*, "Symphony: Distributed hashing in a small world," in *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, vol. 4, 2003, pp. 10–10.

[9] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[10] D. Hastorun, M. Jampani, G. Kakulapati, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazons highly available key-value store," in *In Proc. SOSP*. Citeseer, 2007.

[11] S. Zoels, Z. Despotovic, and W. Kellerer, "Cost-based analysis of hierarchical dht design," in *Peer-to-Peer Computing, 2006. P2P 2006. Sixth IEEE International Conference on*. IEEE, 2006, pp. 233–239.

[12] M. Pandey, S. Ahmed, and B. Chaudhary, "2t-dht: A two tier dht for implementing publish/subscribe," in *Computational Science and Engineering, 2009. CSE'09. International Conference on*, vol. 2. IEEE, 2009, pp. 158–165.

[13] 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*. IEEE, 2004, pp. 263–272.

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

[15] National Oceanic and Atmospheric Administration. (2012) The north american mesoscale forecast system. [Online]. Available: http://www.emc.ncep.noaa.gov/index.php?branch=NAM

[16] Wikipedia Contributors. (2012) Geohash. [Online]. Available: http://en.wikipedia.org/wiki/Geohash

[17] The Apache Software Foundation. (2012) Apache hbase. [Online]. Available: http://hbase.apache.org

[18] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[19] A. Bialecki, M. Cafarella, D. Cutting, and O. OMALLEY, "Hadoop: a framework for running applications on large clusters built of commodity hardware," *Wiki at http://lucene. apache. org/hadoop*, 2005.

[20] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*. Ieee, 2010, pp. 1–10.

[21] S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan, "Clustering data streams," in *Foundations of computer science, 2000. proceedings. 41st annual symposium on*. IEEE, 2000, pp. 359–366.

[22] C. Aggarwal, J. Han, J. Wang, and P. Yu, "A framework for clustering evolving data streams," in *Proceedings of the 29th international conference on Very large data bases-Volume 29*. VLDB Endowment, 2003, pp. 81–92.