DISSERTATION

LOW-LATENCY, QUERY-DRIVEN ANALYTICS OVER VOLUMINOUS MULTIDIMENSIONAL, SPATIOTEMPORAL DATASETS

Submitted by

Matthew Malensek

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Summer 2017

ABSTRACT

LOW-LATENCY, QUERY-DRIVEN ANALYTICS OVER VOLUMINOUS MULTIDIMENSIONAL, SPATIOTEMPORAL DATASETS

Ubiquitous data collection from sources such as remote sensing equipment, networked observational devices, location-based services, and sales tracking has led to the accumulation of voluminous datasets; IDC projects that by 2020 we will generate 40 zettabytes of data per year [1], while Gartner and ABI estimate 20-35 billion new devices will be connected to the Internet in the same time frame [2, 3]. The storage and processing requirements of these datasets far exceed the capabilities of modern computing hardware, which has led to the development of distributed storage frameworks that can scale out by assimilating more computing resources as necessary. While challenging in its own right, storing and managing voluminous datasets is only the precursor to a broader field of study: extracting knowledge, insights, and relationships from the underlying datasets. The basic building block of this knowledge discovery process is analytic queries, encompassing both query instrumentation and evaluation.

This dissertation is centered around query-driven exploratory and predictive analytics over voluminous, multidimensional datasets. Both of these types of analysis represent a higher-level abstraction over classical query models; rather than indexing every discrete value for subsequent retrieval, our framework autonomously learns the relationships and interactions between dimensions in the dataset (including time series and geospatial aspects), and makes the information readily available to users. This functionality includes statistical synopses, correlation analysis, hypothesis testing, probabilistic structures, and predictive models that not only enable the discovery of nuanced relationships between dimensions, but also allow future events and trends to be predicted. This requires specialized data structures and partitioning algorithms, along with adaptive reductions in the search space and management of the inherent trade-off between timeliness and accuracy. The algorithms presented in this dissertation were evaluated empirically on real-world geospatial time-series datasets in a production environment, and are broadly applicable across other storage frameworks.

ACKNOWLEDGEMENTS

One of the most valuable aspects of working towards a Ph.D. is not the academic pursuit itself, but rather the opportunity to grow as a person — to meet and collaborate with people from diverse backgrounds and see the world from more than just a single lens. Throughout my studies, I have benefited from the kindness, generosity, and insight of countless individuals. I appreciate not only the help I received along the way, but the conversations, laughs, stories, and memories that were woven into the journey. My sincerest thanks to all these colleagues and friends.

I deeply appreciate the love and support of my parents, Suzi and Monte, who always believed in the value of education and taught me to be honest and open-minded. My brother, Nicholas, has been an ally and friend no matter the situation. I am thankful for my grandparents, Rudy, Bev, and Ed, who have always been there for me, and my in-laws, Purabi and Manoj, who traveled halfway across the planet to support me. And perhaps my greatest achievement during my Ph.D. studies was meeting my wife and best friend, Devadatta, who inspires me every day, convinced me not to give up on my dreams more times than I can count, was with me through the ups and downs, and was always there with corny jokes, chocolates, and a whole lot of love when the going got rough.

Throughout my studies, I was lucky enough to be guided by two of the most hard-working, kind, dedicated, and inspirational figures in my life: Shrideep and Sangmi Pallickara. I thoroughly enjoyed all of our discussions, meetings, and most of all, getting to tackle challenging problems every day. I appreciate all of the members of the Distributed Systems Group and Big Data Lab for being a source of great ideas and constructive criticism. Finally, I would like to specifically thank Wim Böhm, Jay Breidt, Bruce Draper, Elliott Forney, Sudipto Ghosh, Paul Hansen, and Wayne Trzyna for all their help and support during my time at Colorado State University.

Additionally, this dissertation was supported the US Department of Homeland Security (HSHQDC-13-C-B0018 and D15PC00279) and the US National Science Foundation (ACI-1553685 and CNS-1253908).

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 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 of year. Multiple features, and the values they take on, comprise the *feature space*.

Gleaning insights involves exploring characteristics of these multidimensional datasets. Besides identifying properties associated with individual features, this also includes identifying relationships: how features correlate with each other and also how their values change with respect to each other. Also of interest are the probabilities associated with these features; for example, pairwise conditional probabilities between features that allow use of Bayesian statistics to predict one feature when measurements are unavailable for the other feature. A broader insight users might be interested in is how the feature space is impacted by time and geography. Our objective is to provide a set of queries that:

- Speed up the discovery process
- Allow fast, near real-time exploration of the feature space
- Provide a base to facilitate more complicated processing

We are interested in supporting three types of analytic queries: *approximate*, *exploratory*, and *predictive*. Approximate queries correspond to an ad hoc analysis of the feature space that may lead to further investigation. 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 [4–6]. Exploratory queries help with understanding the feature space: how features evolve, and how they relate to each other. In other words, their objective to get a deeper understanding of the data. Predictive analytic queries assist in forecasting what is likely to happen in the future. An example of a predictive analytics query is: "What will the temperature be in Bozeman, Montana at 4:00 pm tomorrow?"

This dissertation explores analytic queries in the context of our storage framework, *Galileo* [7–10]. Galileo is a modular, distributed storage platform for multidimensional data that supports pluggable data partitioning schemes, indexing algorithms, and physical storage strategies. To ensure scalability and load balancing, Galileo employs a *distributed hash table* (DHT) network design. DHTs are *key-value stores* that utilize a hash function to facilitate storage and retrieval operations across a network of computing resources. To store a unit of information, called a *value*, a unique *key* is provided, which is passed through a hash function to determine where the information should be stored in the network. Once stored, information can be located by hashing the key again. DHTs allow efficient lookup operations and require minimal data migration while scaling (expanding or contracting the cluster). Galileo uses a *controlled dispersion* strategy [10] that incorporates the Geohash algorithm [11] to account for spatiotemporal properties during data partitioning, which ensures similar observations will be placed on the same physical machine. This facilitates spatial range queries and also helps constrain the scope of retrievals.

In the interest of providing query capabilities beyond the standard key-based lookups supported by traditional DHTs, Galileo incorporates hierarchical *analytic base trees* (ABTs) that maintain cross-feature relationships and support retrievals via an SQL-like syntax. When multidimensional data is stored in the hierarchy, similar observations are represented by overlapping *paths* to conserve memory. As a further optimization, a global, coarse-grained version of the ABT allows clients to reduce the search space of their queries by pinpointing the set of *storage nodes* that are likely to hold relevant data. In a distributed setting, these search space reductions are crucial in providing low-latency results and avoiding unnecessary processing. Galileo also supports spatial proximity queries, retrieval of data within arbitrary polygons, and nearest-neighbor searches backed by a GPU-accelerated, compressed bit vector index [12, 13]. The system has been used to store and analyze billions of files composed of petabytes of observations, and has been leveraged in a variety of use cases, including the analysis of atmospheric datasets [14], anomaly detection [15], discrete event simulations [16–18], epidemiology [19, 20], and visualization [21, 22].

To our knowledge, none of the current DHT-based systems accommodate the types of queries or retrieval operations we have proposed 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 [23] jobs or distributed computations. Once exploration of the feature space has been completed, computations can be launched directly on relevant storage nodes to preserve data locality.

1.1 Scenario

Consider a Galileo user that wishes to explore a voluminous, multidimensional atmospheric dataset. The dataset likely contains valuable insight into weather phenomena, and the user wishes to understand what factors are most influential. This could involve locating feature vectors that indicate low pressures over warm ocean waters during the summer, which may be normal or could signal an impending hurricane. The user issues a set of queries to pinpoint features that match these characteristics, and then requests a statistical summary to understand their scope.

While exploring the dataset, the user can adjust query parameters to fine-tune the subset of features under inspection; feature information is retrieved and processed in real time on the server side to avoid transmitting large quantities of data to the client application. Once the user has selected a particular subset of the feature space, he/she generates visualizations of the data and begins training machine learning models in parallel across the cluster to help understand what features contribute the most to the weather phenomenon of interest.

1.2 Research Challenges

Supporting query-based analytics over voluminous, time-series, and geospatial datasets involves several challenges, including:

1. ***Data Volumes***: The quantity and overall size of the data is high, precluding visualization of every point or sifting through the data sequentially.

2. ***Continuous Arrivals***: Data is streamed into the system on a regular basis, requiring the knowledge base to be constantly updated.

3. ***Data Dimensionality***: Each observation has several features (dimensions) of interest. Furthermore, evolution of the feature space has spatiotemporal characteristics that must be accounted for.

4. ***Managing the Memory Hierarchy***: While main memory accesses complete in nanoseconds, disk access times are much slower. Due to dataset sizes, queries often target large quantities of information, which makes careful management of in-memory data structures critical.

5. ***Circumvention of I/O Hotspots***: When some of the distributed nodes take a disproportionate amount of the storage load, hotspots will occur. This means that data partitioning must be done in a balanced manner.

Given the data volumes being dealt with, it is possible that some queries will be compute-intensive and hence have longer turnaround times. We must balance these processing overheads to ensure high throughput is maintained.

1.3 Research Questions

The goals and challenges associated with this study led us to formulate the following research questions:

1. How can we discover and account for relationships between features at a particular geographic scope? How can this be used to inform predictions and explorations?

2. How can we account for the spatiotemporal characteristics of different features at various geographical extents?

3. How can we ensure near real-time query evaluation given the data volumes and the speed differential between the memory and disk access times?

4. How can we facilitate exploration at scale while maintaining fast response times and high overall query evaluation throughput?

5. Given the continuously arriving data streams, how can we maintain an up-to-date knowledge base?

1.4 Summary of Approach

Our approach to supporting query-based analytics over voluminous, time-series, geospatial datasets takes a holistic view that targets: (1) alleviation of I/O hotspots, (2) coping with the differential between memory and disk speed, (3) construction of multiple metadata models to ensure specificity, and (4) online updates to summary statistics that capture feature space evolution. 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 queries when particular conditions are met

To cope with I/O latencies, *analytic base trees* (ABTs) maintain feature synopses in memory. These include statistics at each vertex, with leaf nodes pointing to physical data blocks on disk. ABTs improve upon traditional indexing techniques by providing direct access to higher-level information about the underlying data points, including their distributions, trends, and cross-feature correlations. This enables real-time knowledge extraction to derive both coarse- and fine-grained insights across voluminous datasets. The trees can also be reoriented to reduce their memory footprints (via reduction of vertices) or query evaluation times (via reduction of edges that must be traversed). The breadth and depth of information represented by the ABT directly impacts our ability to avoid costly disk accesses.

While processing ABT queries, storage nodes that do not contain relevant data points are eliminated from the search space. Next, the queries are resolved in parallel across the remaining nodes and are merged to produce a final result set that can be traversed and used directly for analysis. Galileo supports both singular and continuous queries [24]. This work focuses on the following features of the indexing subsystem:

- Autonomous adaptation of index granularities
- A tree-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 tree traversals during query evaluations to control for timeliness. This ensures sustained throughput over time and prevents starvation of processes with complex query profiles.

Both exploratory and predictive queries depend on the construction of data models that capture feature relationships. We rely on a multiplicity of distributed models rather than a single all-encompassing model. This approach allows for specificity, where each model is responsible for a particular geographical scope. Since the models are dispersed over multiple machines and localized, this approach scales with increases in data volumes as well as reductions to finer-grained geographical scopes. Models include multiple linear regression and artificial neural networks to capture both linear and non-linear relationships, respectively.

1.5 Contributions

This dissertation describes a distributed framework for the evaluation of analytic queries over datasets that are voluminous, multidimensional, time-series, and geospatial in nature. Its contributions include:

1. A platform for low-latency, query-driven analysis with an emphasis on iterative refinement.

2. Judicious management of in-memory data structures to resolve queries and facilitate a variety of analytic tasks without requiring time-consuming disk I/O. This stems from the ability of the system to manage the trade-off space between accuracy and timeliness.

3. Support for high-throughput query evaluations over voluminous datasets.

4. Online modeling of the feature space at various scales with autonomous reconfiguration to cope with evolution in the data.

5. Dynamic construction of linear and nonlinear models, including ARIMA for time-series projection, probabilistic estimates for features, and queries that return results in the form of

probability density functions (PDFs), joint PDFs, or in-memory tree-based datasets that can be leveraged by external frameworks.

6. The algorithms described herein are broadly applicable to other storage systems, such as Cassandra [25], Dynamo [26], and Chord [27], among others.

A key innovation in this work is achieving these contributions at scale and in a timely fashion. Our empirical evaluations are performed over a real-world ***1 Petabyte*** dataset encompassing over ***20 billion*** files.

1.6 Experimental Setup and Dataset

The dataset used in this dissertation was sourced from the National Oceanic and Atmospheric Administration (NOAA) North American Mesoscale Forecast System (NAM) [28]. The NAM conducts frequent collection of atmospheric data in GRIB format, and contains several features across a timespan of over ten years. Features include climate attributes including surface temperature, visibility, cloud cover, precipitation, humidity, snow fall, vegetation, and albedo.

Our data ingest process involves reading data from the NAM and streaming it into the system as samples from discrete spatial locations. Each sample consisted of about 10 KB of raw feature data, along with another 40 KB of graphical tiles designed to be overlaid on a map for visualization. The total dataset encompassed 20 billion files at approximately 50 KB each for a total dataset size of 1 Petabyte.

The benchmarks included in this dissertation were carried out on a heterogeneous 78-node cluster consisting of 48 HP DL160 servers (Xeon E5620, 12 GB RAM) and 30 HP DL320e servers (Xeon E3-1220 V2, 8 GB RAM). Each machine was equipped with four hard disk drives. In some cases, our analysis was restricted to the 48-node cluster. The hardware was divided into 13 groups of 6 nodes each. Galileo was run under the OpenJDK Java runtime, version 1.7.0_65.

1.7 Dissertation Organization

The rest of this dissertation is organized as follows: Chapter 2 explores related work, contrasts with our approach, and analyzes the areas that are not addressed or considered by the current state of the art. Chapter 3 describes the Galileo system architecture, including the network design, fault tolerance functionality, and basic retrieval operations. Chapter 4 discusses our distributed indexing implementations: Analytic Base Trees for multidimensional feature data (including time series), and Geoavailability Grids for geospatial data. Chapter 5 explains the analytic queries supported by Galileo and provides benchmark results, followed by Chapter 6 with our conclusions and future research directions.

Chapter 2

Related Work

Galileo supports functionality that encompasses both storage and computation, while also exposing machine learning models and leveraging a variety of data structures. Herein we survey related work and complimentary technologies/approaches.

2.1 Storage Systems

While traditional file systems and relational databases provide a strong storage foundation, they were not designed specifically for scientific data [29] or horizontal scalability where additional computing resources can be added (or removed) incrementally to form a cluster of machines. Distributed storage systems include key-value stores with flat *get*/*put* operations, document-oriented databases that manage discrete (but often unstructured) units of data, wide-column storage with a tabular, but less rigid structure compared to databases, and *data warehouses* that perform offline batch processing.

Like Galileo, Cassandra [25] is a zero-hop distributed hash table (DHT) designed for high-throughput management of tabular, multidimensional information, and was originally built for write-intensive workloads at Facebook. The system employs a wide-column storage format similar to Google BigTable [30] and allows users to create their own partitioning schemes. However, the active partitioning algorithm also affects information retrieval; for instance, using the random data partitioner backed by a hash algorithm does not allow range queries or future reconfiguration of the partitioning algorithm. This ensures retrieval operations are efficient, but also limits flexibility. Besides the usual *put* and *get* operations for retrievals, Cassandra supports a query model similar to SQL called CQL (Cassandra Query Language). Integration with Hadoop [31–34] provides data manipulation and analysis functionality, but the system does not offer approximate, ad hoc query capabilities or preprocessing focused on machine learning.

Apache HBase [35] is an open-source implementation of BigTable [30] and its wide-column data model. HBase provides consistency guarantees similar to those of traditional RDBMS technologies, and has seen considerable use in areas dealing with high dimensionality. One notable guarantee provided by HBase is strictly consistent reads and writes. HBase is built on HDFS [36], the distributed file system included with Hadoop [31–34], resulting in a mature API and strong support ecosystem. It is also worth noting that the design of HDFS was inspired by the Google File System (GFS) [37], one of the first large-scale distributed storage solutions on commodity hardware. While HBase has excellent support for column-oriented storage, its indexing functionality is limited to a single primary key per table, and the consistency features it supports come at the cost of higher latency.

SciDB [38–40] is a scientific storage system for multidimensional data that supports incremental scalability. During development, the creators of SciDB identified three unique aspects of managing scientific data: first, data sources generally have a location associated with them. Second, the data being collected in scientific applications also usually requires complex processing before and/or after storage. Third, sensors and other scientific instruments generate data on the order of petabytes. The system provides built-in computation and analysis tools that closely resemble the functionality found in traditional relational database management systems, but with a scientific focus. Metadata is stored in a centralized *system catalog* implemented as a PostgreSQL database, contrasting with the tree-based indexes in Galileo.

MongoDB [41] shares several common goals with our approach, but is a document-centric storage platform that does not support exploratory queries or analytics directly. However, the system has rich geospatial indexing capabilities and supports dynamic schemas through its JSON-inspired binary storage format, BSON. MongoDB utilizes the Geohash algorithm [11] for one of its spatial index implementations, and is backed by a B-tree data structure for fast lookup operations. For load balancing and scalability, the system supports sharding ranges of data across available computing and storage resources, but also places restrictions on the breadth of analysis that can be performed in a clustered setting.

Panda [42] is a probabilistic, spatiotemporal query processor that operates in a non-distributed environment. The system leverages its predictive capabilities to determine where objects are most likely to move at a future point in time, and can autonomously cache frequently accessed objects to improve response times. While its data structures and architecture support a limited geographic scope, Panda is useful in a variety of applications such as location-based services, traffic management, or ride sharing where the resolution of the data and object behaviors are consistent. The system also supports continuous queries and can optimize results to eliminate object movement processing that would not produce relevant results.

Apache Hive [43] is a data warehouse that runs on Hadoop [31–34] and HDFS [31]. As an analysis platform, it is capable of a wide range of functionality, including summarizing datasets and performing queries. Unlike Galileo, the system is intended for batch use. In Hive, users can perform analysis using the HiveQL query language, which transforms SQL-like statements into MapReduce [23] jobs that are executed across the Hadoop cluster. The *Metastore*, a system catalog, provides an avenue for storing pre-computed information about the data in the system. Hive emphasizes scalability and flexibility in its processing rather than focusing on low latency.

BlinkDB [44] provides approximate query processing (AQP) functionality based on Hive [43]. The system allows analysts or practitioners to query large datasets while specifying time and error bounds to ensure adequate coverage for their particular use case. BlinkDB supports high-level summaries created by sampling randomly across the entire dataset with two methods: a broad dataset-wide sample, and a focused sample that includes frequently-accessed items. The sampling framework in BlinkDB also supports stratification to better represent outliers or underrepresented data points. By generating indexes based on samples, queries can be resolved quickly without reading information from the disk. However, using the system in a predictive or exploratory capacity is generally limited to the insights that can be derived directly from the sampled data.

FastRAQ [6] 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) [45] provides estimated responses to queries while 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*, where more results are 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.

Table 2.1 outlines the broad categories that these storage systems fall into and compares their functionality in terms of our objectives. In general, systems that support queries also target relational functionality, which comes at the expense of latency. Key-value stores, on the other hand, meet nearly all of our objectives but only support the basic *put* and *get* operations. Existing approaches do not consider support for both geospatial and time series data, fuzzy queries, or integrated modeling. Additionally, support for approximation is often limited to building a specific set of summaries beforehand, which prevents adaptation of the accuracy-timeliness gradient.

2.2 Computation Engines

While Galileo was primarily designed to be a storage system, it also supports a variety of distributed computations. Herein we survey related approaches for orchestrating distributed computations and how they compare to the query-based exploration and retrieval functionality in Galileo.

Hadoop [33] is an open-source implementation of MapReduce, providing a scalable solution for managing and processing large datasets. MapReduce is a distributed programming paradigm composed of the *Map* phase where filtering or processing occurs, followed by the *Reduce* phase

Table 2.1: A comparison of storage systems in the context of our research objectives.

Technology	Query Support	Low Latency	High Throughput	Minimal Intervention	Flexible Schema
Databases	✔	✔			
Key-Value Stores		✔	✔	✔	✔
Document-Oriented DB		✔	✔		✔
Wide-Column Stores	✔		✔		
Data Warehouses	✔				

that summarizes or aggregates results from the Map phase. While its underlying distributed file system (HDFS [36]) does not partition records based on their content, the system is widely available and compatible with several distributed computation engines. Hadoop provides distributed batch processing capabilities, but native support for real time exploration is limited. Additionally, while the computational facilities in Hadoop could be leveraged to perform data preprocessing, intermediate records would need to be written to the disk and stored as full-resolution blocks.

Applications or algorithms that require iterative, loop-based processing often execute inefficiently in traditional MapReduce environments. This is primarily due to restrictions on how computations can build state over time; generally, state must be stored to disk after each *map* operation and then loaded again for subsequent iterations. HaLoop [46] addresses these issues by adding loop control directives to MapReduce as well as loop-aware caching mechanisms to allow dataset reuse. Caching is also leveraged to allow efficient discovery of termination conditions, which may normally require running additional MapReduce jobs.

Granules [47–49] is an open-source distributed stream processing system with computation support that can be expressed using the MapReduce paradigm or as directed, cyclic graphs. The framework leverages the NaradaBrokering networking substrate [50–52], and handles deploying, scheduling, and orchestrating computations on clusters of machines or in the cloud. Computations can be scheduled to run when data is available or at regular intervals, with a configurable number of execution iterations. Since computations can execute multiple times, it is possible to build state over the course of execution. Granules has been employed in several areas of study, including bioinformatics, brain-computer interfaces, clustering, and scientific data storage. Notably, the ability for computations to be cyclic and maintain state are crucial features when supporting distributed queries efficiently.

Spark [53] is a cluster computing framework that extends MapReduce by supporting memory resident datasets and computations described by directed, acyclic graphs (DAGs). This allows expensive disk accesses to be avoided while also facilitating graph-based workflows and iterative applications such as machine learning algorithms. One of the key features of Spark is its Resilient Distributed Datasets (RDDs), which provide in-memory parallel data manipulation functionality [54]. Spark can process data from a variety of underlying storage services, but lacks the integrated storage and exploration facilities implemented in Galileo. MLBase [55] extends the Spark stack to support machine learning tasks with RDDs. MLlib [56] contains several machine learning algorithms, and an integrated *ML Optimizer* assists users in finding the best parameters for their models.

Vowpal Wabbit [57] (VW) is a distributed, online machine learning system that supports general computations and shared state between computing resources. VW supports several supervised machine learning techniques and includes implementations of regression, classification, and optimization algorithms. The system has been employed in large-scale machine learning applications and focuses on processing efficiency. Like many of the other machine learning systems discussed in this work, VW compliments the Galileo ecosystem and could be used to build machine learning models after the exploration and preprocessing stages are complete.

Weka [58] includes a variety of data mining and machine learning algorithm implementations, along with pre-processing and visualization functionality. While Weka is designed for single-node deployments, final datasets produced by the exploratory analytics process in Galileo could be converted and used to populate Weka machine learning models for further analysis. Weka includes an open source API for developing new models and visualizations, and includes a graphical user interface for parameterizing models and loading data.

2.3 Indexing/Retrieval Approaches

A variety of research has been conducted on supporting flexible, distributed queries. For instance, Gao and Steenkiste [59] maps a logical, sorted tree containing data points to physical nodes, enabling range queries. Chawathe et. al [60] 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.

Galileo ABTs share some common features with k-d trees [61], but do not employ binary splitting and allow much greater fan-out as a result. Similar to Tries [62], identical attributes in a record can be expressed as single vertices, which simplifies traversals and can reduce memory consumption. However, Galileo ABTs support multiple concurrent data types, maintain an explicit feature hierarchy (that can also be reoriented at runtime), and employ dynamic quantization.

Like the summary statistics maintained in our approach, incoming data streams can be represented as wavelets to avoid indexing every data point while still maintaining an approximate model. Cormode et al. [63] employs several different types of wavelets for creating synopses or approximations of incoming data and reviews their efficacy. Yousefi et al. [64] investigates the feasibility of using wavelets for prediction. However, methods that rely on wavelets are generally very problem- or dataset-specific and can limit the feasibility of resolving arbitrary queries over the underlying data.

The TPR*-tree [65] provides predictive spatiotemporal query functionality that can retrieve the set of moving objects likely to intersect a particular spatial window at some future point in time. Intended use cases include meteorology, mobile computing, and traffic control. TPR*-Tree improves upon the Time Parameterized R-tree (TPR-tree) [66] by adding a probabilistic model that accurately predicts disk accesses involved with resolving a query, along with new insertion and deletion algorithms to enhance performance. While TPR*-tree and its related data structures focus on object movements, Galileo also considers predictions across a wide array of dimensions.

P2PR-Tree [67] provides a P2P-based version of the R-Tree [68] spatial index. The system is decentralized and can also service spatial queries while peers are leaving or joining the network. In P2PR-Tree, queries are routed to nodes that may have pertinent information, with a traversal through the network closely resembling a traversal through an R-Tree. Initially, the range of possible spatial values is broken up into *blocks*, with each block being statically divided into a pre-set number of groups. Nodes in the system are then divided into multiple levels of subgroups with neighboring peers maintaining more detailed information about one another. Each peer also maintains a local R-Tree for performing lookups on the data it holds.

SD-Rtree [69] aims to provide a scalable distributed R-Tree implementation. The resulting system takes inspiration from both R-Trees and AVL trees, and provides several access methods with differing scalability and performance. Like a traditional R-Tree, the SD-Rtree inserts information at its leaf nodes, splitting them when they overflow. This property greatly increases the amount of communication required in the early stages of the tree creation, but gradually decreases as more information is stored and the tree structure settles. A key strategy incorporated in SD-Rtree is client-side caching to speed up queries and reduce the amount of traffic directed towards the root of the tree, which also reduces the number of messages sent during insertions and query resolution.

SAUNA [70] proposes a technique for automatically relaxing the constraints of user-defined queries. SAUNA operates on a standard relational database management system, and uses histograms to estimate the cardinality (number of results) of incoming queries. In situations where a query is estimated to return a small number of results, the input ranges of the query are relaxed to retrieve a broader range of items that are close to the desired parameters. This optimization helps reduce the overall number of queries that will be submitted by users of the system, increasing throughput.

Data Cubes [71–74] are a data structure for online analytical processing (OLAP) that provide multidimensional query and summarization functionality. These structures generalize several operators provided by relational databases by projecting two-dimensional relational tables to N-dimensional cubes. Variable precision is managed by the *drill down/drill up* operators to increase and decrease resolution, respectively, and *slices* or entire cubes can be summarized through the *roll up* operator. While Data Cubes provide many of the same features supported by Galileo, they are primarily intended for single-host offline or batch processing systems due to their compute- and data-intensive updates. In fact, many production deployments separate transaction processing and analytical processing systems, with updates pushed to the Data Cubes periodically.

CHAOS [75] builds on Data Cubes in a single-host streaming environment by pushing updates to its *Computational Cubes* more frequently. To make dimensionality and storage requirements manageable, Computational Cubes only index summaries of incoming data that are generated during a preprocessing step. However, full-resolution data is still made available through continuous queries that act on variable-length sliding windows. Additionally, updates to the cube are still generated and published periodically rather than immediately as data is assimilated.

2.4 Discussion

Recent distributed storage systems are generally designed around the concept of providing a small, well-defined feature set that can be scaled out as necessary. Similarly, computation engines such as MapReduce impose restrictions on user applications, but offer automatic parallelization. These design decisions are ultimately made around trade-off spaces such as query latency, correctness of results, ease of use by both programmers and end users, resilience to network failures, and other domain-specific concerns.

Brewer's CAP theorem [76–78] examines these trade-offs in terms of consistency (*C*), availability (*A*), and partition tolerance (*P*). Distributed systems must tolerate network partitions to operate effectively, and therefore must choose between availability (always being accessible) or consistency (presenting a consistent state to client applications) in the face of a network partition. However, recent work by Google on Spanner [79] and F1 [80] uses specialized hardware to provide a consistent representation of time across geographically dispersed data centers, allowing stronger consistency guarantees while remaining highly available. Additionally, the studies revealed that users: (1) generally prefer an expressive language to retrieve information rather than writing ad-hoc computations, and (2) favor automatic conflict resolution over dealing with consistency issues themselves. Over time, new developments in distributed storage and computation engines will likely be accompanied by increased user friendliness and accessibility; one of the core objectives of this dissertation was to investigate how high-level, expressive queries can improve the effectiveness of data exploration by obviating the need for manually writing distributed applications.

Chapter 3

System Overview and Architecture

Galileo is a high-throughput distributed storage framework that supports range-based, exact-match, geospatial, and approximate queries over multidimensional time series data. The system is implemented as a Distributed Hash Table (DHT), allowing incremental scalability and flexible data partitioning. Distributed hash tables are key-value, structured overlay networks created by partitioning a hash space across multiple machines. In general, DHTs use a hash function to allocate network resources, partition their datasets, and perform lookup operations; similar to the hash table data structure, DHTs map unique keys to their associated values. DHTs employ a *shared nothing* architecture, meaning that a central component is not used to maintain state information: all nodes in a DHT are *stateless* and offer identical functionality without sharing memory or other resources. Further, most DHT implementations rely on *consistent hashing* [81], where changes to the network or hash topology have a minimal impact on the system as a whole. These design considerations make DHTs highly scalable and able to deal with frequent network changes, such as failures or the addition/removal of nodes.

Chord [27], CAN [82, 83], Kademlia [84] and Pastry [85] provide canonical examples of such systems. Contrasting with this traditional DHT model, Galileo is a *zero hop* (or *one-hop*) DHT. This means that enough state information is maintained at each node in the DHT to route requests directly to their destination without intermediate hops through the network. Examples of other zero-hop DHTs include Amazon Dynamo [26] and Apache Cassandra [25].

To support the management of voluminous scientific datasets, Galileo places participating machines (called *storage nodes*) in several groups (or subgroups), creating a hierarchical network layout; in other words, a DHT of DHTs. Combined with a multi-tiered hashing scheme, this functionality enables a balance to be struck between uniform load balancing and ordered data placement. Figure 3.1 provides an overview of the Galileo network and demonstrates how queries propagate through the system. Galileo includes several domain-specific features to ease working

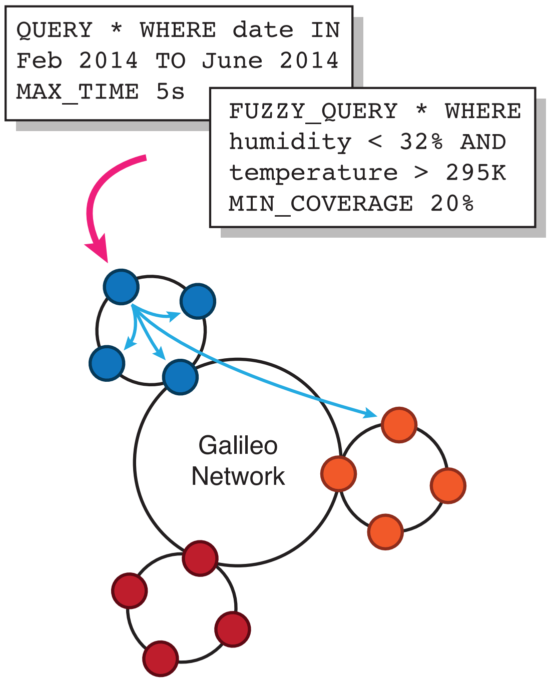


Figure 3.1: Overview of the Galileo network (ring-of-rings DHT), with sample queries.

with scientific datasets such as built-in geospatial and temporal data structures, along with support for scientific formats such as NetCDF [86], BUFR, or HDF5 [87].

Galileo is *eventually consistent* [88], meaning that changes to the system are not visible to all nodes immediately; in other words, Galileo is considered an *AP* system under Brewer's CAP Theorem [76]. This means that network partitions result in consistency being sacrificed; the system remains available and can continue servicing requests, but the global state maintained at each node may not be consistent with the rest of the nodes in the system.

While most DHTs support only key-based retrievals, Galileo provides expressive query support through a multi-tier indexing scheme. Each storage node in the system maintains *analytic base trees (ABTs)* that serve as an index of local files, while the coarser-grained global ABT contains information about where all of the information in the system is located [10]. Both trees rely on a *gossip* scheme to disseminate state information, which propagates from node to node until a steady state is reached. The local ABT is used for locating and retrieving records, whereas the

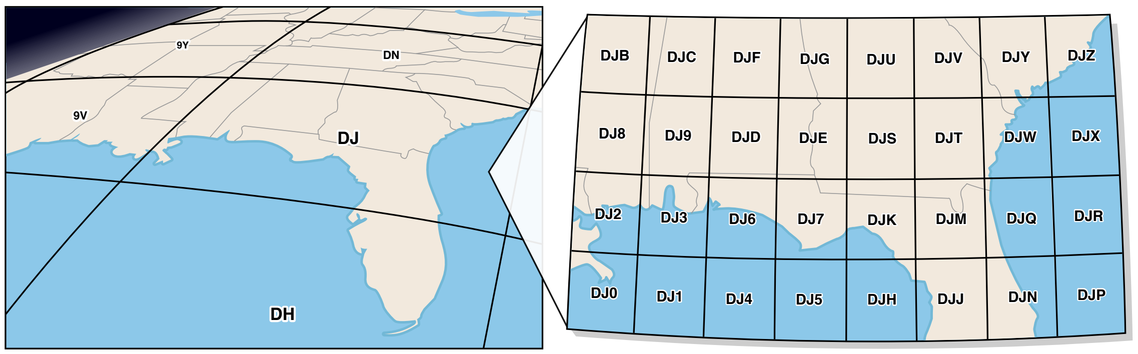


Figure 3.2: An overview of successive spatial subdivisions generated by the Geohash algorithm. In this example, the area represented by Geohash *DJ* is divided into 32 smaller subregions, each about 133×155 km in size.

global ABT helps reduce the search space of distributed queries by eliminating storage nodes that do not contain relevant data.

Users do not need to be aware of the structure or contents of the trees, and queries are formulated in a declarative syntax. Any of the nodes comprising the DHT can accept and evaluate queries. The receiving node computes the subset of machines likely to hold data of interest and then forwards the query on to applicable nodes for evaluation.

3.1 Data Partitioning

We use the Geohash [11] algorithm to partition data across the storage groups in Galileo. Geohash divides the Earth into a hierarchy of bounding boxes that are referenced by Base 32 strings. The longer the string, the smaller (and therefore more precise) the spatial bounding box. For example, the coordinates of N 30.3321 W 81.6556 would translate to the Geohash string *DJMUTCU1Q*. Figure 3.2 demonstrates two successive iterations of the algorithm, where our example location would fall within the *DJ* and *DJM* bounding boxes. Each subsequent iteration of the algorithm adds another character to the string and introduces 32 new spatial subdivisions.

Geohash strings are used by Galileo to succinctly represent spatial points or regions, as well as assign data to groups in a flexible manner that enables both scaling up or down as necessary. With our test dataset, the first two characters of the Geohash for a data point are used to determine group membership. This results in groups being responsible for regions of approximately 1030×620 km, which also helps reduce the search space of distributed spatial queries.

In DHTs, a hash function is used to locate where data will be stored in the system. In the case of Galileo, a two-tiered hashing scheme is used: first, the destination group for the data is determined by computing a Geohash based on the data's spatial information. Then, to determine the storage node within a group, a SHA-1 hash is computed using the data's temporal and feature metadata sets (for instance, the time the observation was recorded along with the current temperature, wind speed, humidity, etc.). The storage node receiving new data points will compute these hashes and then transfer the incoming file to its appropriate destination. To cope with heterogeneous systems, machines can also join multiple groups or represent multiple "virtual" machines within a group. This allows more powerful nodes to be added to the system later while maintaining the balance of load. An important aspect of this approach is that if a query requests information from a specific geospatial area, Galileo can immediately reduce the search space by selecting only the groups of machines that hold relevant data. We call this collocation of similar data points over subsets of nodes *controlled dispersion*.

3.1.1 Evaluating Controlled Dispersion

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 3.3 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 points are spread across all nodes in the cluster; searching for a particular spatial location yields a distributed query plan that must contact each of the 48 nodes.

The distribution results for our hierarchical hashing scheme are shown in Figure 3.4. Unlike the flat SHA-1 hash, we have imposed a greater storage imbalance in the system, but similar data points

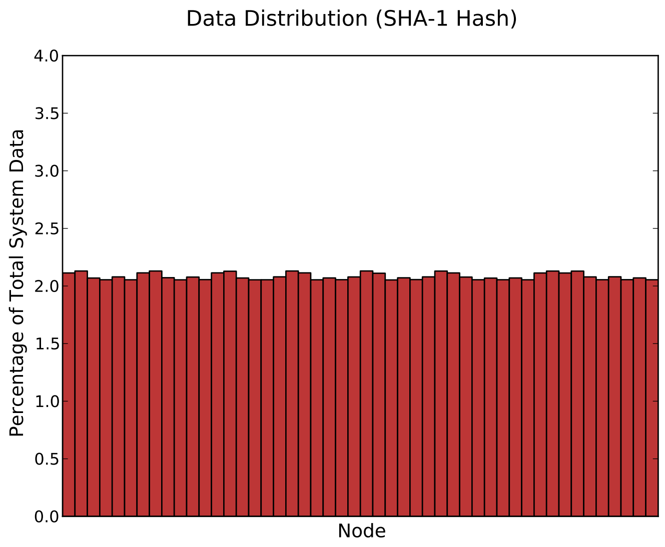


Figure 3.3: Distribution of files in the system using a flat SHA-1 hash across all data dimensions.

are now placed logically closer to each other. Table 3.1 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 3.1: Comparison of data distribution schemes based on the 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
Standard Deviation (%)	0.03	0.41

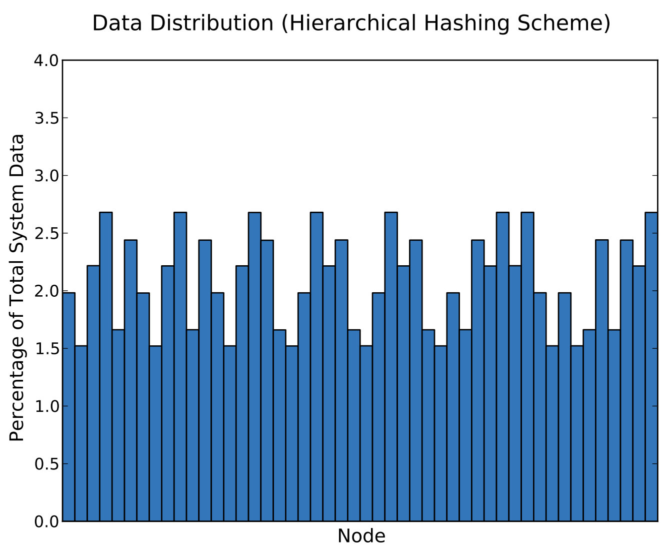


Figure 3.4: Distribution of files in the system using our two-tiered hierarchical hashing scheme.

3.2 On-Disk Storage Format: Galileo Blocks

A Galileo block is a self-describing multi-dimensional array of data, meaning that the files contain all the information required to be read and understood; a block includes feature names (e.g., 'humidity') as well as corresponding value(s) (e.g., '28%'), which may include multiple dimensions or readings collected over time. Blocks are similar to formats such as NetCDF [86] and FITS [89], although in the case of Galileo metadata files are stored separately on the file system alongside raw data. This separation simplifies data management: indexing, lookups, and queries all operate on metadata, while storage and modification operations occur on the blocks themselves. The division also makes it possible to load and retain metadata information in main memory without needing to read an entire block from disk. Blocks support GZIP compression and decompression, which is useful for conserving disk space and reducing network transfer times.

Instead of a single large directory containing all the blocks present on a machine, blocks are separated into a hierarchical directory structure based on their metadata. As the number of files in a directory increases, reading directory indexes becomes more time-consuming, so this structure spreads data across multiple directories to improve performance. The structure also makes it possible to glean some of a block's metadata simply by knowing its location in the hierarchy on disk. In the case of massive datasets where not all the metadata in the system can be stored in main memory or after a failure has occurred, the hierarchy makes it possible to start searching from a starting point that is already close to the desired information.

The initial directory structure is as follows: beginning with temporal information, the year associated with the block is used to determine the directory under the first (root) storage directory for data in the system. Months, days, and hours are used to further sub-divide the directory structure; each year directory contains twelve month directories, and each month directory contains up to 31 days, and so on. Since temporal information could include a range of times, the beginning of the range is used for the disk hierarchy. The next level of subdirectories is determined by using the data's spatial information represented as a Geohash. The precision of these strings determines the number of subdirectories created on the file system, and can be automatically tuned by Galileo to cope with different geographical dispersions of data. If incoming data does not fit into the configured directory structure, it is automatically split into separate blocks by the system.

Our storage scheme offers a few advantages over using one large, contiguous file for storing blocks. For instance, the hierarchy encodes partial metadata in directory names that is available without needing to read any files directly. In addition, this scheme makes the storage format flexible; we can incorporate support for additional formats such as NetCDF [86], HDF5 [87], or FITS [89] easily without needing to overhaul major parts of the system. When non-native formats are stored, a block simply acts as a container of raw data. Since blocks are stored on top of the host file system, any benefits the file system provides will also benefit Galileo. Caching of frequently used blocks can be handled by the host operating system as long as it supports a caching mechanism. Performance characteristics of file systems often involve trade-offs, so this scheme allows the underlying file system to be changed to better match specific workloads, if necessary. For instance, ReiserFS [90] supports dynamic allocation of file system inodes, which can be helpful in situations where a large number of files are being stored.

3.3 Fault Tolerance and Journaling

Making changes to the underlying data structures in Galileo often requires several disk operations. Power failures or system crashes that take place during such operations can leave these data structures in an inconsistent, partially-modified state; the use of *journaling* safeguards against such uncertainties. Every change committed to Galileo is preceded by an update to the system journal. In the case of failures, the journal serves as a checkpoint and the entire journal is read from disk to determine the last operation that was taking place before the failure occurred. Once the pre-failure state has been determined, the operation can be completed, if possible, or rolled back if the data required to complete the operation was lost during the failure.

Each storage node in Galileo executes a separate thread that oversees the verification and replication of data. The designers of Hadoop Distributed File System (HDFS) at Yahoo! observed that around 0.8 percent of their nodes fail each month and that with a replication level of three, the probability of losing a file during one year is less than 0.005 [36]. Therefore, it is ideal to allocate at least three machines per group since replication is done at the group level.

Within a group, machines act as a circular buffer. The parent node for a block will receive the first copy of the block, store it, and then forward the block on to its neighbor. The neighbor then stores and forwards the block on to its neighbor, continuing until the configured replication level is achieved. In the case of machines acting as multiple virtual nodes in the system, the data is forwarded on to the next non-virtual node. This scheme has a few advantages. For one, network load is distributed evenly among nodes participating in replication since multiple copies do not need to be sent to the system directly. Additionally, the parent node will know where replicas will be stored without needing to communicate with any other nodes. Replicated blocks are not included in query results.

Since groups in the system are often formed based on rack locality or some other geographic locality, it can be advantageous to replicate across groups as well. Cross-group replication can be configured manually to guarantee that at least one replica is placed in a different geographic location so data is not lost in scenarios where an entire data center goes down. In cases where groups are already geographically dispersed, the system can handle cross-group replication automatically.

When corruption is detected using checksums stored in metadata files, a node may request a replica from its neighbor. Replication requests are logged and used to determine if a particular node is experiencing a higher rate of corruption than the rest of the nodes. In cases where a node has been determined to be faulty based on its corruption rate, it can be automatically removed from the system.

3.4 Information Retrieval

As described previously, DHTs generally do not offer support for expressive queries due to their retrieval operations being tightly coupled with data partitioning. However, Galileo provides support for additional indexing operations to be applied on observations after they are stored. These indexes are part of a configurable plugin architecture, and allow: (1) reduction of distributed queries' search space, and (2) local retrieval of specific data points on storage nodes.

As multidimensional data points are streamed into the system, they are organized to form a *chain* of feature values. Figure 3.5 demonstrates such a chain for an atmospheric observation collected in Fort Collins, Colorado on June 1st, 2014, at 4pm:

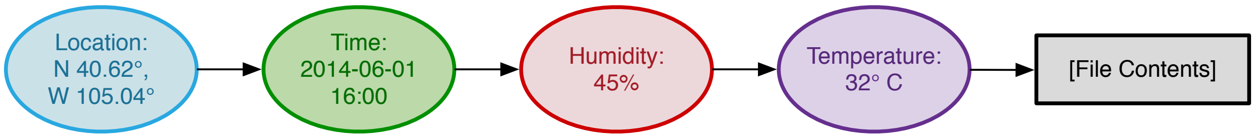


Figure 3.5: A chain of features derived from a multidimensional observation.

Note that each chain ends with a pointer to the raw data (a Galileo file block) associated with the observation. After creation, these chains are inserted into a tree that bears some resemblance to a standard UNIX file system, but with a predetermined hierarchy; humidity may be placed at the root of the tree, followed by the time of observation, and so on. These trees form the basis of the indexes used in Galileo for retrievals. To query the trees, exact values (or ranges) may be

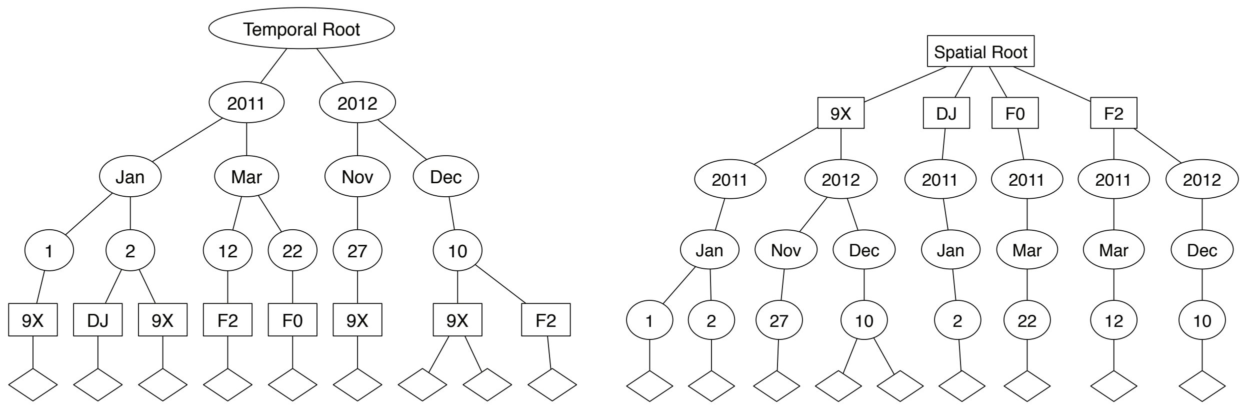


Figure 3.6: Two different orientations of the same index. Circles represent temporal objects, squares represent spatial objects, and diamonds represent feature information.

specified, or if no value is specified for a particular feature then the entire feature *plane* (level of the hierarchy) is traversed. Queries that are more specific select a finer-grained subset of vertices, and the leaves of the tree contain file blocks of interest.

There are two unique properties of this indexing strategy: first, similar paths can *reuse* vertices in the tree, which reduces memory consumption and speeds up traversals. Second, the trees may be *reoriented* to achieve desired performance profiles; for instance, some applications may place a higher importance on temporal information and decide to use it as the root of the tree, which is where traversals begin. On the other hand, there may be applications or users that wish to select a specific geographic region and then traverse through the temporal data within that region, so the tree would be reoriented with spatial information as the root. Figure 3.6 illustrates the reorientation process on an example tree. User-defined features and devices are also part of the dataset, so applications will know what kind of data is available without reading it first. This makes it possible to request all data from a particular type of sensor within a certain spatial region, or to request all readings from a specific date range. The tree can also be reoriented mid-traversal.

Since Galileo expects to have data being continuously streamed into the system, it also provides a callback mechanism to inform clients when their datasets change. To exploit this functionality, clients register their queries with the system and then incoming data can be evaluated as it arrives at the storage nodes. If new data matches the query, its metadata is streamed back to the client for inspection and manipulation [24, 91].

While this hierarchical, tree-based approach makes it possible to execute expressive queries over multidimensional observations, voluminous datasets contain billions of data points that will produce trillions of vertices and edges. Chapter 4 outlines our development of this indexing and query model, with a focus on the data structures and algorithms that allow Galileo to manage such datasets at scale.

Chapter 4

Metadata Extraction and Indexing

Galileo handles a variety of data types: temporal and geospatial, along with multidimensional feature data that can be composed of booleans, intervals, floating points, integers, and strings. Two key data structures that underpin indexing operations in the system are *analytic base trees* (ABTs) and *geoavailability grids* (described in detail in Sections 4.1 and 4.4, respectively). Given the size of the datasets we consider in this work, storing every data point in memory is not feasible, which underscores a key trade-off between timeliness and accuracy. Systems that strive for completely accurate results will require frequent disk I/O, which is orders of magnitude slower than modern CPUs and memory architectures. On the other hand, results that do not achieve an acceptable level of accuracy are simply not useful. The crux of these indexing approaches is to strike a balance between accuracy and timeliness, while leveraging in-memory *summaries* or *sketches* to provide a compact, accurate, and low-latency representation of the data.

4.1 Hierarchical Feature Index: Analytic Base Trees

As previously outlined, Galileo extracts metadata from incoming records and organizes it in hierarchical, tree-based indexes to enable fast query resolution and retrievals. Each level in the hierarchy manages a unique feature type, with queries drilling down through the hierarchy to locate observations of interest. This approach: (1) facilitates the storage of multidimensional data points, (2) allows expressive queries to select particular subsets of the tree, and (3) reduces memory consumption for similar paths through *vertex reuse*. However, when faced with large, diverse datasets, this approach becomes unwieldy due to its memory consumption; high variation in feature values reduces vertex reuse, and datasets that contain several different features require additional levels in the hierarchy, increasing the fan-out of the tree.

To cope with these situations, vertices are assigned to configurable ranges of values called *tick marks*. Tick marks allow varying degrees of index granularity to be achieved, resulting in similar

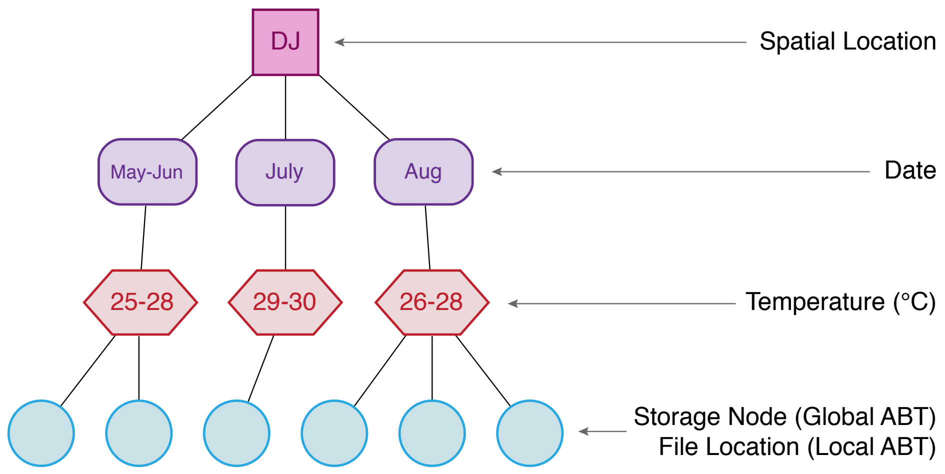


Figure 4.1: An example analytic base tree with variable tick mark granularities; traversals lead to the location(s) of files in the network (in the global ABT) or on disk (in the local ABT).

values being condensed to a single index entry — values of 0.091 and 0.092 might be stored on a vertex managing values from 0.0 to 0.1, for instance. These tick marks can be chosen explicitly by the user or generated autonomously by the system. Figure 4.1 demonstrates an ABT with variable tick mark ranges associated with each vertex. By leveraging tick marks, vertex reuse increases, allowing more information to be stored in the ABT at the expense of accuracy. Note, however, that full resolution file blocks are still maintained on disk.

The ABT operates in two modes: a *local*, high-resolution version for retrieving raw data from the disk, and a *global*, coarse-grained version for determining which storage nodes in the system hold relevant data. In a steady state (i.e., no new files are being added to the system), the global ABT is identical across all storage nodes, while local ABTs are unique. This variable precision is achieved through our *quantization* process, and allows for approximate or exploratory queries that access the appropriate data structures based on user requirements. Global lookups are fast but less accurate, local lookups require more nodes to participate but return higher-resolution trees, and disk accesses provide an option for traditional, full-resolution retrievals at the cost of additional I/O and processing latency.

Table 4.1: Sample granularity configurations for the global ABT: values represent tick mark ranges.

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

4.1.1 ABT Quantization

One approach for mapping a large range of values to a discrete set (called *quantization*) is to divide the range of possible values across fixed intervals. Galileo can be configured to quantize features differently depending on their properties; Table 4.1 provides three sample tick mark granularities for temperature and humidity values. The "coarse" granularity allocates temperature readings within ranges of 10 Kelvin and humidity levels within ranges of 5%. This effectively places all temperatures from 300-309 K into one group, 310-319 K into another group, and so on.

Figure 4.2 demonstrates how increasing or decreasing the granularity of the tick marks impacts query performance of the global ABT. In 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. If an approximate geospatial location is provided, no more than 17% of the nodes in the 6-group system (48 nodes) used for this benchmark will be contacted. The search space within a group can also be reduced similarly as more dimensions are added. Table 4.2 contains statistics on how vertices, edges, and memory consumption increase as granularity increases.

4.1.2 Autonomous Tick Mark Reconfiguration

While our sample tick marks provide a point of reference for how ABTs behave under different configurations, production settings often involve datasets with hundreds of features that cannot be

Table 4.2: Tree statistics with variable tick mark granularities.

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

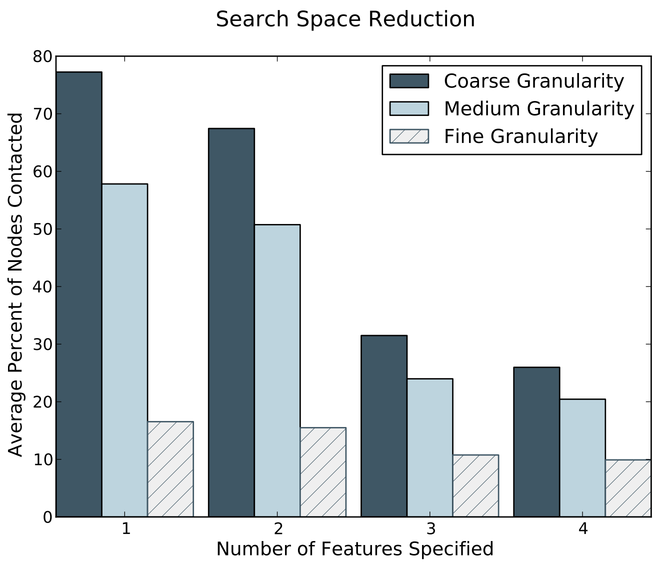


Figure 4.2: Global ABT search space reduction with variable tick mark granularities as more dimensions are specified in the query.

adjusted manually by the user. Additionally, each feature may exhibit a different distribution of values that is best represented by tick marks of variable size. To compensate for these factors, tick mark ranges in the ABTs are autonomously created rather than relying on preconfigured ranges [92]. This optimization helps improve the overall accuracy of traversals, thereby reducing the number of storage nodes that must be contacted during a distributed query. This property improves the performance of queries; regularly-accessed files and readings with frequently-observed values are given priority by the algorithm, improving their precision.

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

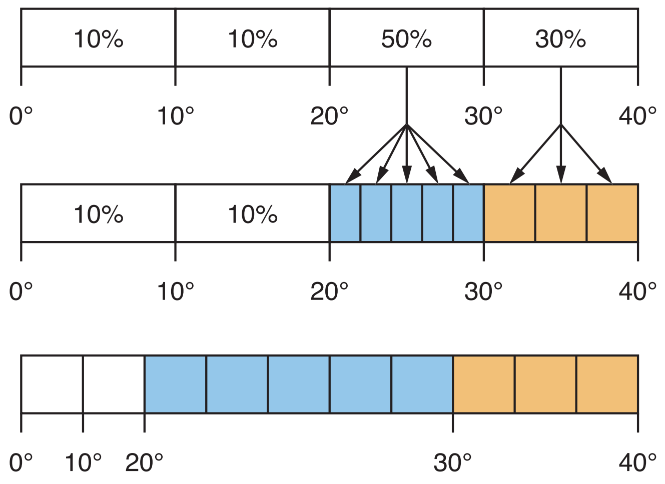


Figure 4.3: Dynamic index reconfiguration on vertices holding temperature values (in °C). In a tree with four vertices, two receive 50% and 30% of the overall storage and retrieval requests. After reconfiguration, load is balanced evenly across 10 vertices.

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 tree, the range will be split into separate vertices to distribute load as uniformly as possible. Figure 4.3 illustrates how tick mark ranges are split in the ABT.

ABT 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:

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

Relatively high values of CV indicate high dispersion. When the CV across a level in the ABT 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 tree and keeping growth in check.

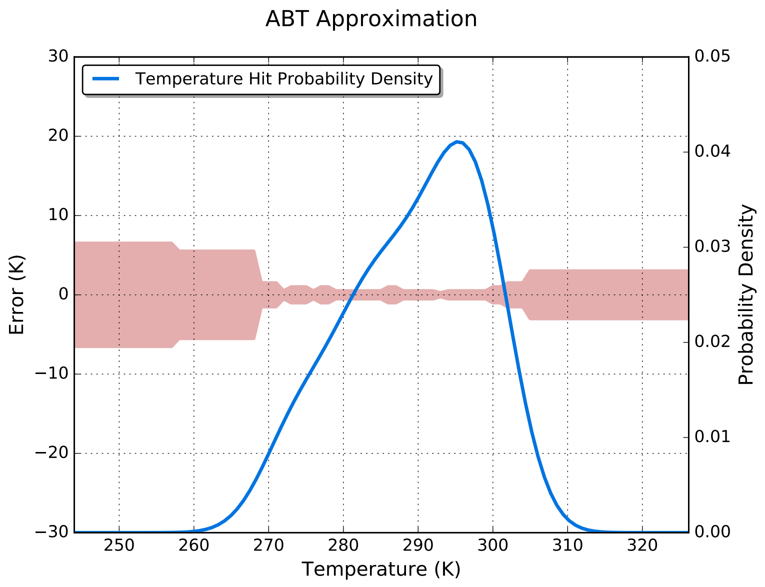


Figure 4.4: Error bounds on approximate results returned by a quantized ABT, with the probability density of hits (query accesses or storage operations) on temperature values.

While modifying the ranges of tick marks in the ABT 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 4.4 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 ABT can provide results within ± 1 Kelvin of the actual value.

4.1.3 Density-Based Quantization

Galileo storage nodes maintain an online kernel density estimation (KDE) for each feature type, which provides an approximation of the features' probability density functions (PDFs). We use the oKDE library [93] to maintain these PDF estimates, which provides information on how each variable is distributed, what the most common values are, and how likely particular ranges of values will occur. The smoothing parameter used to create the PDF, called the *bandwidth*, is selected autonomously using Silverman's rule [94]. Silverman's rule assumes that data tends to follow a normal distribution, which is generally true for naturally-occurring observations. However, we also allow the smoothing parameter to be selectively reconfigured for different problem types. To account for evolution in the feature space, oKDE allows an optional *forgetting factor* to phase out old data over time, as well as a configurable *compression* to influence how much data is retained in memory. This allows dynamic management of the memory-accuracy trade-off space associated with each feature PDF. For our subject dataset, we found that evolution in the feature space did not occur rapidly enough to require old data to be phased out, but we do adjust the compression level based on memory constraints at the storage nodes.

PDF estimation can also be used to generate density-based tick marks. For each feature, PDFs are used to create range boundaries that distribute data points uniformly across the tick marks. Ticks are generated during system initialization, and updated periodically at run time if substantial changes in the distributions occur. Updates are performed on an incremental, targeted basis, where tick marks are either split in two or merged with a neighboring tick mark.

Figure 4.5 illustrates autonomous, density-based ABT quantization for the *surface temperature* feature in our atmospheric test dataset: the highest densities of values are stored in the smallest tick ranges (indicated by vertical lines under the curve), improving overall accuracy. To evaluate accuracy, we compare the mean values of each tick mark with the actual, full-resolution data points. Consequently, the *standard error* ($\sigma_{\bar{x}}$) can also be calculated to judge the accuracy level of the tick marks based on how well they represent the mean: $\sigma_{\bar{x}} = \sqrt{S_n/n^2}$. This information is provided alongside any query results returned by the system. During initialization, we calculate the normalized error for each data point empirically (shown in the lower portion of Figure 4.5). For values that are observed less frequently, the error rate is higher; temperatures from 240 – 260 Kelvin (-33.15 to -13.15 degrees Celsius) reach a normalized root-mean-square error (NRMSE) of about 7%. However, approximately 80% of the values in the tree will be assigned to vertices with an error of about 0.5%. In practice, this means that commonly-observed values will be within 0.25 Kelvin of their actual value.

Table 4.3 compares full-resolution and quantized trees generated from a month of data with 20 unique features, which include atmospheric information such as temperature, humidity, precipitation, and cloud cover. In this configuration, our quantization algorithm reduced memory

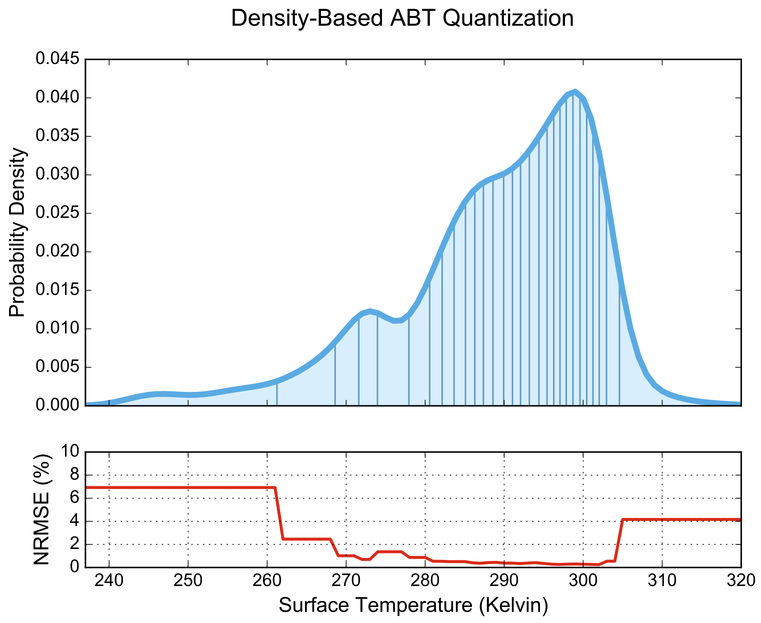


Figure 4.5: Quantization of surface temperature values across 29 tick marks. Each tick mark is indicated by a vertical line under the curve.

consumption by about 62.4%, which allows much more historical data and larger geographical areas to be maintained by Galileo.

Table 4.3: Tree statistics before and after our dynamic quantization algorithm over one month of data.

Metric	**Original**	**Quantized**	**Change**
Vertices	3,104,874	1,238,424	-60.1%
Edges	3,367,665	1,441,639	-57.2%
Leaves	262,792	203,216	-22.7%
Memory	1,710.6 MB	643.1 MB	-62.4%

4.1.4 Modeling Temporal Intervals

While single points in time can be expressed easily through a hierarchy that accounts for various time units (years, months, days, and so on), a timespan represented by a start time and end

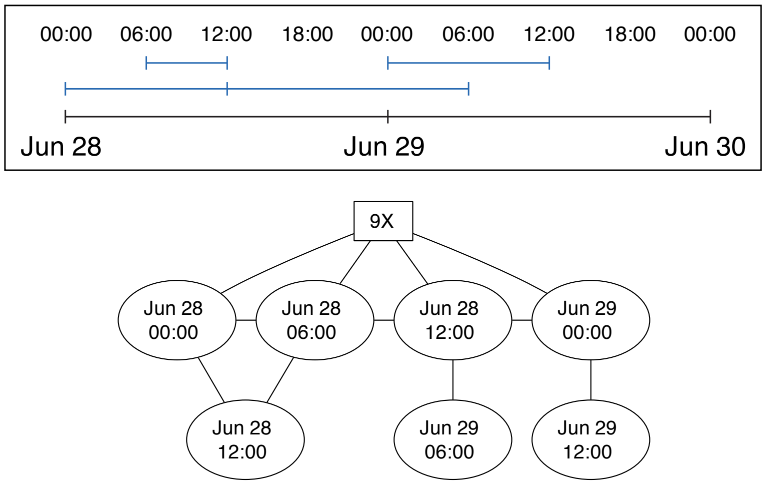


Figure 4.6: Time intervals from samples in a spatial region (represented by the Geohash 9X) across a time frame of two days, along with its tree-based equivalent for indexing purposes. Further descendant feature vertices are excluded for brevity.

time requires a data structure that can handle two-dimensional boundaries. Additionally, a tree based on time units requires the insertion of a large number 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 tree (represented as POSIX time values): 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 tree. Figure 4.6 illustrates how time series data is incorporated into the ABTs in Galileo. While many time series datasets that were 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 ABT, 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 tree with time intervals that start after T_{end} and end before T_{start}.

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:

```
    Year=2013, Month=*, 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 ABT as data is generated. To manage the creation of vertices as they accumulate over time, we rely on our dynamic reconfiguration algorithm with time-specific directives for collapsing older or rarely-accessed intervals. Without specific accommodations for time, ABT 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 (values evenly divisible by five, for example) when tuning the tree to produce divisions between vertices.

4.2 Increasing ABT Precision

By quantizing indexed values and reducing the overall search space, the global ABT enables low-latency distributed queries across voluminous datasets. However, quantization with adaptive tick mark boundaries results in a reduction of index fidelity. Furthermore, simply locating relevant records is only the first step in most analytic processes; given a subset of the data, distributed computations or batch processing must occur to glean insights from the dataset. To address these use cases, the ABT incorporates knowledge extraction as part of the indexing process. As records are streamed into the system, the ABT maintains a variety of statistics at each vertex that describe the underlying data distributions and their interactions. Maintaining this information boosts index fidelity, greatly improves the speed of queries meant to generate synopses, and serves as a platform for development of additional functionality.

4.2.1 Vertex Statistics

By placing incoming data points into discrete bins or "tick marks," Galileo can index voluminous datasets quickly and evaluate queries without excessive memory consumption. While our autonomous tick mark reconfiguration functionality does provide some degree of insight into the data distributions, the ABT further improves expressiveness by calculating statistical synopses of the information that was placed under each vertex. The synopses include:

- Number of data points placed at the vertex
- Smallest and largest values observed
- Mean, variance, and standard deviations

Given that the range of values placed at a vertex is generally small, these statistics provide finer-grained insight into the behavior of the underlying data. Since new information is continually streaming into the system, we use the online method proposed by Welford [95] to avoid re-calculating these values each time a new data point is added. This involves maintaining the count of samples observed thus far, n, along with the current mean of the values, $\bar{x}$, and the sum of squares of differences from the current mean, S_n. When a new data point x_n is added at a particular vertex, the components are updated as follows:

$$\begin{aligned}
&\bar{x}_0 = 0, S_0 = 0 \\
&\bar{x}_n = \bar{x}_{n-1} + \frac{x_n - \bar{x}_{n-1}}{n} \\
&S_n = S_{n-1} + (x_n - \bar{x}_{n-1})(x_n - \bar{x}_n)
\end{aligned}$$

Besides the observation count and running mean, this enables calculation of the variance and standard deviation of the observed values:

$$\sigma^2 = \frac{S_n}{n} \qquad \sigma = \sqrt{\frac{S_n}{n}}$$

Our implementation of Welford's method also maintains the sum of cross products between features to track cross-feature relationships, such as the correlation between temperature values and humidity. These calculations are lightweight, enabling us to update statistics for each vertex affected by the insertion of a new multidimensional data point efficiently. In situations that require updates or removals, the running statistics can be adjusted by effectively reversing the procedure outlined above. When a query or system operation involves multiple vertices, the statistics from each vertex can be merged to provide an aggregate summary. A sample merge operation of vertices A and B follows:

$$\begin{aligned} n' &= n_A + n_B \\ \bar{x}' &= \frac{n_A \bar{x}_A + n_B \bar{x}_B}{n'} \\ S' &= \frac{S_A + S_B + n_A n_B (\bar{x}_A - \bar{x}_B)^2}{n'} \end{aligned}$$

Table 4.4 evaluates the performance of this algorithm by testing each operation over 1000 iterations and averaging their computational overhead in microseconds. Updating the statistics consumes the most CPU time because it triggers several calculations, but all five operations (update statistics, remove data point, calculate mean and standard deviation, merge two instances) complete in less than a microsecond on average; even while accounting for additional traversal overhead, updating an ABT with 100 unique features takes less than 1 ms.

Figure 4.7 contains a demonstration ABT populated with five records from our dataset that were extracted from Florida, USA, in July of 2013. Each record contains three dimensions: temperature, cloud cover, and humidity. The feature values of the paths are shown in Table 4.5, identified by the letters A through E. Note that each level in the tree contains a particular feature type, and that the summary *count* at each level sums to the total number of data points seen at the parent vertex.

Table 4.6 provides an intuition for how summaries in the ABT change over time by updating each statistic at the 50-100% cloud cover vertex (marked Δ in Figure 4.7) as the paths in Table 4.5 were inserted. Column **A** contains the vertex state with only one data point (66%). Note that the

Table 4.4: A performance evaluation of vertex statistics maintenance operations, averaged over 1000 iterations.

Operation	**Mean Time (μs)**	**Standard Deviation (μs)**
Add Data Point	0.942	0.137
Remove Data Point	0.913	0.126
Calculate Standard Deviation	0.416	0.023
Calculate Mean	0.339	0.025
Merge Instances	0.279	0.018

Table 4.5: Sample multidimensional data points used to create the ABT shown in Figure 4.7.

Path	**Temperature**	**Cloud Cover**	**Humidity**
A	295.4 K	66.0%	88.0%
B	296.6 K	64.0%	100.0%
C	294.5 K	72.0%	71.0%
D	293.9 K	20.0%	79.0%
E	294.1 K	100.0%	85.0%

count increases up until column **D**, where the path insertion did not involve the particular vertex being used for demonstration. Once path **E** has been inserted, the summary statistics match those shown in Figure 4.7.

4.2.2 Managing Vertex Specificity

Merging summary statistics across vertices enables Galileo to provide both a coarse- and fine-grained vantage point over the entire dataset: when a linear correlation or event of interest only occurs under specific conditions, statistics at the appropriate leaf vertices capture relationships without being influenced by the rest of the data. On the other hand, vertices near the root of the

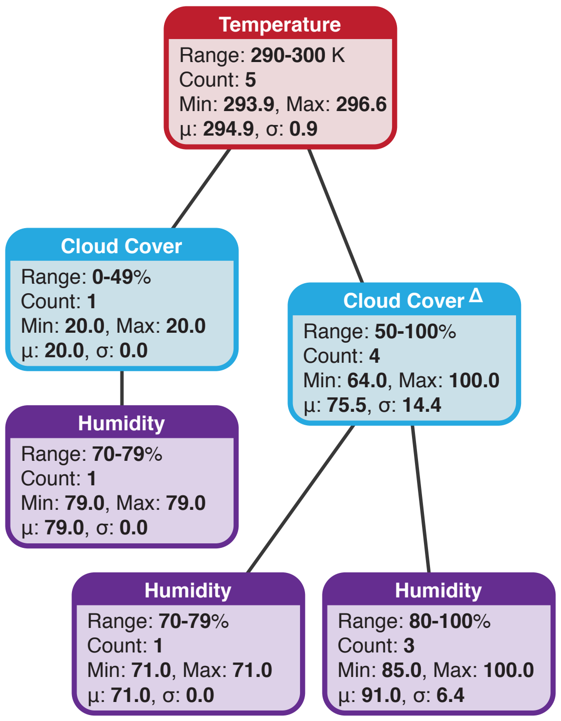


Figure 4.7: A sample ABT populated with multidimensional data points from Table 4.5. File/node pointers are omitted for brevity.

Table 4.6: A step-by-step view of ABT statistics at the cloud cover node handling values from 50-100% (denoted by Δ in Figure 4.7) as each successive path from Table 4.5 is inserted into the tree.

Statistic	**A**	**B**	**C**	**D**	**E**
Count	1	2	3	3	4
Min	66.0	64.0	64.0	64.0	64.0
Max	66.0	66.0	72.0	72.0	100.0
Mean (μ)	66.0	65.0	67.3	67.3	75.5
St. Dev. (σ)	0.0	1.0	3.3	3.3	14.4

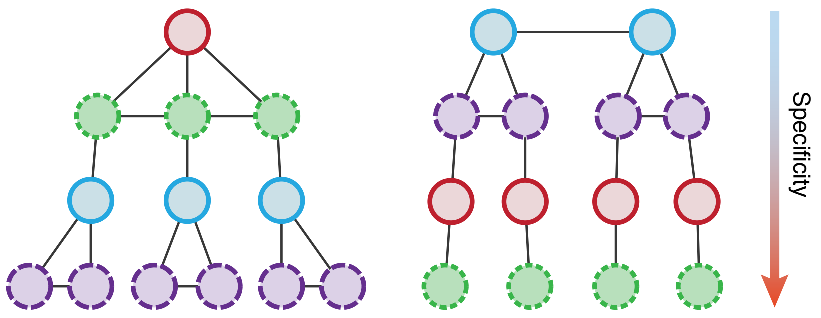


Figure 4.8: Two ABTs modeling the same data with different orientations. Vertices near the top of the hierarchy contain generalized information, whereas vertices closer to the bottom are highly specific.

tree tend to capture highly general information about the subtrees beneath them. By managing the inherent trade-offs of this *specificity gradient*, we can ensure accuracy is maintained while improving query response times and overall system performance. Figure 4.8 illustrates this concept, where two sample trees have been constructed using the same data but different orientations. Vertices in the figure are highlighted with four different colors to represent four unique feature types; as vertex positions change their respective counts change accordingly.

ABT orientations are configured manually by the user or dynamically based on usage trends observed by the system. This requires reorienting the tree to place the desired fine-grained synopses at the leaf nodes, while the coarse-grained synopses are placed near the top of the hierarchy. For example, if a large quantity of queries involves the average temperature of a region, placing temperature vertices at the top of the hierarchy would avoid traversing further through the tree. On the other hand, if a feature is always accessed along with many other features (such as a request for subtrees containing a snow depth greater than 1 cm, temperatures less than 0° C, and a time span of November through February) then it should be placed towards the bottom. Galileo maintains a table of query *feature sets* to monitor these trends. Additionally, ensuring the tree can be reoriented requires each leaf node to maintain *path synopses* that track each incoming path so vertex statistics can be redistributed during a reorientation.

Table 4.7 highlights the differences in creation time, vertex count, and edge count between three different tree orientations (listed hierarchically, with features abbreviated ***H***umidity, ***T***emperature,

Wind speed, and ***S***now depth). The tree 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 4.7: Global ABT sizes and creation times using different orientations.

Orientation	Creation Time (ms)		Vertices	Edges
	Mean	Std. Dev.		
H T W S	58.80	3.41	54,582	84,423
W T S H	49.99	2.81	31,706	61,547
S W H T	50.97	3.43	33,283	63,124

Figure 4.9 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 tree orientation has its own set of trade-offs that must be managed to achieve ideal performance, and these trade-offs continually change over time.

4.3 ABT Retrievals

While the summary statistics and metadata in the ABT are lightweight and incur minimal processing costs, they provide nuanced insights about the underlying dataset. Galileo includes rich retrieval functionality to allow this information to be queried and used individually by end users, aggregated across dimensions, or used to locate phenomena and features of interest autonomously. Queries are implemented as dynamic MapReduce [23] computations that are *pushed* out to relevant storage nodes for parallel evaluation. A query may involve subsets of the ABTs or outputs from analytic models; Figure 4.10 provides an example of both.

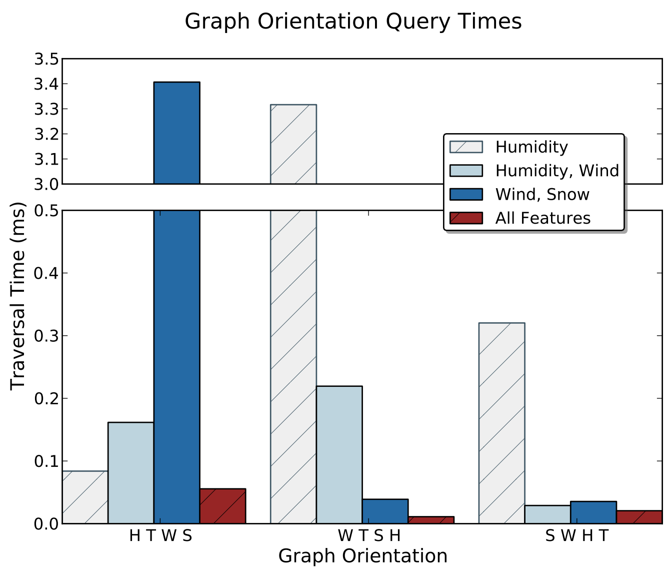


Figure 4.9: Traversal times for random queries across three different global ABT orientations. Queries ranged from broad (one value specified) to explicit (all feature values specified). Test permutations were executed 1000 times.

4.3.1 Result Datasets

As noted previously, the first step of a distributed query in Galileo is to eliminate nodes that do not contain relevant data from the search. This involves performing a lookup on the global ABT, which receives state updates that are gossiped through the network in an *eventually consistent* manner. While the search space reduction process does not guarantee that the remaining nodes will always contain relevant information, it does not produce false negatives. An additional benefit afforded by the summary statistics maintained at each vertex is that irrelevant storage nodes can be detected with greater accuracy. For example, consider a query requesting records with humidity

```
QUERY Humidity, Temperature WHERE
Date IN [Feb 2014 TO Jun 2014]
AS SUMMARY_GRAPH
```

```
PREDICT USING 'Temp-DJ_ARIMA'
WHERE Date =
    [Jun 15 2013 TO Jun 30 2013]
AS TABULAR
```

Figure 4.10: Example queries and requested output types. The first query retrieves temperature and humidity statistics, whereas the second issues a model query to predict temperature values at Geohash *DJ*.

values greater than 32%. In this case, a vertex that is responsible for data points ranging from 25-35% produces a positive match. However, if the vertex in question has a minimum value of 30%, the false positive can be avoided.

After the initial pruning process, queries are pushed to candidate storage nodes for evaluation in the *Map* phase. Query results are represented by traversable *subtrees* or serialized model state information, which is then merged by a subset of the nodes in the *Reduce* phase before being streamed back to the client. ABT statistics facilitate the reduction phase by providing additional information about the data distributions of the query; if a single storage node contains a large percentage of the records being queried, it is selected as a reducer to decrease serialization overhead and network I/O.

The reduce phase operates in one of three modes depending on the end user's goals, requirements, or hardware constraints. These modes include the *tabular layout*, *traversable tree*, or *summary tree*. Modes are selected during query submission, and each has its own set of trade-offs and use cases:

- A ***Tabular Layout***, which provides raw outputs with minimal post-processing occurring in the reduce phase. Results are streamed as table rows (similar to a CSV file or entries in a relational database) directly to clients.
- ***Traversable Trees*** that retain relationships between records. Tree paths that appear more than once are represented as a single path, reducing output sizes.
- ***Summary Trees*** that merge ABT vertices at the same level in the feature hierarchy to summarize the data with fewer records.

The differences between each of these output types are shown in Figure 4.11. Tabular records are best suited for model outputs or streaming applications that prioritize fast responses, while the traversable trees offer additional data inspection and exploration features at smaller output sizes. Summary trees are ideal for fast, ad-hoc exploration and analysis of the feature space, allowing large quantities of data to be sifted and represented in a compact fashion.

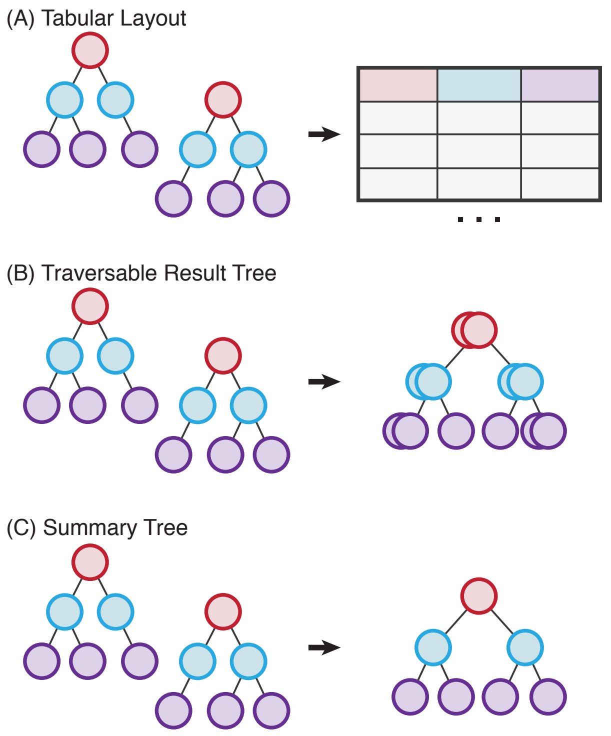


Figure 4.11: Output dataset types supported by Galileo. Intermediate results (left) are transformed by the multi-stage MapReduce process into the final output data structures (right).

Global ABT 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, an ABT query can take as little as 3.2 ms on average to report a negative result (zero matching files). Table 4.8 provides latency statistics across a broad range of randomized queries evaluated by the global ABT, 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 vertex for every result, approximate queries describe a range of values and their respective densities in a single vertex, resulting in faster response times.

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 tree called *path prediction*. This functionality is made possible by generalizing incoming queries across vertex ranges into

Table 4.8: Approximate and rigorous query timing results, averaged over 1000 iterations.

Results	Approximate Query (ms)		Rigorous Query (ms)	
	Mean	**Std. Dev.**	**Mean**	**Std. Dev**
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

orientation-neutral paths, and then testing to determine if the path is present in the tree using a Bloom filter [96]. 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.

Bloom filters are backed by a bit vector and several hash functions. When a new observation is inserted into the filter, it is hashed by each function to produce multiple output values. These outputs are then mapped to the bit vector as array positions, with each position updated and set to a **1**. To determine whether an observation was previously inserted, it is hashed and the corresponding array positions are checked. If each position is set to **1** then the observation *may* have been seen previously; if not, then the bloom filter will return a definitive negative result.

Path prediction involves complete tree paths rather than individual elements and is also applied on vertex ranges rather than discrete values. This allows the bloom 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 global ABT, and is resized or reconfigured during dynamic dimensionality updates. 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 used by the bloom filter stays fixed, as well as the raw path data being of similar sizes, the time consumed checking for a non-matching path stays consistent as long as the number of dimensions being indexed does not change. In our test environment, a path prediction could be performed in as little as 0.01ms, with a standard deviation of 0.0001.

4.3.2 ModelTrees

While the ABT provides a wealth of information about the underlying dataset, its interfaces and functionality also form the basis for more advanced models that facilitate analytic queries. Both exploratory and predictive queries (discussed extensively in the following chapter) build on this technology, with online models ranging from correlation analysis and two-dimensional linear regression to neural networks and ARIMA forecasting. Exploratory queries often involve the discovery of unknown patterns or attributes in the dataset expressed as two-dimensional relationships, so we ensure our summary statistics instances are available at every vertex. On the other hand, predictive queries are intended for situations that require guidance from the user to define the problem domain; while a multiple linear regression or ARIMA model could incorporate every reading received at a storage node, targeted insights are usually much more valuable. Predictive models involve two major phases: *creation* and *maintenance*.

During the creation of predictive models, *ModelTrees* are used to constrain the scope of model inputs to particular subsets of the ABT. User-defined queries filter and select the vertices that compose a ModelTree, resulting in a bounded "view" of the overall dataset. In cases where historical data is required for training, creation of a ModelTree may involve disk accesses. Once the ModelTree is created, nodes under its purview will trigger the maintenance process as they receive new observations. Maintenance intervals determine how often the models are updated, which can be continuous, periodical, or performed based on the availability of a specific number of observations. By default, models are constrained to the same size geographical regions as storage node groups

(1030 $\times$ 620 km in this study) and 12-month time spans, but any geographical, chronological, or feature scope can be incorporated.

Limits on both the scope of model inputs and their lifetimes can be specified by the system administrator to manage resource usage. By default, ModelTrees that have been unused for more than 24 hours are automatically garbage collected and removed from main memory. However, state variables and model calculations generally require consistent and predictable amounts of memory and CPU time, so resource limits can be configured to allow certain models to be maintained for longer time spans (or even indefinitely, with removal performed manually). If storage space is available, these models can also be serialized to disk for future use. In situations where models incorporate future observations, their lifetimes are specified upfront during creation.

To distribute workloads and exploit data parallelism, models with inputs that span multiple storage nodes are maintained separately and can also be queried independently to analyze how different geographic regions or distributions of the data impact the models. However, the most common query pattern involves merging model instances prior to their use to generate an aggregate model. This process is carried out in parallel by our MapReduce framework.

4.3.3 Query Performance Evaluation

To evaluate ABT retrieval performance, we ran an end-to-end benchmark that queried a random 20 by 30 kilometer geographic region with random temperature, humidity, wind speed, and snow depth features. Once submitted to the system, the node receiving the query determines relevant storage nodes by traversing the global ABT and returns a list of nodes to the client. Using the list, the client submits requests to the cluster and waits until the results have been received, which concludes the test. Table 4.9 provides timing information for the initial query that results in a traversal through the ABT as well as the time it takes to complete the query and transfer the requested data to a client. The mode of the number of machines contacted is reported as well; it is clear that adding more machines incurs additional latency during a query operation. Since the queries in this test were random, it provides a good estimate of the worst-case performance of the system for our particular dataset.

Table 4.9: End-to-end query test results. Each result is averaged over 100 runs.

Dataset	**Initial Query**		**Nodes**	**Complete Query**	
Size	**Time (ms)**	**Std. Dev. (ms)**	**Contacted**	**Time (ms)**	**Std. Dev. (ms)**
1	0.202	0.001	1	0.89	0.33
250	0.203	0.001	2	22.59	3.72
500	0.203	0.001	3	28.31	4.02
1000	0.204	0.002	6	85.07	11.23

To complement our end-to-end benchmarks, we also compared the read and write throughput of Galileo to Apache HBase [35] version 0.92.1, an open-source implementation of Google BigTable [30]. We ran HBase on Hadoop [31] and HDFS version 1.0.3. 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 HBase Java API to communicate with the system rather than the interactive shell or a third-party interface.

Table 4.10 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 global ABT; by contacting fewer nodes, Galileo can provide fast retrievals.

Table 4.10: Read throughput comparison: Galileo vs HBase (averaged over 100 iterations).

Number	**Galileo**		**HBase**	
of Blocks	**Read (ms)**	**Std. Dev. (ms)**	**Read (ms)**	**Std. Dev. (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 4.11, 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 4.11: Write throughput comparison: Galileo vs HBase (averaged over 100 iterations).

Number	**Galileo**		**HBase**	
of Blocks	**Write (ms)**	**Std. Dev (ms)**	**Write (ms)**	**Std. Dev. (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.

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 [97] 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 (explored in detail in §5.1.3)
- 20% Exact-match (equality) queries and 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 traversable result trees.

To stress the system under varying usage conditions, we also conducted tests that interleaved a configurable amount of storage and query operations. Figure 4.12 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 4.12 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 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 4.12: Response times observed during concurrent workload tests (averaged over results observed by the first 50,000 clients).

	Response Time (ms)		
Workload	**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

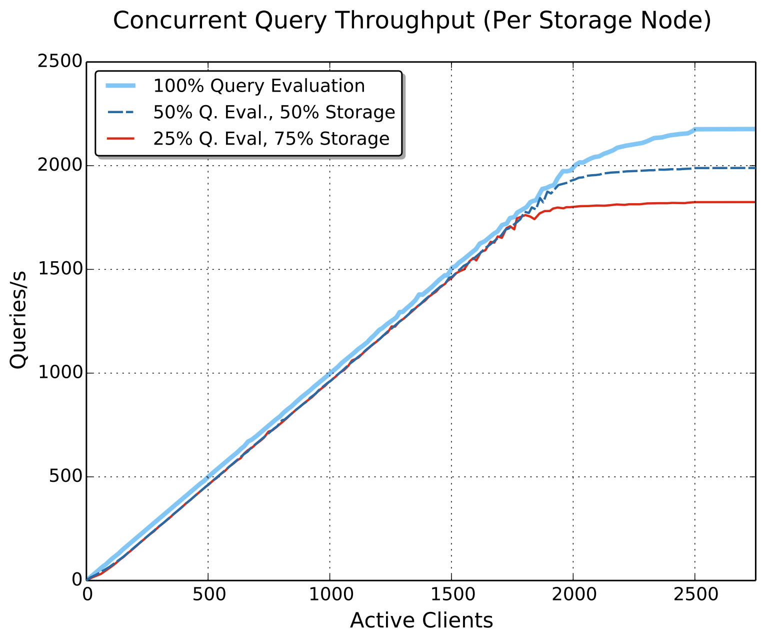


Figure 4.12: Query evaluation throughput at a single storage node as additional clients connect. Each client sends one query per second, with a configurable amount of storage requests interleaved.

4.4 Spatial Index: Geoavailability Grids

Spatial indexing over voluminous, distributed datasets poses several unique challenges; a central index server is a single point of failure and can become a bottleneck, but an index that is shared across all nodes in the system causes consistency problems and excessive state exchange. The R-tree [68] is commonly employed in non-distributed applications for spatial indexing due to its speed and efficiency, but has a number of constraining properties that limit its scalability in distributed applications. Using an R-tree as a global index for billions of files would require a substantial amount of memory, along with a high number of distributed updates due to the frequent rebalancing operations that take place within the tree. Additionally, splitting the tree across a number of nodes and designing a storage network around the data structure is constraining and latency-prone.

To overcome these scalability issues, we have developed the *geoavailability grid*, a distributed spatial indexing data structure that is scalable and fault-tolerant [12, 13]. Geoavailability grids translate points in space to a reduced-resolution coordinate system for indexing purposes. They consist of a vector of bits (represented by the set $\{0, 1\}$) for a given spatial area. Each bit represents a location, and its on-off state indicates whether or not information has been stored there. Due to their concise nature, bitmap indexes have seen considerable research and usage in relational database systems [98], decision support systems [99], and data warehousing [100].

4.4.1 Geocoding

We leverage the Geohash [11] algorithm not only for data partitioning, but also to provide a coarse-grained representation of spatial entities in the system. Quad Trees [101] or the OpenPostcode [102] algorithm could be used to achieve similar results. Using a geocoding algorithm is an essential component of our spatial indexing scheme because it determines the ranges of information that must be stored in each instance of the index. With our test dataset, the first two Geohash characters of a spatial location are used to determine the group of nodes responsible for storing the data. This has two benefits: specifying the first two characters (10 bits) of a Geohash can significantly reduce the search space for spatial queries without additional indexing, and it also means that individual nodes can exclude information from their geoavailability grids that lies outside their geographic scope.

4.4.2 Generating the Spatial Index: Geoavailability Grids

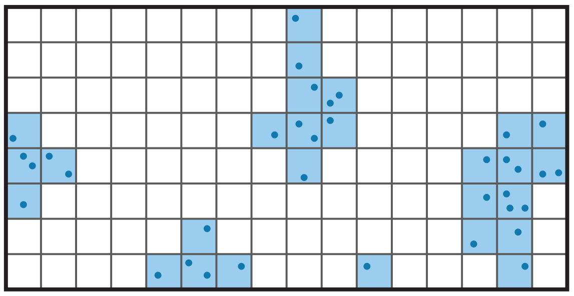
Geoavailability grids are initially configured with a width and height based on geocoding granularity. For example, if a gridded dataset contains readings at intervals of around 30 km, approximately 32 bits of Geohash precision would be required to place samples in separate bins. Choosing an appropriate granularity is highly dependent on the type of information being stored and the intended analysis that will be performed on the data. Finer-grained resolutions allow more specific queries to be resolved, but also increase the overall size of the index.

Each feature of interest (such as humidity or temperature) is accompanied by a unique geoavailability grid on a per-node basis, enabling queries to distinguish between different feature types. For situations where there is a reading present for every type, a catch-all geoavailability grid is generated autonomously to reduce the overall number of bits set in each grid.

For our test dataset, spatial locations are represented by 30-bit Geohashes. After accounting for the first 10 bits that are used to determine group membership, the remaining 20 bits are used to populate the geoavailability grids on each node in the system:

$$9xct1q = \underbrace{\text{01001 11101}}_{\text{Group Hash}}\ \underbrace{\text{01011 11001 00001 10110}}_{\text{Location in Bitmap Index}}$$

This is accomplished by mapping spatial coordinates to their closest bitmap coordinates, and ensuring that the relevant bitmap location is set to a **1** to indicate that one or more data points are present in the location. 20 bits of precision corresponds to 2^{20} Geohash buckets, which is the total number of bits in each index instance. Since Geohashes interleave latitude and longitude values, the width and height proportion of the index changes with each additional bit. Therefore, an index of n Geohash bits would have a width of $2^{\lfloor n/2 \rfloor}$ and a height of $2^{\lceil n/2 \rceil}$. Figure 4.13 illustrates how an example geographic region could be represented as a geoavailability grid.



```
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0
1 0 0 0 0 0 0 1 1 1 0 0 0 0 1 1
1 1 0 0 0 0 0 0 1 0 0 0 0 1 1 1
1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0
0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 0
0 0 0 0 1 1 1 0 0 0 1 0 0 0 1 0
```

Figure 4.13: Data points in a geographic region (left), with the corresponding geoavailability grid (right).

4.4.3 Compressing the Index

Each node in the system manages several index bitmaps for the geographic region under its purview. The bitmaps are distributed across all the nodes in the cluster to ensure that (1) every node in the system is capable of servicing queries, and (2) a node failure does not affect lookup capabilities. This means that the memory consumed by the bitmaps must be low; continually exchanging large amounts of information between nodes is inefficient and reduces the speed of index propagation.

While bitmap indexes support a wide variety of data types, the sheer number of bits required for these representations can prove to be problematic both in memory consumption and processing times for bitwise operations. Extensive research has been conducted on compressed bitmaps, from simple run-length encoding to more advanced schemes such as CONCISE [103] or Word-Aligned Hybrid (WAH) [104]. In this work, we use the Enhanced Word-Aligned Hybrid (EWAH) compression scheme [105, 106] to reduce the effective sizes of our bitmaps. EWAH was chosen due to the compression ratios it achieved on our dataset and its relative speed. Table 4.13 illustrates the difference between uncompressed and compressed bitmap representations for our **entire** dataset. For gridded data, higher resolutions (derived from the second half of the Geohash bits) increase the sparsity of the index and improve compressibility.

Compressed bitmaps are somewhat unique in that they generally do not require decompression before processing occurs. In fact, compression can often speed up bitwise operations. Due to the differences in performance observed across the available bitmap compression algorithms, we pro-

Table 4.13: Bitmap Compression results for 15-, 20-, and 25-bit Indexes

Resolution	Original Size (KB)	Compressed Size (KB)
15-bit	309.0	294.4
20-bit	9879.02	3196.9
25-bit	316090.28	4034.7

vide a universal interface that allows the underlying bitmap representation of a geoavailability grid to be changed at runtime or during initial system configuration – one such alternative is Roaring Bitmaps [107].

4.4.4 Updating the Geoavailability Grid

To ensure that new files' spatial information is disseminated rapidly, geoavailability grid updates are gossiped between groups on a regular basis along with other state information. An update consists of a set of bits that have changed since the publication of the previous update. If a storage node finds itself out of sync with the current updates, neighboring peers can also generate an update or transmit an entire copy of the index. The update interval and maximum number of unpublished updates are configurable parameters depending on network capacity and desired consistency characteristics.

Creating an update involves performing an XOR ($\oplus$) operation on the current index i_c and the previous index state i_p, which produces a set of update bits u that will be relayed to peer nodes: $i_c \oplus i_p = u$. When an update is received at a peer, it can be applied by performing an XOR between the update and the current copy of the remote index ($u \oplus i_p = i_c$). The latest index version number and a checksum are also included in an update message to help ensure consistency. Updates are compressed in the same manner as the indexes, meaning that they generally consume a minimal amount of space; Table 4.14 contains update sizes (including the version number and checksum) for different amounts of new data points that were added in random spatial locations over a 20-bit

Geohash region. A single update will always have a size of 36 bytes since only one bit is set in the compressed bitmap.

Table 4.14: Geoavailability grid update sizes, averaged over 1000 iterations

Modifications	Mean Update Size (bytes)	Standard Deviation (bytes)
1	36.0	0.0
10	179.9	1.0
100	1609.3	10.8
1000	15089.0	98.4

Random updates represent an approximate worst case in terms of message size because the likelihood of an existing bitmap location already being occupied is low; after all, new files that fall within a spatial range already covered by the geoavailability grid do not require any updates to be transmitted. With our test dataset, storage nodes were responsible for approximately 5000 unique bitmap locations on average over the lifetime of the cluster, which would only require about five 1000-bit updates.

One contributing factor in the efficiency of bitmap representations is the bit-level parallelism that can be exploited while evaluating bitwise operations. This property speeds up the creation and application of updates; considering the update sizes and their processing times, keeping the geoavailability grid up-to-date is mostly a function of network latencies. To demonstrate this, we created and applied updates of size 1, 10, 100, and 1000 and averaged the results over 1000 trials. Table 4.15 demonstrates these processing times. In general, the number of modifications has little effect on processing times for both the creation and application of the updates.

Table 4.15: Update creation and application times for bitwise operations, averaged over 1000 iterations

	Creating		Applying	
Modifications	**Time (ms)**	**Std. Dev.**	**Time (ms)**	**Std. Dev.**
1	0.32	0.03	0.18	0.05
10	0.32	0.04	0.19	0.01
100	0.33	0.11	0.20	0.08
1000	0.33	0.08	0.19	0.03

4.5 Spatial Retrievals

Once spatial data has been indexed in geoavailability grids at each storage node, the system can evaluate user-defined geospatial queries derived from polygons surrounding areas of interest. Queries can also contain ranges or exact values of various features to help further reduce the search space. Geospatial query evaluation in Galileo proceeds as follows:

1. A user submits a polygon to retrieve data from.
2. The query is *decomposed* into a number of subqueries by intersecting it with the geometry each group of nodes is responsible for.
3. The geoavailability grids are consulted to determine if data may be available, eliminating any irrelevant nodes.
4. Specified feature values or ranges are queried from the global ABT, allowing additional reduction of the search space.
5. Subqueries are submitted to the remaining set of relevant nodes for evaluation.

4.5.1 Spatial Decomposition

Each group in Galileo is responsible for storing data pertaining to a particular geospatial region. These regions are known by the other nodes in the system and maintained in memory as polygons.

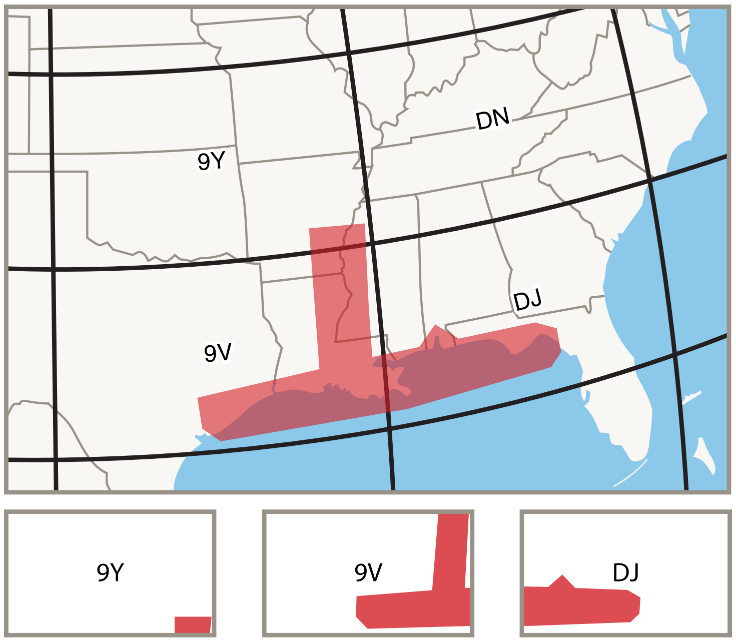


Figure 4.14: A query following the Mississippi river through the Louisiana area and along the shoreline in the Gulf of Mexico (shown in red) decomposed into three subqueries across Geohash group boundaries.

To begin decomposing a spatial query, the minimum bounding rectangle (MBR) is calculated for the query geometry, which is the smallest rectangle that completely surrounds the query polygon. Any group geometries that are overlapped by the query MBR are then intersected with the query polygon. After the intersection operation, the remaining geometries are used to produce a set of groups that are relevant to the query. Figure 4.14 illustrates this procedure.

Decomposing queries in this manner has multiple advantages. Small queries will naturally involve fewer storage nodes, whereas larger queries that are represented by polygons spanning greater geographic regions are processed in parallel across a number of groups. After decomposing the query, geoavailability grids can be consulted to further reduce the search space.

While individual storage nodes can quickly query their internal data structures to determine if relevant information is available, this step completely eliminates communication with nodes that definitely do not contain pertinent data, avoiding any related network latencies and reducing overall cluster load.

4.5.2 Geoavailability Evaluation

Before being evaluated against the collection of pertinent geoavailability grids, query polygons must be projected onto a corresponding bitmap coordinate system. Once this process is complete, a *query bitmap* is created using the polygon geometry.

To create a query bitmap, the spatial area covered by the geoavailability grid can be thought of as a monochrome graphical canvas that will be drawn using standard graphics routines; using the provided query polygon, the regions of interest are filled with color to set the relevant bits within the polygon boundaries to **1**. This effectively converts a user-provided polygon into a geoavailability grid by leveraging existing graphical algorithms and any hardware acceleration available to the system. In cases where large bitmaps are being generated, GPUs can be leveraged if support is available. Another benefit of this strategy is that queries can easily be visualized. Table 4.16 contains information on how long it takes to generate a query bitmap based on index resolution, which is the primary factor involved when drawing query geometry at scale. For this benchmark, the subquery polygon shown in Figure 4.14 for Geohash area *9V* was converted into a query bitmap.

Table 4.16: Query bitmap creation times, averaged over 1000 iterations

Resolution	Bitmap Generation (ms)	Standard Deviation (ms)
15-bit	0.45	0.36
20-bit	1.81	1.14
25-bit	56.31	4.78

While the exact precision of a Geohash is largely dependent on its particular location on the globe, a precision of 25 bits equates to a region of about 230 by 150 meters. For coarser-grained query bitmaps, the storage and processing times decrease. Figure 4.15 illustrates the benefits and drawbacks associated with increasing or decreasing query bitmap resolution; while false positives

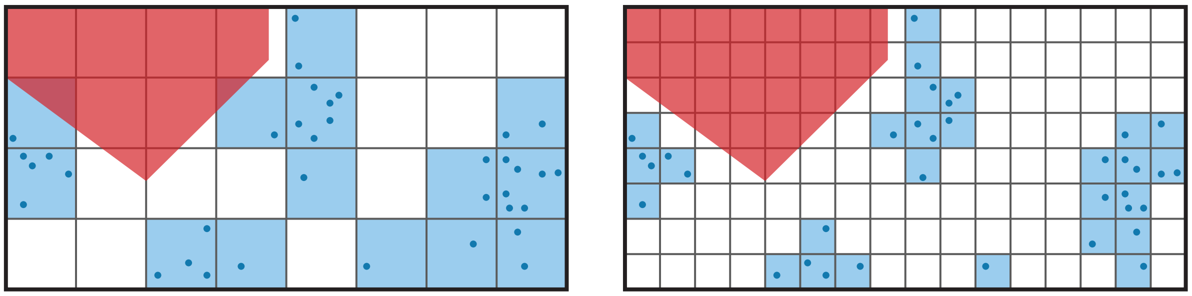


Figure 4.15: Two different query bitmap granularities and an example query polygon. Finer-grained bitmaps generally increase the amount of search space reduction that can be achieved at the cost of consuming more memory.

are less likely with a higher resolution, the memory consumed by the geoavailability grids also increases.

Once a query bitmap has been obtained, evaluating the presence of relevant data within the polygon boundaries is simple; a logical AND ($\wedge$) is performed between geoavailability grids (GG) and the query bitmap (QB). If the resulting bitmap contains *any* bits set to **1**, then there was a region with relevant spatial data that overlapped the query geometry, and the subquery is passed on to relevant storage nodes. In other words, if $GG \wedge QB = \{\}$ for a particular storage node, then it can safely be eliminated from the search without requiring any communication. Table 4.17 contains timing information for processing a geoavailability lookup, which involves several AND operations.

Table 4.17: Geoavailability evaluation speed, averaged over 1000 runs against each group Geohash

Resolution	Lookup Time (ms)	Standard Deviation (ms)
15-bit	0.012	0.021
20-bit	0.163	0.203
25-bit	0.723	0.289

A complete geoavailability evaluation returns a set of storage nodes that contain spatial information within the query boundaries. This set is intersected with results from a global ABT lookup of any other constraints specified by the user, which further reduces the search space by eliminating any destinations that cannot satisfy the entire query. *Subqueries* are submitted to the remaining set of storage nodes for retrieval from local indexes.

4.5.3 Proximity and Nearest Neighbor Searches

While the queries described thus far enable flexible retrieval of geospatial information, they do not account for situations where the desired geometry is not known at query time. This situation occurs frequently in geographic information systems (GIS) or cartographic visualizations; for instance, a user may wish to locate the nearest retail store that sells a particular range of products, but cannot estimate the breadth of his/her search. In these cases, the query geometry is unknown and the search is represented as a single starting coordinate pair. Additionally, proximity queries imply *priority* in their search results: the closest matching points should be returned first, followed by increasingly distant matches.

Given a pair of starting coordinates, proximity queries are evaluated by first checking the corresponding geoavailability grid location. If the bit in this location is set, then the search can be immediately evaluated by the storage node that contains information for the grid point. However, there are often situations where the coarse-grained geoavailability grid cell that was referenced does not contain matching information. In this case, circular query geometry is generated and centered over the starting coordinate pair. To broaden the scope of the search, progressively larger annuli (donut-shaped geometry) are evaluated against the geoavailability grids. This avoids re-querying areas that have already been inspected previously by the algorithm. Figure 4.16a illustrates the process behind evaluating a proximity query, with five iterations of the algorithm shown (including the initial grid cell check).

Each subsequent query annulus retrieves information that is streamed back to clients incrementally; the first results that a client receives will be the closest, followed by additional matches that

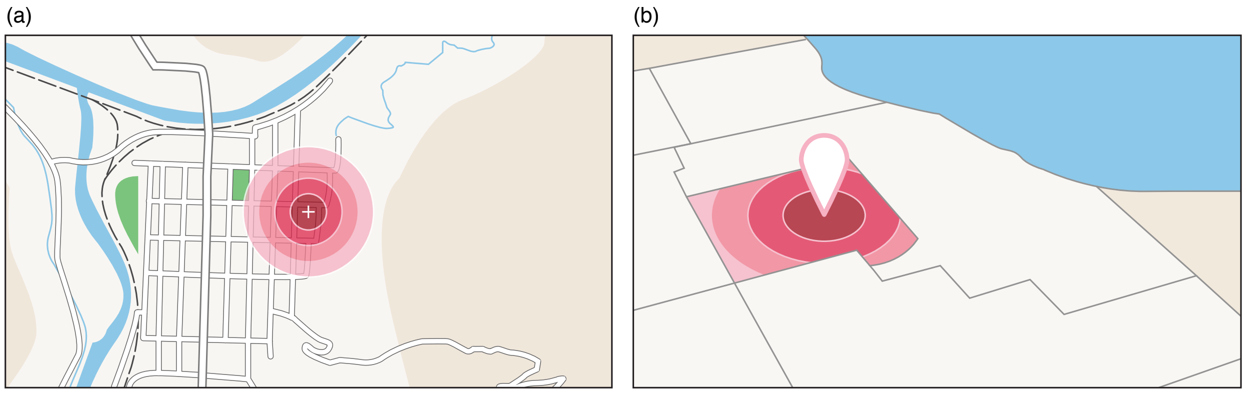


Figure 4.16: (a) Proximity query starting from a given point on a street corner (indicated by a cross) and radiating outward with progressively wider-ranging query annuli; (b) query with boundaries constrained to the geometry of DuPage County, Illinois, USA.

are located geographically further from the original coordinate pair. With 25-bit grids, 8 iterations of the algorithm over a 50km region took 5.35ms with a standard deviation of 0.41ms when averaged over 1000 runs. Using the same parameters on a single-node installation of MongoDB 2.4 [41], the operation took 56.95ms with a standard deviation of 4.53ms. One key performance advantage of our proximity algorithm is that it does not require results to be collected and sorted as a post-processing step.

This approach gives client applications flexibility: the query can be stopped when a particular set of constraints have been met, or the rate at which the query is broadened can be increased to produce more results. Alternatively, conditions for stopping the search can be provided at query time.

Geographically Constrained Proximity Queries

In circumstances that require a proximity search within a specific geographic region, we provide *geographically constrained* proximity queries. As with a standard proximity query, a geographically constrained search begins with an origin point and radiates outward to discover more information, but is limited to a particular set of geometric bounds. Geographically constrained proximity queries can take advantage of Esri shapefiles or datasets such as the US Census Bureau TIGER (Topologically Integrated Geographic Encoding and Referencing) [108] for defining spatial constraints. Figure 4.16b provides an example of a spatially-constrained proximity query in DuPage County, Illinois. While there may be matching data points in the neighboring counties, they are excluded from the search.

Graphics Pipeline Optimizations

Our algorithm for servicing proximity queries makes extensive and frequent use of the underlying graphics framework, which can be accelerated by a compatible GPU. However, there are costs associated with transferring information in and out of graphics memory that must be considered when enabling GPU acceleration. When starting for the first time, storage nodes perform a hardware benchmark to determine the trade-off space for memory latency and image generation times, and use the results to choose the fastest rendering technique available. In many cases, queries that span a small geographic region will be generated by the CPU, but operations on very large bitmaps will be offloaded to the GPU.

One factor that has a strong influence on bitmap generation times is the size of the geoavailability grids being used by the system. A larger geoavailability grid results in a proportionately larger "canvas" that will used for rendering query geometry, which consumes additional memory and processing time. In many cases, a query will span less than 1-5% of the entire geoavailability grid. To avoid the performance penalties associated with creating large images in memory, the maximum bounds of incoming query geometry are calculated and used to create a much smaller *query canvas* in memory that will be used for drawing operations. Once the canvas has been created, the query bitmap is generated and then shifted into place using geoavailability grid coordinates. This can provide significant performance gains; Table 4.18 compares the naive bitmap generation scheme with the optimized version for processing proximity queries with varying radii. Both algorithms were run on the CPU.

4.5.4 Local Retrieval: Geoavailability R-trees

Once a query has been decomposed and evaluated, it can be processed in parallel as a set of subqueries at relevant storage nodes. This process is facilitated by the local ABT for range-based

Table 4.18: Query bitmap generation times for proximity searches with varying radii, averaged over 1000 runs

Modifications	Creation Time (ms)	Standard Deviation (ms)
Naive Algorithm, 25 km	30.63	7.34
Optimized, 50 km	0.85	0.26
Optimized, 25 km	0.30	0.12
Optimized, 5 km	0.09	0.05

and exact-match queries of feature values, but additional indexing infrastructure is required to provide results for polygon-based spatial queries. The R-tree [68] is a fast and efficient spatial index that has seen considerable usage and development in the geospatial domain. Similar to B-trees, R-trees are a balanced search tree and organize data into pages. The basic unit of storage is a rectangle, so polygon-based shapes or queries are first converted to their minimum bounding rectangle (MBR) before use with the index.

R-trees provide an appealing alternative to using geoavailability grids for local retrieval because their accuracy is not based on a preconfigured resolution, and the average complexity of a lookup with maximum page size M is $O(log_M n)$ versus $O(n)$ for a bitmap representation. However, results returned from a geoavailability evaluation do not require post-processing to eliminate rectangles or points that were present in the query MBR but do not fall within the query polygon. Additionally, query bitmaps have already been generated for the geoavailability grids in previous indexing steps, so a local lookup only involves processing a number of AND operations.

To take advantage of the strengths of both approaches, we developed the *Geoavailability R-tree*, which stores geoavailability grid cells in the R-tree rather than individual points, significantly reducing the amount of information stored in the index. To investigate the performance of the geoavailability R-tree, geoavailability grid, and a standard R-tree, we used a polygon subquery in North America from our previous work that followed the Mississippi river down to the Gulf of

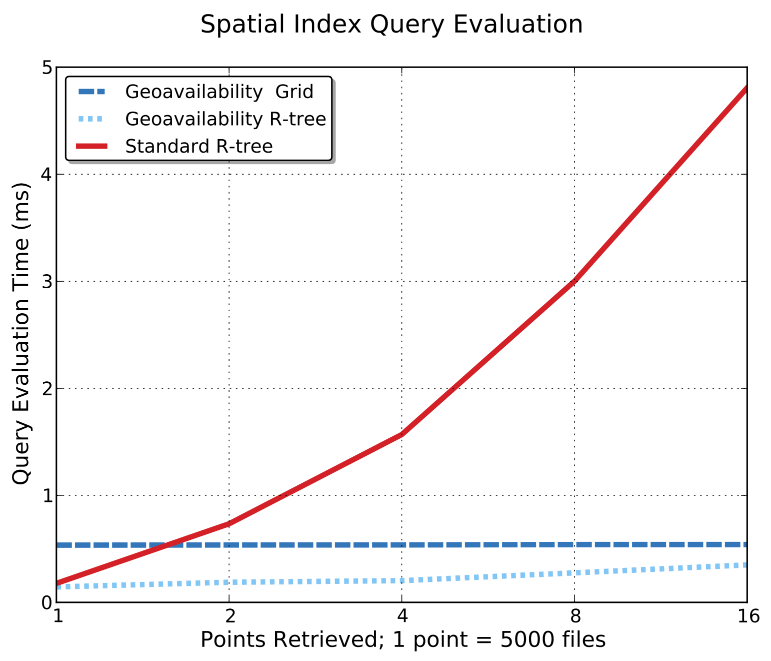


Figure 4.17: Comparison of geoavailability grids, geoavailability R-trees, and a standard R-tree for retrieval operations. Results are averaged over 1000 runs.

Mexico. The Java Spatial Index (JSI) [109] implementation of the R-tree algorithm was used in these benchmarks due to its focus on performance, and the geoavailability grids were configured with 25-bit precision. Through the course of the tests, the query polygon was upscaled to retrieve more files by including a larger portion of the dataset. Figure 4.17 illustrates the performance of all three strategies when retrieving points from our test dataset, with each point representing about 5000 files. While the standard R-tree cannot cope with large amounts of data points, the geoavailability R-tree provides the best overall performance in our benchmarks. However, using a geoavailability R-tree requires additional memory for each feature type, and also exhibits slower performance as the number of points retrieved increases. Conversely, evaluating a query with a geoavailability grid will generally require the same amount of processing time regardless of the size of the query geometry.

One of the strengths of our geoavailability grid implementation is bounded memory consumption: for a given resolution, there is a maximum size of the resulting bitmap whether compression is used or not. We tested the memory consumption of three different geoavailability grid resolutions and an R-tree to gain insights on their resource consumption. Our test dataset was divided among 48 storage nodes in this benchmark, resulting in approximately 20 million points being indexed and tested at each node. Table 4.19 contains the memory consumption results; while geoavailability grids require substantially less memory than the R-tree, it is worth noting that they will trade off some precision based on the Geohash granularity used.

Table 4.19: Memory consumption: geoavailability grids and JSI R-tree.

Implementation	**Memory Consumed (KB)**
15-bit Geoavailability grid	3.98
20-bit Geoavailability grid	44.37
25-bit Geoavailability grid	56.03
JSI R-tree	1120634.88

Chapter 5

Analytic Queries

Galileo supports three types of analytic queries: approximate, exploratory, and predictive. These queries allow fast, iterative exploration and manipulation of datasets to explore relationships, gain insights, and make predictions. After locating observations of interest, the system also allows subsets of the data to be exported in memory to other systems for further analysis. Overall, this functionality greatly reduces the friction between users and their data, leading to faster discoveries and encouraging more in-depth investigation.

5.1 Approximate Queries

Supporting exploratory or approximate queries requires a departure from explicitly-defined query parameters. In this work, we make the distinction between ***approximate*** 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.

5.1.1 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. Galileo provides a time-bound query operator for these situations, which provides 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 acting 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 level of 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 conditions. For instance, an application my request results within a particular time bound (e.g., 5 seconds), but may also elect to have the query complete faster if a minimum accuracy is achieved. This functionality is also 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.

5.1.2 Temporal Reasoning

Datasets consisting of time series information involve both 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.

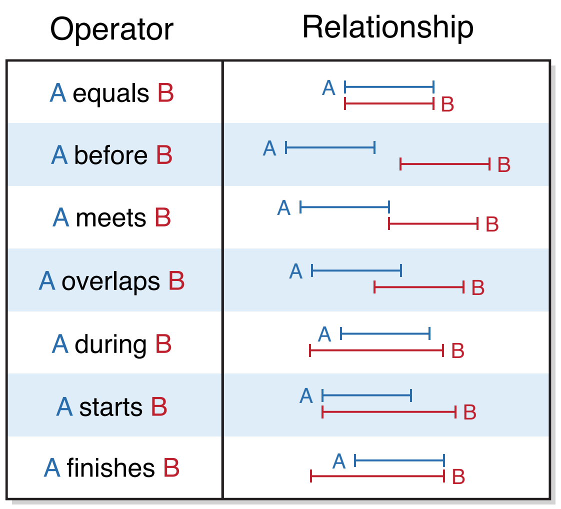


Figure 5.1: Temporal operators from Allen's Interval Algebra. These operators enable queries and analysis across intervals from time series datasets.

For reasoning about the relationships between disparate time intervals in a dataset, we integrated support for the operators described by Allen's Interval Algebra [110, 111] 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 [112] to temporal database management systems [113]. Figure 5.1 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.

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 trees that are returned by queries.

5.1.3 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 [4, 5]. 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 nodes in the tree 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.

5.1.4 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 trees can be returned during a query to track progress, followed by a complete result tree once the conditions for promotion are met.

5.1.5 Dataset Sampling

When a subset of available information will provide a representative model of the complete dataset, a *sample query* on the ABT can be evaluated to provide results in a fraction of the time a rigorous query would take. Each storage node can produce sample trees on demand, or can pre-populate (and update) several sample sizes that cover varying portions of the dataset. The amount of pre-populated sample trees 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 trees 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 result tree, or as a probability density function (PDF) generated through kernel density estimation. Sample queries can be combined with the 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 conditions are met.

Figure 5.2 compares 1% and 5% uniform samples (represented by their probability densities) 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.

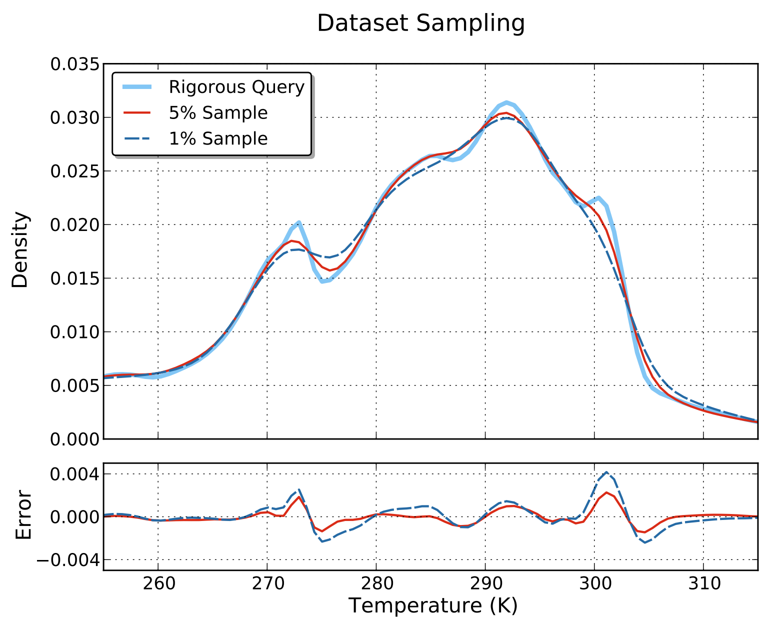


Figure 5.2: Comparison between the density of values in 1%, 5%, and rigorous queries, along with the error associated with both sample sizes.

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 global and local ABTs 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 trees. 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.

5.2 Exploratory Queries

Voluminous datasets often contain a multitude of relationships and insights that may not be obvious or intuitive. For this reason, we provide support for *exploratory analytics* functionality to pinpoint information, trends, and properties of interest in the underlying data. These components include online algorithms for generating both broad and specific statistical synopses, discovering correlations between dimensions, evaluating the significance of feature variations, and analyzing the probabilities associated with events.

5.2.1 Detecting and Quantifying Feature Relationships

Most real-world systems involve several interrelated features. These relationships may represent dependencies or correlations between events or variables; for instance, absolute humidity is impacted by temperature and pressure, and precipitation may be classified as rain, hail, or snow depending on the current temperature (along with other atmospheric conditions). While it is important to note that correlation does not imply causation, the relationships between variables may prove to be valuable from a research perspective and warrant further study.

To autonomously track correlations between features, we augmented the ABT by adding the capability to calculate the Pearson product-moment correlation coefficient (PCC) across 2D feature combinations. PCC measures the degree of a linear relationship between two variables, ranging from $[+1, -1]$. A correlation coefficient of $+1$ or -1 between variables represents a perfect positive or negative linear relationship, respectively, whereas a value of 0 would imply the absence of a linear correlation. Both sides of this spectrum can provide useful insights into how features interact and influence one another.

A correlation query is issued by specifying feature tuples of interest along with any additional constraints (such as limiting results to a specific region or time), and will return a set of correlation coefficients and their two-tailed p-values. p-values represent the probability of producing the same test results if there was no relationship between the features. In other words, a very low p-value (often less than 0.01 or 0.05, depending on user requirements) represents a significant relationship that is unlikely to occur by chance. Client applications have the option to request that these statistics be merged into a single set of correlation coefficients or returned as subtrees for further exploration.

Table 5.1 contains correlations found between several different features during the month of July in Wyoming, USA, along with their corresponding p-values to test the significance of the relationships. These results indicate a negative correlation between sky visibility and total precipitation, a positive correlation between humidity and precipitation, and a negligible relationship between temperature and ground vegetation. As one might expect, snow depth did not exhibit correlations with any of the other features during this summer month, highlighting the fact that both temporal and spatial aspects must be considered when drawing conclusions from the data.

Table 5.1: Correlations and p-values between several features recorded during July in Wyoming, USA.

Feature A	**Feature B**	**Correlation**	p**-value**
Precipitation	Visibility	-0.49	3.39E-39
Humidity	Precipitation	0.37	3.08E-22
Pressure	Visibility	0.36	1.61E-20
Vegetation	Temperature	-0.06	0.12
Temperature	Snow Depth	0.0	1.0

In addition to the correlation coefficient, we also maintain enough information to generate linear models between feature types. These models are provided by two-dimensional linear regression, and include the coefficient of determination, r^2, which measures the predictive quality of the linear models. Features that exhibit strong linear correlations can often be used to make accurate predictions. Our implementation follows the least-squares approach, wherein a straight line is fit to the data such that the sum of squared residuals is minimized. The slope and intercept of the regression line can be retrieved by clients, or the model can be used directly to extrapolate the value of a particular feature.

While updating vertex statistics is computationally lightweight, several more operations are necessary to provide the correlation coefficient, linear regression functionality, and r^2 values. Table 5.2 describes the performance characteristics of the two-dimensional vertex statistics management capabilities added to the ABT. Once again, adding new data points incurs an upfront cost, but subsequent calculations are fast: all operations were completed in less than 2 μs.

Table 5.2: Performance evaluation of the augmented two-dimensional vertex statistics mechanism, averaged over 1000 iterations.

Operation	**Time** (μs)	**Std. Dev.** (μs)
Add Data Point	1.489	0.044
Calculate Correlation	0.723	0.023
Calculate r^2	0.101	0.003
Predict y	0.381	0.016
Merge 2D Instances	0.919	0.103

5.2.2 Significance Evaluation

Regions in close spatial proximity often exhibit similar climactic trends, but variations in the surrounding geography may result in significantly different weather patterns. Similarly, the average temperature for a particular region will vary from year to year, but a researcher would be most interested in situations where the difference was *statistically significant*. While Galileo supports calculating the significance of the variations between two samples using p-values, we have also added functionality to enable the evaluation of queries that invert this problem: given a feature or set of features, a *significance query* locates relevant locations, time spans, or data points across the overall dataset where variations are statistically significant. The features in question are often chosen based information gained from previous exploratory analysis.

We implement significance queries as multi-stage MapReduce computations; each relevant storage node begins by evaluating a standard query with the given parameters, and then retrieves the statistical synopses of the feature in question from the resulting subtree. To reduce network I/O costs, the synopses are combined and transmitted to the node that was projected to contain the largest number of matching records by the initial ABT pruning step, which then performs *Welch's t-test* between synopses. A *t-test* uses the *t-distribution* to determine whether there is a relationship between observations, and Welch's version of this test is used in situations where the variances between two samples are possibly unequal. This produces a set of p-values that measure inter-group variance. In the final step, both the individual and group results are streamed back to the client based on a *p-value threshold* that is specified at query time based on the client's desired significance level. This reduces the amount of information that must be processed on the client side and enables users to quickly evaluate perceived relationships across the dataset. To test this functionality, we submitted queries involving randomized feature sets with significance levels of either 5% or 1% to our cluster. The results were produced in 41.0 ms, on average (over 1000 iterations) with a standard deviation of 0.13 ms.

5.2.3 Probability Density Queries

Feature analysis not only involves the values a particular feature has taken, but also the probabilities associated with them. For instance, high temperatures are more likely to occur during summer months and less likely during winter months. The probability densities associated with these events provide insight as to how features behave under specific conditions, as well as how they evolve over time. Figure 5.3 contains the results of a *probability density query*, illustrating the likelihood of measurable precipitation occurring as the amount of atmospheric cloud cover changed in Wyoming, USA during the month of July in 2013. While the presence of clouds being

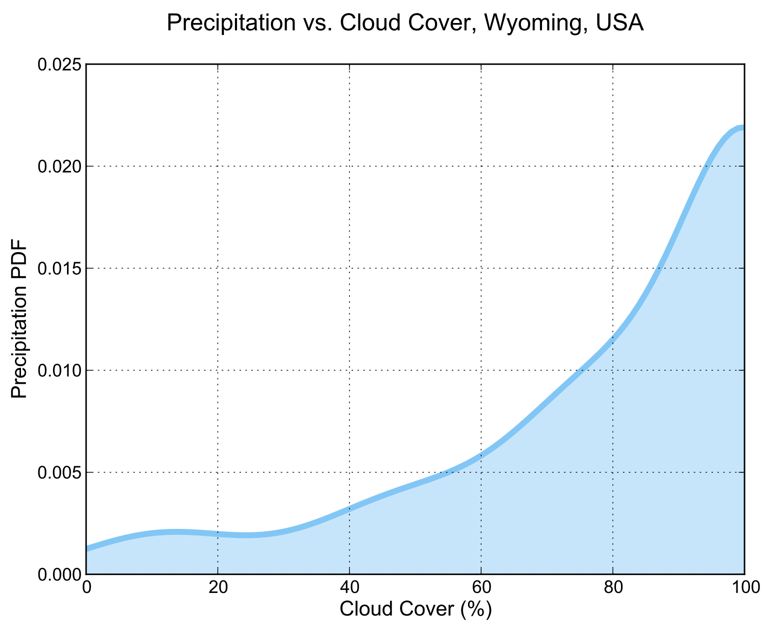


Figure 5.3: Results of a probability density query showing the probability of precipitation occurring as the amount of atmospheric cloud cover varies.

associated with a heightened probability of rain is a fairly intuitive relationship, this query yields the probability of precipitation as a function of cloud cover; with this information, we know that rain is highly unlikely when cloud cover is under 50% in this region. The next step in our analysis might involve inspecting the relationship between cloud cover and precipitation across a variety of spatial locations.

Client applications can issue probability density queries across ranges of feature values, geospatial regions, or time spans. For query shown in Figure 5.3, Galileo uses feature synopses in the ABT to determine the number of times precipitation was greater than zero under the specified constraints and produces a collection of vertices to be passed on to the reduce phase. During the reduce phase, vertices are merged to produce a final histogram that contains the counts for each tick mark range. The resulting set of vertices can either be returned directly to the client to produce a histogram, or passed through a *kernel density estimation* to compute the approximate probability density function. On our test dataset, this query involved a total of six storage nodes and completed in 18.26 ms (averaged over 1000 iterations, with a standard deviation of 0.11 ms).

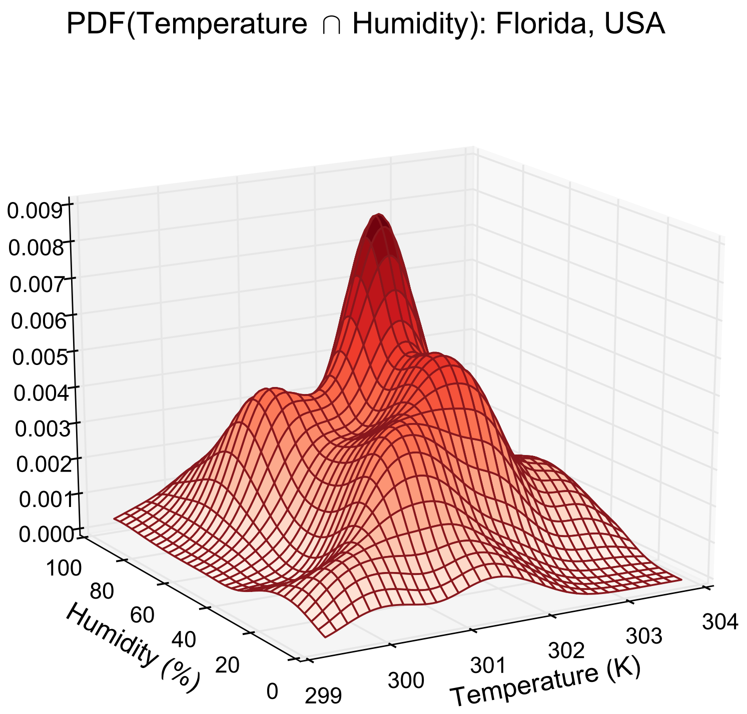


Figure 5.4: Joint probability density of temperature and humidity values in Florida, USA, during July of 2013.

5.2.4 Joint Probability Queries

To model cross-feature influences, *joint probability queries* provide the probability densities of multiple events or feature values occurring at the same time. For example, consider the joint influence of temperature and humidity on the human body's perception of heat: days with high temperatures coupled with high humidity *feel* hotter than those with the same temperature and a low relative humidity (often colloquially referred to as "dry heat"). Figure 5.4 contains the probability density of each temperature-humidity tuple in Florida, USA during the month of July in 2013. Note the prominent peak signaling a high probability of both high humidity and high temperatures for the region. The query involved six storage nodes and was evaluated in 240.10 ms (averaged over 1000 iterations, with a standard deviation of 30.0 ms).

Similar to a probability density query, evaluating joint probability requires creating a subtree of intersecting feature values and then performing a kernel density estimation during the reduce phase. As always, the client-side request for a joint probability query can be combined with our

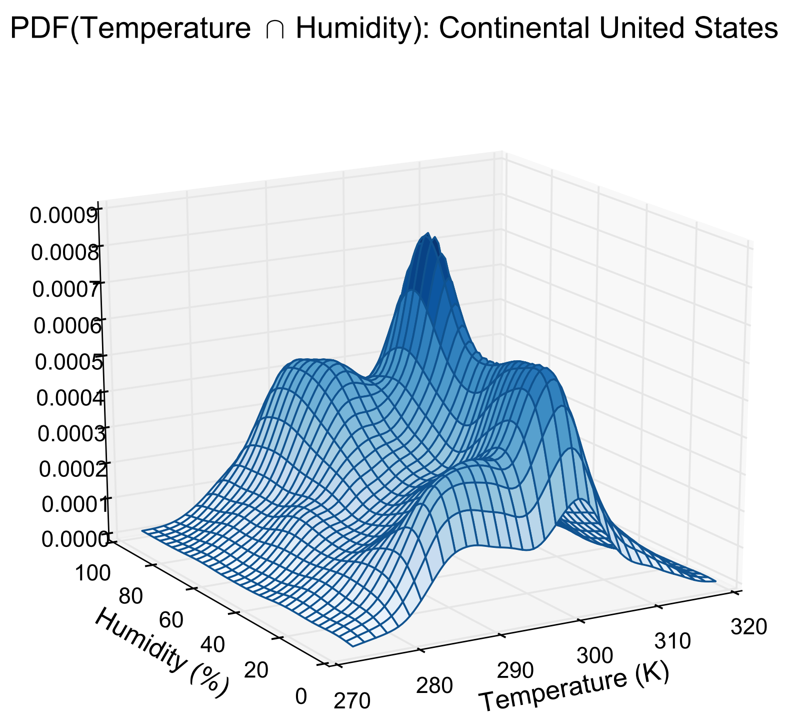


Figure 5.5: Joint probability density of temperature and humidity values across the continental USA during July of 2013.

existing query constructs to manage specificity. Figure 5.5 provides a contrasting view of the same query evaluated across the entire continental United States, which contacted all the nodes in our 78-machine testbed in parallel and completed in 424.19 ms (averaged over 1000 iterations, with a standard deviation of 45.16 ms). Note that in this scenario a peak similar to that in Figure 5.4 exists due to the overall hot and humid summer months, but the probabilities of other temperature-humidity tuple combinations have also risen to account for other regions; particularly, the frequency of high heat and low humidity is much more prominent in this example. In both cases, additional information can be retrieved from the subtrees generated by the query; for instance, a range query could be evaluated against the aggregate subtrees to determine the locations in the continental USA with high temperatures and low humidity.

5.3 Predictive Queries

After exploring the interactions between features in the dataset, *predictive analytics* enables us to capitalize on these relationships through modeling and dynamic forecasting to discover and exploit trends, project future events, and reason about how the dataset is evolving over time. These types of analysis involve making assertions about the strength of relationships to validate statistical hypotheses, projecting future events and conditions with linear and nonlinear models, and making probabilistic estimations. Using this functionality, the wealth of knowledge already stored in the system is leveraged to provide accurate and timely insights into the evolution of the feature space and its corresponding events.

5.3.1 Hypothesis Testing

The exploratory analysis process leads to the development of hypotheses about the dataset and its interactions; an increase in sales might be the result of a successful advertising campaign, or simply due to fluctuations in shopping trends. To evaluate these hypotheses in a statistically sound manner, Galileo provides built-in *hypothesis testing* functionality. Hypothesis testing begins with formulating the *null* and *alternative* hypotheses. The null hypothesis is often considered the "default position" for statistical tests, which states that there is no relationship between particular phenomena. On the other hand, the alternative hypothesis represents the opposite case where outcomes indicate a relationship. When a relationship is present in the data, we *reject* the null hypothesis — depending on the test being carried out, this may or may not be desirable. For instance, it may be advantageous to replace a hardware component with a cheaper version, but only if the null hypothesis holds for failure rates before and after the change.

Hypothesis testing is performed by comparing aggregate summary trees produced by two queries. Storage nodes that are projected to contain a majority of the relevant data for a query orchestrate the reduction process, merging vertices as they are received to produce the final summary trees. A t-test is then carried out on the trees, enabling calculation of the p-value as described in the previous section. We support both one- and two-tailed tests: a one-tailed test is used for relationships that change in only one direction, whereas a two-tailed test is applied when the relationship could be both negative or positive. For example, an advertising campaign could both increase or decrease sales; if the advertisements are negatively received or alienate previous customers, sales may decline. The p-value is then used to decide whether the null hypothesis should be rejected or not, with significance levels of 5% or 1% commonly used to denote a strong presumption against the null hypothesis. To instrument a hypothesis test, clients provide two queries that represent the states being compared, along with a desired significance level. Backed by our vertex synopses, hypothesis testing is an efficient operation: a randomized test involving 250,000 vertices and over 2 million records executed across 24 nodes completed in 26.66 ms on average over 1000 iterations, with a standard deviation of 1.64 ms.

5.3.2 Multiple Linear Regression

While our vertex synopses support two-dimensional regression functionality, there are cases where several explanatory variables can predict a single outcome. To handle these situations, we also support *multiple linear regression* across features to build predictive models that are updated continuously as data is streamed into the system. Since multiple linear regression spans several features, instances are not required at each vertex in the ABT. Instead, we support selective placement of regression instances; the default placement is not linked to any particular location in the tree, assimilating all new readings as they arrive. However, we have also incorporated support for user-defined limitations on the scope of the regression models to particular subsets of the data to ensure expressivity. For example, each vertex representing a month of the year might maintain its own multiple linear regression instance. Users can place or remove multiple linear regression models during index creation or dynamically at runtime, and their meager memory requirements enable a substantial number of instances to be maintained in the index.

In situations where variables in the regression model are highly correlated and begin to exhibit *multicollinearity*, coefficient estimates for individual predictors may become inaccurate. For this reason, we autonomously detect multicollinearity and warn users of its presence using the *variance*

Table 5.3: Dynamic multiple linear regression performance evaluation, averaged over 1000 iterations.

Operation	**Time (μs)**	**Std. Dev. (μs)**
Add Data Point	1.51	0.28
Calculate r^2	0.78	0.02
Calculate RMSE	0.79	0.13
Make Prediction	2.54	0.51

inflation factor (VIF). The VIF provides a measure of multicollinearity that helps judge whether certain variables should be excluded, which is essential when dealing with the large number of models our framework maintains. Client applications can also specify custom VIF thresholds based on their particular use cases.

Like our two-dimensional vertex synopses, multiple linear regression instances are lightweight. Table 5.3 contains performance statistics for various operations applied to the regression instances, which include adding data, calculating r^2, and making predictions. We also include the time taken to compute the root-mean-square error (RMSE) of a regression operation, which measures the accuracy of the predictions using the same units as the dependent variable. Vertices that maintain a multiple linear regression instance create a predictive model for each feature type, meaning fast and efficient updates are critical to ensure overall system performance.

To benchmark the effectiveness of our regression framework, we used models built with data collected in Wyoming, USA during July over a three-year period (2011 through 2013) to predict rainfall. The models included each of the feature types indexed in this study, and were tested with new feature readings from 2014. Figure 5.6 contains a scatter plot of the residuals (difference between predicted and actual precipitation). The RMSE of this test was 0.31 kg/m^2 of rainfall. While an exact measure of rainfall is a useful metric, we can also answer the common question "do I need my umbrella today?" with a binary classifier. To create the classifier, we considered any prediction over 0.31 kg/m^2 to imply that it would indeed rain, whereas a value lower than

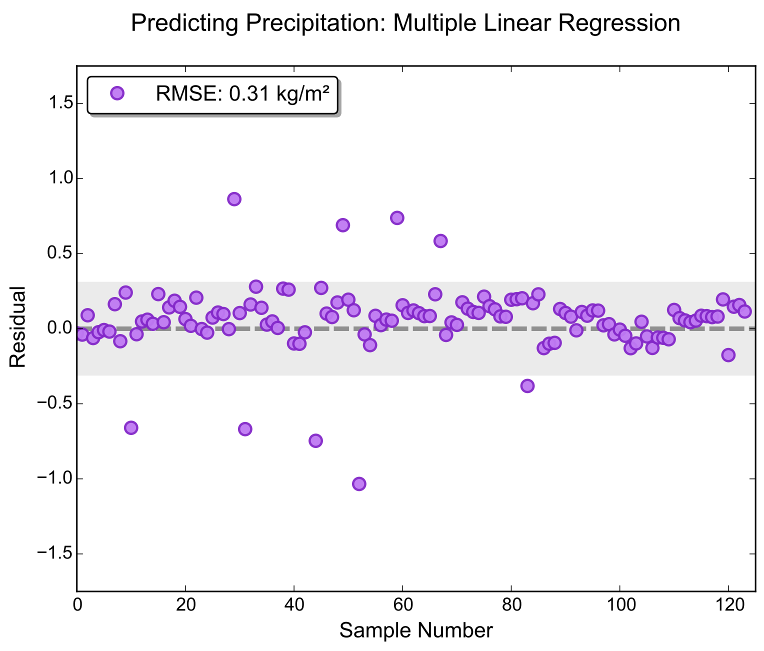


Figure 5.6: Scatter plot of the residuals between predicted and actual precipitation in Wyoming, USA, in July of 2014. The shaded region above and below the reference line represents the RMSE of the predictions.

the threshold would indicate little to no rainfall. In this case, we predicted rain correctly 92% of the time when compared to the actual rainfall data from 2014. While client applications can request multiple linear regression model parameters directly for a particular vertex or feature set and have them streamed back for analysis, they are also given the option to create customized binary classifiers similar to the example described. A classification query specifies features of interest, spatial and temporal ranges, and classification thresholds, and returns a computational model that can be used to predict and classify future events.

5.3.3 Artificial Neural Networks

For some datasets or feature types, linear methods may restrict model fidelity or reduce prediction accuracy; for instance, consider the constant fluctuations present in foreign exchange rates and stock prices or the sinusoidal variations in temperatures that occur over the course of a day. In these cases, nonlinear methods provide an alternative that can produce models that are a better fit for the underlying data. Our framework includes an interface that enables arbitrary models and prediction methods to be placed at ABT vertices in a similar fashion to the multiple linear regression instances, and can produce output datasets in the form of classifications, function approximations, or forecasts. We have incorporated support for online artificial neural networks (ANNs) provided by the Encog [114] machine learning library to accommodate nonlinear predictive models.

Compared to the linear methods discussed in previous sections, ANNs generally involve more complex computations for training. Furthermore, they often do not completely converge on one final set of model parameters, so training is an inexact and iterative process. A neural network created with the Wyoming dataset from the previous section took 682.08 ms on average to train with a single new record (1000 iterations, standard deviation of 128.13 ms). For these reasons, our framework supports a *batch ingest* mode to amortize training costs by collecting a set number of observations before training occurs. Training is also executed as a lower-priority background thread to avoid impacting query throughput, maintaining performance while keeping predictions updated. This allows us to ensure consistent overall system performance while still providing the more compute-intensive functionality afforded by ANNs.

5.3.4 Time-series Forecasting: ARIMA

Autoregressive integrated moving average (ARIMA) models are specifically designed for time-series data and allow predictions to be made on non-stationary data types. ARIMA models are parameterized by three parameters, p, d, and q, which correspond to the autoregressive, integrated, and moving average components of the model, respectively. Our implementation allows these parameters to be chosen autonomously by the system using the Hyndman-Khandakar algorithm [115] or specified at query time by client applications. Query parameters also control which features should be considered by the model and the time bounds of interest (which may include historical data).

Figures 5.7 and 5.8 illustrate the effectiveness of ARIMA for predicting temperatures in Florida and Wyoming, USA, respectively during July of 2013. These models were populated with data from the first 23.5 days of the month, and then were used to forecast subsequent temperatures for the remainder of the month. Predictions for Florida resulted in a root-mean-square error (RMSE) of

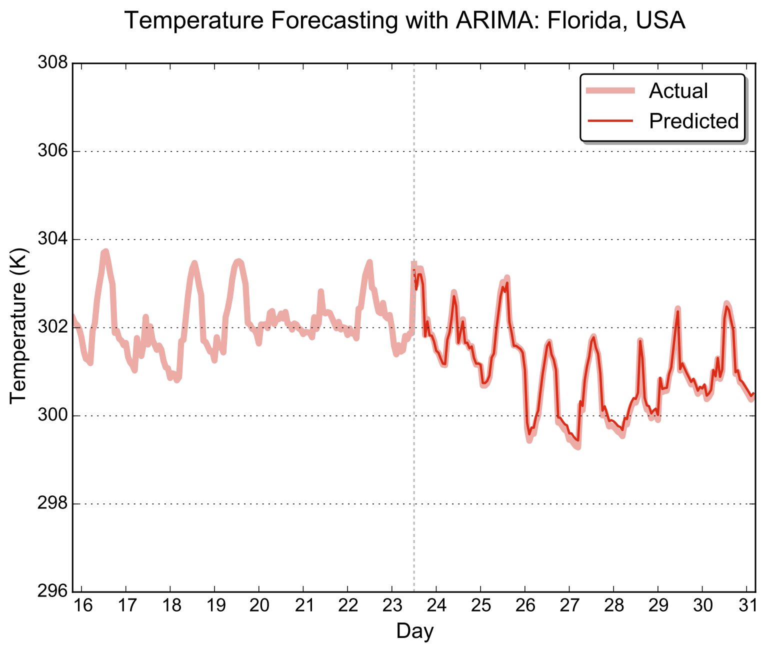


Figure 5.7: ARIMA forecast of temperature values in Florida, USA during the month of July in 2013.RMSE: 0.077 K

0.077 K, while predictions from the model responsible for Wyoming produced an RMSE of 0.818 K; in both cases, the forecasts were highly accurate, and responded to the cyclical temperature shifts that occur during a typical day in both locations. Note that each day generally exhibits the highest temperatures during midday and lowest temperatures during night hours. Compared to Wyoming, temperature fluctuations are less severe in Florida, which likely accounts for some of the difference in prediction accuracy.

5.3.5 Conditional Probability and Naive Bayes Classification

Conditional Probability queries answer questions such as, "what is the probability of rain given cloud cover is greater than 80%?" Computing the answer to this type of query involves determining the intersection of a particular set of events, which can be performed efficiently with the ABT; client-side turnaround times averaged 38.66 ms for queries over Florida, USA (across 1000 iterations with a standard deviation of 2.89 ms). Based on the frequencies stored in vertex synopses, individual probabilities associated with each event can be retrieved through synopsis combinations. This type of probabilistic analysis can also uncover relationships with no direct interaction; two

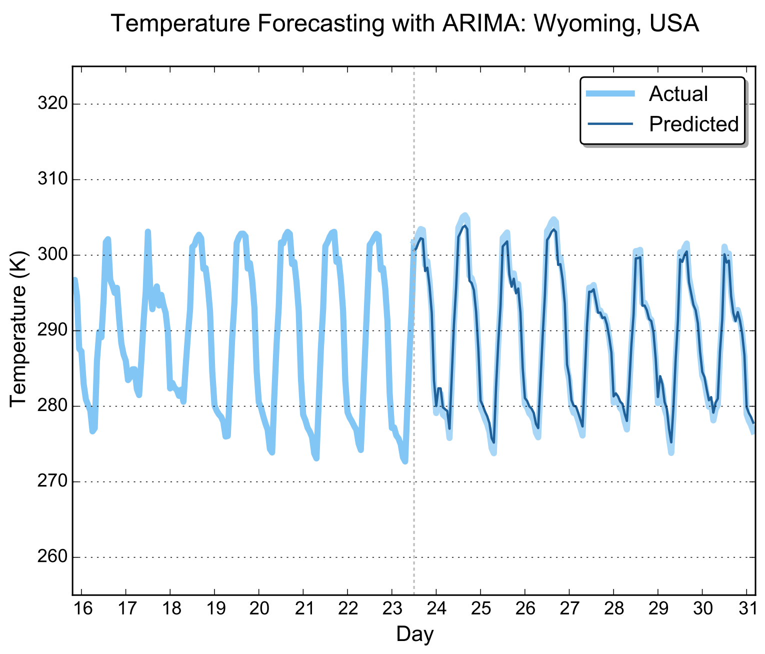


Figure 5.8: ARIMA forecast of temperature values in Wyoming, USA during the month of July in 2013. RMSE: 0.818 K

features may seem unrelated until an additional dimension is considered. Determining whether such a relationship exists requires traversing backwards through the tree hierarchy until a common link is found.

A ***Naive Bayes classifier*** uses the probabilities associated with events or features in the dataset to make predictions. The key assumption of this type of model is that all features are *independent*: completely unrelated to any of the other features. Despite the fact that this assumption may not always hold, naive Bayes has proven to be effective in practice for a variety of classification tasks, including text categorization and medical diagnosis. To classify a given set of samples, naive Bayes uses the combined probabilities of the events to choose the most probable outcome. Given variables such as the current cloud cover, humidity, and temperature, naive Bayes can determine the probability of rain or other events in the dataset. Using our test data from Wyoming in July of 2013 we could achieve 85% accuracy for rainfall predictions in 2014, a query that took 17.68 ms on average (across 1000 iterations with a standard deviation of 1.66 ms, after contacting 6 storage nodes). While the predictions were less accurate than our multiple linear regression models, naive

Bayes can also be used in situations where multiple linear regression does not apply, such as text classification.

5.4 Exporting Result Datasets

When creating visualizations or machine learning models, it is often useful to leverage the capabilities of external frameworks. Rather than retrieving this data, storing it as a CSV or other compatible format, and then loading the data in another framework, Galileo offers in-memory result datasets that are exposed as a virtual FUSE [116] file system (compatible with Linux, Windows, macOS, FreeBSD, and a variety of other platforms). This allows other applications to mount the Galileo file system and transparently access files that may be stored on a variety of machines throughout the cluster. These datasets supply data for training machine learning models or generating visualizations without incurring I/O costs, with live updates propagating to the file system as more data is stored in Galileo.

5.4.1 Visualization

Figure 5.9 demonstrates a climate chart visualization over data in Phoenix, Arizona, USA in 2014. While a standard approach for producing this visualization over voluminous atmospheric data would likely involve several MapReduce computations, the ABT makes all the necessary information readily available through queries, avoiding distributed computations altogether. Furthermore, retrieving the data for this evaluation consumed considerably less time (1.52 ms) than rendering the image on the client side (122.81 ms).

5.4.2 External Machine Learning Libraries

Figures 5.10 and 5.11 demonstrate how data in the ABT can be exported to CSV files and used by external machine learning frameworks. In this benchmark, the scikit-learn Python machine learning library [117] was used to generate a random forest model (Figure 5.10) and gradient

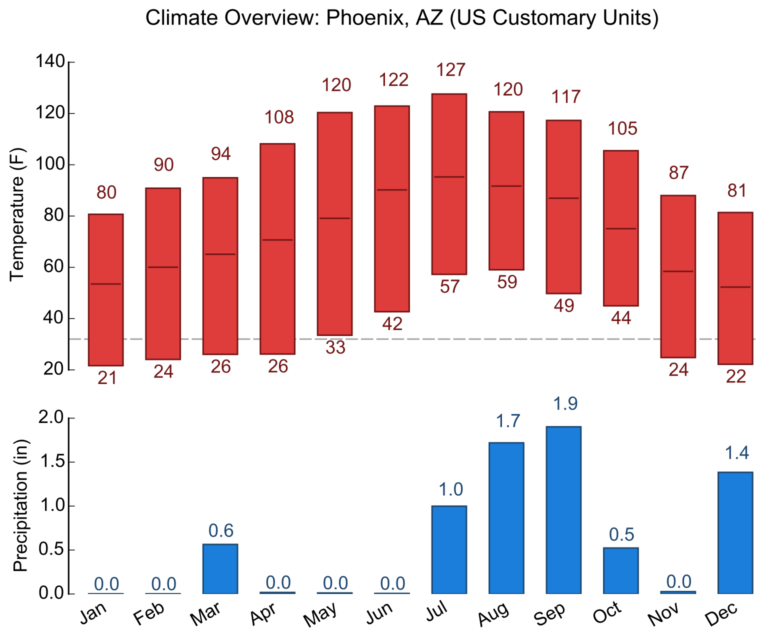


Figure 5.9: A climate chart generated directly from the ABT on observations collected in Phoenix, Arizona.

boosted decision tree model (Figure 5.11) to predict surface temperatures based on the other features in our test dataset. In both cases, the dataset was loaded by scikit-learn in less than 1 second.

5.4.3 Epidemiology Dataset

To demonstrate the performance of Galileo using an alternative dataset, with imported epidemiology data generated by the *Animal Disease Spread Model* (ADSM) [118], which simulates disease outbreaks in livestock populations and has been used in several studies including foot-and-mouth disease [119], avian influenza [120], and pseudorabies [121]. ADSM and its predecessor, the *North American Animal Disease Spread Model* (NAADSM) are Monte Carlo models, which means that each set of parameters, called a *scenario*, is run several times to gain confidence in the outputs. This produces large amounts of input and output data; in this study, we used an ADSM scenario set in Texas, USA, that simulated an outbreak of foot-and-mouth disease across 364,000 farms with direct contact, indirect contact, and airborne disease spread. The outbreak was based on real-world farm and disease biology data, with each scenario variant executed 30 times to account

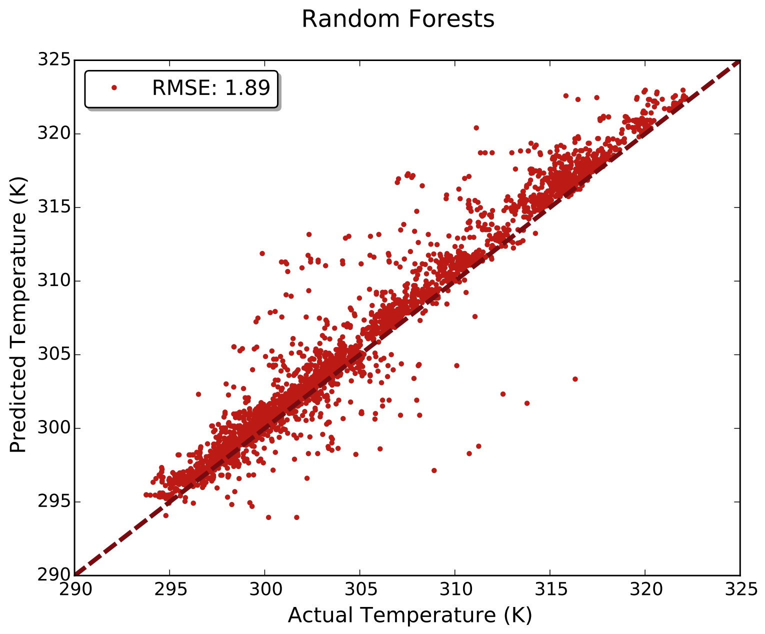


Figure 5.10: Random forest predictions of surface temperature values, generated by scikit-learn through our in-memory virtual file system.

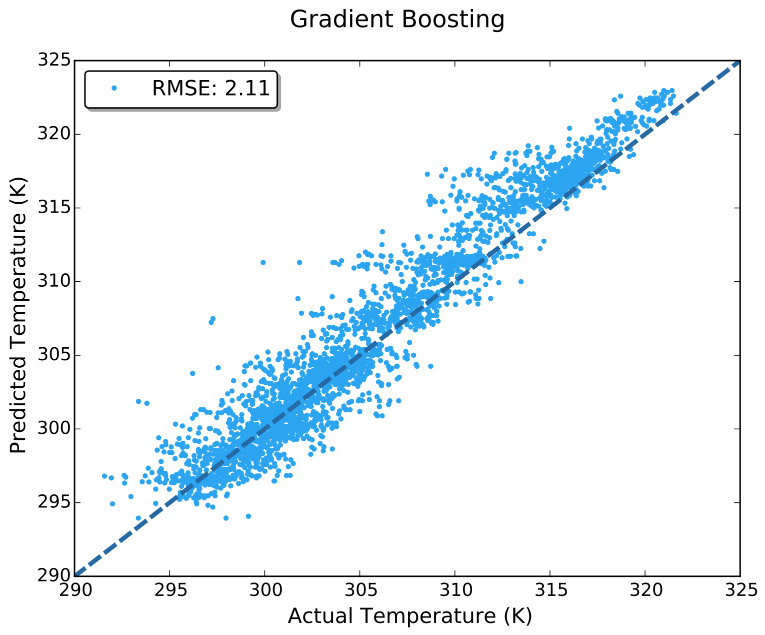


Figure 5.11: Gradient boosted decision tree predictions of surface temperature values, generated by scikit-learn through our in-memory virtual file system.

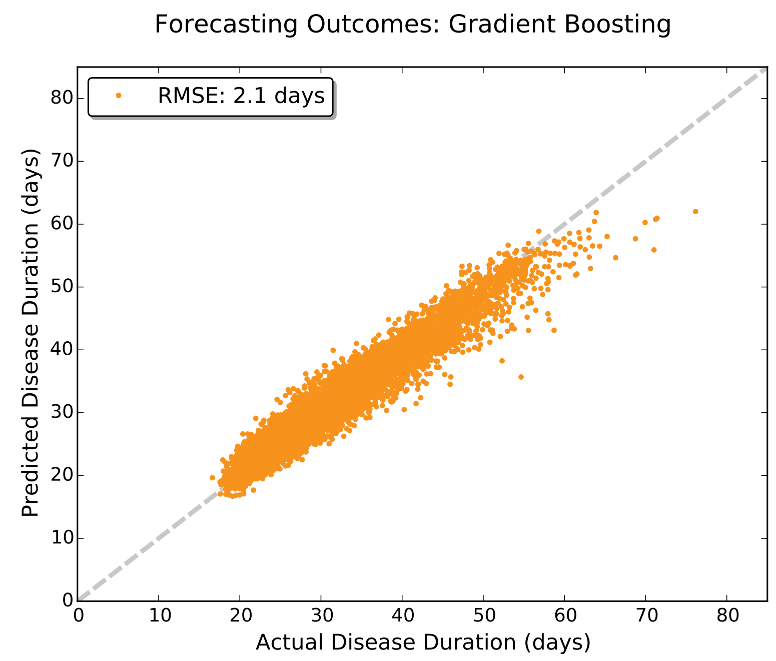


Figure 5.12: Gradient boosted decision tree predictions of disease outbreak durations, generated by scikit-learn through our in-memory virtual file system.

for uncertainty in the results. The overall dataset size was 1.1 petabytes, and included over 2000 unique features.

Our initial test with this dataset leveraged sci-kit learn to predict the *disease duration* output variable from the simulation using another gradient boosted decision tree model. Figure 5.12 demonstrates the accuracy of the model. Next, we evaluated the dataset with Apache Spark [53]. Spark is a popular distributed computation framework that supports machine learning models through its MLlib library [56]. We built an adapter to convert raw Galileo data points to Spark RDDs [54] and then used MLlib to generate another random forest model of disease durations across a cluster of machines, shown in Figure 5.13.

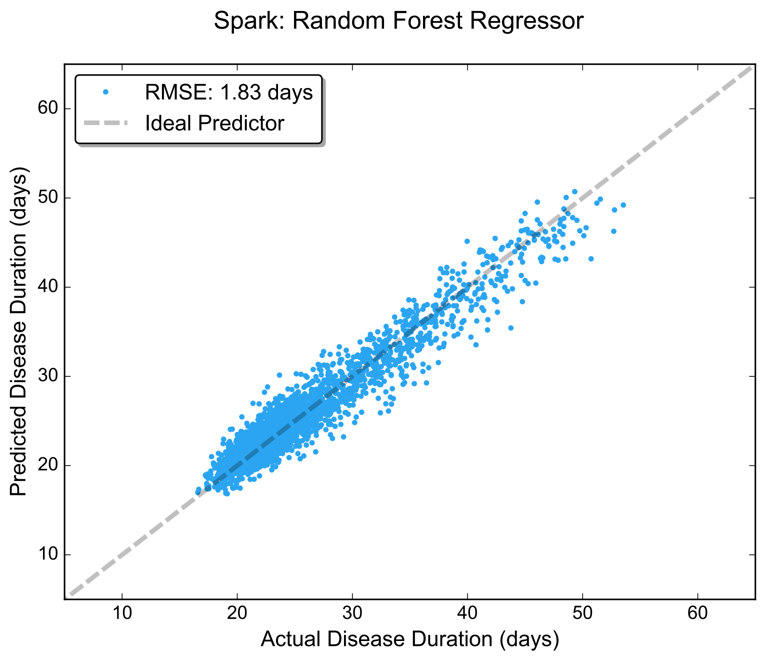


Figure 5.13: Random forest predictions of disease outbreak durations, generated by Spark through our RDD export functionality.

Chapter 6

Conclusions

As data volumes continue to grow, tracking the continually-evolving feature space is a considerable challenge. This is especially true in the case of geospatial, time-series datasets. In this dissertation, we studied the problem of facilitating low-latency, query-based analytics, which required consideration of the following:

- Architectural decisions underpinning distributed storage
- The speed differential between memory accesses and disk I/O
- Trade-offs between accuracy and latency
- Metadata organization and management, and the performance impact of the data structures
- Types of retrievals supported and the dispersion of query load

Addressing these factors enabled us to provide a rich set of exploratory analysis functionality as well as support for predictive models that produce insights beyond the trends present in the dataset.

With its DHT-based network design, Galileo is able to scale out and cope with large datasets. Our partitioning scheme and use of Geohashes also allow the system to localize queries efficiently. By leveraging variable-precision indexes (global/local analytic base trees as well as full-resolution data stored on disk), Galileo can resolve queries quickly while allowing users to be selective about the desired accuracy or latency required for their use cases.

One key aspect of our approach is minimizing disk accesses. This is achieved by carefully maintaining analytic base trees (ABTs) that retain expressiveness for query evaluations but preserve compactness to ensure memory residency. The ABTs remain compact even in situations where individual nodes store hundreds of millions of files. Further, statistical synopses ensure the knowledge base is continually updated with live data streams. We achieve this via the use and adaptation of online algorithms, compact data structures, and lightweight models. This also allows us to perform query evaluations at multiple geographic scales.

The trade-off between accuracy and timeliness of query evaluations can be achieved by automatically controlling query specificity and dataset coverage during evaluations. 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 issues by reducing the fidelity of results under high load.

The trees maintained by Galileo are kept up-to-date with changing trends in values as new information arrives. To cope with these changes, we apply autonomous *quantization* and *vertex reclamation*, which expand or contract vertices to reflect the density and distribution of values. By maintaining tree statistics in an online fashion, we can continually update feature summaries as data is assimilated.

We support a variety of analytic queries to aid in knowledge discovery and leveraging of the insights contained in voluminous datasets. These queries are expressive and allow users to avoid writing complex distributed applications to solve their problems. In many cases, the summary statistics and metadata maintained in memory by Galileo provide enough information to resolve queries without disk I/O. Ultimately, these features reduce the friction between users and their data, encouraging thorough analysis and leading to faster discoveries.

Briefly, the contributions of this dissertation include: (1) a distributed framework for iterative, query-driven analytics over voluminous datasets, (2) support indexing and querying multidimensional, spatiotemporal data at scale, (3) in-memory data structures that help reduce both network and disk I/O while still retaining expressiveness, (4) high-throughput distributed query resolution, (5) online modeling of the feature space and autonomous reconfiguration of the system to cope with evolution in observations, and (6) flexible interfaces that can be applied to a broad class of storage systems.

6.1 Future Work

As more organizations collect and leverage big data, effective analytics and data mining allow insights to be derived that provide business opportunities and a better understanding of real-world phenomena. Our future work will investigate how partitioning and distributing data over clusters of commodity hardware can improve machine learning model training times and prediction/classification performance. This may entail exploiting data locality to ensure timeliness, addressing I/O imbalances [122, 123], or accounting for physical storage properties to better fit access patterns and hardware capabilities. On the hardware front, SSDs and technologies such as 3D XPoint [124] present new possibilities in how data is stored and accessed in analytic environments.

Another insight gained from this work is that giving users accessible tools to manipulate and analyze their data in real time facilitates effective model creation and parameterization. This *feature engineering* process could be augmented with support for Bayesian networks, causality analysis, and 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?" This functionality would also be applicable in areas such as object tracking and smart grids.

As advancements in computing devices shift towards low power and high connectivity, the interaction between the cloud, distributed systems, and Internet of Things (IoT) devices will present opportunities for innovation in computation, storage, and analysis platforms. Future developments will harness federated clouds [125] as well as IoT devices to collect, manage, and analyze voluminous datasets [126, 127]. Cheap, efficient IoT devices such as the Raspberry Pi [128] (shown in Figure 6.1 with sensors attached) acting as *fog* nodes will allow storage and processing activities to be pushed to the edges of the network [129–131]. In these environments, coordination with the cloud allows frameworks to target the individual strengths of participating hardware; for example, fog nodes may be able to quickly resolve queries over data collected recently from onboard sensors while the cloud serves as an archival storage and large-scale analysis platform. In these types of sensing environments, *ensemble learning* methods allow a consensus to be achieved across multi-

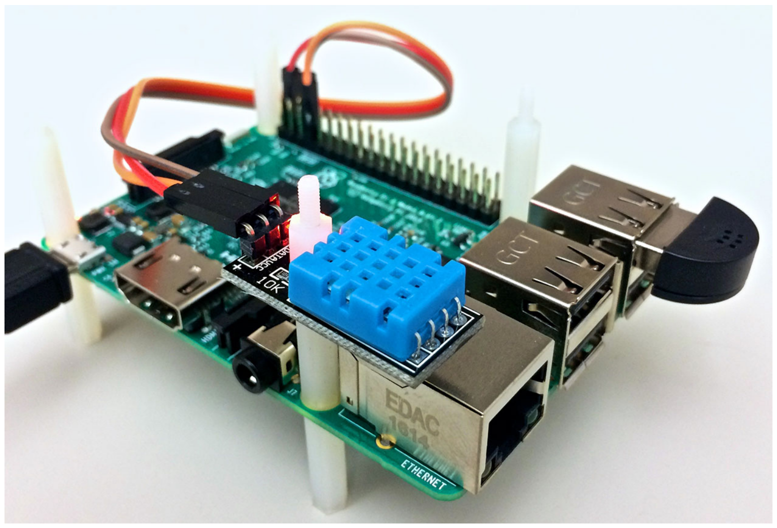


Figure 6.1: A Raspberry Pi single-board computer with temperature, humidity, and audio sensors.

ple predictions or classifications on disparate nodes, which also facilitates anomaly detection. In addition, *collaborative filtering* presents opportunities for predicting behaviors or user preferences as well as making recommendations.

IoT devices often operate in constrained environments with respect to airflow/cooling, power availability, and network speeds. Consequently, frameworks in this space have greater responsibilities when maintaining availability and scheduling tasks. We plan to investigate how these variables interact in large-scale, geographically dispersed hybrid clusters that incorporate fog nodes for distributed storage and computation. In-memory sketches, compact indexes, and energy-aware scheduling are likely to be effective in such an environment. Figure 6.2 provides a query throughput comparison from our initial studies on fog-cloud coordination [131]; with a simplified in-memory index, a cluster of fog nodes is able to achieve higher throughput for smaller queries (short spans of time) due to their I/O bound nature. On the other hand, powerful cloud VMs are better suited for tasks that are CPU-bound (large spans of time in the temporal index). As these technologies continue to evolve, seamless distribution of both storage and computing concerns are crucial when managing future data volumes and devices.

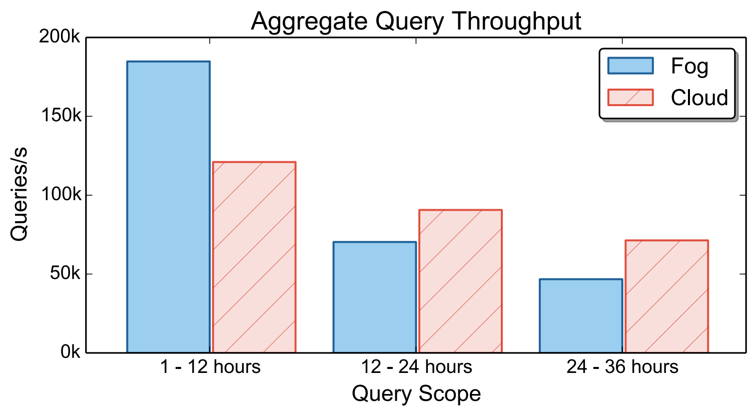


Figure 6.2: Comparison of aggregate query throughput on cloud and fog nodes (48 fog nodes, 16 cloud VMs).

Bibliography

[1] John Gantz and David Reinsel. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the future*, 2007:1–16, 2012.

[2] ABI Research. The Internet of Things Will Drive Wireless Connected Devices to 40.9 Billion in 2020. *https://www.abiresearch.com/press/the-internet-of-things-will-drive-wireless-connect/*, 2014.

[3] Janessa Rivera and Rob van der Meulen. Gartner says the internet of things installed base will grow to 26 billion units by 2020. *http://www.gartner.com/newsroom/id/2970017*, 2015.

[4] Bienvenido Vélez, Ron Weiss, Mark A Sheldon, and David K Gifford. Fast and effective query refinement. In *ACM SIGIR Forum*, volume 31, pages 6–15. ACM, 1997.

[5] John C Shafer and Rakesh Agrawal. Continuous querying in database-centric web applications. *Computer Networks*, 33(1):519–531, 2000.

[6] 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.

[7] Matthew Malensek, Sangmi Lee Pallickara, and Shrideep Pallickara. Analytic queries over geospatial time-series data using distributed hash tables. *IEEE Transactions on Knowledge and Data Engineering*, 28(6):1408–1422, Jun 2016.

[8] Matthew Malensek, Sangmi Lee Pallickara, and Shrideep Pallickara. Fast, ad hoc query evaluations over multidimensional geospatial datasets. *IEEE Transactions on Cloud Computing*, 5(1):28–42, Jan 2017.

[9] Matthew Malensek, Sangmi Lee Pallickara, and Shrideep Pallickara. Exploiting geospatial and chronological characteristics in data streams to enable efficient storage and retrievals. *Future Generation Computer Systems*, 29(4):1049–1061, Jun 2013.

[10] Matthew Malensek, Sangmi Lee Pallickara, and Shrideep Pallickara. Expressive query support for multidimensional data in distributed hash tables. In *Proceedings of the 2012 IEEE/ACM 5th International Conference on Utility and Cloud Computing (UCC)*, pages 31–38, Nov 2012.

[11] Gustavo Niemeyer. Geohash, 2008.

[12] Matthew Malensek, Sangmi Lee Pallickara, and Shrideep Pallickara. Evaluating geospatial geometry and proximity queries using distributed hash tables. *IEEE Computing in Science Engineering (CiSE)*, 16(4):53–61, Jul 2014.

[13] Matthew Malensek, Sangmi Pallickara, and Shrideep Pallickara. Polygon-based query evaluation over geospatial data using distributed hash tables. In *Proceedings of the 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing (UCC)*, pages 219–226, Dec 2013.

[14] Matthew Malensek. On the evaluation of exact-match and range queries over multidimensional data in distributed hash tables. MS Thesis, Colorado State University, Aug 2012. Advisor: Shrideep Pallickara.

[15] Walid Budgaga, Matthew Malensek, Sangmi Lee Pallickara, and Shrideep Pallickara. A framework for scalable real-time anomaly detection over voluminous, geospatial data streams. *Concurrency and Computation: Practice and Experience*, 29(12), Mar 2017.

[16] Matthew Malensek, Zhiquan Sui, Neil Harvey, and Shrideep Pallickara. Autonomous, failure-resilient orchestration of distributed discrete event simulations. In *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference (CAC)*, pages 3:1–3:10, Sep 2013.

[17] Zhiquan Sui, Matthew Malensek, Neil Harvey, and Shrideep Pallickara. Autonomous orchestration of distributed discrete event simulations in the presence of resource uncertainty. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 10(3):18:1–18:20, Sep 2015.

[18] Matthew Malensek, Walid Budgaga, Sangmi Pallickara, Neil Harvey, F. Jay Breidt, and Shrideep Pallickara. Using distributed analytics to enable real-time exploration of discrete event simulations. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, pages 49–58, Dec 2014.

[19] Walid Budgaga, Matthew Malensek, Sangmi Pallickara, Neil Harvey, F. Jay Breidt, and Shrideep Pallickara. Predictive analytics using statistical, learning, and ensemble methods to support real-time exploration of discrete event simulations. *Future Generation Computer Systems*, 56(C):360–374, Mar 2016.

[20] Naman Shah, Harshil Shah, Matthew Malensek, Sangmi Lee Pallickara, and Shrideep Pallickara. Network analysis for identifying and characterizing disease outbreak influence from voluminous epidemiology data. In *Proceedings of the 2016 IEEE International Conference on Big Data (IEEE BigData)*, page (To Appear), Dec 2016.

[21] Jared Koontz, Matthew Malensek, and Sangmi Lee Pallickara. Geolens: Enabling interactive visual analytics over large-scale, multidimensional geospatial datasets. In *Proceedings of the 2014 IEEE/ACM International Symposium on Big Data Computing (BDC)*, pages 35–44, Dec 2014.

[22] Sangmi Lee Pallickara, Matthew Malensek, and Shrideep Pallickara. Enabling access to timeseries, geospatial data for on-demand visualization. In *IEEE Symposium on Large Data Analysis and Visualization, (LDAV)*, pages 141–142, Oct 2011.

[23] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[24] Cameron Tolooee, Matthew Malensek, and Sangmi Lee Pallickara. A framework for managing continuous query evaluations over voluminous, multidimensional datasets. In *Proceedings of the 2014 IEEE International Cloud and Autonomic Computing Conference (ICCAC)*, pages 73–82, Sep 2014.

[25] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[26] D. Hastorun, M. Jampani, G. Kakulapati, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *SOSP*. Citeseer, 2007.

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

[28] National Oceanic and Atmospheric Administration. The north american mesoscale forecast system, 2016.

[29] Sangmi Lee Pallickara, Matthew Malensek, and Shrideep Pallickara. *On the Processing of Extreme Scale Datasets in the Geosciences*, pages 521–537. Springer New York, New York, NY, 2011.

[30] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[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.

[32] Tom White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[33] Chuck Lam. *Hadoop in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010.

[34] The Apache Software Foundation. Hadoop. *http://hadoop.apache.org*, 2016.

[35] The Apache Software Foundation. Apache HBase: A distributed database for large datasets.

[36] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[37] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, October 2003.

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

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

[40] Michael Stonebraker, Jacek Becla, David J. DeWitt, Kian-Tat Lim, David Maier, Oliver Ratzesberger, and Stanley B. Zdonik. Requirements for science data bases and scidb. In *CIDR 2009, Fourth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2009, Online Proceedings*, 2009.

[41] MongoDB Developers. MongoDB. *http://www.mongodb.org/*.

[42] Abdeltawab M. Hendawi and Mohamed F. Mokbel. Panda: A predictive spatio-temporal query processor. In *Proc. of 20th Internatl Conf on Advances in GIS*, SIGSPATIAL '12, 2012.

[43] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: A warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, August 2009.

[44] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion 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*, EuroSys '13, pages 29–42, New York, NY, USA, 2013. ACM.

[45] Phillip B Gibbons, Viswanath Poosala, Swarup Acharya, Yair Bartal, Yossi Matias, S Muthukrishnan, Sridhar Ramaswamy, and Torsten Suel. AQUA: System and techniques for approximate query answering. *Bell Labs TR*, 1998.

[46] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. Haloop: Efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3(1-2):285–296, September 2010.

[47] S. Pallickara, J. Ekanayake, and G. Fox. An overview of the granules runtime for cloud computing. In *eScience, 2008. eScience'08. IEEE Fourth International Conference on*, pages 412–413. IEEE, 2008.

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

[49] Shrideep Pallickara, Jaliya Ekanayake, and Geoffrey Fox. Granules: a lightweight runtime for scalable computing with support for map-reduce. *Cloud Computing and Software Services*, page 201, 2010.

[50] S. Pallickara and G. Fox. On the matching of events in distributed brokering systems. In *International Conference on Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004.*, volume 2, pages 68–76 Vol.2, April 2004.

[51] Geoffrey Fox, Shrideep Pallickara, and Xi Rao. Towards enabling peer-to-peer grids. *Concurrency and Computation: Practice and Experience*, 17(7-8):1109–1131, 2005.

[52] G. Fox and S. Pallickara. Deploying the NaradaBrokering substrate in aiding efficient web and grid service interactions. *Proceedings of the IEEE*, 93(3):564–577, mar 2005.

[53] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[54] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[55] Tim Kraska, Ameet Talwalkar, John C Duchi, Rean Griffith, Michael J Franklin, and Michael I Jordan. Mlbase: A distributed machine-learning system. In *CIDR*, volume 1, pages 2–1, 2013.

[56] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. MLlib: Machine learning in apache spark. *Journal of Machine Learning Research*, 17(34), April 2016.

[57] Alekh Agarwal, Olivier Chapelle, Miroslav Dudík, and John Langford. A reliable effective terascale linear learning system. *J. Mach. Learn. Res.*, 15(1):1111–1133, January 2014.

[58] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009.

[59] Jun 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*, pages 239–250, Oct 2004.

[60] Yatin Chawathe, Sriram Ramabhadran, Sylvia Ratnasamy, Anthony LaMarca, Scott Shenker, and Joseph Hellerstein. A case study in building layered DHT applications. In *Proceedings of the 2005 SIGCOMM*, 2005.

[61] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[62] Edward Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, September 1960.

[63] Graham Cormode, Minos Garofalakis, Peter J. Haas, and Chris Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 4(1–3), January 2012.

[64] Shahriar Yousefi, Ilona Weinreich, and Dominik Reinarz. Wavelet-based prediction of oil prices. *Chaos, Solitons & Fractals*, 25(2):265–275, 2005.

[65] Yufei Tao, Dimitris Papadias, and Jimeng Sun. The TPR*-tree: An optimized spatio-temporal access method for predictive queries. In *Proc of 29th Internatl Conf on Very Large Data Bases*, 2003.

[66] Simonas Šaltenis, Christian S. Jensen, Scott T. Leutenegger, and Mario A. Lopez. Indexing the positions of continuously moving objects. *SIGMOD Rec.*, 29(2):331–342, May 2000.

[67] Anirban Mondal, Yi Lifu, and Masaru Kitsuregawa. P2pr-tree: An r-tree-based spatial index for peer-to-peer environments. In *Current Trends in Database Technology-EDBT 2004 Workshops*, pages 516–525. Springer, 2005.

[68] Antonin Guttman. *R-trees: A dynamic index structure for spatial searching*, volume 14. ACM, 1984.

[69] Cédric du Mouza, Witold Litwin, and Philippe Rigaux. Sd-rtree: A scalable distributed rtree. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 296–305. IEEE, 2007.

[70] Abhijit Kadlag, Amol V Wanjari, Juliana Freire, and Jayant R Haritsa. Supporting exploratory queries in databases. In *Database Systems for Advanced Applications*, pages 594–605. Springer, 2004.

[71] Jim Gray, Adam Bosworth, Andrew Layman, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *Proc. of the 12th International Conference on Data Engineering*, ICDE '96, pages 152–159, Washington, DC, USA, 1996. IEEE Computer Society.

[72] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Implementing data cubes efficiently. In *Proc. of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 205–216, New York, NY, USA, 1996.

[73] Inderpal Singh Mumick, Dallan Quass, and Barinderpal Singh Mumick. Maintenance of data cubes and summary tables in a warehouse. In *Proc. of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 100–111, NY, USA.

[74] Ching-Tien Ho, Rakesh Agrawal, Nimrod Megiddo, and Ramakrishnan Srikant. Range queries in olap data cubes. In *Proc. of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 73–88, New York, NY, USA.

[75] C. Gupta, S. Wang, I. Ari, M. Hao, U. Dayal, A. Mehta, M. Marwah, and R. Sharma. Chaos: A data stream analysis architecture for enterprise applications. In *2009 IEEE Conference on Commerce and Enterprise Computing*, pages 33–40, July 2009.

[76] E. Brewer. Cap twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29, Feb 2012.

[77] Eric A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 7–, New York, NY, USA, 2000. ACM.

[78] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[79] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google&rsquo;s globally distributed database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22, August 2013.

[80] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, John Cieslewicz, Ian Rae, Traian Stancescu, and Himani Apte. F1: A distributed sql database that scales. *Proc. VLDB Endow.*, 6(11):1068–1079, August 2013.

[81] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[82] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. *SIGCOMM Comput. Commun. Rev.*, 31(4):161–172, August 2001.

[83] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of the 2001 Conference on Applications,*

Technologies, Architectures, and Protocols for Computer Communications, SIGCOMM ’01, pages 161–172, New York, NY, USA, 2001. ACM.

[84] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS ’01, pages 53–65, London, UK, UK, 2002. Springer-Verlag.

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

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

[87] 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.

[88] W. Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[89] DC Wells, EW Greisen, and RH Harten. Fits-a flexible image transport system. *Astronomy and Astrophysics Supplement Series*, 44:363, 1981.

[90] Hans Reiser. Reiserfs whitepaper, 2004.

[91] Cameron Tolooee, Matthew Malensek, and Sangmi Lee Pallickara. A scalable framework for continuous query evaluations over multidimensional, scientific datasets. *Concurrency and Computation: Practice and Experience*, 28(8):2546–2563, Jun 2016.

[92] Matthew Malensek, Sangmi Pallickara, and Shrideep Pallickara. Autonomously improving query evaluations over multidimensional data in distributed hash tables. In *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference (CAC)*, pages 15:1–15:10, Sep 2013.

[93] Matej Kristan, Ales Leonardis, and Danijel Skocaj. Multivariate online kernel density estimation with gaussian kernels. *Pattern Recognition*, 44(10-11):2630–2642, 2011.

[94] Bernard W Silverman. *Density estimation for statistics and data analysis*, volume 26. CRC press, 1986.

[95] BP Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, 1962.

[96] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[97] Dan Kegel. The C10K problem. *http://www.kegel.com/c10k.html*, 2006.

[98] Elizabeth Antoine, Kotagiri Ramamohanarao, Jie Shao, and Rui Zhang. Accelerating spatial join operations using bit-indices. In *Proceedings of the Twenty-Second Australasian Database Conference-Volume 115*, pages 123–132. Australian Computer Society, Inc., 2011.

[99] Chee-Yong Chan and Yannis E Ioannidis. Bitmap index design and evaluation. *ACM SIGMOD Record*, 27(2):355–366, 1998.

[100] Thiago Luís Lopes Siqueira, Ricardo Rodrigues Ciferri, Valéria Cesário Times, and Cristina Dutra de Aguiar Ciferri. A spatial bitmap-based index for geographical data warehouses. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1336–1342. ACM, 2009.

[101] Raphael A. Finkel and Jon Louis Bentley. Quad trees a data structure for retrieval on composite keys. *Acta informatica*, 4(1):1–9, 1974.

[102] Open Postcode Ireland. Ireland's postcode. *http://www.openpostcode.org/*, 2013.

[103] Alessandro Colantonio and Roberto Di Pietro. Concise: Compressed 'n' composable integer set. *Info. Proc. Ltrs*, 110(16):644–650, 2010.

[104] Kesheng Wu, Ekow J Otoo, and Arie Shoshani. An efficient compression scheme for bitmap indices. *Lawrence Berkeley National Laboratory Tech Report*, 2004.

[105] Daniel Lemire, Owen Kaser, and Kamel Aouiche. Sorting improves word-aligned bitmap indexes. *Data & Knowledge Engineering*, 69(1):3–28, 2010.

[106] Kesheng Wu, Ekow J Otoo, Arie Shoshani, and Henrik Nordberg. Notes on design and implementation of compressed bit vectors. Technical report, Technical Report LBNL/PUB-3161, Lawrence Berkeley National Laboratory, Berkeley, CA, 2001.

[107] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. Better bitmap performance with roaring bitmaps. *Softw. Pract. Exper.*, 46(5):709–719, May 2016.

[108] United States Census Bureau. Topologically integrated geographic encoding and referencing (TIGER).

[109] JSI (java spatial index). *http://jsi.sourceforge.net/*.

[110] James F Allen. An interval-based representation of temporal knowledge. In *IJCAI*, volume 81, pages 221–226, 1981.

[111] James F Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[112] Peter Van Beek and Dennis W. Manchak. The design and experimental analysis of algorithms for temporal reasoning. *Journal of Artificial Intelligence Research*, 4:1–18, 1996.

[113] Tamas Abraham and John F Roddick. Survey of spatio-temporal databases. *GeoInformatica*, 3(1):61–99, 1999.

[114] Heaton Research, Inc. Encog machine learning framework. *http://www.heatonresearch.com/encog*.

[115] Rob J. Hyndman and Yeasmin Khandakar. Automatic time series forecasting: The forecast package for R. *Journal of Statistical Software*, 27(3):1–22, 7 2008.

[116] Miklos Szeredi et al. File system in user space (FUSE). *https://github.com/libfuse/libfuse*, 2016.

[117] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12:2825–2830, November 2011.

[118] N. Harvey, A. Reeves, M.A. Schoenbaum, F.J. Zagmutt-Vergara, C. Dubé, A.E. Hill, B.A. Corso, W.B. McNab, C.I. Cartwright, and M.D. Salman. The North American Animal Disease Spread Model: A simulation model to assist decision making in evaluating animal disease incursions. *Preventive Veterinary Medicine*, 82(3):176–197, 2007.

[119] D.L. Pendell, J. Leatherman, T.C. Schroeder, and G.S. Alward. The economic impacts of a foot-and-mouth disease outbreak: a regional analysis. *Journal of Agricultural and Applied Economics*, 39(0):19–33, 2007.

[120] C. Green, T. Whiting, G. Duizer, D. Douma, H. Kloeze, W. Lees, and A. Reeves. Simulation modeling of alternative control strategies for an HPAI outbreak using NAADSM. In *Canadian Association of Veterinary Epidemiology Preventive Medicine (CAVEPM) Meeting, May 29 - 30 2010, Guelph, Ontario, Canada*, 2010.

[121] K Portacci, A Reeves, B Corso, and M Salman. Evaluation of vaccination strategies for an outbreak of pseudorabies virus in US commercial swine using the NAADSM. In *ISVEE 12: Proceedings of the 12th Symposium of the International Society for Veterinary Epidemiology and Economics, Durban, South Africa*, page 78, 2009.

[122] Matthew Malensek, Sangmi Lee Pallickara, and Shrideep Pallickara. Minerva: Proactive disk scheduling for qos in multitier, multitenant cloud environments. *IEEE Internet Computing*, 20(3):19–27, May 2016.

[123] Matthew Malensek, Sangmi Lee Pallickara, and Shrideep Pallickara. Alleviation of disk i/o contention in virtualized settings for data-intensive computing. In *Proceedings of the 2015 IEEE/ACM 2nd International Symposium on Big Data Computing (BDC)*, pages 1–10, Dec 2015.

[124] Micron Technology. 3D XPoint Technology. *https://www.micron.com/about/emerging-technologies/3d-xpoint-technology*, 2016.

[125] Matthew Malensek, Sangmi Lee Pallickara, and Shrideep Pallickara. Autonomous cloud federation for high-throughput queries over voluminous datasets. *IEEE Cloud Computing*, 3(3):40–49, May 2016.

[126] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, pages 13–16, 2012.

[127] Flavio Bonomi, Rodolfo Milito, Preethi Natarajan, and Jiang Zhu. *Fog Computing: A Platform for Internet of Things and Analytics*, pages 169–186. Springer International Publishing, 2014.

[128] The Raspberry Pi Foundation. Raspberry Pi. *http://raspberrypi.org*, 2016.

[129] Bo Tang, Zhen Chen, Gerald Hefferman, Tao Wei, Haibo He, and Qing Yang. A hierarchical distributed fog computing architecture for big data analysis in smart cities. In *Proceedings of the ASE BigData & SocialInformatics 2015*, pages 28:1–28:6, 2015.

[130] Prem Prakash Jayaraman, João Bártolo Gomes, Hai Long Nguyen, Zahraa Said Abdallah, Shonali Krishnaswamy, and Arkady Zaslavsky. CARDAP: A scalable energy-efficient con-

text aware distributed mobile data analytics platform for the fog. In *Advances in Databases and Information Systems*, pages 192–206. Springer, 2014.

[131] Matthew Malensek, Sangmi Lee Pallickara, and Shrideep Pallickara. Hermes: Federating fog and cloud nodes to support query evaluations in continuous sensing environments. *IEEE Cloud Computing*, 4(2):54–62, Mar 2017.