A Framework for Managing Continuous Query Evaluations over Voluminous, Multidimensional Datasets

Cameron Tolooee, Matthew Malensek, and Sangmi Lee Pallickara
Department of Computer Science
Colorado State University
Fort Collins, Colorado, USA
(ctolooee, malensek, sangmi)@cs.colostate.edu

***Abstract*—Efficient access to voluminous multidimensional datasets is essential for several scientific applications, including real-time analysis and visualization. Fast evolving datasets present unique challenges during retrievals. Keeping data up-to-date can be expensive and may involve the following: repeated data queries, excessive data movements, and redundant data preprocessing. This paper focuses on the issue of efficient manipulation of query results in cases where the dataset is continuously evolving.**

Our approach provides an automated and scalable tracking and caching mechanism to evaluate continuous queries over data stored in a distributed storage system. Among the storage nodes, one or more nodes are selected using an election algorithm based on CPU and memory utilization. These selected nodes ensure that the query output contains the most recent data arrivals and cache the metadata of the query output. This approach is evaluated in the context of Galileo, our distributed data storage framework. Galileo is designed for managing multidimensional time-series datasets generated in geospatial observational settings; e.g. data generated by remote sensing equipment and sensor networks.

We describe our approach of using the metadata graph to push data preprocessing jobs onto the storage system during the continuous query processing and selectively download subsets of the query output. Our performance benchmarks demonstrate the efficacy of our approach.

Keywords – Continuous Query, Galileo, Query caching, Time series data

I. INTRODUCTION

In recent years, we have witnessed significant increase in the data collected from sensor networks, observational equipment and social network applications. These streaming data enable real-time monitoring and control, where decisions are based on analyzing voluminous, dynamic data. The data from these data sources are often staged into a data storage system continuously at a very high rate. To retrieve the most recently added data items within a fast evolving dataset, users might need to issue similar queries frequently. It is possible that no new results might be available between such successive queries. Repeatedly issuing the same query is not scalable due to the increase in query traffic, a problem that is exacerbated if there are a large number of clients. With traditional data retrieval methods, query evaluations provide a snapshot of a subset at a given time.

A *continuous query* is one where the query is long-lived i.e. it lives beyond the instant when it was first issued and the results made available. In continuous queries once the initial set of results is made available, newly arriving data items that "match" the query are sent back to the client in an incremental fashion. Put another way, a continuous query is a query that is issued once and then logically run continuously over the datasets. This concept has been investigated within the database community [1] and recently in cluster-based storage systems [2, 3]. It has also been explored in real-time data mining systems [4]. Continuous queries can be useful for monitoring events such as traffic, network performance, and financial trend analysis.

Designing a scalable scheme for the evaluation of continuous queries over high-dimensional datasets has been a challenge [5]. As data is integrated from many sources, the amount of the accumulated data grows. There is also the complexity of applying advanced data filters to the output of continuous queries. Data filters such as sampling algorithms or data interpolation algorithms are important not only to improve data quality but also to reduce data transfers significantly.

In this paper, we focus on a framework for evaluating continuous queries at scale. We address this problem from the aspect of a distributed storage framework. We cache the query and its results in distributed storage nodes and autonomously maintain the cached result to include the most up-to-date data. Compared to real-time data stream mining over incoming data [6], evaluating continuous queries within the storage subsystem has the following advantages: (1) queries can be evaluated over features and values across the entire datasets, and (2) query evaluation is not limited by the window size. Therefore, in addition to the nearly real-time monitoring purpose, scientist can use this feature for more

complex analysis over the integrated dataset. Our approach also allows the query evaluation to apply advanced filtering algorithms. During the query evaluation process, pre-defined and user-defined data filters can be applied to the query results. Finally, the query result can be a selective subset of the voluminous dataset containing near-real time updates and it can also be reduced by means of applying filters such as sampling algorithms with a suitable resolution based on the client's resource.

We evaluate our ideas in the context of our Galileo system [7-10]. Galileo is a scalable storage framework for managing multidimensional geospatial time-series data for scientific applications. Data stored and retrieved from Galileo include blocks that are multi-dimensional arrays with temporal information alongside geospatial locations. Users access datasets by specifying multi-dimensional queries that specify bounds or wildcards for one or more dimensions corresponding to the observations/features stored within the system. When new data packets representing observations arrive, it can be added to an existing physical data file or a new data file can be created for the dataset.

A. Scientific Challenges

We consider the problem of efficient and scalable evaluation of long-term continuous queries over voluminous, multi-dimensional datasets. The challenges in this research include the following:

- Data arrives at high rates from a large number of sources. This results in a voluminous dataset that should be dispersed and managed over multiple resources.
- The approach should scale with increases in the number of clients, data volumes, and resources.
- There may be no bound associated with the chronological dimensions specified in the range queries.
- Repetitive queries from the client over the entire dataset can be very expensive. The data storage framework should be able to logically track which portions of the query results need to be updated.
- Distributed data structures should be expressive to define subsets of data to allow users to schedule downloads and apply user-specified data filters.
- Redundant download of results from continuous queries will increase computer network traffic; efficient query management to track the user's query is critical.

B. Research Questions

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

- How can we evaluate and track queries over voluminous datasets, especially in the case of continuous data arrivals and additions to the dataset over a collection of machines?
- How can we strike a balance between managing continuous query evaluations in a scalable fashion while supporting near real-time state updates to the query results?
- How can we avoid hotspots associated with query workload tracking?
- How can we provide an efficient scheme to allow users to browse and interact with query results?
- How can we support data filtering algorithms over continuous, fast-evolving query results?
- And, how can we control the lifetimes associated with query tracking?

C. Overview of Approach

The approach described here is based on our distributed storage framework, Galileo [7-10]. Galileo is a hierarchical distributed hash table (DHT) implementation that provides high-throughput management of voluminous multidimensional data streams. Datasets arriving in Galileo are partitioned and dispersed over a cluster of commodity machines. Data is dispersed by the system based on the Geohash geocoding scheme to ensure geospatial proximity of proximate data points. Query evaluation in Galileo is performed over the metadata residing in the memory at each storage node and query results are represented as a set of metadata. Galileo organizes relations between components of the metadata using a graph data structure.

To support continuous queries, we provide a *distributed updatable cache* over the storage nodes. We define a distributed updatable cache as a query caching feature that autonomously tracks the most recent addition to the query output in a distributed fashion over the storage nodes within the system. A storage node is assigned to cache the continuous query output using an election algorithm, and we call this node *cached continuous query coordinators*. Cached continuous query coordinators are selected from among the storage nodes based on their memory and CPU utilization. The cached continuous query coordinators perform the queries periodically over the metadata of datasets and detect any update of data blocks. As part of the initial response to a continuous query, the client receives the result (metadata) based on initial evaluation over the stored dataset, an identifier for the query, and the address of the cached continuous query coordinators.

The metadata of the current output of the continuous query is retrieved, as the client desires. Clients are allowed to select a portion of query result to download or apply data filters such as sampling algorithms. Sampling of query output is a critical feature for the applications that support multiple resolutions for the same dataset. Users can apply built-in sampling algorithms and also push their own data preprocessing onto nodes that hold the data. The cached continuous query coordinators propagate these customized data preprocessing to the storage nodes within the system that hold data blocks satisfying a particular query.

D. Paper Contributions

This paper presents the design of a storage subsystem supporting continuous query evaluations over a time-series dataset. We use a distributed updatable cache to track the

updates of query results. Caching is performed by a subset of storage nodes selected based on the current workload. Users retrieve only updates of the query results. We use metadata from the matching data blocks to describe the query outputs. The components of the metadata (e.g. features, timestamps, geo-location) are organized as a graph. The metadata graph provides complete access to the data blocks used in Galileo. The metadata graph allows users to utilize the information stored in the nodes of the metadata graph to group the data blocks. The storage subsystem provides protocols to launch the user-defined preprocessing on a selected group of data blocks; this process is automatically performed during caching. This paper also presents the evaluation measurements of the overhead for maintaining distributed updatable cache and user-defined data process over the query output.

E. Paper Organization

The remainder of this paper is organized as follows. In section 2 we provide an overview of related work. In section 3, we include a description of the architecture and query processing capabilities of our system, Galileo. In section 4, we describe our framework for tracking and maintaining continuous query to reflect the most recent update of dataset. Section 5 describes the interactions between query outputs and sampling framework for reduction of output sizes. A performance evaluation of various aspects of the system is presented in Section 6. Finally, conclusions and future work are outlined in Section 7.

II. Related Work

Considerable work on continuous queries has been conducted previously. Most on-going work can be boiled down into three distinct flavors of continuous queries: storage based, data stream based, and real-time database approaches. While Galileo fits in closer to the storage based methodology, it shares characteristics with work from all three areas.

Amazon Kinesis [3] is Amazon's new project that is rapidly growing in popularity. Kinesis provides a fully managed, high-throughput data stream processing framework. It enables sophisticated data stream processing in real-time that plugs into their existing data stores. By partitioning data among their resources using an MD5 hash, the Kinesis system allows for even load distribution among the shards allocated to the stream processing application. Unlike Galileo's distributed updatable cache, Kinesis keeps a 24-hour sliding window over the streamed data. After this, data records are no longer accessible and are potentially lost if not placed in persistent storage. Because of this, Kinesis is unable to efficiently see a holistic view of the data while processing streaming records.

NiagaraCQ [2] supports continuous querying in the vast context of the internet. It leverages similarity in continuous web queries by grouping and sharing common computations. It employs a similar continuous query model as Galileo's distributed updatable cache. A web crawler is deployed to generate a highly optimized xml database for querying. When a continuous query is established, a handle is placed on the relevant XML documents, so that only updated portions of the document are considered for their incremental evaluation. NiagaraCQ is focused on the domain of the internet and does not support sensor network streams.

Borealis [4] is a stream processing engine focused on timely analysis over high rate, sensor network data streams. Its approach, like Galileo's, emphasizes dynamic interactions with queries and results by adding interactions with the sliding window over the data stream. In many scenarios that arise with data streaming, modification of the results or the query itself are desired. Borealis utilizes a network of stream operators and scheduling framework to delay processing of data records during periods of high loads and allow for in-place modifications of queries already being evaluated. Due to the lack of storage, as with most sliding window data streaming engines, Borealis is limited by the size of the window over the stream. While they partially cope with this challenge with their connection point snapshot views, ad-hoc queries on historical data are not feasible.

BeeHive [11, 12] is an application focused real-time global database system. Queries in a real-time database system face the challenges of meeting strict deadlines in their retrievals. BeeHive accomplishes by precisely managing resources, scheduling evaluations, and providing a series of different evaluation resources to asses a variety of query types efficiently. It provides APIs to request for specific time requirements on queries. The strict assurances it delivers restrict the ability to process data as it arrives at high rates.

III. Background: Galileo System Overview

Galileo is a distributed data storage system for voluminous multi-dimensional, geospatial, time-series datasets. Galileo is designed to assimilate observational data, which arrive as streams, from measurement devices such as sensors, radars, and satellites. Data is dispersed and stored over a distributed collection of machines. As soon as a dataset (or a portion thereof) arrives, the dataset is transformed to one or more storage unit(s): *block(s)*. Data blocks are dispersed using an indexing scheme applied on the major dimensions such as geospatial coordinates and temporal information accompanying the measurements.

A. Network Topology

Galileo's topology is organized as a zero-hop distributed hash table. DHT's provide a decentralized, highly scalable overlay network that allow for insertions and retrievals similar to that of a hash table; e.g. *put(key, value)*, and *get(key)*. The class of zero-hop DHT's, such as Apache Cassandra [13] and Amazon Dynamo [14], provide enough state at each node to allow for direct routing of requests to their destination without the need for intermediate hops.

Galileo deviates from the standard DHT in that it employs a hierarchical node partitioning scheme. This scheme leverages characteristics of the data elements to map related data on or near the same node. Each storage node within the system is placed in a group. The size and quantity of groups are a user-configurable parameter that can be adjusted to best

fit the data stored. The two-tiered partitioning structure, where nodes are first placed in groups among similar nodes, then hashed within that group, increases the efficiency of retrieval operations by providing data locality for query evaluations. The data locality also expands data interactions beyond the *put(key, value)*/get*(key)* into more expressive queries such as range-based, wild card, polygon, and approximate queries.

B. Metadata

Instead of using notions of files and directories, the units of encapsulation specified in user queries are the features that describe the dataset. To narrow the search space and to effectively evaluate queries, each node maintains two in-memory metadata structures: a low-resolution *feature graph* that encompasses the entire dataset and a high-resolution *metadata graph* representative of the data stored within that particular node. A typical query evaluation process is as follows: a query is issued to any single node in the system which, in turn, uses the low-resolution feature graph to construct a set of *candidate storage nodes* possibly holding data relevant to the specified query. The candidate nodes then exhaustively evaluate the query at higher-resolutions to retrieve any data blocks that match the specified query. Matching blocks are streamed asynchronously to the issuing client.

The high-resolution metadata graph follows a hierarchal, tree-like structure where each level of the tree corresponds to an indexed feature of the data and the leaves of the tree contain the data access information needed for retrieval. Traversing the graph from root to leaves will discover data that has the properties aggregated along the path, whereas traversing the graph from leaf to root will provide the entire indexed feature information for the data block represented by the leaf. This structure provides many benefits in terms of efficiency of operations and interactivity with the results. By grouping like paths or sub paths, duplicate metadata is avoided and query evaluations following one path returns many results. Once a query is evaluated, the metadata graph can be traversed, reoriented, and/or partitioned to precisely extract the quantity of results and their various attributes without ever reading data from the disk.

Figure 1 depicts a simple metadata graph consisting of three features: spatial location, humidity, and temperature. In this example, all data points share the same spatial characteristic of residing with the 9Q Geohash, discussed in the next subsection. The second level here corresponds to the humidity attribute of the data. As we traverse down the left – most path to data block 1, we observe that data points within that block: reside in 9Q Geohash, have 70.5% humidity, and is 28.9 °C. From this we can tell that this data point is describes a location that is moderately humid and hot.

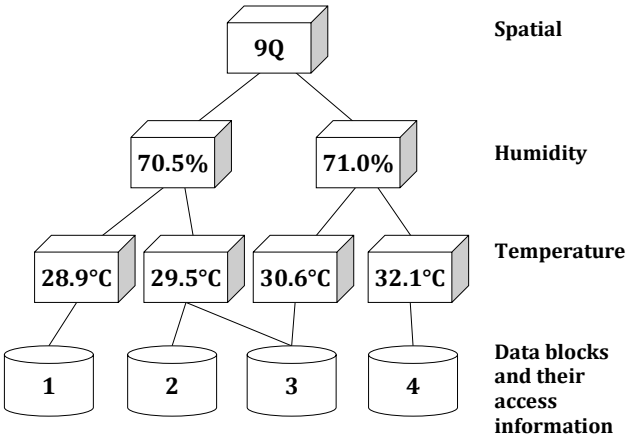

Fig. 1. Example of a simple metadata graph. This is an example metadata of a dataset using three features to index data blocks: Spatial geohash code, humidity, and temperature. The data block 4 contains dataset that has 71% humidity with a temperature of 32.1 °C for the geospatial area 9Q that is mostly southern California and a large portion of Nevada, USA.

Because the bottom level contains data block locations that are fixed within the storage node, it is possible to have multiple edges to the same leaf. This is observed in data block 3 that is connected to two different temperatures.

C. Geohashing and Storage of Data Streams

Galileo supports streaming data that incrementally enters the system from a variety of sources. These data items are constantly evolving over time and can share a number of common attributes. Therefore, simply applying a standard hash function on the incoming data results in an approximately even distribution of files across all the nodes in the system, but does not account for similarity in the data being stored. We employ the Geohash algorithm [15] as the first tier hashing function to partition geospatially similar data points to the same group. The Geohash algorithm divides geographic regions into a hierarchical structure. A Geohash is a string derived from a latitude and longitude coordinate. Each Geohash string represents a bounding spatial box. The length of the string corresponds to the precision of the box; a longer string denotes a more precise, or smaller, bounding box. For example, the latitude-longitude coordinate N40.57, W105.08 is bounded by the Geohash *9XJQBCE*. Appending characters to the string would make it refer to more precise geographical subsets of the original string. Figure 2 [9] illustrates how regions are divided into successively more precise bounding boxes. The Geohash regions provide a natural mechanism to partition geospatially similar data to the same group.

Inspecting the spatial dimensions present in incoming data streams facilitates our controlled dispersion strategy by creating logical groupings of data in the hash space, but also increases the likelihood of storage imbalances across nodes in the system [7]. Using a hierarchical approach allows a balance to be struck; placing logically similar data items in the same groups and then using a second hash function based

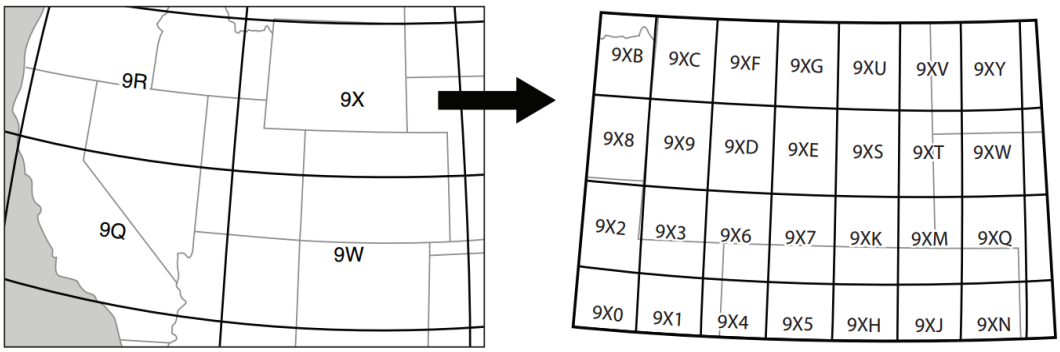

Fig. 2. An illustration of the recursive subdivision of a Geohash bounding box. By adding one base-32 character to the Geohash value, a spatial area is subdivided into 32 sub-blocks. Each of the characters represents interleaving 5 bits that separate the latitude and longitude intervals.

on a feature of the data to place it on specific nodes within the groups. While hierarchical node partitioning imposes a less balanced load distribution, the range of the data dispersion between the two-tiered partitioning and a flat SHA-1 hash is only 1% of the dataset [7].

IV. Distributed Updatable Cache

In an environment where data evolves at a fast rate, challenges arise when up-to-date query results are frequently needed. In many cases, clients must repeatedly reissue queries, filter out new data from the old, and perform redundant processing on the results. Caching is a known solution to many similar problems; however, massive data volumes introduce many complications to the simple caching model.

In general, query caching is performed when the user's data do not change very often and it is particularly useful when the server receives many identical queries that reflect recent changes. Our design is also differentiated from the existing updatable cache in terms of the underlying topologies of the storage nodes. The distributed updatable cache should perform query caching to cope with the nature of the dispersed data over a large number of storage nodes. In-memory management of these caches is also essential to address the gap in the random access performance between main memory and disks. This section will discuss the architecture and algorithm used in the distributed updatable cache to handle these challenges.

A. Cached Continuous Queries

When considering continuous queries over data streams there are many existing techniques for evaluation such as sliding windows, sampling, synopsis data structures, or batch processing [6]. Each technique has usage scenarios for which it performs well and for which it performs poorly. For example, numerous live monitoring applications require prompt, time-independent analysis of streaming data. Because the entire history of the stream is not needed for the analysis, sliding windows are attractive solutions for approximate answers to continuous queries. Conversely, in settings where the analysis performed is dependent on historical data, sliding windows perform poorly. Galileo Cached Continuous Queries (CCQ) strive for a holistic evaluation and thus employ a combination of two techniques: batch processing, where data aggregates over a short time interval and is processed in groups, and data synopsis, where a sketch, or synopsis, of the data is maintained and queried.

A CCQ encompasses the standard Galileo query with the addition of two vital parameters: an *expiration time* and an *update interval*. These two attributes express duration for which the continuous evaluation should be performed and the interval at which each processing cycle will be executed. Once a CCQ is dispatched, the initial query is evaluated and among the query's candidate nodes, an *election phase*, discussed subsection E, is performed to select a node to administer the CCQ. In addition to the initial evaluation, the client receives the information for the node elected to manage the CCQ.

B. Cached Continuous Query Coordinator

The storage node selected to maintain the CCQ spawns a *cached query coordinator* that directs all operations with cached queries on the node. As illustrated in Figure 3, a cached query coordinator consists of four main components, CCQ tracker, cache table, query processors, and election processors. Upon receiving a new CCQ, the CCQ tracker will schedule and initiate the CCQ's processing cycles according to the specified update interval. The value of the update interval determines the window of accuracy in the results. The query results will be at most the update interval time value out-of-date.

At the end of each cycle, the cached query coordinator is responsible for aggregation, compression, and caching of the results into the cache table. If a single cache grows too large, the tracker triggers an election phase to choose a nearby node that has available resources to take over the cache. An election processor is spawned to execute the election phase and update the tracker and cache tables with the results. Each of these tasks is discussed in detail in the subsequent sections.

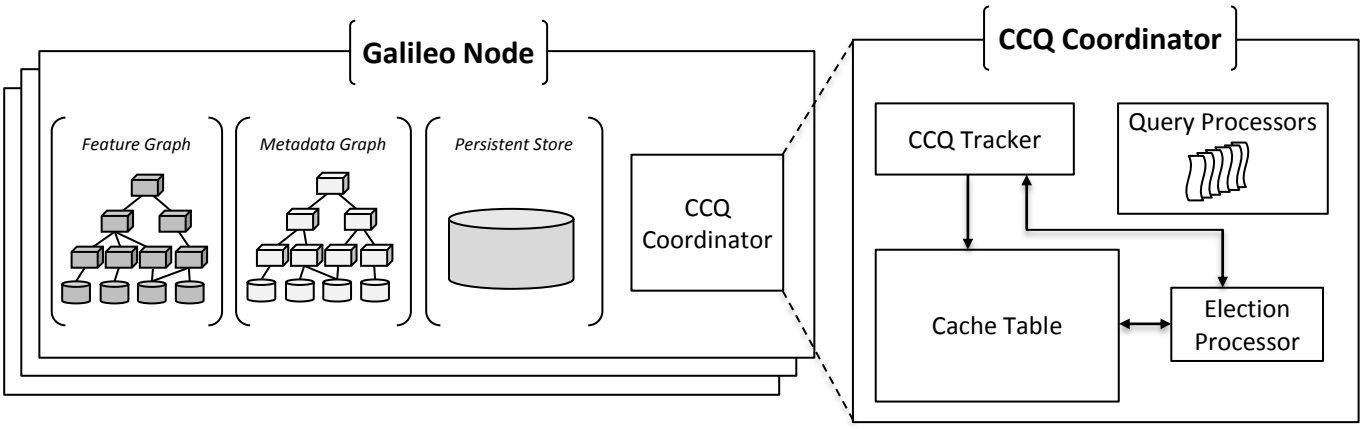

Fig. 3. Galileo's continuous query architecture. Each of the Galileo nodes manages a Cached Continuous Query Coordinator. This Coordinator comprises the following: Tracker, Cache table, query processors and Election Processor.

C. Processing Cycles

The fundamental technique in our continuous query evaluation lies in the batch processing cycle. A batch processing sequence, initiated by the CCQ coordinator, occurs for each CCQ at their respective intervals. The query is reissued to the system and evaluated in a streamlined fashion.

As a result of the redundant nature of continuous queries, nodes involved with a CCQ can perform optimizations during its reevaluations. Cached queries repeatedly explore the same area of the metadata graph and therefore the nodes evaluating them can omit large portions of unrelated graph extents to improve performance. Each query candidate node, obtained from the feature graph, returns its result to the CCQ coordinator for compaction, also derived from the metadata graph.

D. Leveraging the Metadata Graph

One of the goals of the distributed updatable cache is to preserve the data interactivity that Galileo provides via the metadata graph. While simply caching file block identifiers and access information is compact and efficient, the end result is an array of filenames which require direct data accesses to extract any information. Instead, we leverage the current metadata graph in two essential ways to maintain the interaction.

First, the graph is utilized as a synopsis data structure for all the data stored on a node. Since the CCQ coordinator dispatches the cached query only to the candidate nodes, we are able to narrow down the search space immediately. The dispatching to the multiple candidate nodes is performed in parallel. By lowering the dimensionality of the data to only the indexed features, requests can be calculated in near-real time.

The metadata graph also provides a canonical naming scheme for paths that can be utilized as a method of data compaction. Since every path from root to leaf is unique, a discrete path name can be created for each file block. While traversing from the root to a leaf, vertices at each level concatenate its payload to the initially blank name on the way down. The end result is a distinct, static label that represents all the metadata associated with the file block. This label can be reconstructed into a metadata path and, in turn, a metadata graph as depicted in Figure 4. This technique preserves the tree like aspects of the query results that allow client interaction. These compact path labels are cached for each query result.

E. Metadata Path Label Caches

For each re-query result the CCQ coordinator receives, a set of path labels is constructed and cached locally. Local caches introduce new complications such as only caching new data, minimizing cache sizes, and defining maximum cache size thresholds.

1) Caching Recent Data

To ensure only data that has arrived since the previous processing cycle is stored, the CCQ coordinator uses a simple timestamping technique. Every data block that enters the system has an associated timestamp that is updated anytime a block is modified. The CCQ coordinator maintains a time stamp for each tracked CCQ that indicates the time since the last client retrieval. A filter is applied during the query evaluation that drops any data blocks whose timestamp is before the coordinator's timestamp.

2) Local Cache Compression

While each label is compact, cached queries with large outcomes can potentially thrash the main-memory of the caching node. To cope with this, CCQ's provide an optional parameter that introduces compression into the caching process. When enabled, each label cache is streamed through a LZW compression [16] algorithm to reduce the volume of the cache. Due the redundant nature of the cache label content, the high-performance LZW cache achieves excellent compression as shown by Table I. Compression comes with a temporal cost and is not optimal in environments requiring concise response times. Thus, we've made this an optional feature to best fit end user's needs.

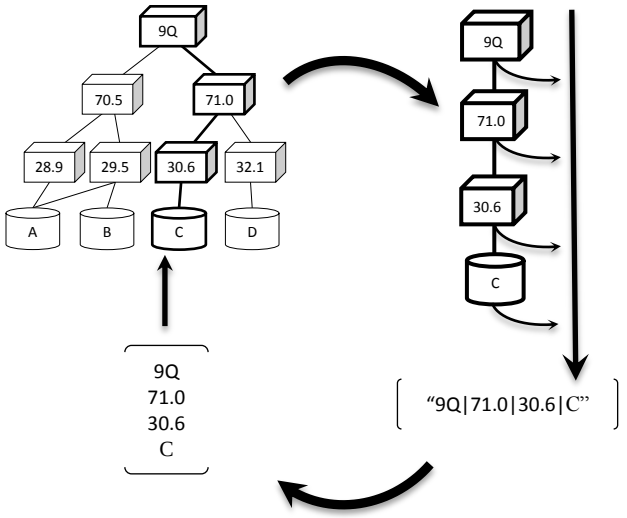

Fig. 4. The compaction and expansion of a metadata graph path. The data compaction process provides a flattened representation of the metadata to maintain and track the dataset for continuous query effectively. Since it preserves the semantics of the metadata, it is easy to merge the sub-graph with the original query results when necessary.

TABLE I.
COMPARISON OF QUERY RESULT SIZES

Number of Results	Original size (kilobytes)	Compacted Size (kilobytes)	Compressed Size (kilobytes)
1,000	102.383	27.483	7.624
100,000	10,187.88	2,743.549	720.745

3) Dynamic Cache Size Thresholds

Even in our best efforts in reducing the volume of query output, large outcomes will eventually exceed a system's memory boundaries. It is not feasible to expect a single storage node's main memory to be sufficiently large. To determine when a cache should be distributed, each cache has a dynamic threshold based on a variety of usage statistics such as available memory, CPU usage, node popularity, and presence of other CCQs. If this threshold is exceeded, the coordinator will initiate an election phase to select a new node to take over the future cache entries.

As the state of the nodes progress this threshold dynamically adjusts to maintain a balanced workload throughout the cluster. For example, the introduction of a second CCQ will decrease the threshold of the existing one; forcing an election phase to redistribute the overflowed cache and maintain equilibrium.

This approach has a ceiling in terms of memory, as ultimately we are bounded by the total memory within the cluster. To reach it, however, takes a bit of work. If a cached query, for example, yields one billion results, the distributed updatable cache system needs 7.3 gigabytes of system wide memory to cope with the continuous query without ever going to disk. Future work, discussed in the final section, includes ways to push this boundary even higher.

F. Election Algorithm

The final piece to our solution is the election phase. Like the processing cycle, the election phase also leverages Galileo's architecture to optimize performance. Once initiated, the node will broadcast an election poll to all the nodes within its hierarchical grouping. If all of the responses fail to meet the needs of the CCQ, the poll is extended further to another group in the Galileo cluster. Broadcasting a poll to the entire cluster simultaneously, would result in a bottleneck of communication when cluster sizes grow large. This approach of polling for a localized optimum provides a scalable election.

As a response to an election poll, a storage node will send the same usage statistics used in the cache threshold calculation (available memory, CPU usage, node popularity, CCQ count) back to the initial coordinator. Based on the incoming statistics and their respective round trip times, each response is ranked. The top rank is selected to be responsible for the future portions of the CCQ cache.

The coordinator tracks the newly elected node in its respective CCQ table entry. Each additional elected storage node will update the original coordinator about their affiliation with the query. As a CCQ cache propagates through the storage nodes exactly one coordinator, the most

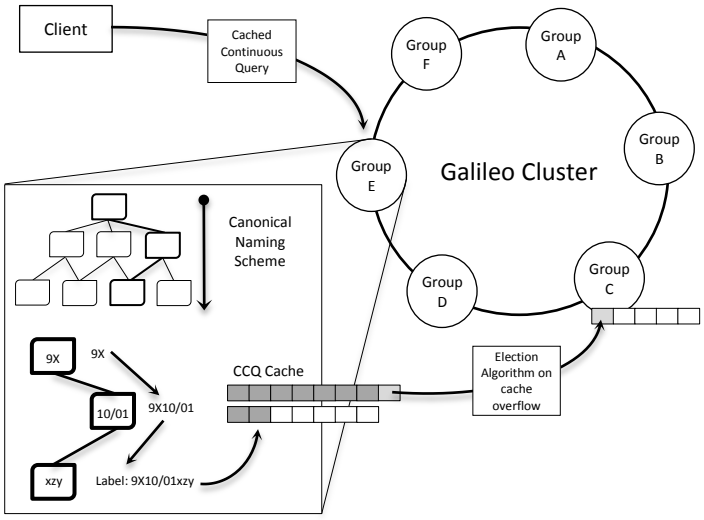

Fig. 5. A high-level overview of the distributed updatable cache and cached continuous query progession. First targeted nodes for the election phase are the nodes in the same group. If all of the nodes in the group are overloaded, the election process will be encompass other groups based on the workloads and the network latency.

recently elected, will initiate processing cycles.

Figure 5 shows a high level view of the distributed updateable cache and the CCQ processing flow. An end user defines and submits a CCQ to any of the storage nodes in the Galileo cluster. Upon receiving the query, the storage node evaluates the query periodically and compacts the results using the metadata graph canonical naming scheme into path labels. The compacted path labels are cached locally in a dynamically sized store. If the cache overflows, an election phase is conducted to choose the locally optimal node to maintain newest data in the cache.

G. Client Interaction

Upon direct request, anomalous event, or any other trigger, the CCQ coordinator will request all the elected nodes to send their caches. Upon arrival, the coordinator aggregates each cache to reconstruct a single metadata graph. This metadata graph is identical to that of a standard query result, and thus, the entire process is transparent to the client. Retrieval for cached continuous queries is more responsive than a standard query because no evaluations are needed. The client receives immediate, up-to-date data that can be combined with data previously accumulated to conduct their studies. After client retrieval, all caches are cleared and the process is repeated until the CCQ expires. The caches are purged from the memory once the CCQ expires.

As the client receives the metadata graph response from any query, the graph is converted to the resource description framework [17] (RDF) data model. Data in RDF is represented by subject-predicate-object triples, where a subjects and objects refer to the resources and predicates describe the relationships between them. The RDF model can be conceptualized as a directed graph with named edges. Vertices in the graph represent subjects and objects, and named edges represent the predicates. For example, if an edge links vertex *P* to vertex *Q*, *P has the property Q*. Galileo's metadata graph lends itself nicely to this RDF directed graph model. By directing edges in the metadata graph downwards towards the leaves and assigning predicate of `parent-of` completely describes the data. This conversion schema allows for the utilization of the graph query language SPARQL to create customized views of the metadata graph to explore the data and create sub-queries. SPARQL [18] is a rich querying language for RDF. Conjunctions, disjunction, triple patterns, and joins are among many of the types of query supported by SPARQL. By converting the metadata graph to RDF, SPARQL queries can be performed to obtain any subset of vertices or to specify a subgraph upon which to do further preprocessing and analysis.

H. Fault Tolerance

Distributed storage systems are often comprised of large numbers of commodity machines to reduce the hardware costs. Consequently, node and disk failures occur frequently and cannot be brushed under the rug as an anomalous event. One common technique for coping with these failures involves assigning replica data to multiple other nodes within the cluster and if a failure occurs, route the request to one of the replicas. The downside of this technique is the storage requirements involved with replicating this information across one or more nodes. In an environment constrained by memory, such as ours, this solution does not scale.

Our approach to handling failures is derived from the fact that all data streamed into the system is stored and can be accessed at any time. Each time a processing cycle distributes its query for evaluation, the storage nodes receiving the CCQ keep track of three fault tolerance data items: (1) the location of the coordinator for the CCQ, (2) the CCQ itself, and (3) the timestamp used as data recency filter, previously described in subsection E1. When a failure is detected, affected coordinators can be determined by checking the tracked coordinator location data at each node. If a coordinator has failed, a new coordinator is elected via a standard election phase. The new coordinator reestablishes the CCQ with the parameters obtained from the stored fault tolerance data. The stored timestamp ensures the consistency of the new label cache with that of the failed coordinator. The new coordinator informs the client that it is the new access point for the CCQ.

V. Sampling Query Results

Many applications, such as data visualization, require the investigator to sample the data after its acquisition from the server; however, downloading the entire query output just to filter out a large portion is inefficient. To cope with this limitation, as part of this effort we incorporate a framework by which clients can provide a user defined sampling function (UDSF) to acquire a sample of the data desired without downloading it first.

Because sampling is often a domain specific function, we provide a mechanism to devise a personalized sampling function to be executed on the cluster. An interface defines the structure for the UDSF to be implemented to ensure compatibility. The client then separately implements, compiles, and compresses the UDSF into a Java ARchive (JAR) file to be sent with the query to the server. After a query is evaluated, rather than dispatching the entire dataset over the network to the client, the JAR is written locally to the server, loaded dynamically, and applied to the evaluated query.

In the cached continuous query framework, the UDSF is loaded and evaluated as normal during the processing cycle. Data blocks are filtered out first by timestamp, to ensure a fair sample, and then by the dynamically loaded UDSF. The sampled results are returned to the coordinator and the rest of the CCQ process continues as normal.

VI. Performance Evaluation

To benchmark the effectiveness of our continuous query framework, we sourced real-world data from the North American Mesoscale Forecast System (NAM) [19], which is maintained by the National Oceanic and Atmospheric Administration (NOAA). The NAM is run four times daily,

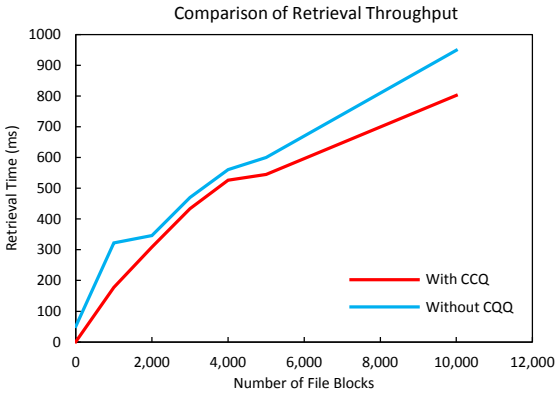

Fig. 6. Comparison of retrieval performance of queries of varying sizes between standard and continuous cached queries.

and we sampled data recorded from 2009-2012 using our NetCDF input plugin to generate a dataset containing one billion (1,000,000,000) Galileo blocks, each of which is 8 kilobytes. The data attributes we indexed and queried against included the spatial location for the sample, temporal range during which the data was recorded, percent maximum relative humidity, surface temperature (Kelvin), wind speed (meters per second), and snow depth (meters).

We composed a series of test scenarios to measure three main aspects of our design: speed, memory consumption, and continuous retrieval throughput. Each experiment was conducted 100 times in our heterogeneous 75-node cluster composed of 47 HP DL160 servers (Xeon E5620, 12 GB RAM, 15000 RPM Disk) and 28 Sun Microsystems SunFire X4100 servers (Opteron 254, 8 GB RAM, 10000 RPM Disk).

To ensure the continuous cached queries performed as expected, we ran a series of retrieval throughput benchmarks in comparison with Galileo's standard query evaluation method. Figure 6 compares the retrieval times of the two querying techniques over a number of file block results. By altering the query parameters, we adjusted the number of returned file blocks to range from 0 to 10,000. These results demonstrate that the time costs of compression, caching, and reassembly are less than that of distributing queries to nodes in parallel, then aggregating results; as the standard method does.

While promising, the previous benchmark exemplified that caching can indeed make things faster, which is to be expected. In this context, the challenge is to attain the speedups caching can provide while maintaining a small memory footprint. To examine the memory efficiency of our continuous query framework, we reduced the volume of the dataset 1,000,000 file blocks to increase the severity of the memory increases we will incur. This benchmark involved issuing 10 distinct CCQ's yielding 100,000 results each; doubling the number of represented in-memory file blocks. To measure the memory discrepancy, we populated the cluster with the one million file blocks and measured the memory usage without the presence of any cached queries. Subsequently, all 10 CCQ's were dispatched and the memory usage was measured once again, after the CCQ's were able to equilibrate. Figure 7 represents the percentage of change in memory usage before and after the CCQ's were issued on a node by node basis. If no memory changes were realized, the chart would appear as two identical rectangles meeting at the 0% line. Each "after" bar that drops below the 0% mark indicates a memory increase on that particular node caused by the distributed updatable cache. The magnitude of this increase is denoted by how far below it drops. It is clear the memory requirements did not double despite doubling the number of represented in-memory file blocks. In fact, only an 11.71% increase was realized after the introduction of the 10 cached continuous queries.

Figure 7 displays the success of the election algorithm's ability to distribute the caches. Though the light bar troughs may seem random in location and length, recall that Galileo utilizes a two-tiered hashing scheme to distribute data initially. This means that from the start, the load distribution in memory is not even amongst the nodes. For instance, in Figure 7, the largest memory increase, the third large difference from the left, occurred on node 17; that had the smallest allocation in the initial distribution. The addition of one million represented file blocks increased the standard deviation of the distribution by .8602 megabytes.

Our final benchmark targets the continuous retrieval process. We deployed a CCQ into an empty cluster, with the varying update intervals and streamed in blocks that match the query. At three second intervals, continuous query update retrieval is requested, the time and number of file blocks was recorded. A summary of the results can be found in Table II. Our batch processing technique allows for consistently fast retrieval times in the face of large query output volumes. Longer update interval times produce a large number of file blocks retrieved. This is because there is more time for incoming data blocks to accumulate before a processing cycle is initiated.

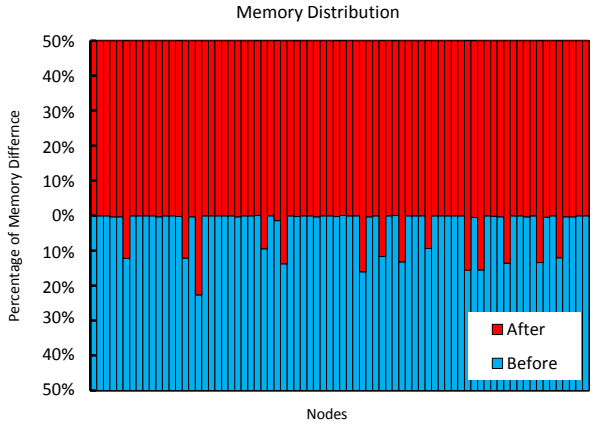

Fig. 7. Difference in memory usage after deploying 10 continuous cached queries having 100,000 results each over a dataset with 1,000,000 file blocks.

TABLE II.
BENCHMARKS FOR VARIOUS CACHED CONTINUOUS QUERY SIZES

Number of Blocks Streamed	Average Retrieval Time (ms)	Average Number of Blocks per Update Interval		
		1 s	5 s	10 s
1,000	100.66	330	384	595
100,000	109.05	423	515	615
1,000,000	177.85	411	506	522

Because only updates since the most recent retrieval are returned each time, the volume of the data transfers are minimized which greatly improves the performance versus downloading all of the redundant data.

VII. CONCLUSIONS AND FUTURE WORK

A. Conclusions

Cost-effective access to voluminous multidimensional datasets is a challenging problem when the data is evolving quickly. Keeping things up-to-date can be expensive and may involve the repeated data queries, excessive data movements, and redundant data preprocessing. Our approach to solving this problem provides a scalable caching mechanism to evaluate continuous queries over data stored in Galileo, our distributed storage system. A cached continuous query defines our batch query processing parameters. The cache continuous query coordinator at each node is responsible for initiating the query at the defined intervals. Upon receiving results from other storage nodes, the coordinator aggregates, compresses, and caches the most recent results into the cache table.

We benchmarked several aspects of our continuous query system such as the efficiency of our cache and compression mechanisms, the distribution of the election phase, and the throughput of our distributed cached continuous queries. Our benchmarks show the efficacy of our approach.

B. Future Work

Our vision of holistic continuous query evaluation has many ideas for improvement and expansion. With a memory limit in mind, the scheduled cached continuous query processing cycle displays a timely, repetitive pattern that we can leverage. By writing caches to disk, knowing they won't be needed until the next interval, we free memory for other queries or processing. As an interval for a cache on disk approaches, we can prefetch the contents before they are needed. This scheme could potentially expand the memory limit our approach imposes, particularly in environments with cached continuous queries possessing long update intervals.

A far more flexible continuous querying scheme could be achieved by broadening our approach to incorporate some of the more common data stream processing techniques such as a sliding window over the incoming data. Such an implementation gives clients the choice of tradeoffs involved in the various continuous query techniques.

ACKNOWLEDGEMENTS

This research has been supported by funding from the US Department of Homeland Security's Long Range program (HSHQDC-13-C-B0018).

REFERENCES

[1] Chen, Y., Lwin, K., and Williams, S.: 'Continuous Query Processing and Dissemination', in Editor (Ed.)^(Eds.): 'Book Continuous Query Processing and Dissemination' (Citeseer, edn.), pp.

[2] Jianjun Chen, D.J.D., Feng Tian, Yuan Wang: 'NiagaraCQ: a scalable continuous query system for Internet databases'. Proc. ACM SIGMOD international conference on Management of data, New York, NY, USA 2000 pp. Pages

[3] Amazon web services. (2013) "Amazon kinesis Developer Guide" http://awsdocs.s3.amazonaws.com/kinesis/latest/kinesis-dg.pdf

[4] Daniel J. Abadi, Y.A., Magdalena Balazinska, Mitch Cherniack, Jeong-hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Er Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, Stan Zdonik: 'The design of the borealis stream processing engine'. Proc. Conference on Innovative Data Systems Research, Asilomar, CA, USA 2005 pp. Pages

[5] Developers, G.: 'Mobile Backend Starter', 2013

[6] Brian Babcock, S.B., Mayur Datar, Rajeev Motwani, Jennifer Widom: 'Models and issues in data stream systems'. Proc. ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems 2002

[7] Matthew Malensek, S.L.P., Shrideep Pallickara: 'Expressive Query Support for Multidimensional Data in Distributed Hash Tables'. Proc. IEEE/ACM Conference on Utility and Cloud Computing, Chicago, USA 2012 pp. Pages

[8] Matthew Malensek, S.L.P., Shrideep Pallickara: 'Galileo: A Framework for Distrubuted Storage of High-Throughput Data Streams'. Proc. IEEE/ACM Conference on Utility and Cloud Computing, Melbourne, Australia 2011 pp. Pages

[9] Matthew Malensek, S.P., Shrideep Pallickara: 'Exploiting Geospatial and Chronological Characteristics in Data Streams to Enable Efficient Storage and Retrievals', Future Generation Computer Systems, 2013, 29, (4), pp. 1049-1061

[10] Sangmi Pallickara, M.M., Shrideep Pallickara: 'Enabling Access to Time-Series, Geospatial Data for On Demand Visualization'. Proc. IEEE Symposium on Large-Scale Data Analysis and Visualization, Providence, Rhode Island 2011 pp. Pages

[11] Suhee Kim, S.H.S., John A. Stankovic: 'Performance Evaluation on a Real-Time Database'. Proc. IEEE Real-Time Technology and Applications Symposium2002 pp. Pages

[12] John A. Stankovic, S.H.S., Jörg Liebeherr: 'BeeHive: Global Multimedia Database Support for Dependable Real-Time Applications', University of Virginia Dept. of Computer Science Tech Report, 1998

[13] Avinash Lakshman, P.M.: 'Cassandra: a decentralized structured storage system', ACM SIGOPS Operating Systems Review, 2007, 44, (2), pp. 35-40

[14] Giuseppe DeCandia, D.H., Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, Werner Vogels: 'Dynamo: amazon's highly available key-value store', in Editor (Ed.)^(Eds.): 'Book Dynamo: amazon's highly available key-value store' (ACM, 2007, edn.), pp. 205-220

[15] Wikipedia Contributors. (2013) Geohash. [Online]. Available: http://en.wikipedia.org/wiki/Geohash

[16] Welch, T.A.: 'A Technique for High-Performance Data Compression', Computer, 1984, 17, (6), pp. 8-19

[17] W3C (1999). "Resource Description Framework (RDF) Model and Syntax Specification." from http://www.w3.org/TR/PR-rdf-syntax/

[18] W3C (2013). "SPARQL 1.1 Overview." http://www.w3.org/TR/sparql11-overview/http://www.w3.org/TR/sparql11-overview/

[19] NOAA. (2013) The NAM. [Online]. Available: http://www.emc.ncep.noaa.gov/index.php?branch=NAM