Galileo: A Framework for Distributed Storage of High-Throughput Data Streams

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. The shared-nothing architecture in Galileo supports incremental assimilation of nodes, while accounting for heterogeneity in their capabilities, to cope with data volumes. 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

I. 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 (usually a few KB), the frequency of the reported measurements combined with increases in the number and type of devices lead to increasing data volumes.

A. Usage Scenarios

Our targeted usage scenario is in the atmospheric domain where such 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.

B. 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.
4. *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 while storing data that in turn is the basis for efficient retrievals.
5. *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.
6. *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.
7. *Failure recovery*: We must account for any possible failures and data corruptions at individual nodes. Recovery from failures must be fast and consistent.

C. Contributions

This paper describes the design of a demonstrably high-throughput geospatial data storage system. 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. 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 over time.

D. 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, the query system will be explained in section III, followed by a brief survey of related technologies in section IV. Section V presents benchmarks of our system's capabilities, and section VI reports conclusions from our initial research and discusses the future direction of the project.

II. System Architecture

Galileo runs as a *computation* on the Granules Runtime for Cloud Computing [1]. 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 pre-processed 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 [2].

Much like Google File System (GFS) [3] or Amazon's Dynamo storage system, [4] 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 single points 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 that are expected in distributed setting.

A. Granules

Granules [1] 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 schedule 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. A computation can change its scheduling strategy during execution, and the runtime will enforce the new scheduling strategy during the next round of execution. A computation is also allowed to specify a ceiling on the maximum number of times that it can be scheduled for execution. The runtime allows a computation to specify a hybrid scheduling strategy that is a combination of data availability, periodicity, and the maximum number of execution.

Granules allows these computations to be developed in C, C++, C#, Java, Python and R. Some of the domains that Granules has been deployed in include bioinformatics, brain-computer interfaces [5], multidimensional clustering algorithms, handwriting recognition [6], and epidemiological modeling. Granules is an open-source effort.

B. Blocks

A Galileo block is an multi-dimensional array of data, similar to how data is represented in systems like SciDB [7, 8] or formats such as NetCDF [9] or FITS, [10] 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.

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 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 on-disk graph. The next level of subdirectories is determined by using the data's spatial information to compute a *Geohash* [11]. 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 subsection E. 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 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 storage format flexible; if we decide to incorporate support for NetCDF, [9] HDF5, [12] or FITS [10] as our file block storage format at a later time, we can do so easily without needing to overhaul major parts of the system. Migration of data in the face of failures or for load-balancing purposes is also simplified, as single units of storage can be copied directly to other nodes.

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 disk-caching mechanism. Performance characteristics of file systems often involve tradeoffs, so this scheme allows the underlying file system to be changed to better match specific workloads, if necessary.

C. Metadata

When designing a database specifically for scientific data, the creators of SciDB [7] 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 involves 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.

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.

Galileo metadata files also contain checksum information for both the metadata itself and the blocks they are associated with. This information is used to detect data corruption that is generally expected when running in a distributed environment on commodity hardware. When corruption occurs, replicas are provided from other machines to replace the faulty files.

D. Journaling

Making complex structural changes to the underlying data structures in Galileo can often require many separate disk operations. Power failures or system crashes that can 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.

When adding a file to the system, partial metadata is read to determine a location for the block in the in-memory graph.

The graph destination for the block may require a number of vertices and edges to be created, so this information is written to the journal before storing any files. When recovering from a failure, this information allows the system to recreate its in-memory graph quickly and begin servicing queries without needing to re-read all the metadata from disk.

E. Geohashes

The Geohash algorithm [11] 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.

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 half-way 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.

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.

F. Network Organization

Galileo employs many features of *distributed hash tables*, (DHTs) much like systems such as Chord [13] or Dynamo [4]. Like Dynamo, Galileo is a *zero-hop* DHT, where each node knows enough about the network topology to route requests directly to their 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. Groups operate in isolation from the other nodes in the system, apart from storage nodes being able to route incoming data directly to its destination stream, which may be handled by a node in a different group.

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. 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.

G. 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 [14]. 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.

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.

III. 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 available in the system and returning the raw data, Galileo streams metadata of the matching blocks back to the requestor 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. The dataset also contains information about the size of the data blocks it describes. 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. Knowing the location of the data also allows applications to push computations directly to relevant storage nodes instead of requesting the blocks be transferred, avoiding network IO costs.

Since some storage nodes may respond faster than others depending on server load or the size of the data requested, datasets are streamed incrementally to client applications. This way processing can start before the entire dataset has been returned.

IV. Related Work

Hadoop [15] and its accompanying file system, HDFS [14] 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 geospatial metadata makes possible. In addition, the Granules runtime allows computations to build state over time, which contrasts with Hadoop's *exactly-once* semantics.

The Hadoop and HDFS combination has been used specifically for geospatial data [16, 17]. Akdogan, Demiryurek, Banaei-Kashani, and Shahabi found that the MapReduce paradigm is effective for a number of geospatial operations and scales linearly as nodes are added to the system [16]. 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 [7, 8] 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 and frameworks such as Granules [1, 18] are used for processing information. 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) [19]. Galileo distributes metadata and index information across all the nodes in the system.

PostGIS [20] 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 [19]. 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 [8]. 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 [21] 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 [3], 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 [22] 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 [21] in its map-based data model and Dynamo [4] 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.

V. 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 Runtime Environment, version 1.6.0_20.

One billion 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.

A. 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 I. 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 I. 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 II. 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 II. 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

B. 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 III. outlines recovery times for a single node after a system failure for three scenarios involving different number of stored blocks.

TABLE III. 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

C. 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 IV. 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 IV. 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
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.

Timing data for each query type is outlined in TABLE V. 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.

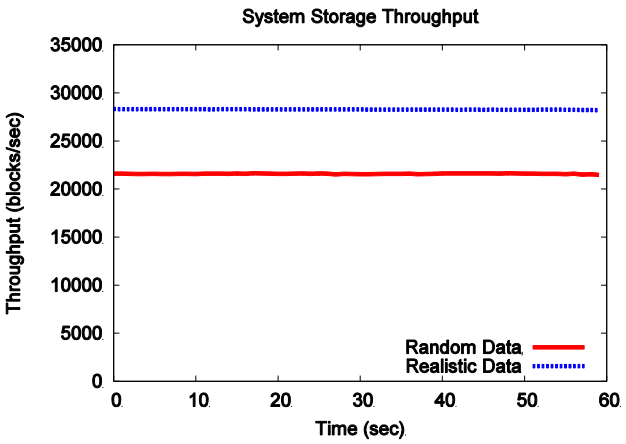

Figure 1. Cumulative storage throughput over 60 seconds

D. *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 1.

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 V. QUERY RESULTS FOR 1 BILLION BLOCKS: EACH DATA POINT IS THE RESULT OF REPEATING THE EXPERIMENT 100 TIMES

Query	First Result (ms)	First Result Std. Dev. (ms)	Last Result (ms)	Last Result Std. Dev. (ms)	Dataset Size	Dataset Creation (ms)	Creation Std. Dev (ms)	Download Time (ms)	Download Std. Dev.
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

VI. Conclusions and Future Work

A. 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, and (4) distributed, concurrent evaluation of queries. Continuous streaming of partial results to a query enables us to achieve faster response times. 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. Finally, our benchmarks demonstrate the feasibility of designing a scalable storage system from commodity nodes.

B. 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.

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 will reduce the number of operations when opening and closing files. Another option may be to combine metadata files to improve recovery times and dataset generation.

Finally, given our separate-file storage scheme, we also may explore adding support for NetCDF, HDF5, or another well-known and supported scientific format. This would make the Galileo platform more appealing for users that are already invested in a particular data format but wish to store and retrieve their data in a distributed setting. This would also make *in situ* access possible, where the system can apply computations to datasets that have not been previously entered into the system. SciDB [7] supports this feature, as the overhead for importing data into a running system can be quite high.

References

[1] 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*, 2009, pp. 1-10.

[2] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Communications of the ACM,* vol. 51, pp. 107-113, 2008.

[3] S. Ghemawat, H. Gobioff, and S. T. Leung, "The Google file system," 2003, pp. 29-43.

[4] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," *ACM SIGOPS Operating Systems Review,* vol. 41, pp. 205-220, 2007.

[5] K. Ericson, S. Pallickara, and C. W. Anderson, "Analyzing Electroencephalograms Using Cloud Computing Techniques," in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, 2010, pp. 185-192.

[6] K. Ericson, S. Pallickara, and C. W. Anderson, "Handwriting Recognition Using a Cloud Runtime."

[7] P. G. Brown, "Overview of sciDB: large scale array storage, processing and analysis," presented at the Proceedings of the 2010 international conference on Management of data, Indianapolis, Indiana, USA, 2010.

[8] P. Cudré-Mauroux, H. Kimura, K. T. Lim, J. Rogers, R. Simakov, E. Soroush, P. Velikhov, D. Wang, M. Balazinska, and J. Becla, "A demonstration of SciDB: a science-oriented DBMS," *Proceedings of the VLDB Endowment,* vol. 2, pp. 1534-1537, 2009.

[9] R. Rew and G. Davis, "NetCDF: an interface for scientific data access," *Computer Graphics and Applications, IEEE,* vol. 10, pp. 76-82, 1990.

[10] D. Wells, E. Greisen, and R. Harten, "FITS-a flexible image transport system," *Astronomy and Astrophysics Supplement Series,* vol. 44, p. 363, 1981.

[11] W. Contributors, "Geohash," *Wikipedia.org,* 2011.

[12] 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.

[13] 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,* vol. 31, pp. 149-160, 2001.

[14] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," 2010, pp. 1-10.

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

[16] A. Akdogan, U. Demiryurek, F. Banaei-Kashani, and C. Shahabi, "Voronoi-based Geospatial Query Processing with MapReduce," 2010, pp. 9-16.

[17] Y. Wang and S. Wang, "Research and implementation on spatial data storage and operation based on Hadoop platform," 2010, pp. 275-278.

[18] 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*, 2008, pp. 412-413.

[19] P. G. D. Group, "PostgreSQL, http://www.postgresql.org/," 2011.

[20] P. Ramsey, "PostGIS manual," *Refractions Research Inc,* 2005.

[21] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS),* vol. 26, pp. 1-26, 2008.

[22] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review,* vol. 44, pp. 35-40, 2010.