# Exploiting Geospatial and Chronological Characteristics in Data Streams to Enable Efficient Storage and Retrievals

Matthew Malensek, Sangmi Lee Pallickara, and Shrideep Pallickara

*Department of Computer Science*
*Colorado State University*
*Fort Collins, USA*
{*malensek, sangmi, shrideep*}*@cs.colostate.edu*

**Abstract**

We describe the design of a high-throughput storage system, Galileo, for data streams generated in observational settings. To cope with data volumes, the shared nothing architecture in Galileo supports incremental assimilation of nodes, while accounting for heterogeneity in their capabilities. To achieve efficient storage and retrievals of data, Galileo accounts for the geospatial and chronological characteristics of such time-series observational data streams. Our benchmarks demonstrate that Galileo supports high-throughput storage and efficient retrievals of specific portions of large datasets while supporting different types of queries.

*Keywords:* data storage, commodity clusters, distributed systems, scale-out architectures, observational streams, query evaluations

## 1. Introduction

There has been a steady increase in the number and type of observational devices. Data from such devices must be stored for (1) processing that relies on access to historical data to make forecasts, and (2) visualizing how the observational data changes over time for a given spatial area. Data produced by such observational devices can be thought of as time-series data streams; a device generates the packets periodically or as part of configured change notifications. Data packets generated in these settings contain measurements from multiple, proximate locations. These measurements can be made by a single device (e.g., volumetric scans generated by radars) or from multiple devices (e.g., sensors send data to a base station that collates multiple observations to generate a single packet).

Observational data have spatio-temporal characteristics. Each measurement represents a *feature* of interest such as temperature, pressure, humidity, etc. The measurement is tied to specific location and elevation, and has a timestamp

associated with it. While individual packets within an observational stream may not be large, (often in the order of kilobytes) the frequency of the reported measurements combined with increases in the number and type of devices lead to increasing data volumes.

### *1.1. Usage Scenarios*

Our targeted usage scenario is in the atmospheric domain where data from such measurements are used as inputs to weather forecasting models and visualization schemes. These usage patterns entail access to historical data to validate new models, identify correlations or trends, and visualize feature changes over time. We need to be able to access specific portions of the data efficiently to ensure faster completions of the aforementioned activities.

### *1.2. Research Challenges*

Research challenges in designing a storage framework for such observational data include the following:

1. *Support for a scale-out architecture:* An extensible architecture that can assimilate nodes one at a time to support increased data storage requirements.
2. *High throughput storage of data:* Given the number of data sources, we must be able to store data streams arriving at high rates. We measure *throughput* in terms of the total number of stream packets stored by the system over a period of time.
3. *Efficient retrievals of specific portions of the data:* Given the large data volumes involved we must support fast sifting of stored data streams in response to queries that target a specific feature at a specific time for a given geospatial area. To accomplish this, we must account for the spatial and temporal characteristics of the data during the storage process and in turn use this metadata for efficient retrievals.
4. *Fast detection of non-matching queries:* Often the query parameters are adjusted based on results from past queries. To support fine-tuning of queries, we must have accurate and efficient detection of situations when there are no data that match a specified query.
5. *Range query support:* We must be able to support range queries over both the spatial and temporal dimensions while ensuring that support for such queries do not result in unacceptable overheads.
6. *Dynamic indexing strategies:* The system should allow its indexing functionality to be adaptively reconfigured to better service different usage patterns and reduce latencies.
7. *Extensive data format support:* There are vast amounts of data stored in established scientific storage formats. Our system must not require a particular input format so that it is useful for researchers that have already invested in an existing format, and we must allow the system to read and understand a variety of metadata without any loss of fidelity from conversion.

8. *Efficient processing of stored data:* Our system should not only facilitate data retrieval, but also simplify launching distributed computations for processing data using the system's indexing semantics as inputs.
9. *Integrated support for visualization:* The system should provide functionality for visualizing data in a number of ways to facilitate analysis.
10. *Failure recovery:* We must account for any possible failures and data corruptions at individual nodes. Recovery from failures must be fast and consistent.

### *1.3. Contributions*

This paper describes the design of a demonstrably high-throughput geospatial data storage system, *Galileo*. The storage framework is distributed and is incrementally scalable with the ability to assimilate new storage nodes as they become available. The storage subsystem organizes the storage and dispersion of data streams to support fast, efficient range queries targeting specific features across the spatial and temporal dimensions. A dynamic indexing scheme is utilized to help the system respond to differing load conditions, and a flexible framework is provided to allow a number of observational data formats to be read and understood by the system. To sustain failures and recover from data corruptions of specific blocks the system relies on replication. Most importantly, our benchmarks demonstrate the feasibility of designing high-throughput data storage from commodity nodes while accounting for differences in the capabilities of these nodes. Leveraging heterogeneity in the available nodes is particularly useful in cloud settings where newer nodes tend to have better storage and processing capabilities.

### *1.4. Paper Organization*

In the following section, the architecture of Galileo will be discussed, including an overview of how data is stored to disk, the network layout, and how data is positioned and replicated within the system. Next, an overview of the system's scientific data format support infrastructure is detailed in Section 3, including information about the runtime plugin architecture and implementation of NetCDF format support. In Section 4, the dataset and query system will be described, followed by details of Galileo's built-in distributed computation API in Section 5. Section 6 explores using the system for client-side data visualization. Section 8 provides a brief survey of related technologies in the field, and Section 7 presents benchmarks of our system's capabilities. Section 9 reports conclusions from our research and discusses the future direction of the project.

### *1.5. Paper Extensions*

Since the initial publication of *Galileo: A Framework for Distributed Storage of High-Throughput Data Streams* [1], we have added several key extensions for this special edition issue. Updates to the system architecture are detailed, including block versioning, compression support, and cross-group replication.

We also added support for launching distributed computations on the data stored in Galileo, streaming visualization, and converting other storage types to our metadata format through a plugin architecture. Finally, a number of new benchmarks have been added, including a comparison with Hadoop and timing information for visualization, computation launching, graph reorientation, and metadata conversion.

## 2. System Architecture

Galileo runs as a *computation* on the Granules Runtime for Cloud Computing [2]. Granules is an ideal platform to build upon because it provides a basis for streaming communication between nodes in the system and for incoming data streams as well. As data enters the system, it can be sifted and preprocessed with the Granules runtime and then stored in a distributed manner across multiple machines with Galileo. When accessing data, users have the option of pushing their computations out to relevant Galileo *storage nodes* where they can be run locally to avoid incurring IO costs associated with transferring large amounts of data across a network. This also makes it possible to support distributed programming paradigms such as MapReduce [3].

Much like Google File System (GFS) [4] or Amazon's Dynamo [5] storage system, Galileo is a high-level abstraction that utilizes the underlying host file systems for storing data on physical media. This allows Galileo to be portable across operating systems and hardware while also coexisting with other files. It also means that Galileo does not require an entire disk or partition be devoted to the system. In Galileo, storage units are called *blocks* and are stored as files on the host file system. Each block is accompanied by a set of metadata that is specifically tailored for scientific applications.

As extremely large datasets can require massive computing resources, Galileo is designed with scalability in mind. The system employs a *shared nothing architecture*, meaning storage nodes operate autonomously and do not share their state with any other nodes. This simplifies the process of meeting increased storage demands because additional nodes can be added to the system without requiring excessive communication or migration of data.

To ensure the system is as fault-tolerant as possible, Galileo does not utilize master nodes or fixed entry points that could result in a single point of failure. In fact, Galileo will continue to provide query and storage facilities even when failures occur so that applications that do not require complete access to all information in the system can continue to run. Blocks are also replicated across machines to cope with general hardware failures and planned outages that are expected in distributed setting.

### *2.1. Granules*

Granules [2] is a light-weight distributed stream processing system. In Granules computations can be expressed as MapReduce or as directed cyclic graphs and the runtime orchestrates these computations on a set of available machines.

In Granules individual computations can specify a scheduling strategy that allows them to be scheduled for execution when data is available or at regular intervals specified in milliseconds. Computations in Granules can have multiple, successive rounds of execution during which they can retain state.

Granules is an open-source effort, and allows computations to be developed in C, C++, C#, Java, Python and R [6]. Some of the domains that Granules has been deployed in include bioinformatics, brain-computer interfaces [7], multidimensional clustering algorithms, handwriting recognition [8], and epidemiological modeling.

### *2.2. Blocks*

A Galileo block is an multi-dimensional array of data, similar to how data is represented in systems like SciDB [9, 10] or formats such as NetCDF [11] or FITS [12], although in the case of Galileo metadata files are stored separately on the file system alongside their respective data blocks. This separation simplifies operations: 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. Combined with the on-disk metadata journal, a storage node's entire state can be quickly recovered after a crash.

Blocks support seamless compression and decompression, which is useful for several operations. For instance, compression can be used as a means to conserve disk space when storing blocks on physical media, and also reduce network transfer times by decreasing the amount of raw data being exchanged. Compression can be turned on and off for network transfers or storage operations by users dynamically at runtime depending on whether it benefits their usage patterns or not.

Instead of just one or a few large directories containing all the blocks present on a machine, Galileo separates blocks into an on-disk hierarchical directory structure using the blocks' metadata. As the number of files in a directory increases, reading directory indexes becomes more and more time-consuming, so this structure spreads data across multiple directories to avoid this performance penalty. 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 point that is already close to the desired information. The number of subdirectories used for storing the data can be tuned at runtime to cope with shifting storage demands and disk usage patterns.

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 on-disk graph. The next level of subdirectories is determined by using the data's spatial information to compute a *Geohash* [13]. Geohashes are strings that can be used to divide data into arbitrarily-sized spatial bounding boxes, where shorter strings correspond with larger geographic regions, and therefore more blocks. 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 geographic dispersions of data. Further details of the Geohash algorithm are discussed in Section 2.5.

Our storage scheme offers a few advantages over using one large, contiguous file for storing blocks. For instance, the on-disk hierarchy encodes partial metadata in directory names that is available without needing to read any files directly. In addition, this scheme makes the on-disk storage format flexible; we can incorporate support for additional formats such as NetCDF [11], HDF5 [14], or FITS [12] 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.

### *2.3. Metadata*

When designing a database specifically for scientific data, the creators of SciDB [10] identified some major differences between scientific data and business-oriented data. Two of these differences involve file metadata: first, scientific data usually has a spatial aspect, involving location or elevation information. Additionally, scientific data storage needs are massive and continuously growing in size; systems dealing with such information should be able to handle data on the petabyte-scale. This generally requires an indexing scheme to speed up access.

Galileo aims to address these scientific needs as well. Each file in Galileo is accompanied by a rich set of metadata. The metadata contains spatial information, which can include elevation and a range of coordinates that form a bounding box or a single spatial point. This information can be used to query and apply computations on specific geographic regions. Since measurements are often performed over a range of time, temporal information is also encoded in files' metadata. This allows users to retrieve a wide array of constantly changing information bundled as a single dataset, or apply transformations across a specific time interval.

Versioning information is another another relevant metadata item to have when storing and processing scientific data. Versioning is used extensively in Google's BigTable [15] storage system for a variety of scenarios including garbage collection of old data. There are several situations in Galileo where versioning and garbage collection are useful. For instance, scientific users often need to be able to reproduce the steps taken to create a dataset. In this case, each intermediate step can be stored as a different version of the same Galileo file. Unlike the temporal ranges stored in metadata, versioning information is not bounded. In the case of extremely large datasets that are being used to make real-time predictions, the garbage collection feature could be used to remove old data that is no longer relevant to conserve disk space.

In an effort to make Galileo data blocks *self-describing*, meaning all the information needed to interpret the data is included in the files, metadata contains a number of user-defined *features*. Features could include environmental attributes such as wind speed, pressure, humidity, or some other application-specific attribute. Support is also included for encoding device identifiers in the metadata so datasets can be built from specific sensors or instruments. This gives the file format flexibility to fit a wide range of use case scenarios.

To cope with massive data storage needs, Galileo uses temporal and spatial metadata attributes to create a hierarchical graph-based index of the data residing in the system. This index is stored in main memory on the Galileo storage node, so locating data is as fast as possible.

### *2.4. Journaling*

Making complex changes to the underlying data structures in Galileo can often require many separate 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 the on-disk data stored in 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. The journal also allows the system to recreate its in-memory graph quickly and begin servicing queries without needing to re-read all the metadata from disk.

### *2.5. Geohashes*

The Geohash algorithm [13] can be used to divide geographic regions into a hierarchical structure. A Geohash is derived by interleaving bits obtained from latitude and longitude pairs and then converting the bits to a string using a base-32 character map. A Geohash string represents a fixed spatial bounding box. For example, the latitude and longitude coordinates of N 40.57, W 105.08 fall within the Geohash bounding box of *9xjqbce*. Appending characters to the string would make it refer to more precise geographical subsets of the original string. Figure 1 illustrates how regions are divided into successively more precise bounding boxes.

To obtain the latitude and longitude bits from an initial pair of coordinates representing a target point in space, the algorithm is applied recursively across successively more precise geographical regions bounding the coordinates. The remaining geographical area is reduced by selecting a halfway pivot point that alternates between longitude and latitude at each step. If the target coordinate value is greater than the pivot, a 1 bit is appended to the overall set of bits; otherwise, a 0 bit is appended. The remaining geographic area that contains the original point is then used in the next iteration of the algorithm. Successive iterations increase the accuracy of the final Geohash string.

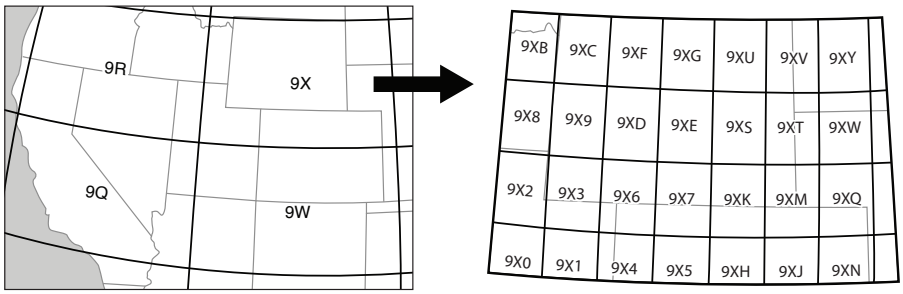


Figure 1: An illustration of successively increasing precision in the Geohash algorithm.

An appealing property of the Geohash algorithm is that nearby points will generally share similar Geohash strings. The longer the sequence of matching bits is, the closer two points are. This property is exploited in Galileo to support simple range-based spatial queries that return data in a given Geohash region, allowing users to specify more- or less-precise hashes to select smaller or larger areas. It is also possible to use Geohashes for quick proximity searches. In addition to queries, Geohashes can also be used to group similar data. If a collection of data gets too large, simply using more precise Geohashes allows the system to create more specific, and therefore smaller, groupings of data. This property also allows quick retrieval of similar data blocks for use in computations.

### *2.6. Network Organization*

Galileo employs many features of *distributed hash tables*, (DHTs) much like systems such as Chord [16], Pastry [17], or Dynamo [5]. Like Dynamo, Galileo is a *zero-hop* DHT, where each node knows about the network topology and hash algorithm used to route information directly to its destination. Individual storage nodes running on machines in the system are divided into *groups*. These groups can represent arbitrary collections of machines, or could be used to arrange machines with geographic locality or common hardware attributes. Each group is assigned its own UUID stream, so it is also possible to broadcast information to an entire group if necessary. Group members form a ring and communicate with their neighbors over a socket connection on a separate thread. This connection is used for detecting when a neighboring node has failed and maintaining replication levels when failures occur.

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. Using the group and storage node hashes, clients can determine a UUID for the particular node they wish to communicate with and begin publishing data on the node's UUID stream. In the system's present state, this scheme simply distributes data evenly across all available machines, but in

the future the algorithm could be changed to group data based on its content or metadata.

To cope with heterogeneous systems, machines can also join multiple groups or represent multiple "virtual" machines within a group by running multiple cooperating Galileo instances. This allows more powerful storage nodes to be added to the system later and still balance load across available machines efficiently.

Galileo is an *eventually consistent* system. Nodes and groups can be added or removed from the system at will, but this may affect the availability of data that must be migrated during changes to the network topology. This property allows applications to continue with their computations if they can be completed with partial information. Once data migrations are complete, the system resumes its usual operating state and all the information stored in the system is available once again.

### *2.7. Data Replication*

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 block during one year is less than 0.005 [18]. 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.

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. Scientific Data Format Support

While the scientific community has invested considerable time and effort in collecting and storing vast amounts of data, there has also been a great deal of investment in the formats used for storing these volumes of data. The storage formats in question could include simple comma-separated plain text files, relational database management systems, proprietary storage formats, distributed storage, or multi-dimensional arrays stored as binary files. This diversity in storage formats is largely due to organizations having unique usage patterns and storage requirements for their data.

Galileo has its own internal storage format that can be leveraged directly by users if they choose, but also provides rich functionality for supporting other formats. Innovation in storage formats is not our goal for Galileo. Instead, we wish to facilitate the storage, management, and retrieval of vast amounts of data in any storage format the end user desires. This approach ensures that the system will be useful without requiring long conversion processes that could be costly or result in partial data loss.

Much of Galileo's power is derived from its ability to reason about the data stored in the system, particularly its metadata. Therefore, in order to support multiple storage formats, we provide an interface that allows programmers to specify how relevant fields in a foreign format should be converted to Galileo's native format. If available, spatial, temporal, and feature data must be read from incoming data formats before being stored in the system. Other relevant fields can be included as well, if deemed necessary by the programmer. This approach is somewhat similar to databases such as Cassandra [19] or mongoDB [20] in its *schemaless* data representation. Queries and datasets can be built around the foreign data format once relevant fields have been stored as Galileo metadata.

### *3.1. Plugin Architecture*

Available plugins and their capabilities are loaded by clients at runtime. To maintain a separation of concerns in the system, format support is a client-side functionality; if reading and converting metadata was done at each storage node, then users would have to ensure that all nodes have the same plugins available. Utilizing this architecture also means that the client-side algorithm for file placement in the network only needs to deal with the native Galileo storage format.

At runtime, plugin capabilities are examined and a mapping of file extensions to plugins is created. Since file extensions are not always used or inaccurate, plugins also implement an interface method that can examine the headers of input files and determine whether or not the file can be interpreted by the plugin. Users can also choose to invoke a particular plugin directly during a storage operation if they are dealing with files of a specific known type.

The metadata conversion interface is simple: plugins are given an arbitrary array of bytes to interpret, and using this information a Galileo metadata object is produced by the plugin. This Galileo-formatted metadata is returned to the

system, and the file can then be dispersed and stored across the network as usual. Since the input file contents are not being modified in any way, there is no data loss involved in this process. In this case "conversion" simply refers to reading an input file and creating additional information about the file that can be understood by the system.

### *3.2. NetCDF*

One data storage format used particularly often in atmospheric sciences is the Network Common Data Format (NetCDF) [11]. NetCDF was developed at the University Corporation for Atmospheric Research (UCAR) primarily for massive meteorological datasets, and now provides a machine-independent interface that is relevant for storing multitudes of different data types. NetCDF files are self-describing, multi-dimensional arrays and can be read and written through rich software libraries supported by UCAR.

NetCDF files can be composed of ASCII text and binary data and contain three properties: *dimensions*, *variables*, and *attributes*. Dimensions are particularly relevant to Galileo as they are used to describe application-specific properties such as time, elevation, or spatial position. Variables represent the core data of a NetCDF file, and are represented by multi-dimensional arrays of homogeneous values. Attributes provide extended metadata that describes the variables in the file to aid in processing and analysis.

A wide range of software applications support NetCDF, partially due to its simplicity and relative ease of use. Due to the large install base of NetCDF users, it is the first external format Galileo supports. Our NetCDF plugin utilizes the Unidata NetCDF Java Library, which provides a rich interface for reading NetCDF files as well as several other observational data formats. In fact, the latest version of the library supports Hierarchical Data Format 5 (HDF5) [14], as well as a number of other formats including BUFR, CINRAD, NEXRAD-3, UniversalRadarFormat, GRIB, and OPeNDAP. Support for these formats make Galileo a useful across a wide array of scientific disciplines.

Since NetCDF does not impose a particular schema on its users, there is no guarantee that information such as latitude, longitude, or elevation are stored using the same NetCDF variable names. This issue is resolved by utilizing a set of naming conventions that users adhere to. One such convention is the Climate and Forecast (CF) metadata standards, which were developed by a number of international organizations. Our NetCDF plugin supports a simple text-based system for describing particular conventions and mapping their variables to Galileo metadata attributes. Support for a number of conventions is included with the plugin, and users can create their own mappings quickly with a simple text editor.

## 4. Information Retrieval: Datasets

Galileo's information retrieval process is different from traditional databases or key-value stores. Instead of matching user-submitted queries against the data

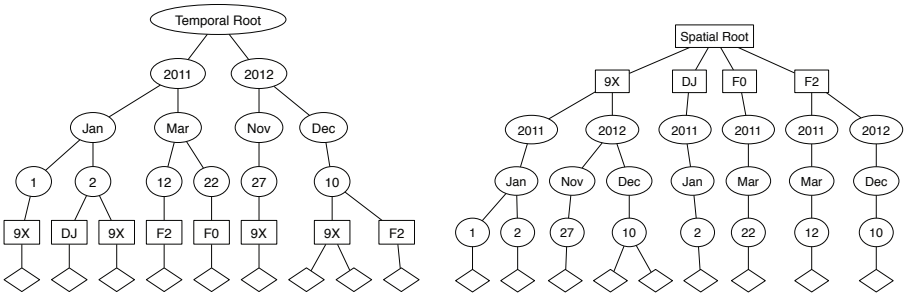


Figure 2: Two different views of the same in-memory graph. Circles represent temporal objects, squares represent spatial objects, and diamonds represent feature information.

available in the system and returning the raw data, Galileo streams metadata of the matching blocks back to the requestor incrementally and our client-side API transparently collates these metadata blocks into a traversable *dataset graph*. This dataset is a subset of Galileo's in-memory metadata graph, and describes the attributes of the blocks that match a query. This allows applications to determine how many result blocks are available and what their various attributes contain without needing to read any data from the disk. Once a dataset has been obtained, applications can have blocks transferred directly to them or further fine-tune the dataset by traversing through it or selecting more specific portions of the graph. Client applications can use this information to deploy computations on storage nodes that hold data relevant to the computation.

### *4.1. Traversing a Dataset*

Dataset graphs can be re-oriented and traversed by the user. For instance, some applications may place a higher importance on temporal information and decide to use it as the root of the graph, 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 graph would be re-oriented with spatial information as the root. Figure 2 illustrates the reorientation process on a simple example graph. 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 graph can also be re-oriented mid-traversal.

Allowing the metadata graph to be oriented in different ways based on access patterns also creates a number of possibilities for employing machine learning techniques to predict future usage and orient the graph accordingly for efficient traversals. This capability also makes it possible to react to large fluctuations in traffic; if Galileo is running as a backend storage platform for a website, it could reorient its index to cope with a large influx of traffic requesting a particular range or type of data.

## 5. Distributed Computation API

Galileo is tightly integrated with the Granules [2] cloud runtime, which supports expressing computations using the MapReduce paradigm or as directed, cyclic graphs. In this case, a computation simply refers to a distributed, executable task that is deployed and run across a cluster of machines. These computations could include operations such as data analysis, pre-processing, transformation, and visualization. Galileo runs within the Granules computation framework as well, underscoring the fact that computations can be fully fledged distributed applications. Since these operations often involve large datasets, we provide an integrated API to assist developers in deploying computations that operate on data stored in the Galileo file system.

To facilitate launching computations on particular subsets of data stored in Galileo, the system allows users to launch Granules computations using a Galileo dataset as one of its parameters. The client-side API accepts a dataset graph returned from a Galileo query as its input, and then launches specified computations on the nodes that host data relevant to the computation. In a cloud-based setting, this architecture ensures that the penalties incurred from IO latencies are as low as possible. More sophisticated user-specified task placement is also possible for situations where a problem requires a more specific execution pattern.

In general, once a computation has launched on a node it begins reading its input data. In the case of a MapReduce computation, processing done in the map phase will often be passed on to a reduce phase where results are collected. Once collected, results can be written directly back into the Galileo file system to be dispersed across the cluster and replicated. For visualization or other client-based activities, the results can also be returned to the original client that started the query and computation process. The Galileo API can be invoked at any time during a Granules computation's lifecycle, meaning that subsequent stages of execution can continue to be placed near their relevant data blocks and partial computation progress can be stored in Galileo at user-defined intervals, if necessary.

### *5.1. Usage Scenarios*

Since Granules provides support for computations that are represented as directed, cyclic graphs, there are several applications beyond the standard MapReduce framework that can also benefit from Galileo. Allowing cycles and long-running computations provides users with an avenue for continuously processing observational data from a range of sources. These data items often require pre-processing before storage, which can be accomplished by launching a Granules computation on incoming data streams and then storing the results in Galileo. The Microsoft Dryad project [21] is a prime example of a graph-based computation framework, which gives users flexibility in determining data flow and dependencies during processing. Its graph-based approach allows for a number of interesting applications, such as a distributed SQL query engine.

*Workflow* systems are commonly used in the sciences to model computations and their dependencies. These systems, such as Kepler [22] and Pegasus [23] frequently involve executing complex programs on input datasets. Structurally, these computations are graph-like, and require a particular sequence of events to unfold during their execution. In general, a workflow involves querying large databases of collected observations and then having results streamed from a central database server to the clients that request it. Galileo can provide similar functionality for scientific workflow users while pushing computations directly to their data, thus speeding up the entire workflow process.

## 6. Client-Side Visualization

One common use case for scientific data stored in a distributed file system like Galileo is real-time visualization. These visualizations could include biological, medical, or meteorological data and can provide insights to researchers in these fields. Another related area includes Geographic Information Systems, (GIS) which can involve visualizing organization-specific data with an object-oriented approach. GIS systems are often powered by a database backend which is used to provide detailed information about the objects in the system. Common uses of GIS include military planning, municipal utility management, and cartography.

Galileo includes a client-side visualization API that integrates with its query and dataset workflow. This functionality is comparable to technologies such as Google's Fusion Tables [24]. Fusion Tables' primary focus is on making data access and visualization simple and efficient for users by abstracting away the underlying database and focusing on the useful actions that users may wish to carry out on their data. Much like Fusion Tables, Galileo supports importing data from a variety of sources through its data format API. Once data has been imported into the system, it can be queried and filtered before being passed on to the visualization API. Our current implementation can export data to Keyhole Markup Language, (KML) enabling visualization in products such as Google Earth. This functionality could also be implemented as a web service that could be queried from Google Earth or other client-side applications dynamically.

Specific regions and attributes can be selected using Galileo's query system, and in some cases simply reading metadata provides enough information to begin visualization. If the specific data items required for visualization cannot be accessed using simple queries, results from an initial query can be passed through one or more MapReduce phases and output to clients. By streaming results to clients, Galileo provides an excellent interface for incrementally updating a visual representation of data in real time. The API also makes it possible to export data directly for use in graphics or plotting software.

Another important aspect of visualization is time series information. The metadata stored in Galileo includes a timestamp that could represent an actual calendar date, or versioning metadata could be incremented each time new data is recorded in a region. This information can be streamed from the system incrementally to provide real-time video-based visualization. For continuously-updating visualizations, Galileo provides a callback functionality that monitors

when new data is added to the system that matches active queries. When new relevant data has entered the system, clients are notified and can update their visual representation.

Our current visualization functionality is implemented as a Granules MapReduce computation through the Galileo Computation API. When users submit a query the resulting metadata is streamed back to the client, which then automatically launches visualization computations. Data is parsed in the Map phase and passed on to the Reduce phase where it is collated and written to one or more files that contain the processed data. In the case of KML output, these objects include geographic properties and shape information. The output from this component can also be passed on to a plotting program for visualization in a number of graphical formats.

## 7. Benchmarks

To test the capabilities of Galileo's storage system, we ran benchmarks on a 48-node Xeon-based cluster of servers with a gigabit Ethernet interconnect. Each server in the cluster was equipped with 12GB of RAM and a 300-gigabyte, 15,000 RPM hard disk formatted with the ext4 file system. The benchmarks were run on the OpenJDK Java Runtime Environment, version 1.6.0_22.

One billion (1,000,000,000) random data blocks were generated for the experiments and dispersed across the 48 machines, each containing 1,000 simulated sensor readings and accompanying metadata. Readings had a feature set that included pressure, temperature, and humidity. This configuration resulted in blocks that consumed 4,000 bytes of disk space each and metadata files that were approximately 120 bytes each. Random temporal ranges were chosen within the years of 2002-2011, and random spatial locations for the data were constrained to the continental United States. In the interest of testing the system with the biggest dataset possible, replication was disabled on the storage nodes to conserve disk space. The total size of the billion-block dataset was approximately 8TB.

To simulate source "sensor arrays" that stream data into the system, machines outside the cluster were used to generate the random blocks and stream them into the system across the network. We also generated "realistic" data by having subsequent blocks share some characteristics with previously generated blocks. For example, one block may be generated with metadata for July 1st at 3:00 and the next block would contain information about readings from July 1st at 4:00. Both the random and realistic datasets were used in our benchmarks.

### *7.1. Storage*

Data blocks enter the system as a stream of bytes. Once the data is received, the metadata portion of the block is de-serialized so it can be read and indexed in the in-memory graph, and then the data is written to disk. Table 1 contains timing information for each part of the process in a scenario where a 10,000-block burst of data is streamed into the system.

Table 1: Per-Block Mean Storage Time: 10,000 Blocks

| **Operation** | **Mean Time (ms)** | **Standard Deviation (ms)** |
|---|---|---|
| De-serialization | 0.0298933 | 0.0283722 |
| Indexing | 0.0186978 | 0.0129819 |
| Writing to Disk | 0.133983 | 0.0425601 |

In these tests, the majority of the time storing a block is spent writing to disk. In cases where files much larger than 4kB are stored, the overhead incurred by de-serialization and indexing should be even more insignificant when compared to write times.

Our storage scheme involves creating several filesystem-level objects. As a block enters the system, its metadata and content are stored separately on disk, which creates two files and therefore consumes two inodes on the ext4 file system. In addition, directories are also created for the on-disk hierarchy that resembles the in-memory graph. Table 2 contains a summary of disk space usage (using the number of 1024-byte blocks consumed) and inode utilization as more blocks are added to the system.

Table 2: Disk Usage

| **Number of Blocks** | **1-K Disk Blocks Used** | **Inodes Used** |
|---|---|---|
| 1,000,000 | 9,020,432 | 2,069,110 |
| 10,000,000 | 88,009,576 | 21,217,563 |
| 20,000,000 | 166,358,044 | 40,239,375 |

### *7.2. Recovery*

In the event of a system crash, power loss, or scheduled reboot, Galileo must recover its state from disk after being restarted. Recovery first involves reading the system journal, which contains enough edge information to restore the system graph that is used to create datasets. Table 3 outlines recovery times for a single node after a system failure for three scenarios involving different number of stored blocks.

### *7.3. Retrieval*

For our initial query tests, 100 million random blocks were submitted and stored in the system in the manner discussed earlier. Then we queried for all pressure readings generated in July of 2011 within a spatial range roughly covering the state of Colorado.

Table 3: Recovery Times

| **Blocks Stored at a Node** | **Graph Recovery (sec)** |
|---|---|
| 1,000,000 | 3.062 |
| 10,000,000 | 28.91 |
| 20,000,000 | 65.18 |

Table 4 summarizes the results of the query. It includes timing information for creating a dataset from memory, creating a dataset from disk, (simulating post-failure conditions) and also for transferring raw data blocks across the network to a client ("downloading" the dataset contents).

Table 4: Small Dataset Query Results

| **Result Type** | **First Result (ms)** | **Last Result (ms)** |
|---|---|---|
| Dataset (in-memory metadata) | 60.76 | 668.42 |
| Dataset (metadata from disk) | 84.81 | 1309.12 |
| Full Block Download | 542.96 | 5769.21 |

The query returned 105,556 blocks, which results in a dataset of about 10 megabytes and roughly 400 megabytes of raw block data.

Our next query benchmark involved the entire billion-block dataset. We created six different query types to test various access patterns:

1. *No Match:* This is the case where none of the blocks in the system match the query.
2. *One Match:* This is a specially designed query where only a single block (of $10^9$ blocks) matches the query.
3. *Standard Query:* The query requests blocks for a particular feature over a given geospatial location at a specific time (specified using year, month, day, hour).
4. *Temporal Range:* Returns all blocks with the desired feature for a given geospatial area that fall between a specified start and end time range.
5. *Spatial Range:* Blocks that fall within coarser or finer grained ranges of a specified geospatial bounding box. Our tests include querying for blocks within the entire continental United States, Colorado, and the northeast quarter of Colorado.
6. *Exhaustive feature search:* Within a given year, locate all measurements of a specific feature regardless of the corresponding geospatial location. This query evaluation requires an exhaustive search of the year's subgraph.

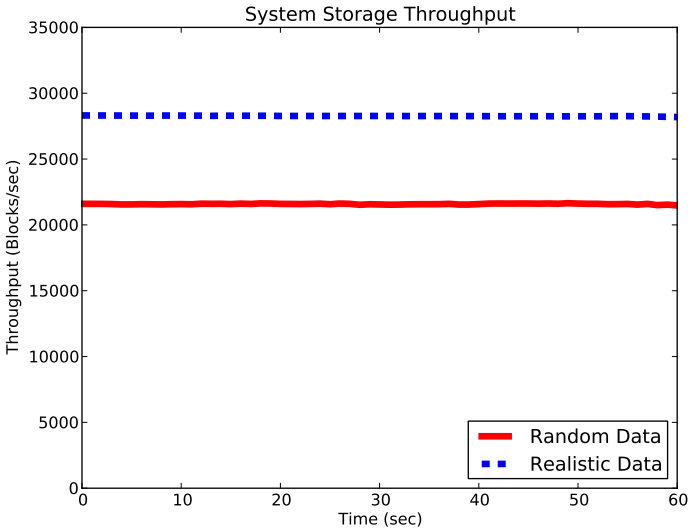


Figure 3: Cumulative storage throughput over 60 seconds

Timing data for each query type is outlined in Table 5. This includes the time to return the dataset's metadata components, the size of the dataset, how long the system spent creating the dataset (which requires traversing the in-memory graph) and also how long it took to download the block information for each dataset. Each data point represents the result of running queries 100 times to ensure stable results were collected; we also report the corresponding standard deviations.

### *7.4. Storage Throughput*

To further test the storage capabilities of Galileo, we also performed a cumulative storage throughput test. Data blocks were streamed into the system from five separate sources outside the cluster and stored on disks that were initially empty. We sampled the number of blocks being stored for a 60 second window at each node, and then the readings were summed to determine the cumulative storage rate. Results from this benchmark are depicted in Figure 3.

These results show that our realistic data simulation does yield higher performance than completely random data. This trend is largely due to disk write access patterns; in the case of random data there will generally be no commonality in destination directory between subsequent blocks, whereas the realistic dataset produced less variance in destination directory. For the realistic scenario we were able to achieve a sustained cumulative throughput of 28,269 blocks per second.

Table 5: Query Results for 1 Billion Blocks: Each Data Point is the Result of Repeating the Experiment 100 Times

| **Query** | **First Result (ms)** | **First Result SD (ms)** | **Last Result (ms)** | **Last Result SD (ms)** | **Dataset Size (blocks)** | **Dataset Creation** | **Creation SD (ms)** | **Download Time (ms)** | **Download SD (ms)** |
|---|---|---|---|---|---|---|---|---|---|
| No Match | 42.09 | 0.67 | 47.05 | 1.72 | 0 | 0.01 | 0.004 | N/A | N/A |
| One Match | 42.96 | 1.07 | 50.39 | 4.29 | 1 | 0.01 | 0.008 | 50.47 | 4.22 |
| Standard Query | 44.10 | 5.26 | 55.57 | 9.11 | 1,411 | 0.02 | 0.01 | 241.45 | 69.05 |
| Temporal Range | 47.54 | 5.40 | 588.80 | 17.12 | 98,535 | 0.29 | 0.57 | 9,142.36 | 119.92 |
| Spatial Range (US) | 48.07 | 14.99 | 261.81 | 26.50 | 31,413 | 0.05 | 0.01 | 1,845.67 | 42.97 |
| Spatial Range (CO) | 43.08 | 0.45 | 57.73 | 8.92 | 1,643 | 0.01 | 0.01 | 252.03 | 41.63 |
| Spatial Range (NE CO) | 42.81 | 0.52 | 57.23 | 2.45 | 398 | 0.01 | 0.01 | 62.13 | 9.50 |
| Exhaustive Feature Search | 53.97 | 2.84 | 64,069.30 | 444.42 | 8,230,612 | 3.66 | 0.17 | 459,297.52 | 169.33 |

*7.5. Launching Computations*

To test the latencies involved with launching a computation through the Granules runtime, we ran 100 iterations of a demo application that queried the system and then launched a computation on its resultant metadata graph. In this benchmark, all blocks in the continental United States that were created on February 1st, 2001 at 12:22 am were queried, which returned 164,852 matching blocks. The computation that was launched read corresponding data blocks from disk, performed a SHA-1 checksum on the blocks to ensure their integrity, and then returned the hashes to the client application. The steps taken in this experiment were as follows:

1. Submit a query to the system
2. Wait for query results to be streamed back to the client
3. Collate results into a complete metadata graph
4. Launch the computation on the graph
5. Wait for all checksum results

Table 6 details the results of this experiment; the entire process took about 3.6 seconds to complete, with roughly 90 milliseconds spent performing the computation on average. Untimed code paths represented about 14.71 milliseconds of the overall time, and about 647.72 MB total data was read from disks across all machines in the system.

Table 6: Computation Launch Benchmark: 100 runs

| **Operation** | **Mean Time (ms)** | **Standard Deviation (ms)** |
|---|---|---|
| First Query Result | 257.206 | 20.2707 |
| Last Query Result | 3506.57 | 64.8951 |
| Computation Time | 89.9173 | 11.0152 |
| Total Time | 3611.2 | 60.5 |

*7.6. Metadata Conversion*

One key feature that ensures Galileo can be used for a variety of problems is its format support. In this experiment, we benchmarked our NetCDF file format plugin to determine how much overhead was incurred by converting metadata on the client side before storing it in the system. Data was submitted to the system from outside the network, read by the format conversion plugin, and then the data and its accompanying metadata were stored in the system as usual.

Dataset 1 is an example NetCDF file provided by UCAR from the Community Climate System Model (CCSM). Dataset 2 is another NetCDF example

with humidity readings. Datasets 3 and 4 are output from the Weather Research and Forecasting (WRF) Model [25], which is a next-generation mesoscale numerical weather prediction system designed to serve both operational forecasting and atmospheric research needs. The WRF model provides a 3-dimensional variational data assimilation. To generate the time-variant dataset for a longer time period, we repeatedly executed the WRF model every hour for the area under consideration. Each of the model run includes the resultant sub-dataset associated with the internal time steps. The datasets are encoded as NetCDF format and follow the CF convention.

Table 7: Metadata Conversion: 1,000 Runs

| **Dataset** | **Size (MB)** | **Features** | **Mean Time (ms)** | **SD (ms)** |
|---|---|---|---|---|
| 1 | 59.0 | 5 | 0.3343 | 0.3773 |
| 2 | 2.7 | 12 | 0.5269 | 0.3421 |
| 3 | 18.0 | 121 | 2.8810 | 0.5187 |
| 4 | 1608.55 | 124 | 2.9473 | 0.3024 |

We can conclude from these test results, shown in Table 7, that file size does not greatly influence the amount of time the system spends reading metadata; rather, the amount of features being read from the file affects conversion time the most. This property results in a tradeoff for Galileo users depending on whether they need fast storage times or richer query support. This also means that the particular NetCDF storage convention being used will have an affect on conversion times.

### *7.7. Visualization*

To evaluate the effectiveness of our client-side visualization API, we ran benchmarks on a query that requested blocks on a specific day from a spatial area that spanned the entire continental United States. The query resulted in 3,478 matching blocks. The benchmark was run 100 times. Table 8 contains the amount of time elapsed as the dataset is streamed back to the client; this feature allows multi-threaded applications to begin updating their graphical display before all data has been received.

Table 9 provides a summary of the visualization results, including the time elapsed before the first result is returned, the total time, and the time taken to write the file to disk in Keyhole Markup Language (KML). This process could also be implemented as a web service, in which case the file write times provide an estimate of transfer times over a fast network connection.

### *7.8. HDFS*

While we have demonstrated that Galileo is a viable storage platform for a number of use cases, it is also beneficial to compare its performance with

Table 8: Time elapsed as data is streamed incrementally back to a client for visualization

| **Percentage Complete** | **Total Time Elapsed (ms)** | **SD (ms)** |
|---|---|---|
| 25 | 110.345 | 5.4683 |
| 50 | 130.462 | 14.1453 |
| 75 | 150.948 | 20.35 |
| 100 | 179.318 | 55.8448 |

Table 9: Visualization benchmark results

| **Operation** | **Time (ms)** | **Standard Deviation (ms)** |
|---|---|---|
| Processing First Result | 90.8379 | 2.5547 |
| Processing Last Result | 179.318 | 55.8448 |
| Writing KML | 28.5625 | 3.8344 |

other distributed file systems. For this comparison, we choose the de facto standard open-source MapReduce implementation, Hadoop, and its distributed file system, HDFS. We used Hadoop and HDFS release 1.0 on the same 48-node cluster we tested Galileo on and leveraged the Hadoop Java API for file system access.

Since HDFS does not provide the same indexing and query capabilities as Galileo, we ported several Galileo APIs to run on HDFS, including the in-memory graph and native Galileo data types. Graph-based indexing was done on the client side to avoid making massive changes to the underlying Hadoop architecture. We stored blocks generated from our "realistic" dataset in both systems. Table 10 summarizes read and write performance of HDFS and Galileo.

The performance profiles found in our benchmarks highlight how the two systems diverge in their design and intended use cases. In Galileo, where target usages involve large numbers of small files being stored and retrieved, performance suffers somewhat when reading or writing a single block at a time. On the other hand, Galileo excels when dealing with larger quantities of files. The difference in single-block performance on Galileo is largely due to the fact that a query must be executed before the location of the desired data is found, whereas in HDFS the query operation was done on the client side and then the file was requested from HDFS directly. In addition, Galileo is an event-based system, meaning nodes must explicitly notify clients when operations have completed rather than simply closing a connection. Both of these factors contribute to the overhead incurred when dealing with a single file, but these penalties become much less significant when dealing with large quantities of files.

It has been well-documented that HDFS is most efficient when dealing with

large files, [26, 27, 28, 29] which also helps explain the performance disparity between the two systems. Liu et al. [26] circumvented this limitation by combining many small files into larger, indexed files before storing them in HDFS. Version 0.18.0 of Hadoop also introduced HAR (Hadoop Archive) files, which provide a means to store smaller files as indexed archives. These techniques could improve the performance of HDFS in our tests, but it is clear that dealing with this particular use case was not one of its original goals.

There are also a few architectural differences between Galileo and HDFS that impact performance. In HDFS, the *Namenode* handles indexing and storage operations and represents a single point of the failure in the system. In Galileo, clients directly contact nodes that each retain their own index. The single index on the Namenode in HDFS makes it difficult to store a large number of files due to memory constraints; with today's modern hardware storing billions of files would not be possible in HDFS [29]. Like Galileo, HDFS spreads its on-disk files across a number of subdirectories, but they are not grouped together based on metadata for more efficient access patterns.

Ultimately, the performance differences seen in these benchmarks emphasize that the use cases Galileo targets are much different than those targeted by Hadoop and HDFS. These results do not represent a weakness in HDFS but rather underscore a difference in objective. It is clear that in our case, designing a specialized system can provide significant performance benefits over using a more general solution.

### *7.9. Graph Reorientation*

Allowing the in-memory metadata graph to be reoriented by users or automatically by the system at runtime provides a powerful way to adaptively improve the performance of the system. For this benchmark, we recorded the time it takes to completely reconstruct the metadata graph and use time, space, or feature information as the starting point for traversals. Table 11 contains timing information for this benchmark.

The results from this experiment show that reorientating the graph on-the-fly at runtime on the storage nodes is certainly a viable option to increase performance. The graph API can also be used locally by clients to reorient query results to better suit their processing needs. Exact timing for these operations will vary depending on the data stored in the system; in this benchmark, the temporally-oriented graph resulted in fewer object creations due to the resulting graph having fewer vertices than the spatial and feature graphs.

## 8. Related Work

Hadoop [30] and its accompanying file system, HDFS [18] share some common objectives with Galileo. Hadoop is an implementation of the MapReduce framework, and HDFS can be used to store and retrieve results from computations orchestrated by Hadoop. A primary difference between HDFS and Galileo is the role of metadata in the two systems; HDFS is designed for more general-purpose storage needs, and cannot perform the indexing optimizations Galileo's

Table 10: Write Performance: Galileo and HDFS

| **Operation** | **Galileo (ms)** | **HDFS (ms)** | **SD: Galileo (ms)** | **SD: HDFS (ms)** | **Galileo Blocks/Sec** | **HDFS Blocks/Sec** |
|---|---|---|---|---|---|---|
| Read 1 block | 49.26 | 2.2808 | 3.92 | 0.5172 | 20.30 | 438.4426 |
| Read 100 blocks | 52.10 | 165.381 | 3.89 | 12.42 | 1,919.38 | 604.6643 |
| Read 200 blocks | 57.61 | 306.379 | 7.72 | 21.47 | 3,471.61 | 652.7862 |
| Read 400 blocks | 63.21 | 603.634 | 9.18 | 30.07 | 6,328.11 | 662.6531 |
| Read 800 blocks | 122.68 | 1,208.5 | 28.60 | 46.87 | 6,521.03 | 661.9776 |
| Read 1600 blocks | 250.12 | 2,404.7 | 39.80 | 66.59 | 6,396.92 | 665.3636 |
| Write 1 block | 51.62 | 22.19 | 7.90 | 43.55 | 19.37 | 45.05 |
| Write 100 blocks | 71.80 | 1,738.61 | 12.27 | 277.19 | 1392.75 | 57.51 |
| Write 200 blocks | 120.39 | 3,491.35 | 12.95 | 369.7 | 1661.26 | 57.28 |
| Write 400 blocks | 191.22 | 6,047.61 | 14.10 | 673.87 | 2091.83 | 66.14 |
| Write 800 blocks | 335.80 | 11,939.9 | 14.05 | 754.99 | 2382.37 | 67.00 |
| Write 1600 blocks | 612.27 | 23,996.9 | 21.32 | 1,318.68 | 2613.22 | 66.67 |

Table 11: Metadata Graph Reorientation Benchmark

| **Graph Root** | **Time (ms)** | **Standard Deviation (ms)** |
|---|---|---|
| Temporal | 2914.28 | 20.041 |
| Spatial | 3722.69 | 23.237 |
| Feature | 3641.99 | 19.066 |

geospatial metadata makes possible. HDFS is also tuned for a relatively small number of large files rather than a large number of small files as in Galileo. In addition, the Granules runtime allows computations to build state over time, which contrasts with Hadoop's run-once semantics.

The Hadoop and HDFS combination has been used specifically for geospatial data [31, 32]. Akdogan et al. found that the MapReduce paradigm is effective for a number of geospatial operations and scales linearly as nodes are added to the system [31]. Their implementation uses an index based on Voronoi diagrams, which helps speed up operations on geospatial areas, but does not include a temporal component.

SciDB [10, 9] also shares many characteristics with Galileo. It is a science-oriented database management system (DBMS) which deals with multi-dimensional arrays of data in a shared nothing architecture. SciDB has modular support for data processing and querying facilities, allowing users to write their own extensions to run within SciDB. This makes writing powerful, application-specific queries possible. Conversely, Galileo places computational responsibilities outside the system by utilizing the Granules [2, 33] framework for processing information, but the two are tightly integrated. Another key difference between the two systems is metadata handling. In SciDB, metadata and information about nodes in the system are indexed in a centralized *system catalog* which is backed by the PostgreSQL Object-Relational Database Management System (ORDBMS) [34]. Galileo distributes metadata and index information across all the nodes in the system.

PostGIS [35] provides an alternative approach to storing data with geospatial attributes: instead of being designed as a standalone system, it is an extension that runs on the PostgreSQL ORDBMS [34]. PostGIS includes geospatial data types and queries that coexist with traditional database functionality. The geospatial queries allow some processing work to be offloaded to the database itself. PostGIS is an ideal system for users with information that fits the tabular database storage model well, but in general multi-dimensional arrays are often a better fit for many forms of scientific data, as discovered in a panel held by the SciDB creators [9]. Scaling PostGIS may also be more difficult, as scaling options frequently involve replicating the database to other servers or splitting data manually between multiple servers, complicating the application logic used to interact with the database.

BigTable [15] is a database-like storage platform that maps row, column, and time values to byte arrays. In BigTable, data is stored in lexicographic order by row keys. Rows with consecutive keys are grouped into *tablets*, which are distributed across machines to provide load balancing. Since multiple versions of data can be present in the database at a given time, timestamps are used to distinguish between different versions. BigTable stores its data on the Google File System [4], which handles the splitting and distribution of files. While BigTable has been used for Google Earth, queries that are explicitly geospatial are not supported by the system.

Cassandra [19] was created by Facebook for dealing with massive amounts of textual data in the form of user message inboxes. Unlike other distributed data stores that focus on read performance, Cassandra is heavily optimized for write-heavy workloads. This feature is provided by doing extensive journaling and flushing large amounts of buffered data to the disk while performing large, sequential writes. Cassandra is similar to BigTable [15] in its map-based data model and Dynamo [5] in its network organization. In Cassandra, node addition and removal is a more involved process than in Galileo and may require data migration. Cassandra is also not designed for geospatial queries and generally is deployed across hundreds of machines rather than thousands.

PAST [36] is a peer-to-peer storage network based on the Pastry [17] distributed object location and routing substrate. PAST is DHT-based and uses a consistent hashing algorithm to distribute files evenly across all the storage nodes in the system, but facilities for data center or rack locality-based placement are not provided. Each node maintains a routing table so it can forward requests towards their destination, which means that file requests can be submitted from any node participating in the network. A caching mechanism is employed to speed up queries; if a file is being requested often, nodes will cache a local copy until its popularity drops. To cope with load balancing issues, PAST provides an innovative feature that allows nodes to maintain "pointers" to files in the system. These pointers act as a logical location for a file while referring to the file's actual physical location. This way the properties of the DHT can be maintained while still being able to relocate files in the system. Unlike Galileo, PAST does not have an indexing or query mechanism, so retrieval requires an exact file name.

SciMATE [37] also supports scientific data formats with an extensible plugin API. The system operates on data in-place rather than requiring it be imported before processing, and provides extensions to the standard MapReduce API to help facilitate shuffling large amounts of key-value pairs. In SciMATE, foreign data formats are read and interpreted by the system, and then a uniform interface is presented to programmers wishing to process the data.

Another system that deals with the storage of many small files was developed by Thain and Moretti [38] in the Chirp distributed grid file system [39]. Instead of simply combining a number of small files into larger files as done in many existing systems, this work focuses striking a balance between the strengths of FTP and NFS to provide better performance for small files in a distributed setting.

## 9. Conclusions and Future Work

### *9.1. Conclusions*

A shared-nothing architecture allows incremental addition of nodes into the storage network with a proportional improvement in system throughputs. Efficient evaluation of queries is possible by:

1. Accounting for spatio-temporal relationships in the distributed storage of observational data streams.
2. Separating metadata from content.
3. Maintaining an efficient representation of the metadata graph in memory.
4. Distributed, concurrent evaluation of queries.

Continuous streaming of partial results to a query enables us to achieve faster response times and provide real-time visualization support. Returning only the metadata associated with the content in the query response allows selective downloads and quick estimates for the total size of the dataset and expected download times. Two query evaluation features in our system enable fine-tuning of queries: fast turnarounds for queries with non-matching data and support for range-queries over the spatial and temporal dimensions.

The use of journaling at individual storage nodes allow us to make (and complete) complex structural changes to on-disk data despite failures that may take place at the node. Journaling also reduces recovery times after a failure. Replication of content allows us to sustain failures and data corruptions while satisfying queries that match data held in affected blocks.

We provide rich support for reading data from other storage formats through our flexible plugin framework. Our experience with assimilating NetCDF helped refine our plugin architecture and we expect that this will help as we proceed with supporting other observational data formats. NetCDF is one of the most popular observational data formats and we expect that this should allow us to leverage translators that are available for NetCDF. This plugin functionality is used to provide metadata information to the system so it can understand and reason about the data it is storing; our indexing strategy can be dynamically reconfigured at runtime to suit a number of usage patterns and reduce latency. The indexing scheme also provides a flexible means for users to traverse subsets of the data in the system and then execute distributed computations using the datasets directly as input parameters. Our visualization API facilitates novel analysis and research pursuits on the data stored in the system as well.

Finally, our benchmarks demonstrate the feasibility of designing a scalable storage system from commodity nodes. We also show that our specialized approach can outperform a general distributed storage solution in a number of situations.

### *9.2. Future Work*

While exact-match and range-based queries are useful for a number of applications, we plan to continue to add functionality to the query system. This

may involve implementing support for an existing query language or creating a simple language that interacts with our dataset format directly. Another primary focus of our future work lies in contacting only a subset of the nodes in the system while processing a query, which will greatly improve Galileo's scalability and responsiveness.

A possible improvement to the on-disk storage format would involve combining multiple blocks, or possibly even entire directory structures, into single indexed blocks. This approach will reduce inode consumption and may allow for faster disk access patterns; often queries will involve blocks that are spatially or temporally similar, so combining the related blocks into a single file would reduce the number of file open and close operations. Another option may be to combine metadata files to improve recovery times and dataset generation. We also may experiment with other file systems that support dynamic inode allocation, but we do not plan on making any specific file system a prerequisite for using Galileo.

The PAST [36] peer-to-peer storage system provides an innovative pointer feature that could benefit Galileo as well. When nodes are added or removed from the system, pointers can be created so that data does not have to be fully migrated between nodes immediately for the system to stay operational. Data could also be migrated on-demand; when a query requests data that is referenced by a pointer, the relevant blocks could be transferred to their new parent storage node on the fly. This feature should provide tremendous performance and availability benefits.

Finally, the authors of SciDB determined that one common complaint among scientific users was that they spent a large amount of time waiting for data to be stored in a system before they could begin their research [10]. This problem was solved in SciDB by allowing *in situ* access to data, where the system can apply computations to datasets that haven't been previously entered into the system. For this reason, we plan to allow the Galileo computation API to be used on files stored on local file systems as well.

## References


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

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

[3] J. Dean, S. Ghemawat, Mapreduce: Simplified data processing on large clusters, Communications of the ACM 51 (2008) 107–113.

[4] S. Ghemawat, H. Gobioff, S. Leung, The google file system, in: ACM SIGOPS Operating Systems Review, volume 37, ACM, pp. 29–43.

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

[6] K. Ericson, S. Pallickara, Adaptive heterogeneous language support within a cloud runtime, Future Generation Computer Systems (2011).

[7] K. Ericson, S. Pallickara, C. Anderson, Analyzing electroencephalograms using cloud computing techniques, in: Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on, IEEE, pp. 185–192.

[8] K. Ericson, S. Pallickara, C. Anderson, Handwriting recognition using a cloud runtime, Colorado Celebration of Women in Computing (2010).

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

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

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

[12] D. Wells, E. Greisen, R. Harten, Fits-a flexible image transport system, Astronomy and Astrophysics Supplement Series 44 (1981) 363.

[13] W. Contributors, Geohash, Wikipedia.org (2011).

[14] Q. Koziol, 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).

[15] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, R. Gruber, Bigtable: A distributed storage system for structured data, ACM Transactions on Computer Systems (TOCS) 26 (2008) 4.

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

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

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

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

[20] mongoDB Developers, mongodb manual, http://www.mongodb.org/ (2011).

[21] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly, Dryad: distributed data-parallel programs from sequential building blocks, ACM SIGOPS Operating Systems Review 41 (2007) 59–72.

[22] J. Wang, D. Crawl, I. Altintas, Kepler+ hadoop: a general architecture facilitating data-intensive applications in scientific workflow systems, in: Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science, ACM, p. 12.

[23] E. Deelman, G. Singh, M. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. Berriman, J. Good, et al., Pegasus: A framework for mapping complex scientific workflows onto distributed systems, Scientific Programming 13 (2005) 219–237.

[24] H. Gonzalez, A. Halevy, C. Jensen, A. Langen, J. Madhavan, R. Shapley, W. Shen, J. Goldberg-Kidon, Google fusion tables: web-centered data management and collaboration, in: Proceedings of the 2010 international conference on Management of data, ACM, pp. 1061–1066.

[25] J. Michalakes, J. Dudhia, D. Gill, T. Henderson, J. Klemp, W. Skamarock, W. Wang, The weather research and forecast model: Software architecture and performance, in: Proceedings of the 11th ECMWF Workshop on the Use of High Performance Computing In Meteorology, volume 25, World Scientific, p. 29.

[26] X. Liu, J. Han, Y. Zhong, C. Han, X. He, Implementing webgis on hadoop: A case study of improving small file i/o performance on hdfs, in: Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on, IEEE, pp. 1–8.

[27] B. Dong, J. Qiu, Q. Zheng, X. Zhong, J. Li, Y. Li, A novel approach to improving the efficiency of storing and accessing small files on hadoop: a case study by powerpoint files, in: Services Computing (SCC), 2010 IEEE International Conference on, IEEE, pp. 65–72.

[28] T. White, Hadoop: The definitive guide, Yahoo Press, 2010.

[29] T. White, The small files problem, http://www.cloudera.com/blog/2009/02/the-small-files-problem/ (2009).

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

[31] A. Akdogan, U. Demiryurek, F. Banaei-Kashani, C. Shahabi, Voronoi-based geospatial query processing with mapreduce, in: Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on, IEEE, pp. 9–16.

[32] Y. Wang, S. Wang, Research and implementation on spatial data storage and operation based on hadoop platform, in: Geoscience and Remote Sensing (IITA-GRS), 2010 Second IITA International Conference on, volume 2, pp. 275 –278.

[33] S. Pallickara, J. Ekanayake, G. Fox, An overview of the granules runtime for cloud computing, in: eScience, 2008. eScience'08. IEEE Fourth International Conference on, IEEE, pp. 412–413.

[34] P. G. D. Group, Postgresql, http://www.postgresql.org/ (2011).

[35] P. Ramsey, Postgis manual, http://postgis.refractions.net/ (2011).

[36] A. Rowstron, P. Druschel, Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility, in: ACM SIGOPS Operating Systems Review, volume 35, ACM, pp. 188–201.

[37] Y. Wang, W. Jiang, G. Agrawal, Scimate: A novel mapreduce-like framework for multiple scientific data formats (2012).

[38] D. Thain, C. Moretti, Efficient access to many samall files in a filesystem for grid computing, in: Proceedings of the 8th IEEE/ACM International Conference on Grid Computing, IEEE Computer Society, pp. 243–250.

[39] D. Thain, C. Moretti, J. Hemmes, Chirp: a practical global filesystem for cluster and grid computing, Journal of Grid Computing 7 (2009) 51–72.