

Fast, Ad Hoc Query Evaluations over Multidimensional Geospatial Datasets

Matthew Malensek, Sangmi Pallickara, and Shrideep Pallickara, *Members, IEEE*

Abstract—Networked observational devices and remote sensing equipment continue to proliferate and contribute to the accumulation of extreme-scale datasets. Both the rate and resolution of the readings produced by these devices have grown over time, exacerbating the issues surrounding their storage and management. In many cases, the sheer scale of the information being maintained makes timely analysis infeasible due to the computational workloads required to process the data. While distributed solutions provide a scalable way to cope with data volumes, the communication and latency involved when inspecting large portions of an overall dataset limit applications that require frequent or rapid responses to incoming queries.

This study investigates the challenges associated with providing approximate or exploratory answers to distributed queries. In many situations, this requires striking a balance between response times and error rates to produce meaningful results. To enable these use cases, we outline several expressive query constructs and describe their implementation; rather than relying on summary tables or pre-computed samples, our solution involves a coarse-grained global index that maintains statistics and models the relationships across dimensions in the dataset. To illustrate the benefits of these techniques, we include performance benchmarks on a real-world dataset in a production environment.

Index Terms—Approximate query processing, ad hoc exploration, multidimensional data, distributed hash tables

1 INTRODUCTION

The past decade has seen substantial growth in data volumes, with a large portion of this information falling broadly into the category of geospatial, time-series datasets. Such datasets are useful in understanding phenomena such as weather, traffic conditions, or disease spread. They also facilitate threat assessments based on video and activity tags, or can provide location-based services relating to commerce. The proliferation of observational devices such as Doppler radars, satellites, and in-situ sensors alongside falling hardware costs, better network connectivity, and increased measurement rates have all contributed to the growth of this data.

Generation of such datasets involves measuring *features* of interest at particular geospatial locations. Examples of features could include temperature, humidity, traffic density, or viral infections, with each measurement having an associated geolocation and time. These readings are taken at regular intervals or when an environmental change has been detected. Such measurements can help in understanding how features (and their underlying phenomena) evolve. This can be useful for a range of activities, including planning, forecasting, or disaster mitigation. Features often take on values that evolve spatiotemporally; for example, the range of temperature values at a particular location depends on both the time of the day and day within the year. Multiple features and the values they take on comprise a *feature hyperplane*.

As datasets continue to grow in size, efficient query evaluations underpin knowledge extraction, visualization, and processing. We classify queries in such settings into two categories: *rigorous* and *approximate*. Rigorous queries have a clear specification of attributes and are exhaustively evaluated, which often results in disk accesses or other related I/O activity. The objective during rigorous query evaluations is to ensure that

results are accurate with no false positives or negatives.

On the other hand, approximate queries correspond to an exploratory, ad hoc analysis of the feature space. Such queries allow insights to be gained about the underlying dataset and can drive further investigation if deemed necessary. Studies have shown that developers and end users often rely on a continuous query cycle to calibrate attributes and constraints, with several preliminary inquiries leading up to the final set of query parameters [1], [2], [3]. Unlike rigorous queries, the most critical aspect of approximate queries is often evaluation time, with speed playing a significantly more important role than accuracy. By their very nature, we expect the number of approximate queries being serviced by a system to exceed that of rigorous queries.

Storage and analysis activities at this scale have exceeded the capabilities of standalone commodity hardware, leading to the development of distributed solutions that are highly scalable and elastic. The approximate query processing engine (AQP) described herein was implemented in the context of our multidimensional storage system, Galileo [4], [5]. Galileo's design is based on distributed hash tables (DHTs), which have well-known scalability properties that have been demonstrated in systems such as Facebook's Cassandra [6] and Amazon's Dynamo [7]. DHTs have also been used in BitTorrent, the Coral CDN, and the YaCy peer-to-peer search engine. However, our algorithms are not bound to any particular network design. In fact, we have adapted several of our features to MongoDB [8], a popular storage system that supports geospatial information.

To our knowledge, none of the current DHT-based systems accommodate the types of queries or retrieval operations we have developed in this work, which were performed in the context of voluminous, time series datasets. Through expressive, massively parallel queries, we enable user-friendly analysis that can be used to supplement or completely replace custom MapReduce [9] jobs or distributed computations. Once exploration of the feature space has been completed, distributed computations can be launched directly on relevant data points while preserving data locality.

• M. Malensek, S. Pallickara, and S. Pallickara are with the Department of Computer Science, Colorado State University, Fort Collins, CO 80523. E-mail: {malensek,sangmi,shrideep}@cs.colostate.edu

1.1 Research Challenges

The primary goal of this study is to enable fast evaluation of approximate queries over geospatial, time-series datasets distributed across resources in a cloud or clustered environment. To be practical, these evaluations must also be reasonably accurate. The challenges involved in doing so include the following:

- The datasets over which these query evaluations are performed are voluminous. Furthermore, explorations may involve multiple approximate queries in rapid succession.
- The datasets are multidimensional. Individual feature types may be single- or double-precision floating-points, integers of varying size, strings, or enumerations.
- The most common constraint within approximate queries involves range. In addition to the specification of ranges over features, ranges could also be specified along the spatial and temporal dimensions.
- Query evaluations must be performed in the presence of other concurrent rigorous and approximate queries.
- Since data is continually arriving and being added to the datasets, the feature space is always evolving. This impacts the distribution of feature values and the evaluation of range constraints.

Since approximate queries must be evaluated in real time, disk accesses should be rare. Schemes that prefetch data blocks into memory in anticipation of incoming queries may incur high disk I/O costs. In the presence of concurrent approximate queries, this could also lead to high rates of page faults and thrashing when queries target blocks that were not paged in.

1.2 Research Questions

Key research questions that we explore in this paper include the following:

- 1) How can we manage the trade-off between accuracy and timeliness of query evaluations, while enabling configurable coverage of the accuracy gradient?
- 2) How can we minimize disk accesses while ensuring a low memory footprint for our data structures?
- 3) How can we establish error bounds even in cases where ranges are specified for particular feature values?
- 4) How can we support explorations of the feature space for portions of the hyperplane where the availability of data is disproportionately sparse?
- 5) How can interference between queries, both rigorous and approximate, be reduced?

All of these goals must be achieved in the presence of continually-arriving data and feature space evolution.

1.3 Overview of Approach

Enabling efficient approximate query capabilities involved the following components: (1) expressive query specification, (2) algorithms and data structures for evaluating the queries, and (3) refinements to allow for faster evaluations. The features described in this paper were implemented in our distributed storage framework, Galileo, but are applicable to other frameworks as well.

The query specification constructs provided by this work include:

- Time- and error-based constraints
- Limits or requirements on the number of results returned
- Operators for exploring temporal interactions

- *Fuzzy* queries that allow parameters to be incrementally relaxed to discover additional or related information
- *Promotion* of approximate queries to fully-fledged rigorous queries when particular conditions are met

Query evaluation is handled by a multi-tier framework with each component working in tandem. The storage nodes in Galileo maintain fine-grained metadata about their portion of the overall dataset, along with a copy of coarse-grained metadata that describes the dispersion of information throughout the entire system. Additions to the indexing subsystem include:

- Autonomous adaptation of index granularities based on changing storage densities
- A graph-based representation of temporal intervals
- Embedded summary statistics for each feature type
- Online dataset sampling

To address interference between different types of queries, we have incorporated mechanisms to automatically adjust dataset coverage, the degree of fuzziness, and the depth of graph traversals during query evaluations to control for timeliness. This ensures sustained throughput over time and prevents starvation of processes with complex query profiles.

We have also added two refinements that help increase approximate query performance: the first relates to graph reorientations, which allow reductions in the graph size that result in faster traversals. The second refinement involves Bloom filters, which are used to identify queries that will not produce matching results across feature ranges, allowing the system to quickly resolve the query without a graph traversal. Finally, we provide detailed performance analysis of our system design.

1.4 Paper Contributions

The work presented herein demonstrates the feasibility of evaluating approximate queries over multidimensional, geospatial, time-series datasets. Contributions include:

- We have incorporated support for ad hoc, approximate queries in the context of a distributed hash table (DHT) based storage system. Such systems are often used to manage semi-structured and unstructured data [6], [7].
- The research presented in this work could also be applied to other distributed storage systems. Rather than retrieving information based on keys, we provide value-based queries that span multiple dimensions and types.
- Our empirical evaluations demonstrate the feasibility of our approach at large scales. In our benchmarks, the test dataset spans a billion files, each of which contain multiple feature values for specific locations.
- We allow discovery of attribute ranges across multidimensional feature values that have not been expressed. This enables discovery of unusual feature combinations. We also allow the exploration of the outlier space with stratified sampling. To our knowledge, no other system provides such functionality in the context of time-series data.
- We not only provide a framework for the specification of approximate queries, but also allow specification of error bounds across ranges of feature values.

1.5 Paper Organization

The rest of the paper is organized as follows: Section 2 provides an in-depth overview of the methodology used in this work. Section 3 describes our distributed storage system, Galileo.

Section 4 outlines the query functionality that guided our implementation, followed by Section 5 which describes the components of our query evaluation framework. Section 6 describes how our framework was successfully adapted to another storage system, while Section 7 includes performance benchmarks in a virtualized setting. Section 8 provides a survey of related work from the literature, followed by Section 9 which concludes the paper and describes our future research direction.

2 METHODOLOGY

Our approximate query functionality involves both query specification and efficient distributed evaluation; the user-facing specification features described herein contributed to the development of the algorithms and data structures outlined in the paper. On the query specification front, we focus on the simplification of common usage patterns that are frequently observed in large data stores. In many cases, these flexible queries obviate the need for user-defined distributed computations (e.g., MapReduce) that would traditionally be used for analyzing and aggregating data. Examples of such analysis include creating climate charts that summarize weather patterns in a region over time, determining political affiliations in social networks, or optimizing model parameters based on a given set of outputs. The backend implementation of these features combines aspects of systems design, data structures, graph theory, interval algebra, information theory, and statistics. The interactions between these elements are also considered; for instance, controlling the degree of data dispersion during storage operations has an impact on the network design of the system as well as the amount of information that can be stored in local data structures.

2.1 Query Specification

Our approach supports both user-initiated and system-driven instrumentation of queries. Queries can involve a complete set of feature parameters, or can specify ranges of acceptable values to be returned. Unlike a tabular data store, Galileo can provide results in the form of a graph, which retains the relationships between data points and allows for further analysis and traversal.

To supplement these queries with approximation, we consider three query dimensions: the number of results returned, the time elapsed, and the overall accuracy (which may be measured by a confidence interval, or percent coverage of the underlying data). These dimensions are inter-related: allowing a query to run for a longer period of time will often return more results and improve accuracy, while limiting accuracy requirements will result in a faster response.

When dealing with temporal properties, interactions often occur between particular intervals. For instance, one event may precede another, overlap, or occur during the same time period. To support these types of analysis, we have integrated Allen’s Interval Algebra [10], [11] into our query engine. These query operators allow users to discover causal relationships or inspect correlated events.

Since the entire feature space is often unknown, *fuzzy queries* allow automated, incremental relaxation of the bounds specified on the features within a query. This effectively increases feature and dataset coverage, and is especially useful when a particular set of constraints is not satisfied by the underlying dataset but the nearest-matching results would still be useful.

We allow specification of both the features or dimensions along which these bound relaxations are made, along with the degree to which the relaxation is performed for a particular dimension. Fuzzy queries may have bounds on the number of matching results or turn around times associated with their evaluations to expand or contract exploration.

Approximate queries are often used for exploratory purposes, with a rigorous query following the initial exploration phase. With *query promotion*, users can transform an approximate query to its rigorous equivalent automatically on the server side when a particular set of conditions are met. Additionally, each path a query takes through the index structures in Galileo is recorded, allowing the “paths least traveled” or most popular data points to be discovered and analyzed. This functionality is used to improve the accuracy of the sampling algorithms in the system, but can also highlight potentially unusual or anomalous data.

2.2 Distributed Query Evaluation

One key aspect of our approach is to limit the number of nodes involved in a given query. This is achieved through several data structures that are specifically designed for search space reduction, which ultimately decrease latency and improve response times. Optimizations include those that minimize communication, the footprint of the data structures, and the time required to evaluate queries. These involve dynamic reconstruction of our indexes, embedded statistics, and sampling methods, and strive to reduce any required user intervention. The space complexity associated with the data structures used as part of the query evaluations must also be low. Increased memory footprints associated with data structures can lead to increased paging and thrashing that in turn leads to increased disk I/O.

In our system, we use graphs to manage metadata. Modeling the metadata as a graph allows us to support approximate queries by controlling the paths, comprising vertices and edges, that are traversed during query evaluations. There are two graphs maintained at each storage node in Galileo: the *feature graph* that maintains a coarse-grained view of all data blocks in the system and the *metadata graph* that maintains a fine-grained view of the data blocks stored at the particular node. The feature graph helps with reduction of the search space during distributed query evaluations and the metadata graph helps with restricting file-block retrievals to only those blocks with data matching the specified query.

The feature graph allows us to ensure that every query does not target all nodes in the system but rather only a subset of nodes that are most likely to hold the data. This helps us avoid cases where the system throughput and response times for queries drop significantly because every storage node is involved in evaluation of every query. Vertices in the graph represent range of values for a particular feature; the size of the range represented by a vertex reflects the granularity and, correspondingly, the expected accuracy for query evaluations that results in traversals through that vertex. We use these graphs to evaluate queries and also report on the error bounds associated with our evaluations for a particular feature.

To improve accuracy and reduce the memory footprint of the graphs, we dynamically tune the resolution of the information they manage. The resolution is dynamically adjusted so that there is greater resolution in places where the density of values for a particular feature is the highest. To accomplish this,

we rely on a statistical measure, the coefficient of variation (CV). The CV is computed for each vertex and measures the scattering of values within the range represented by that vertex. We split a vertex with high CV to improve accuracy and reduce error bounds. Conversely, we combine vertices (representing a contiguous range of value) that have a low degree of scattering to conserve space. To estimate the error bounds associated with the query evaluations, we rely on *online* updates of summary statistics associated with various paths.

Graph hierarchies are dynamically reconfigured at runtime to improve query evaluation times and conserve memory. Different graph orientations result in a different set of memory footprints for managing the same information — this is because each orientation results in a different number of edges and vertices. We perform these reconfigurations based on popular or dominant queries so that the number of vertices and edges traversed during evaluations is reduced.

Enabling support for temporal analysis also required modifications to our previously-developed graph functionality. When dealing with intervals, the graphs include two layers of vertices that represent start and end times. Edges are then used to link the two points in time, enabling both range queries and analysis through Allen’s interval algebra. This storage format also serves to avoid storing redundant data; each timestamp will only be stored once, with the links between vertices used to distinguish between unique intervals.

We have also incorporated support for sampling the metadata graphs and using these sampled representations to service queries. The first sampling method used in the system is uniform sampling, which preserves the distribution of feature values. While uniform sampling can provide insights over large quantities of information, it tends to underrepresent outliers or anomalous data points. To explore these corner cases, we provide support for *stratified sampling* and *path inspection*. Stratified sampling deliberately overrepresents rarely-occurring values (or outliers) to enable better representation of their characteristics, and path inspection involves highlighting paths through the graph that have not been traversed as often.

To support fast detection of queries that will not have matching results we use Bloom filters. Bloom filters are space-efficient with the property that there may be some false-positives but never false-negatives. Here we convert queries into orientation neutral paths, which we then use to query the Bloom filter.

2.3 End-to-End Query Process

Each node in the Galileo network is capable of servicing queries. When a query is submitted to the system, the receiving node will perform a feature graph lookup to determine which nodes may hold relevant data. If the query contains approximation operators (see Figure 1 for example queries), both the node proximity and estimated quantity of matching data is taken into account during the communication process; for queries with tight time bounds, local neighbors will be consulted for data first, with additional nodes contacted to increase the retrieval scope and ensure a more representative sample is acquired. In Galileo, nodes may be grouped by their physical location (rack, datacenter) so neighboring nodes will often be able to respond faster. Finally, results are combined to form a traversable *result graph*, which is transferred back to the client. The result graph includes a variety of statistics about the underlying data, such as means, variances, confidence intervals, etc. Information about the query coverage of the entire dataset is also provided.

2.4 Experimental Dataset and Test Environment

In this study, Galileo was populated with data provided by the National Oceanic and Atmospheric Administration (NOAA) North American Mesoscale Forecast System (NAM) [12]. The NAM involves frequent collection of atmospheric information in GRIB format, which contain several features, including the spatial location of the samples, the time they were recorded, percent relative humidity, surface temperature (Kelvin), wind speed (meters per second), and snow depth (meters). We sampled from the data provided by the NAM to create our test dataset, which consisted of one billion (1,000,000,000) files of approximately 8 KB each.

The benchmarks included in this paper were carried out on a 48-node cluster of HP DL160 servers equipped with a Xeon E5620 CPU, 12 GB of RAM, and a 15000-RPM disk. Galileo was run under the OpenJDK Java runtime version 1.7.0_60.

3 SYSTEM OVERVIEW

Galileo [4], [5] is a high-throughput distributed storage system. A primary goal of Galileo is incremental scalability, which led to its network design being modeled as a distributed hash table (DHT). Much like Apache Cassandra [6] or Amazon Dynamo [7], Galileo is a *zero-hop* (or *one-hop*) DHT; rather than forwarding requests through intermediary nodes in the network, files are sent directly to their final destination to reduce latency. Diverging from traditional DHTs such as Chord [13] or Pastry [14], the network design in Galileo is hierarchical and supports multiple levels of hash functions. Ultimately, these design decisions allow Galileo to provide functionality outside the scope of the standard DHT architecture.

The primary use case for Galileo is the storage and analysis of voluminous, multidimensional datasets commonly seen in the scientific domain. This focus enables a number of problem-specific optimizations, such as built-in support for spatial and temporal metadata. The dimensions (features) seen in these datasets are managed by the system to provide insights and enable expressive retrieval capabilities. Galileo can import or store data from a wide range of scientific data formats, such as NetCDF [15], HDF5 [16], or its own multidimensional array format.

To handle retrieval operations, Galileo employs a multi-tier indexing scheme. Individual *storage nodes* in the system maintain a *metadata graph* instance that indexes local files. On a global scale, the *feature graph* [5] is a complete, coarse-grained representation of all the information stored in the system. Traversal through the feature graph leads to specific storage nodes that will likely hold data relevant to a given query.

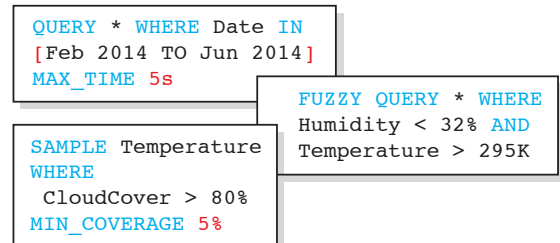


Fig. 1. Example queries supported by the approximate query processing engine in Galileo.

For two- or three- dimensional spatial information, bitmap-based *geoavailability grids* [17] record spatial regions that have relevant data points, allowing the system to evaluate queries constrained by polygon shapes. Both the feature graph and geoavailability grid are updated through a gossip protocol to quickly disseminate global state among nodes, and can provide dramatic reductions in search space over large, distributed clusters of computing resources. These indexing algorithms are integrated directly in Galileo, but are broadly applicable across a variety of distributed storage frameworks.

3.1 Partitioning

By supporting multiple indexing strategies, Galileo effectively decouples its storage and retrieval components, departing from the design of traditional DHTs. This allows hash functions to be changed at runtime to adjust storage properties without impacting retrieval operations, and also gives users an opportunity to develop their own hash functions that account for the unique characteristics of their particular datasets.

Galileo allows individual storage nodes to be placed into *groups* or *subgroups* of computing resources. In a two-level hierarchy, a hash function can be used to determine the group membership for a particular data point, followed by a second hash to determine the destination storage node within the group. In this particular study, group membership is assigned based on the spatial region the reading was produced from, combined with an SHA-1 hash of remaining feature values to determine the destination storage node. This configuration places data with spatial similarity in close proximity on the network while distributing load uniformly across the group.

Placement of incoming spatial data is facilitated by the Geohash algorithm [18]. Geohashes represent a hierarchy of successively more precise spatial bounding boxes, which are derived by interleaving bits obtained from two-dimensional coordinate pairs (e.g., latitude and longitude). Each additional bit in a Geohash increases precision by halving the spatial bounding box being represented, and bits are mapped to a Base32 character set to produce human-readable strings. Longer Geohash strings are more precise; for example, a Geohash of 9X references a bounding box in the western United States that includes Wyoming and parts of northern Colorado, while 9XJQ would roughly encompass Fort Collins, Colorado.

3.2 Metadata and Information Retrieval

For local query evaluation, each node in Galileo maintains a *metadata graph* instance. Metadata graphs contain feature information for all the files stored at a given node, and a traversal through the graph hierarchy incrementally narrows down the set of matching information stored on disk. Results from a Metadata graph traversal are returned in the form of a subgraph, which can be traversed, manipulated, or used to download raw data from the storage node. Exact-match and range-based queries are supported on string, numeric, spatial, or interval features stored in the graph, and can be described using a simplified dialect of SQL.

To reduce the search space of distributed queries, the *feature graph* contains a coarse-grained view of all the data available in the system. As new files are stored, the feature graph is updated in an eventually consistent manner through a gossip protocol. The granularity of the index can be changed at runtime, and places incoming samples into ranges called “tick marks.” This form of quantization helps make a globally-distributed

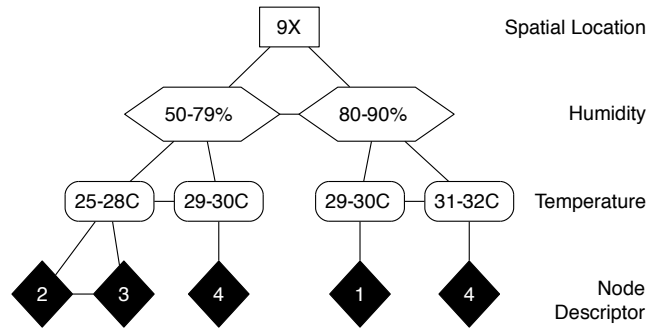


Fig. 2. A simplified feature graph showing humidity and temperature ranges for the spatial region represented by Geohash 9X. A traversal of the graph leads to storage node descriptors that hold file locations and statistics.

index possible by reducing its memory consumption and the size of updates that must be transferred over the network. The feature graph helps pinpoint storage nodes that will likely hold relevant information, reducing the amount of communication and processing required to perform a distributed query.

Additionally, the feature graph is a critical component in our approximate query processing engine. Several insights about the information in the system can be obtained from the feature graph: the availability of specific feature combinations in the dataset and their relationships, the number of files matching a particular set of characteristics, and how files are dispersed over the entire cluster. The feature graph is consulted as the first step in any distributed query, regardless of whether approximate or complete results are required. Figure 2 contains a simplified feature graph that illustrates its functionality; queries across a number of feature values will traverse through the graph based on the *tick marks* the values fall within, which are ultimately represented by vertices. On each level of the hierarchy, links are maintained between the vertices to allow scanning across values. The final vertex in a feature graph traversal points to a *node descriptor*, which collects a variety of information about the *path* that was traversed. In the feature graph shown in Figure 2, querying for:

Location=9X, Humidity=52-57%, Temperature=30C

leads to node descriptor 4, which would provide the locations of matching files in the network. Note that the exact tick mark range does not have to be specified in the query.

Both the Metadata and Feature graphs share some similarity with existing graph- or tree-based structures. For instance, the hierarchical organization of dimensions in a k-d tree [19] is somewhat similar to the Galileo indexes, but both the Metadata and Feature graph implementations allow for significant fan-out to occur at each level in the hierarchy instead of employing binary splitting. Additionally, the graphs share similarities with Tries [20], where traversals incrementally build upon a data point’s “prefix” by adding features as each dimension is explored. Our indexes are designed to be highly expressive and fast to traverse.

Different spatial locations often exhibit unique trends or patterns, which led to the development of bitmap-based *geoavailability grids* in Galileo. Geoavailability grids allow users to reduce the search space of distributed geographical queries by specifying polygons of interest with spatial coordinates.

To provide this functionality, Galileo divides the Earth into a hierarchy using Geohashes. The variable granularity of the bounding boxes provided by the Geohash algorithm allow geoavailability grids to capture a coarse-grained representation of the spatial areas covered by a given dataset, while still eliminating regions that do not contain relevant values. By generalizing spatial regions into bitmaps, queries can be evaluated quickly using bitwise operations. The evolution of a dataset over a spatial region often provides several valuable avenues for analysis, but can be limited by the speed of query evaluations when the data reaches extreme quantities or sizes. The features described herein provide a means for performing approximate spatial analysis, which could then be followed by rigorous analysis when phenomena of interest are discovered.

4 QUERY SPECIFICATION

Supporting exploratory or approximate queries requires a departure from explicitly-defined query parameters. In this work, we make the distinction between *approximate* (or exploratory) queries and traditional *rigorous queries* that involve exact results. To accommodate a wide range of use cases and promote expressivity, we designed a collection of operations to guide the implementation of our approximate query processing engine. These user-facing constructs influenced and guided our implementation. They include time-, error-, quantity-based constraints, temporal logic expressed through interval algebra, and fuzzy queries that enable parameters to be automatically relaxed to include more results. These operators allow fine-grained control over the scope of each query, and are designed to be combined with the standard constructs present in languages such as SQL or existing features in Galileo such as range-based and exact-match queries.

4.1 Time and Error Constraints

Several problem domains require performing computations under time constraints. For example, visualization tools may need to render a scene within a few hundred milliseconds to produce smooth animations, or a busy website could be required to service requests within a particular time frame to avoid violating service-level agreements. In these situations, results are requested using the time-bound query operator, which will provide a response within a specified amount of time, along with statistics on the accuracy of the results. When dealing with time bounds, queries are scheduled dynamically to avoid starvation in longer-running processes. In the case of an extremely busy storage node, the depth or breadth of time-bound queries is scaled back to meet constraints.

Conversely, there are scenarios that require information to be retrieved as quickly as possible, but only if the results reach a particular level of accuracy defined by error bounds. One example of this usage pattern is buying and selling commodities on a short-term basis, which could require rapidly monitoring pricing trends and taking action only when the error margin falls below an acceptable threshold. When predicting weather patterns, certain calculations may require timely results, but are only useful if a reasonable accuracy is achieved. Support for error-based constraints also implicitly includes functionality for requesting subsets of the overall dataset, which could involve returning a specific number of results or a percentage of the total items that meet query directives.

Time- and error-bound operators can also be combined; if certain conditions hold true, an application may instruct

the server to favor one approach over the other, effectively adapting its requirements based on changing environmental conditions. This functionality is highly relevant in cases where an operation would introduce too much error, requiring the use of an alternate approach. For fine-grained control, users can also specify the priorities for each operator being used.

4.2 Temporal Reasoning

Datasets consisting of time series information often involve several data points that span across uniform or non-uniform time intervals. Significant insight can be gained by inspecting the time at which a sample was taken; for instance, temperatures fluctuate over the course of a day and may influence other environmental conditions. In these cases, an average of all observed values can mask significant patterns in the underlying dataset. This makes accounting for time in a high-resolution fashion critical in many areas of analysis.

For reasoning about the relationships between disparate time intervals in a dataset, we integrated support for the operators described by Allen’s Interval Algebra [10], [11] into our query language. Allen’s Interval Algebra is an expressive calculus for temporal reasoning that has seen wide application in areas from artificial intelligence [21] to temporal database management systems [22]. Figure 3 provides an overview of the relationships that can be represented by these operators, which make inexact comparisons possible across temporal events. Usage examples include querying for data points that overlap, lie outside a specific range, or happen during a common time period. If the exact time that an event occurred is not known, these query operators complement the other approximation functionality in the system; for example, a user may query for all intervals that ended at a specific time, effectively creating a sequential list of event start times leading up to a particular circumstance of interest.

Efficient evaluation of queries on intervals, temporal or otherwise, requires problem-specific indexing functionality. For instance, an exact-match query between the start and end times of an interval should return a result even if a physical index record is not present at the requested time. Simply “filling” all possible units of time between the start and end of an interval needlessly consumes memory and disk space. Furthermore, the resolution of a point in time should not have to be reduced

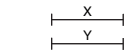
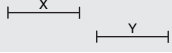
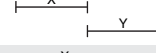
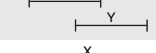
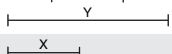
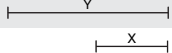
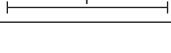
Temporal Operator	Relationship
X equals Y	
X before Y	
X meets Y	
X overlaps Y	
X during Y	
X starts Y	
X finishes Y	

Fig. 3. Temporal operators from Allen’s Interval Algebra. These operators enable queries and analysis across intervals from time series datasets.

to fit into a predefined minimum unit. Our indexing strategy for intervals is described in depth in Section 5. Each temporal record is represented by high-resolution start and end pairs, which can be compared efficiently and converted directly to user-friendly time units (such as minutes, hours, or days).

These temporal operators have been integrated into our indexes and can be combined with any of the other query constructs available. They obviate the need for involved manual definition of temporal comparisons by the end user, and also allow the relationships between time bounds to be modeled in the result graphs that are returned by a query.

4.3 Fuzzy Queries

In situations where the boundaries of a range query are unknown or approximate, fuzzy queries allow selective relaxation of constraints to produce more results. One example use case for fuzzy queries is hotel room booking; a user may wish to search for rooms with desired attributes across the span of their vacation, but if no results are found then the closest matches available can still be returned without having to submit another query. This usage pattern of continuous refinement has been observed in several domains [1], [2]. For datasets that include a high number of dimensions, query parameters across a broad range of features may be too selective. In these cases, the user can specify features to be expanded, or opt for retrieving results that require the smallest percent change in constraints.

The default operating mode for a fuzzy query attempts to find the closest matching values to the query parameters provided. For example, if a fuzzy query requests all records with humidity values of 50% but none exist, records with readings of 48% may be returned instead. Of course, if exact matches can be achieved, then the results returned by the fuzzy query will be no different than that of a rigorous equivalent. However, in cases where an exact match is not available, the nearest neighboring n nodes in the graph can be located in $O(\log n)$ steps (similar to the performance of a binary search). Both the higher and lower values can be returned by the fuzzy query, along with their differences from the original value. Additional vertices can be returned to the user as well; for example, a specific use case may require the nearest five matches to be included in the results. Fuzzy queries can drastically reduce the number of repetitive exploratory searches that a user may need to make before settling on a final set of query parameters.

4.4 Query Promotion

While approximate queries can provide useful results in a fraction of the time a rigorous query would take, there are situations where exploratory results may need to be confirmed later with a rigorous query. In this case, *query promotion* allows the scope of a query to be expanded to include the entire dataset as long as particular user-defined constraints are met. Queries can be extended based on response times, error margins, or when the set of matching information meets specific criteria, such as the result count or presence of a particular relationship between features. To keep extended queries current, time bounds can also be updated to include new information that was recently stored. In some cases, query promotion can completely replace long-running exploratory computations; several approximate result graphs can be returned during a query to track progress, followed by a complete result graph once the conditions for promotion are met.

5 QUERY EVALUATION FRAMEWORK

In this section, we describe our framework and algorithms for approximate query evaluation. Given the data volumes and concurrent requests involved, our primary objective is to ensure that query evaluations are fast and high-throughput. This is achieved by adapting indexes autonomously at runtime to improve performance, optimizing the representation of temporal intervals when stored in a graph-based structure, embedded low-latency access to summary statistics, and online sampling support.

5.1 Dynamic Feature Graph Dimensionality

To compensate for the inevitable imbalances in the distribution of feature values in a dataset, tick marks in the feature graph are dynamically adjusted at runtime rather than relying on preconfigured ranges [23]. This optimization helps improve the overall accuracy of traversals, thereby reducing the number of storage nodes that must be contacted during a distributed query. Approximate queries also benefit from this property; regularly-accessed files and readings with frequently-observed values are given priority by this algorithm, which improves the precision of the information that can be gleaned by performing a feature graph query.

Tick mark granularities are modified based on two metrics: the distribution of feature values in new files, and observed trends from previous queries. Both of these events are considered *hits* that impact the load characteristics of individual vertices. If incoming samples are biased towards a specific range of values grouped into a single vertex in the graph, the range will be split into separate vertices to distribute load as uniformly as possible. Figure 4 illustrates how tick mark ranges are split in the feature graph.

Feature graph reconfiguration is facilitated through a *dispersion function* that calculates the imbalances in load across vertices. In this work, we use the *coefficient of variation (CV)* as our dispersion function, which is defined as the ratio of the standard deviation to the mean, $CV = \sigma/\mu$. Relatively high values of CV indicate high dispersion. When the CV

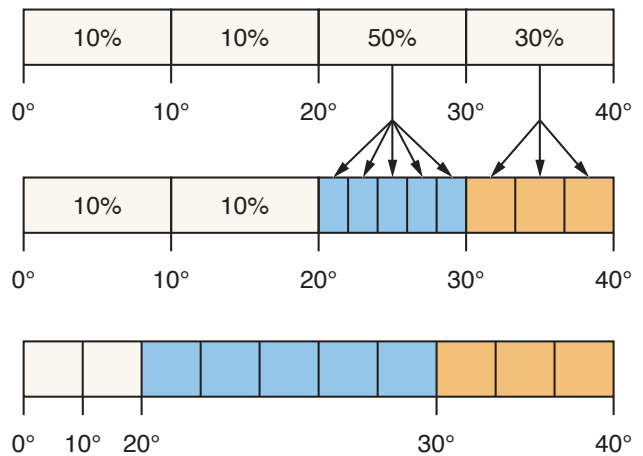


Fig. 4. Dynamic index reconfiguration on vertices holding temperature values (in °C). In a graph with four vertices, two receive 50% and 30% of the overall storage and retrieval requests. After reconfiguration, load is balanced evenly across 10 vertices.

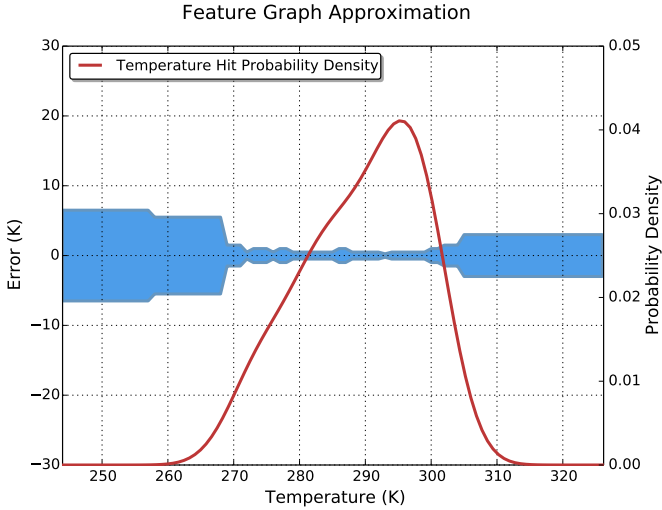


Fig. 5. Error bounds on approximate results returned by the feature graph, with the probability density of hits on temperature values (Kelvin) in the dataset.

across a level in the graph hierarchy cannot be substantially decreased by our splitting algorithm, the process stops until more data is collected. In general, this process completes in less than a millisecond and can be managed with a configurable upper bound on iterations. The algorithm executed in reverse, called *vertex reclamation*, merges lesser-used vertices with their neighbors, decreasing the overall size of the graph and keeping growth in check.

While modifying the ranges of tick marks in the feature graph improves performance, it also ensures that the ranges receiving a larger share of storage and retrieval requests will provide more accurate results for approximate queries. Figure 5 demonstrates the error bounds for queries against temperature values along with the observed probability density of hits across the values. For the majority of queries, the feature graph can provide results within ± 1 Kelvin.

Feature graph queries are fast because they require clients to only contact a single node in the system and report synopses rather than complete results. In our test configuration, a feature graph query can take as little as 3.2 ms on average to report a negative result (zero matching files). Table 1 provides latency statistics across a broad range of randomized queries evaluated by the feature graph, and also shows the amount of time rigorous queries would take (which were extended from approximate queries). Note that while a rigorous query must include a graph vertex for every result, approximate queries describe a range of values and their respective densities in a single vertex, resulting in much faster response times.

Although approximate queries are considerably faster on average, there are situations where a fuzzy query could continually expand its scope due to the sparsity of the feature space. In such a scenario where no time bounds are provided, the query evaluation could continue to incrementally expand its scope, resulting in longer response times. To cope with this scenario, especially in the face of heavy load from other clients, the system limits fuzzy queries to a maximum time span that can be configured based on the particular deployment. For example, a high-traffic website may impose a tighter limit than a deployment being used primarily for internal analysis.

TABLE 1
Approximate and rigorous query timing results, averaged over 1000 iterations.

Results	Approximate Query (ms)		Rigorous Query (ms)	
	μ	σ	μ	σ
0	3.20	0.50	7.65	2.41
25,000	4.59	1.22	139.31	3.84
50,000	6.92	1.36	264.62	2.92
100,000	13.12	3.90	521.01	4.80
200,000	18.77	2.39	922.20	10.36

5.2 Modeling Temporal Intervals

While singular points in time can be expressed easily through a graph hierarchy that accounts for various time units (years, months, days, and so on), a timespan represented by a start time and end time requires a data structure that can handle two-dimensional boundaries. Additionally, a graph hierarchy based on time units requires the insertion of a large amount of vertices and edges for individual units of time, resulting in increased memory usage and slower traversals. Rather than addressing the hierarchical nature of time representations directly, we chose to effectively flatten these intervals into two entries in the graph hierarchy: one level containing start times, and a second level containing end times. For data points that consist of a single time value, called *timestamps*, the start and end times of the interval are equal and only occupy a single vertex in the graph. Figure 6 illustrates how time series data is incorporated into the feature graph and metadata graphs in Galileo. While many time series datasets generated by observational devices commonly deal with uniform intervals, this approach supports non-uniform time spans as well.

By integrating time series information directly into the graph hierarchy, queries can be simplified to basic numeric relationships. For example, a query to retrieve samples that fall within a timespan represented by (T_{start}, T_{end}) would eliminate all vertices in the graph with time intervals that start after T_{end} and end before T_{start} . As with any other features stored in the graph, vertices maintain links to neighboring values in the

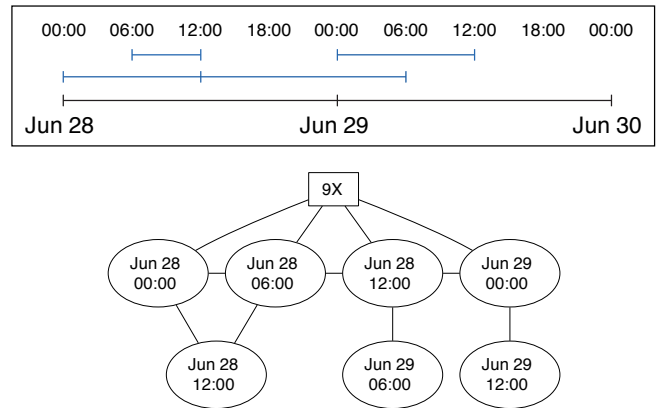


Fig. 6. Time intervals from samples in a spatial region (represented by the Geohash 9X) across a time frame of two days, along with its graph-based equivalent for indexing purposes. Further descendant feature vertices are excluded for brevity.

hierarchy to facilitate operations that scan across available time values.

Queries across time intervals are powerful, but some applications require non-contiguous queries, such as “retrieve all samples gathered on the fifth day of each month in the year 2013.” To support this particular use case, time-based queries can also be defined using a date hierarchy along with wild cards, such as $\text{Year}=2013, \text{Month}=\ast, \text{Day}=5$. In this scenario, the query is transformed into a set of non-contiguous intervals before being evaluated by the index in parallel.

Unlike most feature types, time is unique because new vertices must be continually added to the graph structures as data is generated. To manage the creation of vertices as they accumulate over time, we rely on our dynamic graph reconfiguration algorithm with time-specific directives for collapsing older or rarely-accessed intervals. Without specific accommodations for time, feature graph tick marks may be autonomously expanded into awkward increments, e.g., intervals of 63 minutes. Unless otherwise configured, the system favors commonly-used fractions of time (for example, values evenly divisible by five) when tuning the graph to produce divisions between vertices.

5.3 Graph-Embedded Statistics

To supplement value-based queries, node descriptors in the feature graph contain several statistics. This information is useful in a variety of situations, including those that deal with data points in lower-resolution tick mark ranges. For each path through the graph, node descriptors include:

- A count of matching items (density of values)
- Minimum and maximum values of each feature type
- Mean, variance, and standard deviation of the values

We use the method given by Welford [24] to avoid recalculating the mean, variance, and standard deviations of feature values as new files are continuously added to the system. This is made possible by maintaining a count of samples, n , along with the previous mean of the values and the sum of squares of differences from the current mean, S_n :

$$S_n = S_{n-1} + (x_n - \bar{x}_{n-1})(x_n - \bar{x})$$

Given these items, the population variance is calculated with $\sigma^2 = S_n/n$. This optimization eases the burden placed on individual storage nodes in providing up-to-date information in a timely manner, and helps reduce the amount of updates that must be gossiped through the system to reach a consistent state. If a user has requested specific feature values in their query, these statistics can generate confidence intervals for the approximate results returned by the feature graph, or could estimate the insights a complete query would provide.

5.4 Hierarchical Reorientation

In a consistent system, information stored in the feature graph at each node will be identical. However, it is often advantageous to represent the data in different ways by *reorienting* the levels in the graph hierarchy to better suit the unique workload trends at each node. Reorientation impacts both the traversal times of incoming queries and the overall memory consumption of the graph.

A reorientation involves expanding the current graph state to orientation-neutral paths and then determining where vertices belong in the new hierarchy. Given a collection of paths, the

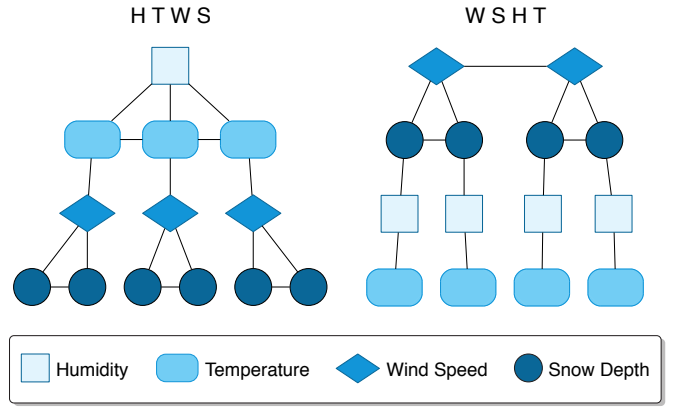


Fig. 7. Two example graph orientations of the same underlying data, with four feature types. Graph orientations are named based on the first initial of the features in hierarchical order.

maximum number of vertices and edges at hierarchy level n can be calculated in advance using the overall set of feature values for the level, F_n , with the following recurrence relations:

$$\begin{aligned} V_n &= V_{n-1} + F_n F_{n-1} \\ E_n &= E_{n-1} + 2F_n F_{n-1} - F_{n-1} \\ F_0 &= 1, V_0 = 1, E_0 = 0 \end{aligned}$$

Figure 7 illustrates two different graph orientations, along with the naming conventions for graph configurations used in this study. The graph on the left includes *Humidity*, *Temperature*, *Wind* speed, and *Snow* depth, which is shortened to *HTWS* for brevity. Note that while the same dataset was used to generate both example orientations, the vertex and edge counts are different for both representations.

Table 2 highlights the differences in creation time, vertex count, and edge count between three different graph orientations. The graph can be reoriented quickly to react to changing trends and usage patterns, which is crucial in providing low-latency results for approximate queries. In this test, creation time includes generating orientation-neutral paths and moving vertices into place within the specified hierarchy.

TABLE 2
Feature graph sizes and creation times with different orientations.

Orientation	Creation Time (ms)		Vertices	Edges
	μ	σ		
HTWS	58.80	3.41	54,582	84,423
WTSHT	49.99	2.81	31,706	61,547
SWHT	50.97	3.43	33,283	63,124

Figure 8 illustrates how traversal times vary across different orientations; while the SWHT configuration provides the best overall performance, it may not be suitable in situations where humidity values are frequently accessed (in which case, the HTWS configuration should be selected instead). Each graph orientation has its own set of trade-offs that must be managed

Graph Orientation Query Times

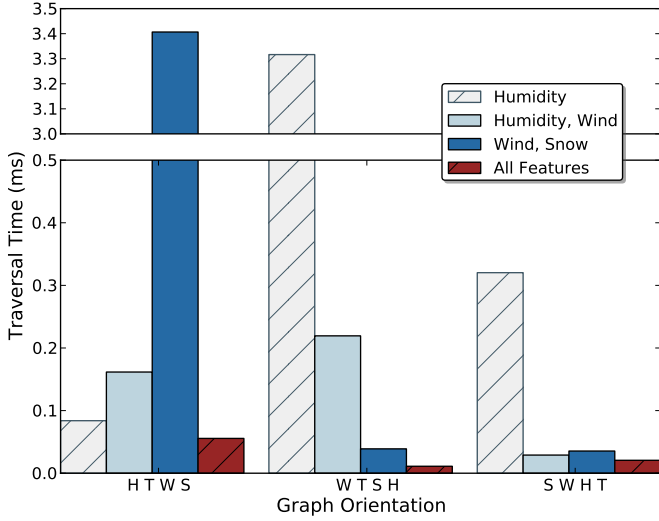


Fig. 8. Traversal times for random queries across three different feature graph orientations. Queries ranged from broad (one value specified) to explicit (all feature values specified). Test permutations were executed 1000 times.

to achieve ideal performance, and these trade-offs continually change over time.

5.5 Path Prediction Using Bloom Filters

For situations where a query would not return any results, we developed an optimization to enable short-circuit evaluation of requests without traversing the graph called *path prediction*. This functionality is made possible by generalizing incoming queries across vertex ranges into orientation-neutral paths, and then testing to determine if the path is present in the graph using a Bloom filter [25]. Bloom filters are a space-efficient way to determine whether an element is present in a set. The filter may produce false positives, but *never* false negatives, making it an excellent way to resolve queries that do not match any vertices.

Path prediction involves complete graph paths rather than individual elements and is also applied on vertex ranges rather than discrete values. This allows the filter to be relatively small while maintaining accuracy. The size of the filter and number of hash functions used is influenced by the amount of vertices in the feature graph, and is resized or reconfigured during dynamic dimensionality updates to ensure acceptable performance. Path metadata is sorted and structured such that two paths containing the same feature values in a different order will be processed identically by the bloom filter.

To enable fast evaluation using path prediction, we devised a serialization format that allows the raw path data to be fed to the bloom filter directly. Since the number of hash functions stays fixed, as well as the raw path data being of similar sizes, the time consumed checking for a non-matching path stays very consistent as long as the number of dimensions being indexed does not change rapidly. In our test environment, a path prediction could be performed in as little as 0.01ms, with a standard deviation of 0.0001.

5.6 Non-Blocking I/O and Concurrent Workloads

Galileo was designed around a non-blocking event-based architecture, a popular approach for handling substantial amounts of concurrent requests. Many of the servers addressing the c10k problem [26] use non-blocking I/O to support a large number of simultaneous clients. In this class of problem, the network interface becomes the bottleneck rather than the CPU: when each client requests a 4 KB file every second, a Galileo storage node can manage around 32,000 concurrent clients with a single gigabit interface, and upwards of 64,000 clients on dual gigabit interfaces using link aggregation. While web-centric server benchmarks like c10k are useful in measuring the scalability of a server, they do not fully capture the usage profile Galileo is intended for. In many cases, thousands of files will be requested in a single query, and approximation features generally trade higher CPU utilization for fewer disk accesses and reduced latency.

To evaluate the scalability of our approximate query processing engine, we designed a set of benchmarks that were inspired by the c10k problem. In these tests, a steadily-increasing number of clients connected to the node and began submitting randomized queries every second. The requests contained combinations of approximate query types, which consisted of:

- 55% Range queries
- 20% Fuzzy queries
- 20% Exact-match queries
- 5% Inequality queries

Each of the aforementioned queries stress the system differently. Inequality queries exclude only a single record from a given feature type, so their use was limited to avoid scenarios where almost all the data on the storage node was retrieved. On the other hand, range queries are the second most computationally intensive query type, and were selected frequently in our tests to help illustrate real-world performance under high load situations. Finally, the entire dataset was inspected to ensure the randomized ranges being used were reasonable for each feature type; relative humidities were limited to the range of 0-100%, temperatures did not exceed 60° C, etc. Results were streamed back to the client as metadata graphs.

To stress the system under varying usage conditions, we also conducted tests that interleaved a configurable amount of storage and query operations. Figure 9 shows the results for the baseline query-only workload, along with benchmarks that were composed of 50% and 75% storage operations. The number of queries per second submitted by each client was held constant across the three tests; i.e., the 50% query, 50% storage benchmark saw each node processing about 4,000 client requests per second at its maximum throughput due to the additional storage operations. A high level of concurrency was achieved in the storage tests as threads blocked on I/O could be interleaved with those doing query processing operations: for instance, the 25% query, 75% storage workload increased total requests from the baseline by about 70% while query throughput decreased by only 15%. For the three test cases discussed, the system reached its peak query throughput at 2,170, 1,980, and 1,820 concurrent clients per server, respectively.

Table 3 reports the minimum, maximum, and average response times observed by client applications while our 48-node cluster was managing 50,000 concurrent connections. Although the random nature of the queries being evaluated caused response times to fluctuate, an average of the results indicated a slight upward trend in latency as more storage requests were

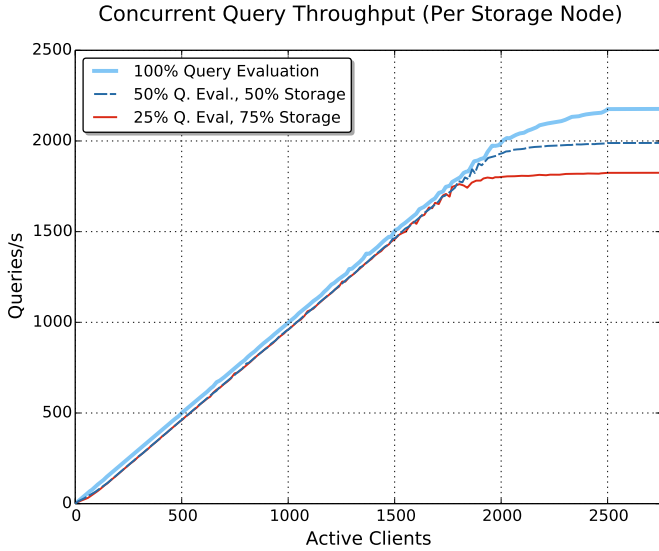


Fig. 9. Query evaluation performance at a single storage node as additional clients connect. Each client sends one query per second, with a configurable amount of storage requests interleaved.

made. This outcome matched our observations on the server side, and also underscores our emphasis on throughput. While certain queries can consume a significant amount of processing time, the amount of overall requests evaluated per second on each storage node remains fairly predictable over time.

TABLE 3

Response times observed during concurrent workload tests (averaged over results observed by the first 50,000 clients).

Workload	Response Time (ms)		
	Min	Max	Mean
100% Query Evaluation	0.42	35.11	4.82
50% Q. Eval., 50% Storage	0.47	91.63	6.21
25% Q. Eval., 75% Storage	0.52	184.22	9.87

5.7 Dataset Sampling

When a subset of available information will provide a representative model of the complete dataset, a *sample query* on the metadata graph can be evaluated to provide results in a fraction of the time a rigorous query would take. Each storage node can produce sample graphs on demand, or can pre-populate (and update) several sample sizes that cover varying portions of the dataset. The amount of pre-populated sample graphs maintained is generally problem- and hardware-specific, where the trade-off space involves both the memory consumed and query evaluation times. Galileo can optionally monitor sampling trends and automatically create pre-populated sample graphs for commonly-requested sample sizes if memory and processing resources are available.

The results generated by a sample query can be represented as raw data points, a traversable metadata graph, or as a probability density function (PDF) generated through kernel density estimation. Sample queries can be combined with the

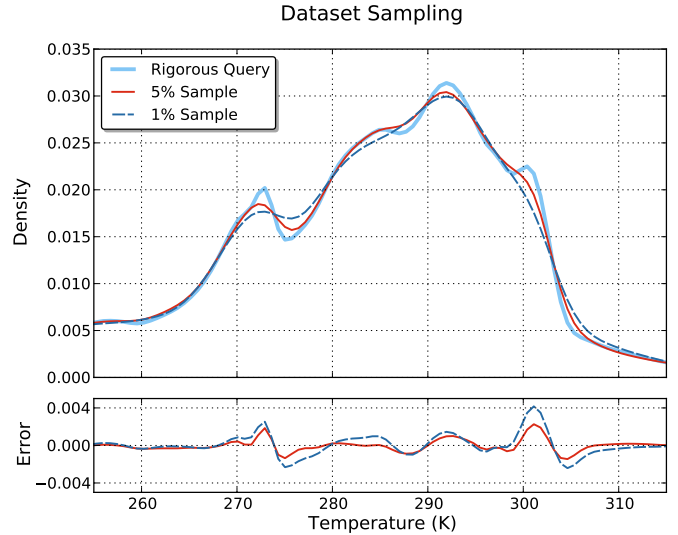


Fig. 10. Comparison between the density of values in 1%, 5%, and rigorous queries, along with the error associated with both sample sizes.

other query constructs supported by Galileo. For instance, all features could be sampled across a particular time interval, or the probability density of a specific feature could be requested when a set of environmental conditions are met.

Figure 10 compares 1% and 5% uniform samples (represented by their probability density estimation) with a rigorous query across a particular range of temperature values. In general, sampling produces a fairly representative model of our particular dataset. Note how the sharp peaks in the curve for the rigorous query correspond with higher observed sampling error; in these cases, the system can use both *stratified sampling* and *path inspection* to help reduce the margin of error.

When using stratified sampling, homogeneous subpopulations in the dataset are selected and sampled from independently using either random sampling or systematic sampling. Contrasting with a uniform approach, this method helps ensure that the resulting samples are highly representative of the underlying subpopulations, even if some of the data points could be considered outliers. In situations where a sampled population is highly diverse, or the population varies in density based on region, stratified sampling will often produce a result with less sampling error. However, this approach is only feasible when it is possible to generate subpopulations of the overall dataset.

Path inspection locates paths through the metadata and feature graphs that are accessed more or less frequently than usual. Similar to stratified sampling, path inspection can be used to boost the importance of outliers, but is performed by empirically measuring the read and write operations conducted on each vertex in the graphs. Data points found with this approach help pinpoint anomalies or areas of the dataset that could warrant further analysis, and can be discovered through the metadata generated by both storage and query requests. To avoid skewing the results over time, path inspection does not modify the access counters at the vertices, and both the read and write counters can be used independently, if desired. This functionality enables analysis of not only the dataset itself, but users of the system as well.

6 APPLICABILITY TO OTHER STORAGE FRAMEWORKS

To test the algorithms used by our approximate query processing engine in the context of another system, we adapted several of the constructs outlined in this work to MongoDB, version 2.6.1 [8]. MongoDB is a document store that supports clustering along with incremental scalability through horizontal partitioning. Similar to Galileo, MongoDB can evaluate range queries and launch MapReduce computations on its datasets, but was designed around BSON, a binary storage format similar to JSON. MongoDB can utilize the Geohash algorithm for its spatial indexing functionality, with general indexes backed by a B-tree data structure for fast lookup operations. The system also natively supports imposing time limits on individual queries, and can produce summary statistics using server-side aggregation.

While MongoDB imposes some hard limits (especially in a clustered environment) that make managing voluminous datasets challenging, we were able to translate most of the functionality supported by our approximate feature graphs to MongoDB. The cluster we designed acted as a distributed approximate index with links to a secondary storage tier responsible for fulfilling any rigorous, full-resolution queries. We implemented the second tier using both Galileo storage nodes and standalone MongoDB instances, ensuring the two tiers were loosely coupled and our approximate index could be adapted to other systems in the future.

Our dataset was converted from NetCDF files to JSON documents before being streamed to the MongoDB-backed index. This conversion increased the size of incoming records by about 25%. MongoDB does not support pre-storage processing hooks or trigger functionality seen in many relational databases, so records were stored immediately in a collection set aside for staging new data. After initial storage, the *oplog* was used in combination with a *tailable cursor* to trigger post-storage processing, which includes updating statistics about the dataset and performing binning based on the dynamic tick marks discussed in Section 5.1.

Table 4 contains latency statistics for a range of randomized queries evaluated with our feature graph adaptation in MongoDB. Compared to Galileo, MongoDB excels when handling smaller queries, but tends to require more time to process large requests. One factor that influences MongoDB evaluation times is overhead from the increased duplication of information required to produce BSON documents, which are complete, stand-alone representations of each data point. With minor adjustments, this document-oriented implementation of the feature graph could be employed in tabular storage systems as well.

TABLE 4
Timing results for approximate queries evaluated with MongoDB, averaged over 1,000 iterations.

Results	Evaluation Time (ms)	
	μ	σ
0	2.16	0.20
25,000	4.15	0.91
50,000	8.75	1.38
100,000	17.93	2.21
200,000	34.56	4.22

7 PERFORMANCE IN VIRTUALIZED SETTINGS

The overhead associated with virtualized I/O has traditionally made bare metal the ideal platform for high-throughput storage systems; for instance, Amazon Dynamo [7], Facebook Cassandra [6], and Google BigTable [27] all were designed and tested on bare metal. However, recent advancements in virtualization technology and the ubiquity of cloud infrastructure have challenged conventional wisdom. To evaluate the performance of our algorithms in a virtualized environment, we deployed VMs on the same hardware used for our bare metal tests. We ran the VMs on KVM (Linux 3.13), allocated one CPU core to each VM, and created a 48-node VM deployment to perform a direct comparison between the VM and bare metal installations. Table 5 provides latency information for randomized queries evaluated against the feature and metadata graphs in a virtualized environment, repeating the tests performed in Section 5.1.

TABLE 5
Approximate query timing results on virtual infrastructure (48 VMs), averaged over 1000 iterations.

Results	Approximate Query (ms)		Rigorous Query (ms)	
	μ	σ	μ	σ
0	3.28	0.53	7.75	2.81
25,000	4.73	1.47	140.61	3.03
50,000	7.13	1.62	272.01	3.75
100,000	13.39	2.47	526.97	4.91
200,000	18.92	4.17	947.65	11.66

While the overhead in this test was about 2% on average, we observed that variability was the main cause in the timing differences, with the VMs frequently performing as well as the bare metal hosts but occasionally exhibiting increased latency. Overall, these tests confirm that VMs are an effective solution for evaluating approximate queries, while also providing several distinct advantages, such as lower cost and greater elasticity. We were also able to run this benchmark on a 192-VM cluster with similar results (2.4% average overhead).

Persisting data to disk in an efficient manner is also a major concern for our system, so we evaluated several disk formats (with both pre-allocated and dynamically-resizing images, if applicable) in Table 6. The formats tested include raw block images, the *qemu qcow2* format, and the *VirtFS* file system passthrough described by Jujjuri et al. [28], which allows the guest VM to mount a host file system directly using the 9P2000.L protocol. Bare metal disk performance and Amazon EC2 results are included for reference.

While this test does illustrate some overhead in virtualized disk performance, the results provided by the raw block images and *qcow2* format were both reasonably competitive with bare metal. Dynamic images carry slightly less file system read overhead, likely due to the significantly smaller on-disk sizes when the images are not used to capacity. For our particular use case, raw images with dynamic resizing provide the best blend of performance and flexibility, since host storage space will not be occupied until it is actually used by the VM. Although the passthrough results were somewhat surprising, we suspect that the large number of metadata operations performed by Galileo may impact performance.

TABLE 6

Disk format performance results for representative Galileo workloads on a 48-VM cluster, averaged over 1000 iterations. Amazon EC2 instances included for reference.

Benchmark Type	Read (MB/s)	Write (MB/s)
Bare Metal	233.8	194.7
raw (pre-allocated)	227.0	191.4
raw (dynamic size)	228.7	191.1
qcow2 (pre-allocated)	182.4	182.1
qcow2 (dynamic size)	185.4	179.7
File System Passthrough	33.0	22.8
EC2 m1.xlarge	157.7	71.1
EC2 i2.xlarge (with SSD)	448.5	516.3

8 RELATED WORK

Cassandra [6] is similar to Galileo in its network layout and storage capabilities, as both systems are designed around the DHT paradigm. Cassandra’s primary use case is the high-throughput management of tabular, multidimensional information. The system allows users to create their own partitioning schemes, but the partitioning algorithm used directly affects information retrieval as well; for instance, using the random data partitioner backed by a hash algorithm does not allow range queries or future reconfiguration of the partitioning algorithm. Cassandra scales out linearly as more hardware is added, and supports MapReduce computations.

Apache Hive [29] is a data warehouse that runs on the Hadoop [30] and HDFS [31] platform. As an analysis platform, it is capable of a wide range of functionality, including summarizing datasets and performing queries. Unlike Galileo and a number of other storage frameworks, the system is intended for batch use rather than online transaction processing (OLTP). In Hive, users can perform analysis using the HiveQL query language, which transforms SQL-like statements into MapReduce jobs that are executed across a number of machines in a Hadoop cluster. The *Metastore*, a system catalog, provides an avenue for storing pre-computed information about the data stored in the system. Hive emphasizes scalability and flexibility in its processing rather than focusing on low latency.

A considerable amount of research has been conducted on supporting query types beyond the standard *get* and *put* operations of DHTs. For instance, Gao and Steenkiste [32] maps a logical, sorted tree containing data points to physical nodes, enabling range queries. Chawathe et. al [33] outlines a layered architecture for DHTs wherein advanced query support is provided by a separate layer that ultimately decomposes the queries into *get* and *put* operations, decoupling the query processing engine from the underlying storage framework.

Popivanov and Miller [34] explores the issues surrounding managing and performing similarity searches over large quantities of time series information. To effectively summarize large datasets, this approach employs several wavelets that can outperform the commonly-used Haar wavelet and accurately estimates values for a wide range of data types. However, methods that rely on wavelets or synopses are generally very problem- or dataset-specific and can limit arbitrary queries.

BlinkDB [35] is an approximate query processing (AQP) system that extends the functionality found in the Hive query engine to support responsive approximate queries. The system

performs sampling at two scales: a broad, random sample of the dataset, along with focused sampling over frequently-accessed items. This hybrid approach requires much less information to be read from disk to compute query responses. Like Galileo, BlinkDB queries can specify time and error bounds.

FastRAQ [3] considers both the storage and retrieval aspects associated with range-aggregate queries. To manage the error bounds of these approximate queries, partitioning is based on stratified sampling: a threshold is used to control the maximum relative error for each segment of the dataset. Like Galileo, data is assigned hierarchically to groups and then physical nodes. Queries are resolved using adaptive summary statistics that are built dynamically based on the distributions of the data.

The Approximate QUery Answering System (AQUA) [36] intends to provide estimated responses to queries by avoiding direct access to the data itself. The system collects *synopsis data* in a number of ways: observing new information as it arrives, periodically inspecting the underlying data warehouse, or directly contacting the data warehouse during a query. AQUA returns its query results alongside an accuracy measure (such as a confidence interval), and can support *continuous reporting*, wherein more results can be streamed to a client as accuracy increases. Unlike Galileo, AQUA does not support time bounds or target error rates in queries. It is also designed for batch processing rather than online transaction processing, and cannot respond to real-time changes in the dataset.

9 CONCLUSIONS AND FUTURE WORK

As data volumes continue to grow, tracking the continually-evolving feature space can prove to be challenging. This is especially true in the case of time-series datasets. In this work, we considered the problem of approximate query support with a geospatial component in addition to a number of arbitrary features. Efficient explorations over time-series datasets can be supported by (1) incorporating query constructs that allow non-contiguous time intervals to be easily specified; (2) assisting in the discovery of unusual feature range combinations that have not been explored; and (3) by biasing dataset representations where feature value outliers are overrepresented.

The trade-off between accuracy and timeliness of query evaluations can be achieved by automatically controlling the degree of query specificity and dataset coverage during evaluations. Disk accesses, which can significantly delay query evaluations, are minimized by keeping compacted versions of the metadata graphs in memory and performing evaluations over their contents. The use of Bloom filters allows us to quickly identify cases where measurements are not available for a portion of the feature space. We target interference between queries, and the concomitant timeliness issue, by reducing the fidelity of results under conditions of high load.

The graphs maintained in Galileo are kept up-to-date with changing trends in values as new information arrives. To cope with these changes, we apply *vertex reclamation*, which collapses vertices into ranges that reflect the density and distribution of values. By maintaining graph statistics in an online fashion, we can continually update summary statistics for feature values as data is assimilated.

Given the data volumes described in this work, schemes that require constant user intervention quickly become untenable. Our strategies for metadata compaction, data assimilation, outlier biasing, discovery of unexpressed feature combinations, updates to summary statistics, and fast detection of unavailable data are all performed without user intervention.

Our future work will focus on support for probabilistic query evaluations. This would allow a user to explore questions such as, "What is the probability that relative humidity levels are greater than 50% during the Summer in Fort Collins, Colorado?" Supporting such queries could lay the foundation for construction of Bayesian Models and causality analysis.

ACKNOWLEDGMENTS

This research has been supported by funding from the US Department of Homeland Security's Long Range program (HSHQDC-13-C-B0018) and the US National Science Foundation's Computer Systems Research Program (CNS-1253908).

REFERENCES

- [1] B. Véléz, R. Weiss, M. A. Sheldon, and D. K. Gifford, "Fast and effective query refinement," in *ACM SIGIR Forum*, vol. 31, no. SI. ACM, 1997, pp. 6–15.
- [2] J. C. Shafer and R. Agrawal, "Continuous querying in database-centric web applications," *Computer Networks*, vol. 33, no. 1, pp. 519–531, 2000.
- [3] X. Yun, G. Wu, G. Zhang, K. Li, and S. Wang, "Fastraq: A fast approach to range-aggregate queries in big data environments," *Cloud Computing, IEEE Transactions on*, 2014.
- [4] 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*, vol. 29, no. 4, pp. 1049–1061, 2013.
- [5] —, "Expressive query support for multidimensional data in distributed hash tables," in *Utility and Cloud Computing (UCC), 2012 Fifth IEEE International Conference on*, Nov. 2012.
- [6] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.
- [7] D. Hastorun *et al.*, "Dynamo: Amazon's highly available key-value store," in *SOSP*. Citeseer, 2007.
- [8] MongoDB Developers, "MongoDB," <http://www.mongodb.org/>.
- [9] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
- [10] J. F. Allen, "An interval-based representation of temporal knowledge," in *IJCAI*, vol. 81, 1981, pp. 221–226.
- [11] —, "Maintaining knowledge about temporal intervals," *Communications of the ACM*, vol. 26, no. 11, pp. 832–843, 1983.
- [12] National Oceanic and Atmospheric Administration. (2013) The north american mesoscale forecast system. [Online]. Available: <http://www.emc.ncep.noaa.gov/index.php?branch=NAM>
- [13] I. Stoica *et al.*, "Chord: A scalable peer-to-peer lookup service for internet applications," *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4, pp. 149–160, 2001.
- [14] 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.
- [15] 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.
- [16] 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.
- [17] M. Malensek, S. Pallickara, and S. Pallickara, "Polygon-based query evaluation over geospatial data using distributed hash tables," in *Utility and Cloud Computing (UCC), 2013 Sixth IEEE International Conference on*, Dec. 2013.
- [18] G. Niemeyer. (2008) Geohash. [Online]. Available: <http://en.wikipedia.org/wiki/Geohash>
- [19] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, Sep. 1975.
- [20] E. Fredkin, "Trie memory," *Commun. ACM*, vol. 3, no. 9, pp. 490–499, Sep. 1960.
- [21] P. Van Beek and D. W. Manchak, "The design and experimental analysis of algorithms for temporal reasoning," *Journal of Artificial Intelligence Research*, vol. 4, pp. 1–18, 1996.
- [22] T. Abraham and J. F. Roddick, "Survey of spatio-temporal databases," *Geoinformatica*, vol. 3, no. 1, pp. 61–99, 1999.
- [23] M. Malensek, S. Pallickara, and S. Pallickara, "Autonomously improving query evaluations over multidimensional data in distributed hash tables," in *Proceedings of the 2013 ACM International Conference on Cloud and Autonomic Computing*, Aug. 2013.
- [24] B. Welford, "Note on a method for calculating corrected sums of squares and products," *Technometrics*, vol. 4, no. 3, pp. 419–420, 1962.
- [25] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.
- [26] D. Kegel. (2006) The C10K problem. [Online]. Available: <http://www.kegel.com/c10k.html>
- [27] F. Chang *et al.*, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.
- [28] V. Jujjuri, E. Van Hensbergen, A. Liguori, and B. Pulavarty, "VirtFS—a virtualization aware file system passthrough," in *Ottawa Linux Symposium (OLS)*. Citeseer, 2010, pp. 109–120.
- [29] A. Thusoo *et al.*, "Hive — A warehousing solution over a Map-Reduce framework," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1626–1629, 2009.
- [30] 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.
- [31] A. Bialecki, M. Cafarella, D. Cutting, and O. O'Malley, "Hadoop: a framework for running applications on large clusters built of commodity hardware," 2005. [Online]. Available: <http://hadoop.apache.org>
- [32] J. Gao and P. Steenkiste, "An adaptive protocol for efficient support of range queries in dht-based systems," in *Network Protocols, 2004. ICNP 2004. Proceedings of the 12th IEEE International Conference on*, Oct 2004, pp. 239–250.
- [33] Y. Chawathe *et al.*, "A case study in building layered DHT applications," in *Proceedings of the 2005 SIGCOMM*, 2005.
- [34] I. Popivanov and R. J. Miller, "Similarity search over time-series data using wavelets," in *Data Engineering, 2002. Proceedings. 18th International Conference on*. IEEE, 2002, pp. 212–221.
- [35] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica, "BlinkDB: queries with bounded errors and bounded response times on very large data," in *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 2013, pp. 29–42.
- [36] P. B. Gibbons *et al.*, "AQUA: System and techniques for approximate query answering," *Bell Labs TR*, 1998.



Matthew Malensek is a Ph.D. student in the Department of Computer Science at Colorado State University. His research involves the design and implementation of large-scale distributed systems, data-intensive computing, and cloud computing.



Sangmi Pallickara is an assistant professor in the Department of Computer Science at Colorado State University. She received her Masters and Ph.D. degrees in Computer Science from Syracuse University and Florida State University, respectively. Her research interests are in the area of large-scale scientific data management, data mining, scientific metadata, and data-intensive computing.



Shrideep Pallickara is an Associate Professor in the Department of Computer Science at Colorado State University. He received his Masters and Ph.D. degrees from Syracuse University. His research interests are in the area of large-scale distributed systems, specifically cloud computing and streaming. He is a recipient of an NSF CAREER award.